

# DevTest Solutions

## Using CA Service Virtualization

Version 8.0



This Documentation, which includes embedded help systems and electronically distributed materials, (hereinafter referred to as the "Documentation") is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be disclosed by you or used for any purpose other than as may be permitted in (i) a separate agreement between you and CA governing your use of the CA software to which the Documentation relates; or (ii) a separate confidentiality agreement between you and CA.

Notwithstanding the foregoing, if you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2014 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

# Contact CA Technologies

## Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At <http://ca.com/support>, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

## Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to [techpubs@ca.com](mailto:techpubs@ca.com).

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at <http://ca.com/docs>.





# Contents

---

Chapter 1: CA Service Virtualization	11
Introduction to Virtualization.....	11
Types of Service Virtualization .....	11
Service Virtualization Overview .....	12
High-level Virtualization Steps .....	13
Virtualization of Messaging Systems.....	15
Chapter 2: Installation and Configuration	17
How to Install CA Service Virtualization .....	17
System Requirements .....	18
How to Configure CA Service Virtualization .....	19
Set Up Users and Monitor Usage .....	19
Set the Proxy for localhost .....	20
Define Other Settings in local.properties.....	20
Install the Database Simulator .....	21
Other Startup Properties.....	23
DDLs for Major Databases.....	23
EclipseLink Properties for a Session .....	24
EclipseLink Properties for Schema .....	28
APPC Agent.....	31
APPC Agent Technical Deployment Information .....	32
Install the APPC Agent.....	33
Configure the APPC Agent.....	35
Chapter 3: Understanding CA Service Virtualization	39
How to Work with CA Service Virtualization .....	39
CA Service Virtualization Components.....	40
Virtual Service Models .....	41
Service Images.....	42
Imported Transactions .....	43
How Virtualization Works .....	43
How Conversational Requests are Handled .....	45
Magic Strings and Dates .....	47
Magic Strings.....	48
Magic Dates.....	52
Understanding VSE Transactions .....	53

---

Stateless and Conversational VSE Transactions .....	53
Navigation Tolerance .....	56
Logical Transactions .....	57
Match Tolerance .....	58
Argument Match Operators.....	59
Meta Transactions and Specific Responses .....	60
How to Debug Match Failures.....	60
Track Transactions.....	61
 Chapter 4: Using the DevTest Portal with CA Service Virtualization .....	 63
Open the DevTest Portal .....	63
 Chapter 5: Create a Virtual Service .....	 65
Record a Website (HTTP or HTTP/S) .....	66
Configure a Virtual Service .....	69
Save a Virtual Service .....	71
 Chapter 6: Editing Virtual Services .....	 75
Open a Virtual Service .....	75
Stateless Transactions View .....	76
Conversation View .....	81
Unknown Responses .....	82
Search for Text in a Virtual Service .....	83
Add a Note to a Signature .....	84
Add a Label to a Specific Transaction.....	84
Updating a Virtual Service Manually .....	85
Find the Transactions that Match a Request .....	94
Virtual Service URLs .....	95
Deploy a Virtual Service .....	96
 Chapter 7: Using the Workstation and Console with CA Service Virtualization .....	 99
 Chapter 8: Creating Service Images .....	 101
Open a Service Image.....	101
Combine Service Images .....	102
Delete a Service Image.....	102
Create a Service Image.....	103
Create a Service Image from Scratch .....	104

---

Create a Service Image from a WSDL.....	104
Create a Service Image from WADL .....	108
Create a Service Image from RAML.....	110
Create a Service Image from Layer 7 .....	112
Create a Service Image from Request/Response Pair .....	113
Create a Service Image from PCAP .....	117
Create a Service Image by Recording.....	119
Create and Deploy a Virtual Service with VSEasy .....	207
Work with VSMs.....	208
Using Data Protocols .....	209

## Chapter 9: Editing Service Images 273

Legacy Service Images.....	273
Open a Service Image to Edit .....	274
Service Image Tab .....	275
Transactions Tab .....	279
Transactions Tab for Stateless Transactions .....	280
Transactions Tab for Conversations .....	291
Conversation Editor .....	293
Service Images for JMS Transport Protocol .....	304

## Chapter 10: Editing a VSM 305

Virtual Service Router Step .....	307
Virtual Service Tracker Step .....	308
Virtual Conversational/Stateless Response Selector Step .....	309
Virtual HTTP/S Listener Step .....	310
Virtual HTTP/S Live Invocation Step .....	312
Virtual HTTP/S Responder Step.....	314
Virtual JDBC Listener Step .....	315
Virtual JDBC Responder Step .....	316
Socket Server Emulator Step.....	317
Messaging Virtualization Marker Step .....	319
Compare Strings for Response Lookup Step .....	320
Compare Strings for Next Step Lookup Step.....	322
Virtual Java Listener Step .....	324
Virtual Java Live Invocation Step.....	325
Virtual Java Responder Step.....	326
Virtual TCP/IP Listener Step .....	327
Virtual TCP/IP Live Invocation Step.....	329
Virtual TCP/IP Responder Step.....	330
Virtual CICS Listener Step .....	331

---

Virtual CICS Responder Step .....	331
CICS Transaction Gateway Listener Step.....	332
CICS Transaction Gateway Live Invocation Step .....	334
CICS Transaction Gateway Responder Step .....	335
Virtual DRDA Listener Step.....	336
Virtual DRDA Response Builder Step.....	336
Virtual DRDA Live Invocation Step .....	337
IMS Connect Listen Step .....	338
IMS Connect Live Invocation Step.....	339
Virtual IMS Connect Responder Step .....	340
JMS VSE Steps .....	341
JCo IDoc Listener Step .....	346
JCo IDoc Live Invocation Step.....	347
JCo IDoc Responder Step .....	347
JCo RFC Listener Step .....	348
JCo RFC Live Invocation Step .....	349
JCo RFC Responder Step.....	350
 Chapter 11: Desensitizing Data .....	 351
Dynamic Desensitization .....	351
Static Desensitization .....	352
Data Desensitizer Data Protocol Handler.....	352
 Chapter 12: Virtualizing a Service .....	 353
Preparing for Virtualization .....	354
Deploy and Run a Virtual Service .....	355
Running Live Requests .....	357
Session Viewing and Model Healing .....	369
VSE Metrics .....	371
 Chapter 13: VSE Manager - Manage and Deploy Virtual Services .....	 381
Install VSE Manager.....	381
Use VSE Manager .....	382
 Chapter 14: VSE Commands .....	 387
VSE Manager Command - Manage Virtual Service Environments .....	388
ServiceManager Command - Manage Services .....	389
ServiceImageManager Command - Manage Service Images .....	390
VirtualServiceEnvironment Commands .....	395

---

Chapter 15: Java Agent VSE Properties	397
Glossary	401



# Chapter 1: CA Service Virtualization

---

This section contains the following topics:

[Introduction to Virtualization](#) (see page 11)

[Types of Service Virtualization](#) (see page 11)

[Service Virtualization Overview](#) (see page 12)

[High-level Virtualization Steps](#) (see page 13)

## Introduction to Virtualization

*Virtualization* usually refers to hardware virtualization, where the behavior of a physical asset, such as a server or application in a software emulator, is simulated and the emulator is hosted in a virtual environment. The virtual environment provides the same communication with the emulated asset as the physical environment.

Virtualization has the following advantages:

- Better manages physical assets, which eases change and configuration management
- Improves the utilization of physical capacity, which better leverages the capacity of the physical assets
- Improves agility, which avoids the costly delays that happen while waiting for IT to reconfigure or switch servers

CA Service Virtualization provides service virtualization. The process is in principle the same as that for hardware virtualization.

## Types of Service Virtualization

CA Service Virtualization has two product configurations, optimized for specific customer applications:

- CA Service Virtualization
- CA Service Virtualization for Performance

CA Service Virtualization is best for use cases in development, integration, testing, and user acceptance. Instances of these products service up to 10 parallel transactions simultaneously, or approximately 10 transactions a second.

CA Service Virtualization for Performance is specifically for performance testing applications, and is more scalable, only limited by the underlying hardware and network.

## Service Virtualization Overview

*Service virtualization* is the imaging of software service behavior and the modeling of a virtual service to stand in for the actual service during development and testing. Service virtualization is complementary to hardware virtualization and addresses its limitations. The word *virtualization* refers to service virtualization in this documentation.

You may not want or be able to stay connected to the server for your quality assurance tasks. CA Service Virtualization emulates the behavior of the server.

Within CA Service Virtualization, you can use a virtual service environment (VSE) as the client to drive the service to be recorded. However, you will most likely exercise the service for recording with your client (for example, a browser or an in-house application).



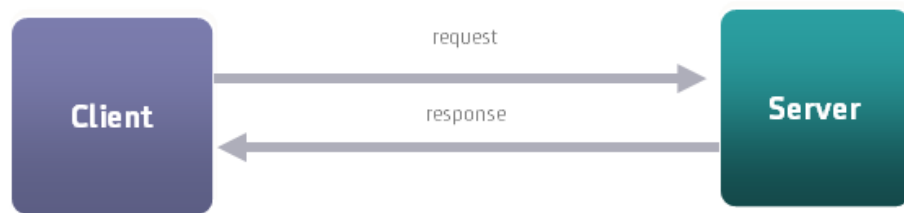
## High-level Virtualization Steps

The high-level steps in CA Service Virtualization are:

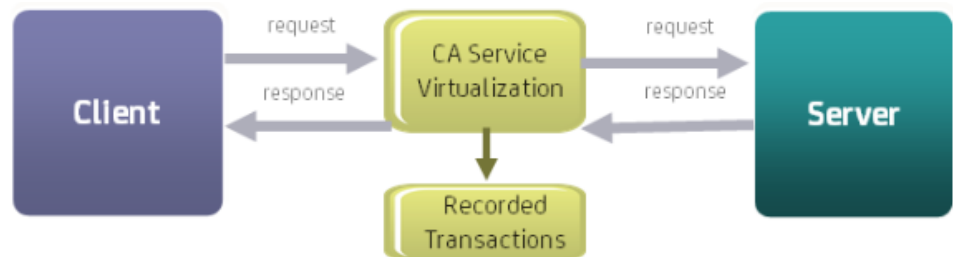
1. Take an image of the service behavior (service image); record transactions that the server handles.
2. Construct the virtual service from the behavior (virtual service model or VSM).
3. Deploy the VSM to the Virtual Service Environment (VSE). The VSM then looks at the captured service images to find the appropriate responses for requests coming in to the VSE.

The following graphics show that when recording the image, VSE acts as the pass through mechanism between the client and server. While VSE passes the requests and responses along, it records the transactions.

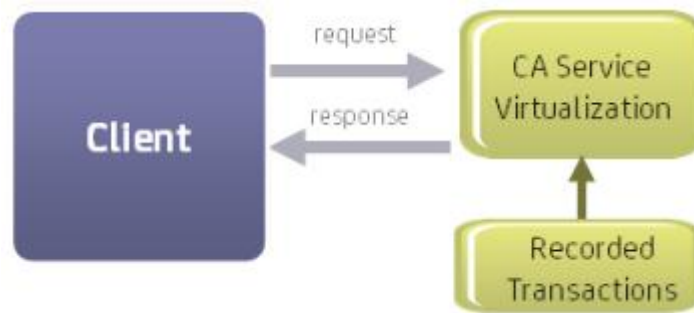
### Normal Operation



### Recording



At the time of virtualization, in the absence of the server, VSE responds to the client requests by consulting the recorded transactions.



## Virtualization of Messaging Systems

Message-oriented middleware (MOM), or messaging systems, are services that provide a means to enable asynchronous communication between two or more software applications. This communication always happens in the form of messages. The messages are posted to message destinations configured in the MOM.

The types of message destinations are:

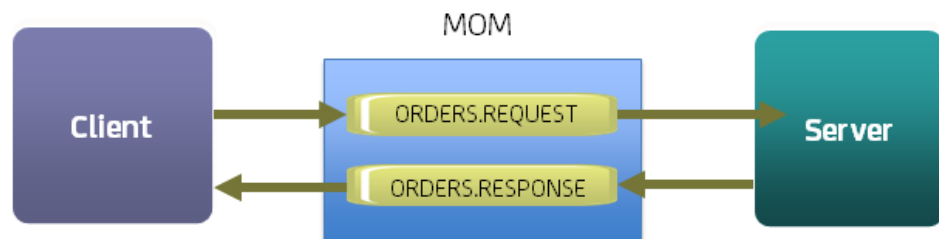
### Queues

A publisher adds a message to the queue, and a subscriber pulls messages from the queue, in a "first in, first out" fashion.

### Topics

A publisher publishes a message to a topic, and all subscribers that subscribe to the topic receive the message.

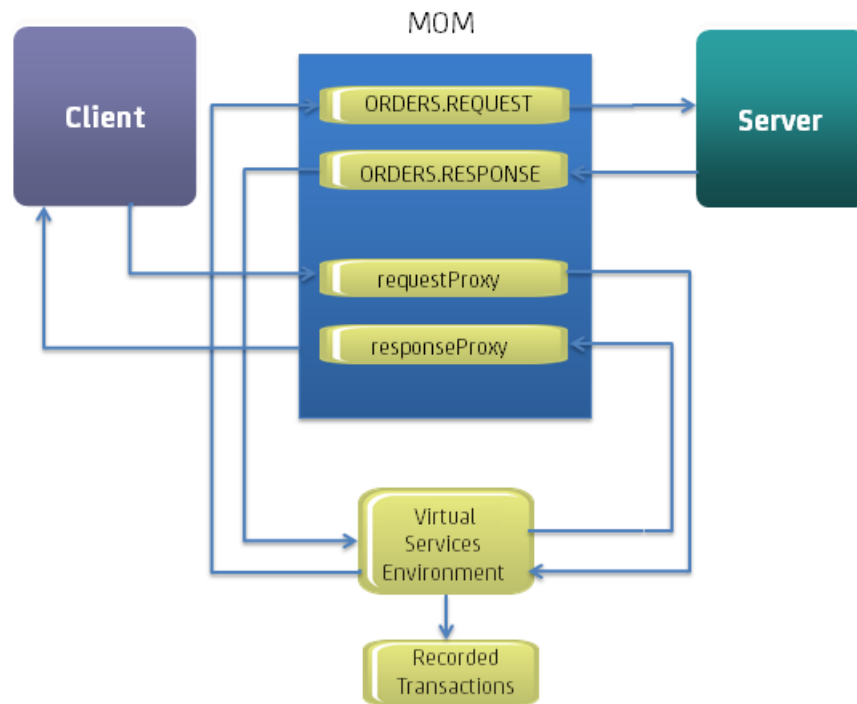
The following graphic shows a simple message-based service. In this scenario, the client adds messages to a queue (**ORDERS.REQUEST**), which the server picks up. The response from the server is in the form of messages added to another queue (**ORDERS.RESPONSE**). The client then picks them up.



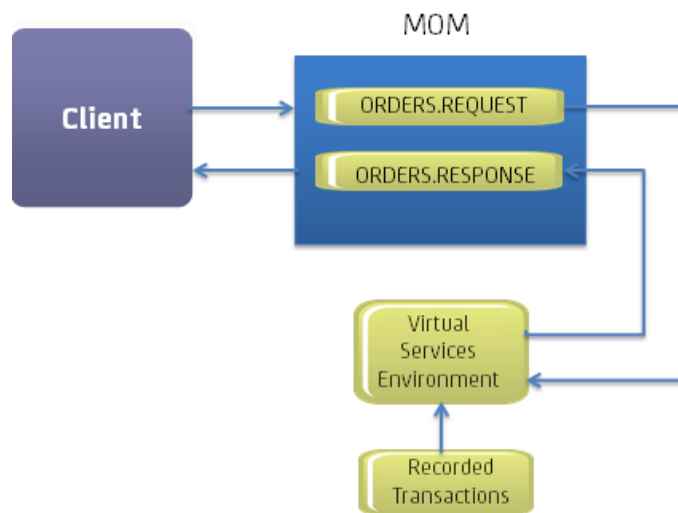
Possible variations include:

- Using topics instead of queues
- Multiple responses to a single request, which can possibly target different destinations

CA Service Virtualization aims to virtualize the server. In the recording mode, VSE requires extra proxy destinations (**requestProxy** and **responseProxy** queues in the following graphic) that the client uses instead of their counterparts. The server still listens and posts to the real destinations. VSE acts as a pass through between these proxy and real destinations. VSE records the traffic to create the VSM and the service image that it needs for the virtualization.



Later, when VSE virtualizes the server, it works with the real destinations. VSE does not need the proxy destinations.



# Chapter 2: Installation and Configuration

---

This section contains the following topics:

[How to Install CA Service Virtualization](#) (see page 17)

[How to Configure CA Service Virtualization](#) (see page 19)

[Install the Database Simulator](#) (see page 21)

[DDLs for Major Databases](#) (see page 23)

[APPC Agent](#) (see page 31)

## How to Install CA Service Virtualization

The CA Service Virtualization software is installed with DevTest Server. For information about the installation and configuration of DevTest Solutions, see *Installing*.

To use CA Service Virtualization, the following processes (or services) must be running:

- Registry (DevTest Registry Service)
- VirtualServiceEnvironment (VSE Service)

As a server level service, CA Service Virtualization can coexist with a registry that has an attached coordinator and simulator. The simulator and coordinator are not mandatory to run CA Service Virtualization.

## System Requirements

The following system resources for CA Service Virtualization are baseline requirements only:

- **CPU:** 2 GHz or faster
- **RAM:** 2 GB or more
- **Disk Space:** 5 GB of free space
- **OS:** Recommended: 64-Bit operating system. Supported: Windows 2008, 7, 8, Linux, Solaris, AIX 6.1 (LISA 5.0 and later)

For larger scale CA Service Virtualization deployments, we recommend the following resources:

- 256 Virtual Service Threads for each VSE instance
- 1 Processor Core and 2GB RAM for each VSE instance

### Example: 1,000,000 transactions each day

- 1 thread for each service to support functional tests
- About 6 threads for each service to support virtualization for load and performance tests.
- Eight cores are equal to 2,048 concurrent virtual service threads.
- 16 GB RAM (for DevTest).

For more information about other system requirements, see System Requirements and Prerequisites in *Installing*.

## Software Directory Structure and Files

The following list describes the DevTest Solutions directory structure. The directories are in the DevTest root installation folder.

**bin**

Contains executables, such as TestRegistry.exe, Workstation.exe, VirtualServiceEnvironment.exe, and VSEManager.exe.

**DemoServer**

Contains the DevTest demo server.

**doc**

Contains DevTest documentation.

**examples/vse**

Contains examples relating to VSE.

**tmp**

Contains the VSE workspace where VSE temporarily stores conversations and stateless transactions.

**vseDeploy**

Contains deployed VSMs and related data.

## How to Configure CA Service Virtualization

To configure CA Service Virtualization properly, complete the following tasks:

1. [Set up users and monitor usage](#) (see page 19).
2. [Set the proxy for localhost](#) (see page 20).
3. [Define other settings in local.properties \(optional\)](#) (see page 20).

## Set Up Users and Monitor Usage

Your DevTest Solutions license is for the entire product. The license agreement provides for a given maximum number of concurrent users of the SV Power User user type (among other user types). An administrator grants CA Service Virtualization users various permissions that are associated with the SV Power User user type. Periodically, an administrator generates a Usage Audit Report to monitor compliance with the maximum concurrent usage. See *Administering*.

## Set the Proxy for localhost

When DevTest acts as the HTTP client for VSE, the HTTP traffic from DevTest must be passed to VSE. A common way to pass HTTP traffic is to set VSE as the web proxy. However, proxy use is disabled by default for simple names like "localhost".

You can override this behavior in the **local.properties** file.

### Follow these steps:

1. Open the **local.properties** file in the root DevTest installation folder.
2. Uncomment the **lisa.http.webProxy.nonProxyHosts.excludeSimple** property.
3. Set the value for this property to false.

**lisa.http.webProxy.nonProxyHosts.excludeSimple=false**

4. Save and close the file.

## Define Other Settings in local.properties

You can optionally set the following extra configurations in **local.properties**:

- **lisa.vseName=VSENAME**

To rename the VSE server, add this property and change the value where *VSENAME* is the name of the VSE server.

- One of the following:

**lisa.registryName=REGISTRY**

or

**lisa.registryName=tcp://111.666.11.198:2010/REGISTRY**

### To connect to another test registry:

1. Add the **lisa.registryName** property to **local.properties**.
2. Change the values of *REGISTRY* to the name of the new test registry.



## Install the Database Simulator

To record JDBC traffic, install the DevTest simulation JDBC driver on the database client. The database client uses the DevTest driver instead of the actual driver.

### Follow these steps:

1. Copy the JAR file **lisajdbcsim.jar** in the LISA\_HOME\lib directory to the classpath of your database.

This JAR file contains the DevTest simulation JDBC driver. To help with the use of demo server as the database client, the driver is already copied into the demo server in the DEMO\_HOME\jboss\server\default\lib directory.

2. Set the driver class to **com.itko.lisa.vse.jdbc.driver.Driver**. Set the driver class, depending on the style of JDBC that the database client uses:

#### DriverManager Style

In an application server, it is not likely that the database client uses the Java DriverManager to acquire connections. If the application server does, add the following command to the startup command for the database client:

```
-Djdbc.drivers=com.itko.lisa.vse.jdbc.driver.Driver
```

If this property is already used, add the DevTest driver to the front. Separate the DevTest driver from the rest of the driver class names with a colon ( : ).

#### DataSource Style

If the database client uses the DataSource style for acquiring a connection (as the Demo Server does), then update its configuration. Where the configuration specifies a data source definition, specify **com.itko.lisa.vse.jdbc.driver.Driver** as the JDBC driver to use, and a connection URL.

3. To achieve a pass through, modify the connection URL so that the DevTest Simulation JDBC driver can identify the real driver information. Format the connection URL as follows:

```
name=value[;name=value...]
```

#### **name**

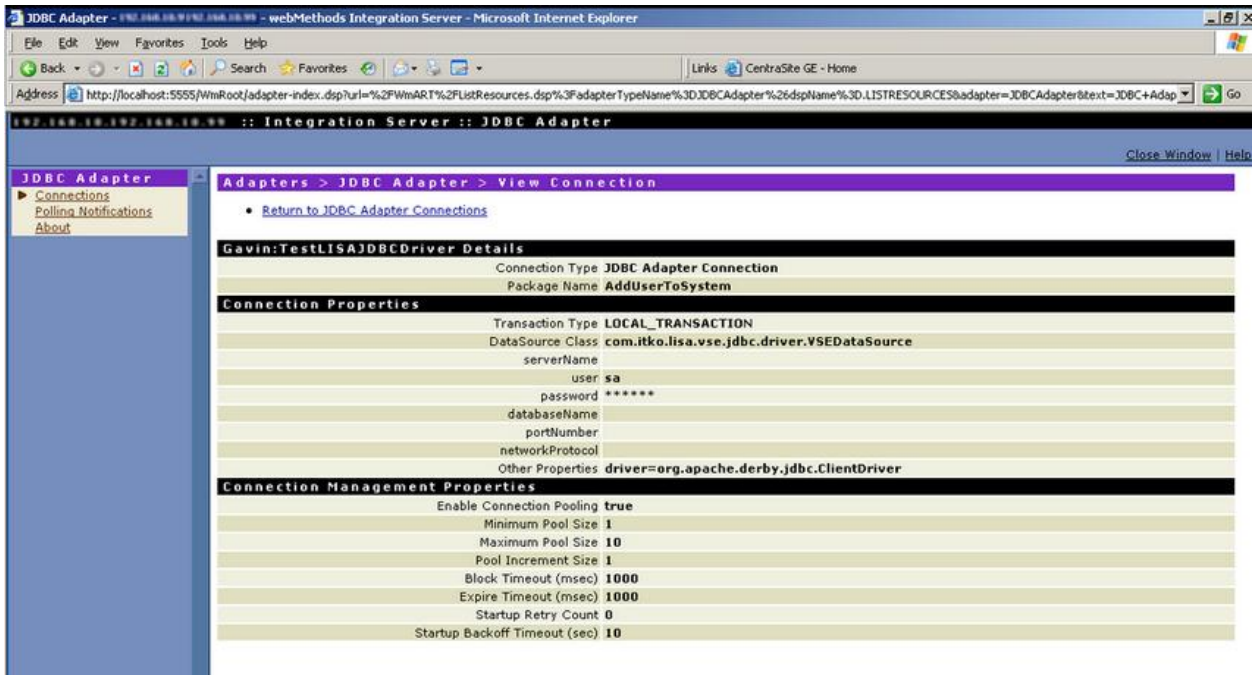
**jdbc:lisasim:driver:** The value must be the fully qualified class name of the real JDBC driver to use.

**url:** The value must be set to the connection URL that the real driver expects. (It must be defined as the last property so that it can contain semi-colons.)

**Note:** VSE does not support the Oracle thin driver as a pass through driver. The Oracle thin driver does not provide the full JDBC implementation. If you use Oracle as a database, use other JDBC drivers for virtualization.

For an example of setting the connection URL, see  
**DEMO\_HOME\jboss\server\default\deploy\itko-example-ds.xml**.

The following graphic shows an example of virtualizing the database in a WebMethods environment.



To use both the simulation and CA Continuous Application Insight drivers, make the simulation driver the "outside" driver. Specify the CAI class and URL. See the **itko-example-ds.xml** file in the **DEMO\_HOME\jboss\server\default\deploy** folder for an example that defines a DevTest data source to JBoss for DevTest demo server.

## Other Startup Properties

Regardless of the connection style of the database client, you can add the following properties to the startup command for the database client to affect the simulation driver.

### **`lisa.jdbc.sim.require.remote`**

Specifies whether the driver requires an active connection with a DevTest Workstation or VSE Server to run. This method is the best way to have a database client synchronize with VSE to record or play back any startup database activity that the server performs.

#### **Values:**

- **true:** The driver blocks until there is an active connection with a DevTest Workstation or VSE Server.
- **false:** The driver does not require an active connection.

**Default:** false

### **`lisa.jdbc.sim.port`**

Defines the IP port on which the driver listens for connections from a recorder or a running virtual service model.

**Default:** 2999

## DDLs for Major Databases

To have DevTest generate a DDL for a file, add the following lines to the **local.properties** file:

```
eclipselink.ddl-generation=create-tables
eclipselink.ddl-generation.output-mode=sql-script
eclipselink.target-database=Oracle
```

Starting DevTest with these properties creates the following files that contain the necessary DDL:

- `createDDL.jdbc`
- `dropDDL.jdbc`

You can generate DDLs for a different DBMS by changing the target-database value. For more information, see:

- [EclipseLink Persistence Unit Properties for Session](#) (see page 24)
- [EclipseLink Persistence Unit Properties for Schema Generation](#) (see page 28)

## EclipseLink Properties for a Session

You can use the following EclipseLink JPA persistence unit properties in a **persistence.xml** file to configure EclipseLink extensions for a session, and as the target database and application server.

### **eclipselink.session-name**

Defines the name by which the EclipseLink session is stored in the static session manager. Use this option if you must access the EclipseLink shared session outside of the context of the JPA. Use this option also to use a preexisting EclipseLink session that is configured through an EclipseLink **sessions.xml** file.

**Values:** A valid EclipseLink session name that is unique in a server deployment.

**Default:** EclipseLink-generated unique name.

#### **Example:**

```
persistence.xml file<property value="MySession"/> Example:
property Mapimport
org.eclipse.persistence.config.PersistenceUnitProperties;pr
opertiesMap.put(PersistenceUnitProperties.SESSION_NAME,
"MySession");
```

### **eclipselink.sessions-xml**

Defines the persistence information that is loaded from the EclipseLink session configuration file, **sessions.xml**.

You can use this option as an alternative to annotations and deployment XML. If you specify this property, EclipseLink overrides all class annotation and the object relational mapping from the persistence.xml, and ORM.xml and other mapping files.

To indicate the session, set the **eclipselink.session-name** property.

**Note:** If you do not specify the value for this property, **sessions.xml** file is not used.

**Values:** The resource name of the sessions XML file.

#### **Example:**

```
persistence.xml file<property value="mysession.xml"/> Example: property
Mapimport
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(Persist
enceUnitProperties.SESIONS_XML, "mysession.xml");
```

### **eclipselink.session-event-listener**

Defines a descriptor event listener to be added during bootstrapping.

**Values:** Qualified class name for a class that implements the `org.eclipse.persistence.sessions.SessionEventListener` interface.

#### **Example:**

Persistence.xml file<property value="mypackage.MyClass.class"/> Example:  
property Mapimport  
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(PersistenceUnitProperties.SESSION\_EVENT\_LISTENER\_CLASS, "mypackage.MyClass.class");

#### **eclipselink.session.include.descriptor.queries**

Specifies whether to copy all named queries from the descriptors to the session by default. These queries include the ones that are defined using EclipseLink API, descriptor amendment methods, and others.

##### **Values:**

- **true:** Enable the default copying of all named queries from the descriptors to the session.
- **false:** Disable the default copying of all named queries from the descriptors to the session.

**Default:** true.

##### **Example:**

Persistence.xml file<property value="false"/>Example: property Mapimport  
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(PersistenceUnitProperties.INCLUDE\_DESCRIPTOR\_QUERIES, "false");

#### **eclipselink.target-database**

Specifies the type of database that your JPA application uses.

##### **Values:**

The following values are valid for use in a **persistence.xml** file and for the **org.eclipse.persistence.config.TargetDatabase**:

- **Attunity:** Configure the persistence provider to use an Attunity database.
- **Auto:** EclipseLink accesses the database and uses the metadata that JDBC provides to determine the target database. This value applies to the JDBC drivers that support this metadata.
- **Cloudscape:** Configure the persistence provider to use a Cloudscape database.
- **Database:** If your target database is not listed here and your JDBC driver does not support the use of metadata that the Auto option requires, configure the persistence provider to use a generic choice.
- **DB2:** Configure the persistence provider to use a DB2 database.
- **DB2Mainframe:** Configure the persistence provider to use a DB2 mainframe database.
- **DBase:** Configure the persistence provider to use a DBase database.
- **Derby:** Configure the persistence provider to use a Derby database.
- **HSQL:** Configure the persistence provider to use an HSQL database.
- **Informix:** Configure the persistence provider to use an Informix database.

- **JavaDB:** Configure the persistence provider to use a Java DB database.
- **MySQL:** Configure the persistence provider to use a MySQL database.
- **Oracle:** Configure the persistence provider to use an Oracle Database.
- **PointBase:** Configure the persistence provider to use a PointBase database.
- **PostgreSQL:** Configure the persistence provider to use a PostgreSQL database.
- **SQLAnywhere:** Configure the persistence provider to use an SQLAnywhere database.
- **SQLServer:** Configure the persistence provider to use an SQLServer database.
- **Sybase:** Configure the persistence provider to use a Sybase database.
- **TimesTen:** Configure the persistence provider to use a TimesTen database. You can also set the value to the fully qualified classname of a subclass of the `org.eclipse.persistence.platform.DatabasePlatform` class.

**Default:** Auto

**Example:**

Persistence.xml file<property value="Oracle"/>Example: property Mapimport  
org.eclipse.persistence.config.TargetDatabase;import  
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(Persist  
enceUnitProperties.TARGET\_DATABASE, TargetDatabase.Oracle);

**eclipselink.target-server**

Specifies the type of application server that your JPA application uses.

**Values:**

The following values are valid for use in the **persistence.xml** file and the **org.eclipse.persistence.config.TargetServer**:

- **None:** Configure the persistence provider to use no application server.
- **WebLogic:** Configure the persistence provider to use Oracle WebLogic Server. This server sets this property automatically. Set it only if it is disabled.
- **WebLogic\_9:** Configure the persistence provider to use Oracle WebLogic Server version 9.
- **WebLogic\_10:** Configure the persistence provider to use Oracle WebLogic Server version 10.
- **OC4J:** Configure the persistence provider to use OC4J.
- **SunAS9:** Configure the persistence provider to use Sun Application Server version 9. This server sets this property automatically. Set it only if it is disabled.
- **WebSphere:** Configure the persistence provider to use WebSphere Application Server.

- **WebSphere\_6\_1:** Configure the persistence provider to use WebSphere Application Server version 6.1.
- **JBoss:** Configure the persistence provider to use JBoss Application Server.
- The fully qualified class name of a custom server class that implements the **org.eclipse.persistence.platform.ServerPlatform** interface.

**Default:** None

**Example:**

persistence.xml file<property value="OC4J\_10\_1\_3"/>Example: property  
Mapimport org.eclipse.persistence.config.TargetServer;import  
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(Persist  
enceUnitProperties.TARGET\_SERVER, TargetServer.OC4J\_10\_1\_3);

## EclipseLink Properties for Schema

You can define the EclipseLink JPA persistence unit properties in a `persistence.xml` file to configure schema generation.

### **eclipselink.ddl-generation**

Specifies the Data Definition Language (DDL) generation action to use for your JPA entities. To specify the DDL generation target, see **eclipselink.ddl-generation.output-mode**.

#### **Values:**

You can use the following values in a `persistence.xml` file:

- **none:** EclipseLink does not generate a DDL; no schema is generated.
- **create-tables:** EclipseLink tries to execute a CREATE TABLE SQL command for each table. When you issue a CREATE TABLE SQL command for an existing table, EclipseLink follows the default behavior of your specific database and JDBC driver combination. In most cases, an exception is thrown, the table is not created, and EclipseLink continues with the next statement.
- **drop-and-create-tables** – EclipseLink tries to DROP all tables, then CREATE all tables. If EclipseLink encounters issues, it follows the default behavior of your specific database and JDBC driver combination. EclipseLink then continues with the next statement.

The following values are valid for the **org.eclipse.persistence.config.PersistenceUnitProperties**:

- **NONE**
- **CREATE\_ONLY**
- **DROP\_AND\_CREATE**

If you use persistence in a Java SE environment and you want to create the DDL files without creating tables, define a Java system property **INTERACT\_WITH\_DB** and set the value to **false**.

**Default:** One of the following:

- **None**
- **PersistenceUnitProperties.NONE**

#### **Example:**

```
persistence.xml file<property value="create-tables"/>Example: property
Mapimport
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(Persist
enceUnitProperties.DDL_GENERATION, PersistenceUnitProperties.CREATE_ONLY);
```

### **eclipselink.application-location**

Specifies where EclipseLink writes generated DDL files. Files are written if you set **eclipselink.ddl-generation** to anything other than **none**.



**Value:** A file specification to a directory in which you have write access. The file specification can be relative to your current working directory or absolute. If it does not end in a file separator, EclipseLink appends one that is valid for your operating system.

**Default:** One of the following:

"."+File.separator

or

<tt>PersistenceUnitProperties.DEFAULT\_APP\_LOCATION</tt>

**Example:**

persistence.xml file<property value="C:\ddl\"/>Example: property Mapimport org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(PersistenceUnitProperties.APP\_LOCATION, "C:\ddl\");

#### **eclipselink.create-ddl-jdbc-file-name**

Specifies the file name of the DDL file that EclipseLink generates containing SQL statements to create tables for JPA entities. This file is written to the location specified by **eclipselink.application-location** when **eclipselink.ddl-generation** is set to create-tables or drop-and-create-tables.

**Values:** A file name valid for your operating system. Optionally, if the concatenation of **eclipselink.application-location** with **eclipselink.create-ddl-jdbc-file-name** is a valid file specification for your operating system, you can prefix the file name with a file path.

**Default:** One of the following:

■ createDDL.jdbc

or

■ PersistenceUnitProperties.DEFAULT\_CREATE\_JDBC\_FILE\_NAME

**Example:**

persistence.xml file<property value="create.sql"/> Example: property Mapimport org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(PersistenceUnitProperties.CREATE\_JDBC\_DDL\_FILE, "create.sql");

#### **eclipselink.drop-ddl-jdbc-file-name**

Specifies the file name of the DDL file that EclipseLink generates containing the SQL statements to drop tables for JPA entities. This file is written to the location specified by **eclipselink.application-location** when **eclipselink.ddl-generation** is set to **drop-and-create-tables**.

**Values:** A file name valid for your operating system. Optionally, if the concatenation of **eclipselink.application-location** with **eclipselink.drop-ddl-jdbc-file-name** is a valid file specification for your operating system, you can prefix the file name with a file path.

**Default:** One of the following:

- dropDDL.jdbc

or

- PersistenceUnitProperties.DEFAULT\_DROP\_JDBC\_FILE\_NAME

**Example:**

persistence.xml file<property value="drop.sql"/>Example: property Mapimport  
org.eclipse.persistence.config.PersistenceUnitProperties;propertiesMap.put(PersistenceUnitProperties.DROP\_JDBC\_DDL\_FILE, "drop.sql");

**eclipselink.ddl-generation.output-mode**

Defines the DDL generation target.

**Values:**

The valid values for the use in the **persistence.xml** file are:

**both:**

- Generate the SQL files and execute them on the database.

If **eclipselink.ddl-generation** is set to "create-tables," **eclipselink.create-ddl-jdbc-file-name** is written to **eclipselink.application-location** and then executes on the database.

If **eclipselink.ddl-generation** is set to "drop-and-create-tables," both **eclipselink.create-ddl-jdbc-file-name** and **eclipselink.drop-ddl-jdbc-file-name** are written to **eclipselink.application-location**. Both SQL files execute on the database.

**database:**

- Execute SQL on the database only (do not generate the SQL files).

**sql-script:**

- Generate the SQL files only (do not execute them on the database).

If **eclipselink.ddl-generation** is set to "create-tables," then **eclipselink.create-ddl-jdbc-file-name** is written to **eclipselink.application-location**. The command does not execute on the database.

If **eclipselink.ddl-generation** is set to "drop-and-create-tables," both **eclipselink.create-ddl-jdbc-file-name** and **eclipselink.drop-ddl-jdbc-file-name** are written to **eclipselink.application-location**. Neither is executed on the database. The following values are valid for the

**org.eclipse.persistence.config.PersistenceUnitProperties:**

- DDL\_BOTH\_GENERATION – see both.
- DDL\_DATABASE\_GENERATION – see the database.
- DDL\_SQL\_SCRIPT\_GENERATION – see sql-script.

**Default:** The default for Container or Java EE mode is {{database}},

**Example:**

```
persistence.xml file<property value="database"/>Example:
property Mapimport
org.eclipse.persistence.config.PersistenceUnitProperties;pr
opertiesMap.put(PersistenceUnitProperties.DDL_GENERATION_MO
DE, PersistenceUnitProperties.DDL_DATABASE_GENERATION);
```

**Note:** Override this setting by containers with specific EclipseLink support. See your container documentation for details. Bootstrap or Java SE mode: **both** or **PersistenceUnitProperties.DDL\_BOTH\_GENERATION**.

## APPC Agent

The APPC agent allows you to virtualize APPC transaction programs. The agent is an APPC CPI-C API-based Java program that acts as a proxy between the APPC client and server programs. The agent forwards the messages that an APPC client sends to an APPC server and conversely.

You can configure the agent to virtualize a transaction program that an APPC CPI-C destination name defines. You can use the agent with a VSE recorder to record the requests and responses that are exchanged between the APPC client and server transaction program. In the recording mode, it communicates with the live client and server and records all the messages that are exchanged. The agent creates an APPC virtual service you can deploy to VSE.

In the playback mode, when you deploy the recorded APPC virtual service, the live client continues to communicate with the APPC agent. The responses come from the APPC virtual service running in VSE instead of from the live server. If recording or playback is not in progress, the agent acts in pass-through mode. In pass-through mode, the agent simply forwards the messages between the APPC client and server.

The APPC agent has the following limitations:

- The APPC agent supports one destination name for each agent.
- The APPC protocol does not support live invocation mode and model healing

**To get started with the APPC protocol:**

1. Install and configure the APPC agent, using the steps in [Install the APPC Agent](#) (see page 33).
2. Configure SNA for the APPC agent using [Configure the APPC Agent](#) (see page 35).
3. Record an APPC virtual service using the process in Record APPC Service Images.
4. To start using APPC service playback, follow the standard steps for [deploying and running a virtual service](#) (see page 355).

**The following topics are included.**

[APPC Agent Technical Deployment Information](#) (see page 32)

[Install the APPC Agent](#) (see page 33)

[Configure the APPC Agent](#) (see page 35)

## APPC Agent Technical Deployment Information

### Prerequisites

IBM Communication Server

### System Resources

#### Memory

Minimum 512 MB of available RAM

#### CPU

The [IBM-recommended CPU requirements](#) are adequate to run the APPC agent.

#### Hard Disk Storage

The APPC agent generates file system logs. You need at least 500 MB of disk space to ensure sufficient room for the logs over time. Remove the log archives every 30 to 90 days.

#### Java Version

The APPC agent requires Oracle JDK 1.7.0 or greater

#### Operating System

We support the APPC agent on all Linux platforms that IBM supports for Communication Server.

### Network

- The APPC Agent binds to port 9000 (default) for DevTest internal communication.
- The APPC Agent communicates with the DevTest registry (default port:2010).
- The APPC Agent communicates with the VSE Recorder that runs in DevTest Workstation over AMQ, which is done through the DevTest registry (default port:2010).
- The APPC Agent communicates with VSE over AMQ, which is done through the DevTest registry (default port:2010).

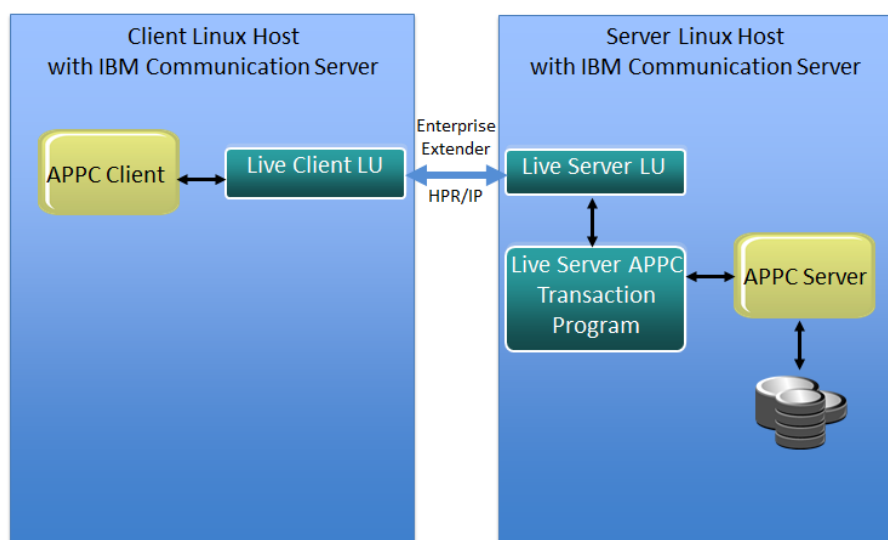
## Install the APPC Agent

The APPC agent is only supported on UNIX, and must be installed on a system that has the IBM Communications Server running. The agent needs the APPC CPI-C library to work correctly.

The agent runs on the APPC client, server, or an independent OS. We recommend that you install on the APPC client because it requires the least setup and allows you to decouple fully the APPC server during playback.

The following graphic shows the architecture of the APPC agent.

### Sample APPC Transaction Architecture



#### To install the agent:

1. Create a directory in LISA\_HOME named appc-agent.

The agent is located in LISA\_HOME/addons/appc-agent/appc-agent.zip.

**Note:** The APPC agent does not require that DevTest Solutions is installed on the target computer. You can copy the agent zip file from an existing DevTest installation and install it on any computer.

2. Unzip the agent in the appc-agent directory.

After you extract **appc-agent.zip**, the **lisa-sna-appc.properties** file will be at the top level of LISA\_HOME/appc-agent/.

3. Copy the cpic.jar file in the appc-agent/bin directory.

4. To configure the agent using environment variables, edit the **set-appc-env.sh** shell script in the **appc-agent** directory.

Environment variables in the shell script set the configuration properties. The configuration properties override any of the properties that are set in the properties file. If the **LIVE\_DESTINATION**, **VSE\_REGISTRY**, **VSE\_LAB**, **APPC\_AGENT\_PORT**, or **lisa.sna.appc.agent.registry.ping.timeout.millis** properties are not set in the shell script, the values in the **lisa-sna-appc.properties** file are used.

To configure the agent using a property file, edit **lisa-sna-appc.properties** in the **bin** directory. Provide the parameters to use if the shell script does not provide them for these fields: **LIVE\_DESTINATION**, **VSE\_REGISTRY**, **VSE\_LAB**, **APPC\_AGENT\_PORT**, and **lisa.sna.appc.agent.registry.ping.timeout.millis**.

The APPC agent uses the following environment variables:

**LD\_LIBRARY\_PATH**

Defines the directory where the SNA shared library is located.

**LD\_PRELOAD**

Defines the SNA library shared objects on your system.

**LIVE\_DESTINATION**

Defines the destination for the transaction program that the agent is virtualizing. For the sample programs included with the APPC agent, the live destination is **VSE\_SAM**. **VSE\_SAM** is a destination CPI-C name for the samples.

**VSE\_REGISTRY=tcp://host:port/Registry**

Defines the registry URL to which VSE and DevTest Workstation connect. Enter the IP address and port where the DevTest registry is running. Validate that the hostname can be resolved and that it does not have any DNS issues.

**VSE\_LAB**

Defines the lab name for the agent installation. "Default" works for standard installations.

**Default:** Default

**APPC\_AGENT\_PORT**

Defines the port to which the agent service binds.

**Default:** 9000

**lisa.sna.appc.agent.registry.ping.timeout.seconds**

Defines (in seconds) the timeout after which the agent stops waiting for a connection to the DevTest registry and the agent continues operation in pass-through mode.

**Default:** 5

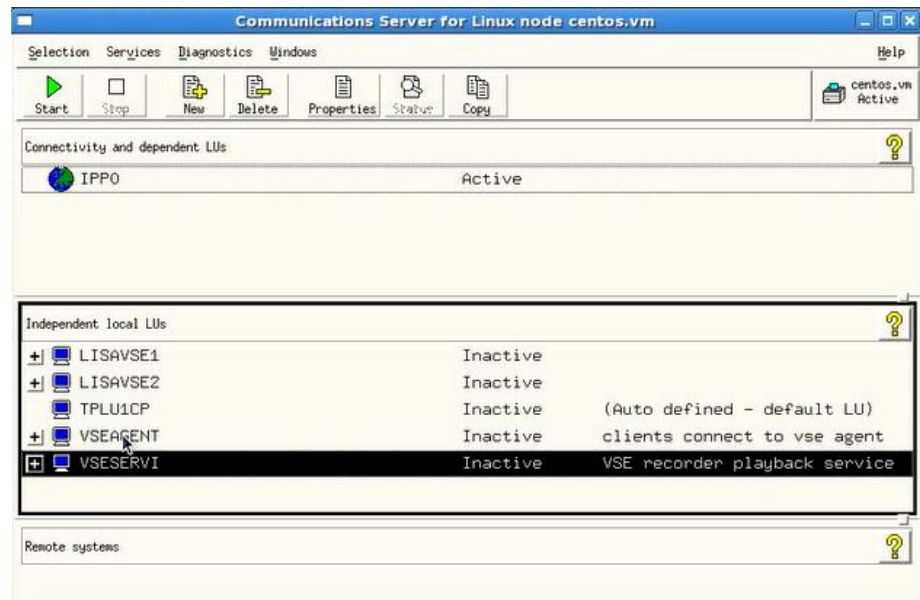
## Configure the APPC Agent

Follow these steps:

1. To specify logging options, edit the **appc-agent-logging.properties** file in the APPC agent bin directory.

For information about configuring logging options for [log4j](#), see the log4j documentation.

2. Configure the SNA properties on the system. To access the SNA Communications Manager, enter *snaconf* on the command line.
3. Configure a local logical unit pair that you can use to configure the agent transaction program and destination. Consult your SNA administrator for appropriate values for your system.



4. Select Services, APPC, Transaction program, Properties.
5. Configure the transaction program to point to the agent home directory/bin/run-appc-agent.sh.
6. Configure the stdout and the stderr paths to some files in the agent directory so that all the log files are in the same location.
7. Enter a description that indicates that this program is the agent transaction program.

**TP Invocation**

TP name  
Application TP /YSE\_AGENT\_TP

Service TP (hex)

LU  
Parameters are for invocation on any LU  
Parameters are for invocation on a specific LU

TP invocation  
☒ Queue incoming Allocates

Full path to TP executable /opt/appc-agent/bin/run-appc-age

Arguments I

User ID root

Group ID I

stdin Path I

stdout Path /opt/appc-agent/appc-agent.stdout

stderr Path /opt/appc-agent/appc-agent.stderr

Environment I

Description YSE Agent Controller TP

OK Cancel Help



8. When you have configured the agent transaction program, select Services, APPC, CPI-C, Properties to define a CPI-C destination.

**CPI-C destination**

Name: VSEAGENT

Local LU:  
☒ Specify local LU alias: VSEAGENT  
☐ Use default LU

Partner LU and mode:  
☒ Use PLU alias  
☒ Use PLU full name: JTKO . VSESERVI  
Mode: LOCHODE

Partner TP:  
☒ Application TP: VSE-AGENT-TP  
☐ Service TP (Hex)

Security:  
☒ None ☐ Same ☐ Program ☐ Program strong  
User ID:  
Password:

Description: VSE Agent Service Destination

OK Cancel Help

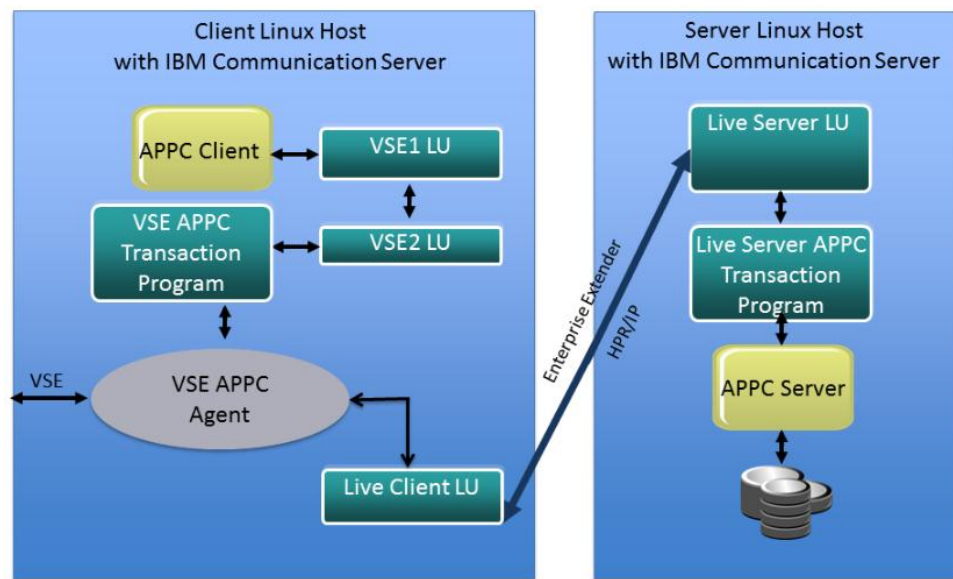
When you have configured the agent, you can either start recording or you can deploy a recorded virtual service to VSE. To point to the agent destination name, use your client programs.

If your client program points to a live destination name, you can switch it to VSEAGENT. Depending on how the destination name is passed to your client program, you could change client command-line parameters, a configuration file, or the client code.

If there is a recording started using DevTest Workstation, the agent records in recording mode. If the service is deployed based on the recording by this agent, it plays back. If recording or playback is not deployed, the service works in pass-through mode. In pass-through mode, it communicates with the live destination and forwards all requests and responses between the live client and the live server.

The following graphic shows the architecture of the APPC agent with VSE.

### Sample APPC Transaction Architecture with VSE



# Chapter 3: Understanding CA Service Virtualization

---

This section contains the following topics:

[How to Work with CA Service Virtualization](#) (see page 39)

[CA Service Virtualization Components](#) (see page 40)

[Virtual Service Models](#) (see page 41)

[Service Images](#) (see page 42)

[How Virtualization Works](#) (see page 43)

[Magic Strings and Dates](#) (see page 47)

[Understanding VSE Transactions](#) (see page 53)

[Match Tolerance](#) (see page 58)

[Track Transactions](#) (see page 61)

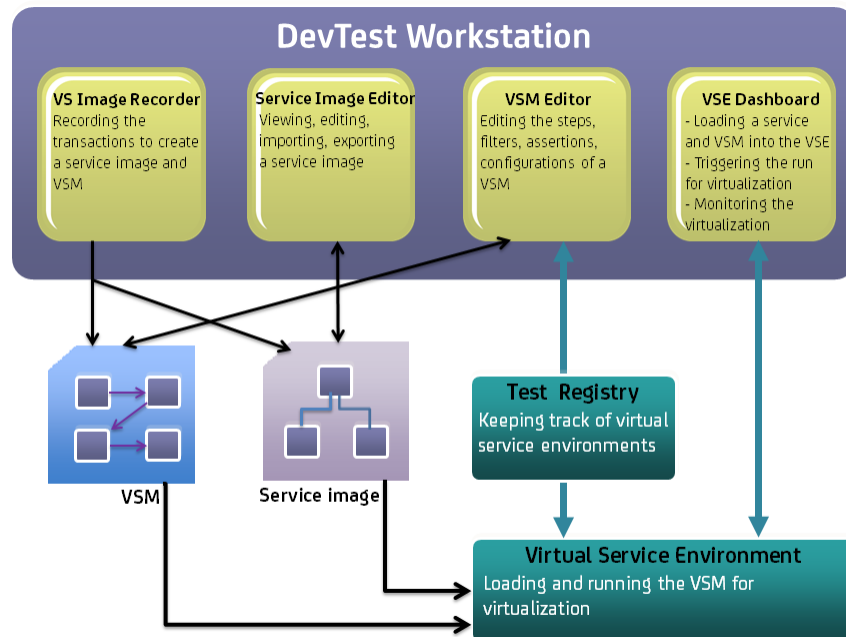
## How to Work with CA Service Virtualization

The general process for working with CA Service Virtualization includes the following steps:

1. Start the Virtual Service Image Recorder.
2. In the Virtual Service Image Recorder, provide basic information about what to record. Select the appropriate protocols. Provide any information that the protocols require.
3. In the Virtual Service Image Recorder, start the recording.
4. Exercise the client communication with the server routed through VSE.  
VSE records the traffic.
5. In the Virtual Service Image Recorder, finish the recording.
6. In the VSE Console, deploy the virtual service model and start the virtual service.
7. Run live requests against CA Service Virtualization.

## CA Service Virtualization Components

The following graphic shows how the components in CA Service Virtualization relate to each other.



The Virtual Service Image Recorder creates a service image and a VSM. The Service Image Editor and the VSM Editor let you view and edit them.

During virtualization, the VSM and service image load and run on a VSE that runs as a service. The Test Registry service tracks one or more VSEs that are running, and DevTest Workstation uses it to connect to the VSE. The VSE Dashboard is the web UI used to monitor and control the VSMs and service images that are loaded onto the VSEs.

## Virtual Service Models

You can conceptualize a virtual service model as a series of steps to be executed when a request is received. The steps of the VSM create and pass back a response to the request. A virtual service model is stored in a .vsm file.

A VSM must contain at least one VSE step from the Virtual Service Environment step list in DevTest Workstation to be deployable to a VSE.

After you record a service image, VSE automatically generates the protocol-specific steps in the VSM. You can modify any of the generated steps. You can also:

- Populate the responses from an Excel spreadsheet or by cross-referencing a database table and doing calculations on inputs
- Add some steps of different step types
- Manipulate the request and response steps before proceeding
- Specify the service image to use from the Response Selection step

After virtualization, a VSM must be deployed to the VSE. The VSM defines how behavior patterns get used and queries the service image to determine how to respond. The VSM knows how to navigate the service image. In general, VSMs are deployable to VSEs only, while test cases can be staged to coordinators, but not to VSEs.

## Service Images

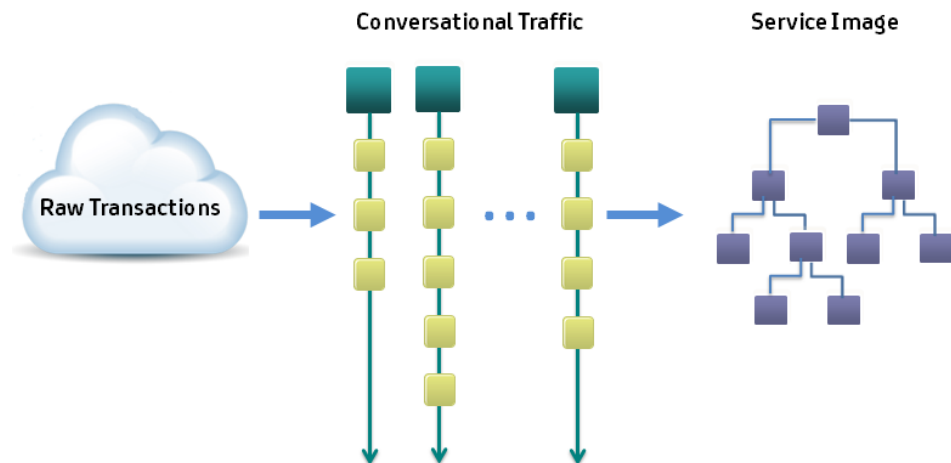
A *service image* is a recording of the interaction between the client and server as created by CA Service Virtualization. A virtual service model references a service image. After a service image is recorded, it is used to deliver the appropriate response to the client in the absence of the server.

A service image contains the following sets of information:

- A list of conversations (requests and responses) recorded as a conversation tree
- A list of stateless transactions (requests and responses)
- The responses to send when unknown conversational or stateless requests are encountered

You can view, edit, or create a service image with the Service Image Editor.

You can visualize a conversation as a series of stateful transactions. However, multiple conversations (from multiple sessions) can be recorded in the same service image. Similar request structures are merged into a single transaction to create a tree, as the following graphic shows.



For example, if multiple users log in to the system with **login()** transactions, all these transactions are merged into a single transaction. But if one user logs in with **login()** and another user logs in with **acquireAuthToken()**, the transactions are not merged.

## Imported Transactions

You can import the following XML documents into a service image being recorded:

### Raw Transactions

The XML document represents raw traffic as if coming directly from the network, characterized by a root element of `<rawTraffic>`.

For an example, see `LISA_HOME\examples\VServices\raw-traffic.xml`.

### Conversational Traffic

The XML document represents traffic that is rearranged into conversations, stateless transaction sets, or both. The traffic is organized into linear lists of transactions, each of which represent a "real" conversation and has a root element of `<traffic>`.

For an example, see `LISA_HOME\examples\VServices\traffic.xml`.

## How Virtualization Works

In the absence of a server, CA Service Virtualization simulates the behavior of the server for its client. This process is known as the virtualization of a server. The process requires loading the service image that a VSM references and running it in the VSE dashboard.

When VSE receives a request, VSE examines the request and tries to match it to an existing conversational state (session) in VSE. For example, a cookie ID or some other session identifier can "tie" requests in stateless protocols such as HTTP. VSE uses the conversational state to determine where in the conversation tree the "current transaction" is, and any other "state" such as a previously submitted authentication token, which may not be part of the current request, but is used in the subsequent response. The user name is an example.

If an existing session cannot be found, the VSE attempts to match the request against the starter transactions of each conversation in the image. If it finds a match, it creates a session and the session returns a relevant response. The session is maintained until two minutes after it has last been "seen" by the VSE. You can change this behavior with the **`lisa.vse.session.timeout.ms`** property. If no conversation starters match, no session is created and the list of stateless transactions is consulted in the order in which they are defined. If there is a match, the appropriate response is returned. If there is still no match, the "unknown request" response is sent.

When VSE finds a previous session and VSE is in a conversation, the next transaction match depends on many factors. Those factors can include navigation and match tolerances.

If VSE does not find a matching transaction in the conversational and stateless transactions, it consults the service image again for the type of response to send for an unmatched conversational or stateless request.

**More information:**

[How Conversational Requests are Handled](#) (see page 45)



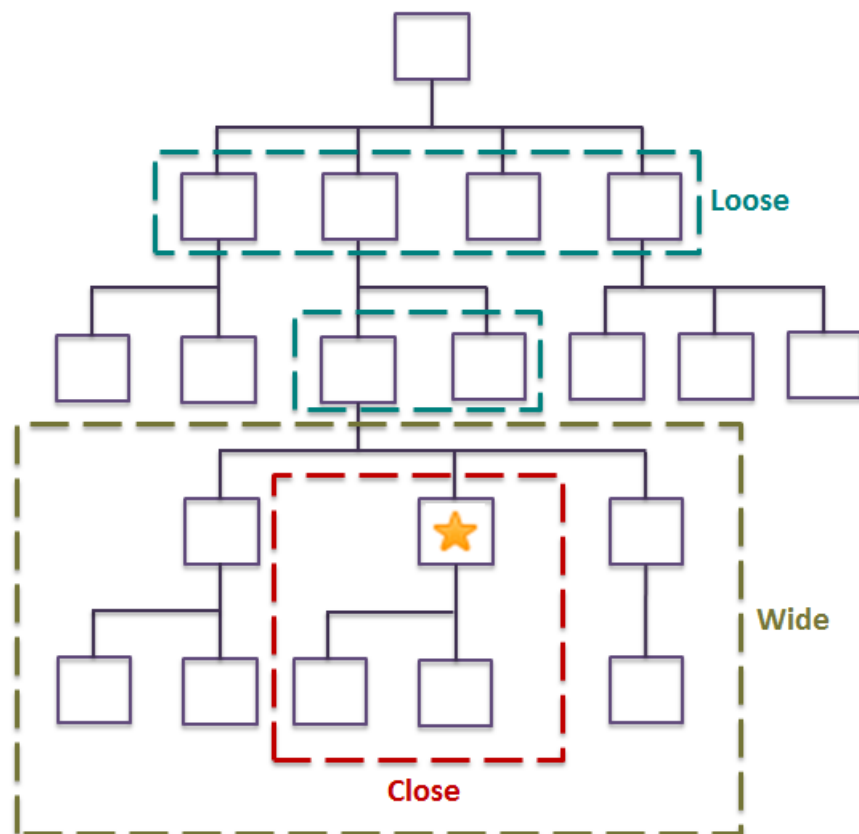
## How Conversational Requests are Handled

The navigation tolerance that you can specify for every node in the tree plays an important role in how the VSM handles a conversation request. The navigation tolerance is used to determine where in the conversation tree a VSM searches for a transaction that follows the specified transaction.

### Navigation Tolerance Levels

- **Close:** The children of the current transaction are searched.
- **Wide:** A close search, including the current transaction plus siblings and nieces/nephews of the current transactions.
- **Loose:** A wide search, plus the parent and siblings of the current transaction, followed by the children of the starting transaction. If both fail to find a match, then the starting transactions for all conversations are checked, resulting in searching the full conversation.

The following graphic shows how the navigation tolerance affects the transactions to be searched in a conversation tree. A star ★ marks the current transaction.



At the time of recording, the VSE recorder allows for initializing the navigation tolerance on transactions the following settings:

**Default navigation**

Defines the default tolerance on all Meta transactions that have child Meta transactions.

**Default:** Wide

**Last**

Defines the default tolerance for Meta transactions that are "leaf" transactions without any child Meta transactions.

**Default:** Loose

You can change these parameters later for each node through the Service Image Editor in DevTest Workstation.

The defaults provide a better match on "right" behavior. VSE responds correctly more often in situations when current run-time sessions restart a conversation without the need to start a new conversation.

**Handling Unknown Requests**

An unknown request occurs in the following situations:

- When there is an active conversation
- When there is not an active conversation

If there is not an active conversation, a request is identified as unknown if there are no stateless transactions that can satisfy the request. In this case, the service image response for unknown stateless requests becomes the reply.

If a request cannot be matched to a follow-on transaction:

- If the navigation tolerance is not CLOSE, the conversation starter transactions are given the chance to satisfy the request.
- If the request is still unmatched and a stateless transaction can produce a reply, then that is sent. The current session continues to remember where it is in its conversational tree.

If that fails, the service image response for unknown conversational requests becomes the reply.

## Magic Strings and Dates

Magic strings and magic dates are central to the dynamic nature of a virtualized service. They enable the virtualized service to return meaningful results for request parameters that were never recorded.

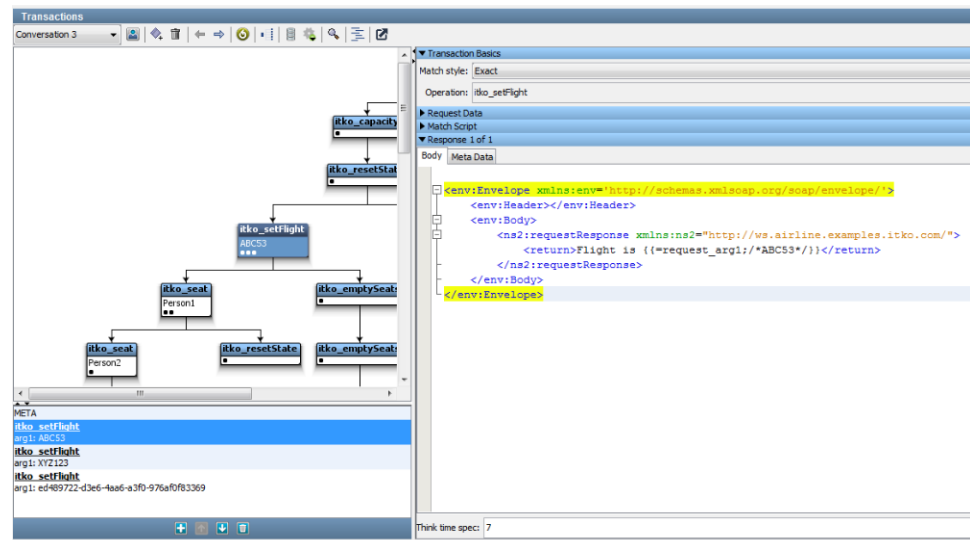
- [Magic Strings](#) (see page 48)
- [Magic Dates](#) (see page 52)

## Magic Strings

During the recording phase, VSE examines each request for arguments or parameters. For example, a weather forecasting web service call could include a city name or an airline booking system request could include a flight number. If the flight number is included in the recorded response, it is classified as a *magic string*.

For example, if VSE is in playback mode and a request comes in for a flight number that was never recorded, the correct flight number is included in the response.

In the following example, we recorded a request to set some "state" in the conversation, in this case the flight number (ABC53). The recorder recognized that the string ABC53 was also in the response, so it converted ABC53 to a magic string. The magic string is saved in the service image database as '=request\_arg1;/ABC53/.



The `{{ }}` notation is a standard VSE property substitution syntax. This notation means "substitute the `{{ }}` text with the runtime value of the first argument of the request." If there is no first argument, use ABC53 as the default.

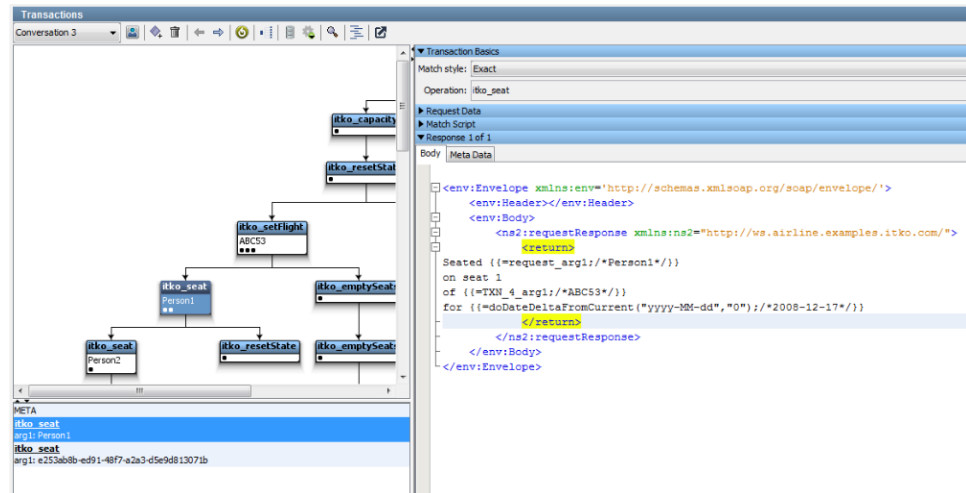
The practical significance is that if a future client requests flight ZZ99, the virtualized service responds with the correct flight number, ZZ99.

VSE detects magic strings not only in a simple request/response pair. If an argument is ever seen in any subsequent response in a conversation, VSE deems it to be magic and stores the argument value in the conversation state.

By default, VSE does not consider anything in an XML tag for magic strings. This exclusion includes XML tags, attribute names, and attribute values. You can change this behavior by setting `lisa.magic.string.xml.tags=true` in the `local.properties` file, in which case normal magic string rules apply.

In other words, if you have `<foo bar="baz"/>`, none of that is considered for magic strings. By contrast, if the structure is `<foo>bar</foo>`, then "bar" (but not "foo") is considered for magic strings.

The following example is later in the previous conversation. The request operation is "itko\_seat" and the single argument is a name, in this case "Person1". The response contains the name in the request, the previously set flight number and a date.



This is the canonical example of conversational state over a stateless protocol, in this case SOAP over HTTP. If the VSE in playback mode sees a request to set the flight to ZZ99 and then a request to seat PersonSomeoneElse, then the correct string, "Seated PersonSomeoneElse on seat 1 of flight ZZ99" is included in the response, even though those details were never recorded.

VSE does not regard the seat number, 1 in this case, as a magic string. The seat number was never seen in the original series of requests that led to this response. Nor is it large enough to be regarded as a magic string. A magic string must be at least three characters long and optionally have whitespace on the left, right, or both sides of the string. You can adjust the length and whitespace parameters in the **local.properties** file.

The `{{ }}` property notation is powerful and flexible and is not confined to using magic strings. For example, we could change the "1" in the response in the example to "DataSetValue" and could define a data set in the virtual service model. Then that data set is used to generate the runtime response value. The data set could be a random string generator, a counter, a call to a database, a reference to an Excel spreadsheet row, or anything. `}}` can execute arbitrary Java code and even generate realistic "everyday" data such as a valid credit card number `{{=:Credit Card:}}`.

For more information, see "Using BeanShell in DevTest" in *Using*.

## Magic String Exclusions

VSE attempts to identify tokens in identical requests and responses (contingent on specific rules), and turn the values in the response to a "magic string." The value of the magic string varies, based on the values in the request.

However, DevTest has no way of knowing if the values match by design or by accident. This match is most common for the following values:

- Booleans
- The **\_\_NULL** token that the DevTest agent and CAI for Java VSE uses. For example:

### Request

```
<GetUserRequest>
<userId>lisaitko</userId>
<includeDetails>true</includeDetails>
</GetUserRequest>
```

### Response

```
<GetUserResponse>
<userId>lisaitko</userId>
<isActive>true</isActive>
<isEmailVerified>true</isEmailVerified>
</GetUserResponse>
```

The two "true" values in the response have nothing to do with each other, or with the value in the request. In real world scenarios, the number of unwanted "magic strings" that is generated is much greater. One way to avoid this issue is to find and replace these magic strings manually after the service image is recorded.

An easier way, however, is to set the **lisa.magic.string.exclusion** property in **lisa.properties**.

This property lets you specify values that are not candidates for magic string identification. DevTest does not try to correlate these values in the response to values in the request during recording. If necessary, you can still manually edit the service image to add magic strings.

## Magic String Case Sensitivity

A *magic string* is a string that in a request argument that is later seen in a response for that conversation. When looking for magic strings, a match is not case-sensitive. Magic string case processing is shown through the following logic:

- If we see "dallas" in the request and "DALLAS" in the response during recording, then we get "austin" during playback, we respond with "AUSTIN".
- If we see "DALLAS" in the request and "dallas" in the response during recording, then we get "AUSTIN" during playback, we respond with "austin".
- If we see "dallas" in the request and "Dallas" in the response during recording, then we get "austin" during playback, we respond with "Austin".
- If we see "dallas" in the request and "DaLLas" in the response during recording, and during playback we see "austin", we send back "austin", not "AuSTin". If you must explicitly deal with this mixed-case response in your service image, alter the response for the meta-transaction to include a callout to your custom Java code such as:

```
{%=MyHelperClass.mixedCase(request_city);%}
```

## Magic Dates

A powerful date parser scans requests and responses are also scanned during recording. Anything matching a wide definition of date formats is recognized and translated to a *magic date*. In the example that is shown in [Magic Strings](#) (see page 48), the magic date is:

```
{{ =doDateDeltaFromCurrent("yyyy-MM-dd", "0D");/2008-12-17}}
```

The use of {{ }} notation is important here.

At run time, this string is translated as "generate a date of the format yyyy-MM-dd that is 0 days from the current date". That is, generate the current date.

If the original recording was taken on 1 February 2009 and the response contained the date 2009-02-10, the magic date string would be:

```
=doDateDeltaFromCurrent("yyyy-MM-dd", "10D");/2009-02-10/
```

The **10D** in the magic string means that the VSE generates a date in the response that is 10 days ahead of the current time. Therefore, if the VSE is in playback mode on 12 June 2010, the response contains the string **2010-06-22**.

The valid parameters for date deltas are:

- **D:** Days
- **H:** Hours
- **M:** Minutes
- **S:** Seconds
- **Ms:** Milliseconds

Another variant of magic dates is:

```
doDateDeltaFromRequest
```

Use the **doDateDeltaFromRequest** variant when a date is used as a parameter in the request and a date is seen in the response. For example, an airline reservation system could accept a seating request for a specific flight on a specific day. If that date is seen in the response, the VSE correctly substitutes the date in any subsequent responses.

A more sophisticated example is if the flight request generated a response that detailed a flight crossing the International Date Line. A flight from Los Angeles to Sydney arrives two days later than the departure according to the calendar date even though the flight time is 14 hours. In this example, the response contains something like:

```
doDateDeltaFromRequest("yy-MM-dd", "2D")
```



If VSE processed a similar request for a flight departing LAX on 19-June-2013, it includes the correct arrival date of 21-June-2013 in the response.

**Note:** You can add any valid date and time formats, including the ones that have zones, to **`lisa.properties`** for magic date calculations.

## Understanding VSE Transactions

A *transaction*, as it applies to CA Service Virtualization, is a complete unit of work that a service performs. A transaction includes the request that the client sends to the service and the response that the service sends to the client.

With most service protocols, including web services, HTTP, and Java, transactions happen synchronously. The request and response are directly related to each other.

With web services and HTTP, the request and response are contained in a single socket connection, and the response typically follows the request. With Java, the request and response are contained in a single thread. The response (the return value) always follows the request (a method call).

Messaging is different because it is asynchronous. The request and response messages do not have to occur in the same socket connection, the same thread, or even in the same day. The client sends a request message and continues working while waiting for the response. When the response is sent from the service, the client receives it and matches it to the original response, completing the transaction.

A transaction typically has only one response (for example, in the case of an HTTP responder). However, the messaging protocol can return multiple responses from a single request.

## Stateless and Conversational VSE Transactions

VSE transactions are classified as *stateless* or *conversational*.

## Stateless Transactions

*Stateless* transactions include no logical relationships between transactions. For example, HTTP and SOAP are stateless protocols. A stateless transaction always has a static response, regardless of what calls were made previously.

In a stateless conversation, each service call is independent of the others. For example:

- What is the weather like in Dallas, TX? (Operation: weather; arg1-city)
- What is the weather like in Atlanta, GA? (Operation: weather; arg1-city)
- What will the weather be like in Dallas, TX tomorrow? (Operation: weather; arg1-city; arg2-date)

## Conversational or Stateful Transactions

Conversational transactions (conversations) are *stateful*. Conversations consist of:

- The logical transaction that, if matched, starts a unique session
- The information necessary to create and identify that session.

A transaction in a conversation always depends on the context that earlier conversations create.

For example, the following stateful conversation involves using an ATM:

1. Connect to the ATM. (Operation: logon)
2. Which account do you want? (arg1-checking)
3. What is my balance? (Operation: balance)
4. Response.
5. Withdraw an amount of money. (Operation: withdrawal)
6. How much? (arg2-amount)
7. What is my balance? (Operation: balance)
8. Response (different based on the amount that was withdrawn in the previous request).
9. End session (Operation: log out)

## Conversation Starters

The first transaction in a conversation is the *conversation starter*. When VSE receives an incoming request, it reviews the conversation starters to determine whether the transaction means we are starting a new conversation.

Because all the transactions in a conversation are related to each other, VSE must have a way of determining this relationship. This determination is typically made by using some kind of special token such as "jsessionid" or a cookie in web transactions, or a custom identifier in message traffic.

DevTest supports the following types of conversations:

### Instance-based conversations

The protocol layer is responsible for identifying the unique string, which is based on different instances of the client. The string identifies server-side sessions for both recording and playback of the service image.

### Token-based conversations

VSE generates the token using a string generation pattern that is stored with the conversation in the service image. After the token is generated, it operates the same as an instance-based conversation. Token-based conversations cannot be automatically inferred during recording. To specify where tokens are found, use the [VS Image Recorder](#) (see page 280).

## Navigation Tolerance

In a conversation, VSE follows specific rules to determine how to find the next conversational transaction. The sets of rules, also known as *navigation tolerances*, are:

### Close

The transactions must go straight down the tree. The only candidates for the next conversational transaction are the children of the current one.

### Wide

The default. Wide tolerance allows navigation to the current transaction, the children of the current transaction, the siblings of the current transaction, and the immediate descendants of the sibling (or "nephews"). The order of precedence is as follows:

1. Children of a current transaction
2. Children of a sibling
3. Sibling of a current transaction

### Loose

The most permissive navigation tolerance. VSE first tries the Close and Wide tolerances, then adds the ability to match the parent of the current transaction, any of the siblings of the parent ("uncles"), the children of the siblings of the parent ("cousins"). If this match fails, navigation is permitted to any transaction in the second or third level of the tree. The order of precedence is as follows:

1. Children of a current transaction (close tolerance)
2. Children of a sibling (wide tolerance)
3. Sibling, or current transaction (wide tolerance)
4. The siblings of the parent transaction ("uncles") but not their children (not cousins)
5. The parent transaction or its siblings (parent or "uncles")
6. The children of the starter transaction for the current conversation (the immediate children of the root of the tree)
7. The starter transactions for all the conversations in the SI

## Logical Transactions

A *logical transaction* appears as a single node in a conversation. A logical transaction consists of one Meta transaction with one or more specific transactions.

When a logical transaction appears in the search for a specific request, the Meta transaction is consulted. The Meta transaction determines if this transaction can respond to the request. If the Meta transaction responds to the request, then all the specific transactions are asked if they can respond to the request. If none of the specific transactions can respond to the request, then the response is taken from the Meta transaction.

This logic can be expressed in the following pseudo code:

```
for each logical transaction \{

    if (meta transaction request matches the given request) \{
        // This node would handle the request for each specific transaction
        in this node \{
            if (the specific transaction matches the given request) \{
                return the response from the specific transaction
            \}
        \}
        // No specific transaction found for the given request
        return the response from the meta transaction
    \}
\}
```

Further, a physical Meta transaction carries a list of specific transactions and a list of child Meta transactions. The list of child Meta transactions allows Meta transactions to be structured as a decision tree.

## Match Tolerance

*Match tolerance* defines how VSE decides whether a specific transaction matches the incoming transaction.

The levels of match tolerance are:

### **Operation**

The loosest match tolerance. The operation name of the incoming transaction must match the name of the recorded transaction.

### **Signature**

The operation name must match and the names of the arguments must match exactly, with no additions or deletions. The order of arguments does not have to be the same.

### **Exact**

In addition to the signature match, the values for each argument must match the values that were recorded, as defined by the argument match operators.

## Argument Match Operators

Every argument in a request has a *match operator*. For an Exact match operation, the operator controls how the value of the argument in the incoming request is matched against the value of the corresponding argument in the service image.

The available match operators are:

### Anything

Always returns true. The virtual service recorder defaults the comparison to **Anything** when it determines that an argument is a date. The incoming value for this argument can be anything. The argument must be present for the signature/META match to work, but the value is ignored and can be blank or null.

### = Equal

Returns true if the values are the same.

### != Not equal

Returns true if the values are different.

### < Less than

Returns true if the inbound value is less, before, or earlier than the value from the service image.

### <= Less than or equal

Returns true if the inbound value is less, before, earlier than or equal to the value from the service image.

### > Greater than

Returns true if the inbound value is greater, after, or later than the value from the service image.

### >= Greater than or equal

Returns true if the inbound value is greater, after, later than or equal to the value from the service image.

### Regular Expression

Returns true if the inbound value (as a string) matches the value, as a regular expression, from the service image.

### Property Expression

The value must be in the form of a double-braced script expression, `{{ }}`. This property or expression is evaluated. If the result starts with Y, y, T, t, or ON, then the argument is deemed to have matched. The argument value in the inbound request is ignored unless referenced in the script.

**Note:** If an argument is marked as a date, the values from the requests being compared are converted to a date before the comparison is done.

## Meta Transactions and Specific Responses

When VSE searches for a conversational match, it only searches Meta transactions. A Meta transaction cannot have a match tolerance of **Exact** (the default is **Signature**). Each Meta transaction has one or more specific responses, which could have any match tolerance (**Exact** is the default).

If none of the specific responses for a Meta transaction match, then the response that is specified for the Meta transaction is used.

### How VSE Selects the Next Response

1. If the incoming request is in a conversation, search for a match that is based on the navigation tolerance and other rules explained previously.
2. If nothing is found in the current conversation, look for another conversation (unless navigation tolerance is CLOSE).
3. If there is a conversational match, with a specific response (instead of a Meta match), use the specific response.
4. Otherwise, look for a specific match in the stateless transactions, and if found, use that one.
5. If there was no specific match in the stateless list, but there is a Meta match in the conversational list, use that Meta match.
6. If there was no conversational match at all, but there is a Meta match in the stateless list, use the Meta match.
7. Fail with "no match found," and send back either the "unknown conversational response" or the "unknown stateless response" specified in the service image. The response could be overridden based on the protocol and handlers used.

## How to Debug Match Failures

If you get failures to match, open the **LISA\_HOME\logging.properties** file and set **log4j.logger.VSE** to **DEBUG** or **TRACE**. This setting causes VSE to be verbose to the **vse\_XXX.log** file, where **XXX** is the service image name. The **log4j.logger.VSE** property also tells you exactly what matched and what did not match.

Set the **log4j.logger.VSE** property to **INFO** or **WARN** for production use. Do not leave it set to **DEBUG** or **TRACE** longer than necessary.



## Track Transactions

In VSE, you can track the transactions that are done through the ITR facility.

When using a virtual service in the ITR, set the execution mode in the configuration file that the project uses.

**Follow these steps:**

1. Select Actions, View tracked responses from the ITR run.
2. To track transactions of a specific virtual service, set the execution mode in the active configuration file by setting the **lisa.vse.execution.mode** property to TRACK.

A new window in the editor lists the tracked transactions.



# Chapter 4: Using the DevTest Portal with CA Service Virtualization

---

This section contains the following topics:

[Open the DevTest Portal](#) (see page 63)

[Create a Virtual Service](#) (see page 65)

[Editing Virtual Services](#) (see page 75)

[Deploy a Virtual Service](#) (see page 96)

## Open the DevTest Portal

You open the DevTest Portal from a web browser.

**Note:** For information about the server components that must be running, see Start the DevTest Processes or Services in *Installing*.

**Follow these steps:**

1. Complete one of the following actions:
  - Enter **http://localhost:1507/devtest** in a web browser. If the registry is on a remote computer, replace **localhost** with the name or IP address of the computer.
  - Select View, DevTest Portal from DevTest Workstation.
2. Enter your user name and password.
3. Click Log in.



# Chapter 5: Create a Virtual Service

---

The Create a Virtual Service window lets you record a virtual service.

To create a virtual service, enter Recording Information on the top pane of the CA Service Virtualization Recorder.

**Follow these steps:**


1. Enter the Recording Name. To display a list of previously saved recording configurations, click List. To clear a selected recording information from the recorder and start with a fresh recording, click New Recording.

When you display a list of recordings, you can sort the list by name, id, or status. To

delete a recording, highlight it and click Delete .

2. Select the VSE Server from the drop-down list.
3. Enter a Description for this virtual service.

4. Click Txn  to display transactions as the virtual service is recording.

5. Click Status  to see details about the status of the recording.

This section contains the following topics:

[Record a Website \(HTTP or HTTP/S\)](#) (see page 66)

[Configure a Virtual Service](#) (see page 69)

[Save a Virtual Service](#) (see page 71)

## Record a Website (HTTP or HTTP/S)

To record a virtual service from an HTTP/S website, enter information about the HTTP transport protocol in the bottom pane of the Create Virtual Service window.

HTTP Transport Protocol

1 Record 2 Configure 3 Save

VS Recorder

Client

Server

Use SSL

10.132.50.233

8001

Use SSL

tp://localhost:8080

☒ Do not modify host header parameter received from client

Start Recording

Next

### Follow these steps:

1. CA Service Virtualization has chosen a default host for the VS Recorder and displays that IP address. To change the name or IP address of this host where the recorder is running, select another IP address from the drop-down. If there are multiple host IP addresses, and you want to record on all of those IP addresses, "All" is provided as the first option in the drop-down and is selected by default.
2. To enter or change the target port for the VS Recorder, click the lock icon to enable the field for input. Enter a port number that the recorder uses as a listen port of incoming requests.
3. To specify the website to record, enter the URL in the Enter Target URL field. The following formats are accepted:

- <ip>:<port>
- <hostname>:<port>
- <server>.<domain>

**Note:** As the target URL is typed into this field, VSE validates that the URL format. Once the field is fully typed, and you tab out or enter, VSE makes a call to the target server to ensure that it is reachable. An appropriate error message is shown.

4. Select or clear the Do not modify host header parameter received from client check box.

If selected, this option instructs the live invocation to forward the host header that is received from the client application to the target server. If cleared, the live invocation regenerates the host header parameter to be **host: <target host>:<target port>**.

**Note:** As you enter information, watch the top of the window for error messages that highlight invalid entries.

**Note:** Notice that the graphic for the recording changes color as the components are validated. In the preceding graphic, both the VS Recorder and the server are colored, indicating that they have been validated.

The status of the recording changes from "Draft" to "Ready" when all the mandatory information required for recording is entered. The status shows on the blue Status button. To get more details for the recording, click Status.

5. Choose one of the following options:

- To record without using SSL, leave the Use SSL check boxes cleared. Click Next.
- To record using SSL, select one or both of the Use SSL check boxes. The Use SSL check box on the left lets you enter information about the SSL certificate to send to the client. The Use SSL check box on the right lets you enter information about the SSL certificate to send to the server.

**Note:** For more information about SSL, see "[CA Service Virtualization and SSL](#) (see page 68)".

6. (Optional) If you select one or both Use SSL check boxes, complete the following fields:

**SSL keystore file**

Specifies the name of the keystore file.

To find a keystore file on the file system, click Browse.

**SSL Keystore password**

Specifies the password that is associated with the specified keystore file.

**SSL Key alias**

Designates an alias for a public key.

**SSL Key password**

Specifies the password that is associated with the specified alias in the keystore file.

7. (Optional) To validate the SSL parameters, click Validate. As you enter data into the SSL fields, the system performs progressive validation.
8. Click Start Recording.
- Note:** If you start a recording at the recording panel, record some transactions, click Next, then click Prev to go back to the recording panel, you go back to the configuration steps.
9. When you have finished recording, click Next.

## CA Service Virtualization and SSL

During CA Service Virtualization recording, the SSL Server application is the System Under Test.

### **Use SSL (between Recorder and target server)**

If the Use SSL check box between the recorder and server is selected, the recorder communicates with the target server through an SSL connection. By default, the recorder trusts all server certificates.

If the target server requests Client Authentication (two-way SSL), the recorder sends the certificate that is specified in the fields below the Use SSL check box. If no certificate information is specified, the recorder uses the values that are specified in the local.properties (the **ssl.client.cert.\*** properties) file of the VSE server where recording is performed.

### **Use SSL between Client and Recorder**

If the **Use SSL** check box between the client and recorder is selected, the client communicates with the recorder through an SSL connection. In this portion of the recording, the recorder is the delegate for the target server and must send back a server certificate when the client initiates a request. The certificate that is sent back to the client is the one specified in the fields below the Use SSL check box.

### **Playback of the VSM**

If you select Use SSL to Client, an SSL handshake occurs between the client application or test case client and the VSM. The keystore that is provided in the VSM Listen step is used as the Server certificate for one-way authentication. There is no SSL handshake between the client and the VSM. The handshake is straight HTTP.

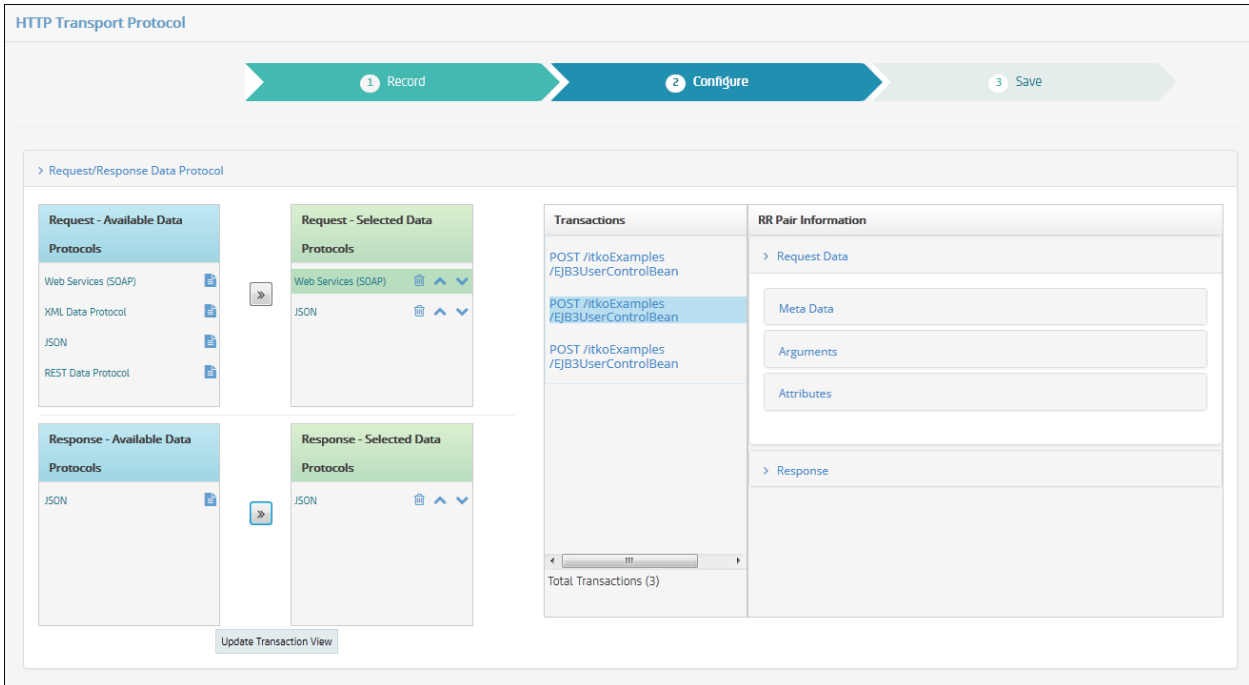
If the Live Invocation step was executed, an SSL handshake occurs between the VSM client to the real server. If the real server requests client authentication, the keystore in the HTTP/S Protocol Live Invocation step is used.

If the keystore contains multiple certificates, CA Service Virtualization uses the first one.




## Configure a Virtual Service

The Configure pane lets you accept, add, or delete data protocols to the recording you completed. You can also see the effect the data protocol has on the transactions.



The column on the left lists the available data protocols for the request and the response side. Data protocols that were auto-detected are highlighted in green and show in the Selected list. All data protocols available for the request and response are displayed.

To add a data protocol, highlight it and click Move  to move the data protocol into the Selected Data Protocols column. You can move multiple data protocols into the selected column. To delete, move up, or move down the selection, use the icons at the top of the column.

Transactions display on the right side of the pane, with request and response data. When you click Update Transaction View, the system applies those data protocols to the recorded transactions. The data protocols, when applied to a transaction, parse requests and responses differently, depending on the protocol. For more information, see "[Using Data Protocols](#) (see page 209)" in *Using CA Service Virtualization*.

The updated transactions, with parsed requests and responses, are retrieved from the VSE server and shown in the "preview" Transactions pane on the right of the Configure pane.

**Note:** The selected data protocols apply and are saved for the virtual service only when you click Update Transaction View. Changes you make in the Available and Selected lists are only UI changes.

## Save a Virtual Service

The Save pane of the CA Service Virtualization Recorder displays the following options:

- Save & Close
- Save & Deploy Virtual Service
- Save & Edit Virtual Service

For each option, complete the following fields:

### Select Project

Determines the project where the virtual service is stored. Select a project from the drop-down list. Projects that are listed in the drop-down are projects that exist in the Projects directory,

### Treat all transactions as stateless

Specifies whether to treat all recorded transactions as stateless. In most cases, leave this option cleared.

**Default:** Selected

### Allow duplicate specific transactions

Specifies whether to record duplicate specific transactions.

**Default:** Cleared

To display more options, click YES for Advanced Mode at the upper right corner of the window. Optionally, complete the following fields:

### VSM Style

Specifies whether to generate a VSM including the prepare steps.

**Values:**

- **More Flexible:** Includes prepare steps (leading to a five-step model).
- **More Efficient:** The prepare steps are absent (leading to a three-step model).

**Default:** More Efficient

### Default navigation

Specifies the navigation tolerance that determines where in the conversation tree a VSM searches for a transaction that follows the specified transaction. Select the default navigation tolerance for all except the last (leaf) transactions.

**Values:**

- Close: The children of the current transaction are searched.

- **Wide:** A close search, including the current transaction plus siblings and nieces/nephews of the current transactions.
- **Loose:** A wide search, plus the parent and siblings of the current transaction, followed by the children of the starting transaction. If both fail to find a match, then the starting transactions for all conversations are checked, resulting in searching the full conversation.

**Default:** Wide

#### **Leaf Navigation**

Specifies the navigation tolerance that determines where in the conversation tree a VSM searches for a transaction that follows the last (leaf) transactions.

##### **Values:**

- **Close:** The children of the current transaction are searched.
- **Wide:** A close search, including the current transaction plus siblings and nieces/nephews of the current transactions.
- **Loose:** A wide search, plus the parent and siblings of the current transaction, followed by the children of the starting transaction. If both fail to find a match, then the starting transactions for all conversations are checked, resulting in searching the full conversation.

**Default:** Loose

## [Save and Close](#)

Save & Close saves the virtual service and closes the recorder tab.

## Save and Deploy Virtual Service

Save & Deploy Virtual Service saves the virtual service and deploys it immediately.

Complete the following fields:

### Group Tag

Specifies the name of the [virtual service group](#) (see page 402) for this virtual service. If deployed virtual services have group tags, they are available in the drop-down list. A group tag must start with an alphanumeric character and can contain alphanumeric characters and the following special characters:

- period (.)
- dash (-)
- underscore (\_)
- dollar sign (\$)

### Concurrent capacity

Specifies a number that indicates the load capacity. *Capacity* is how many virtual users (instances) can simultaneously execute with the VSM. Capacity here indicates how many threads there are to service requests for this service model.

VSE allocates a number of threads equivalent to the total concurrent capacity. Each thread consumes some system resources, even when dormant. Therefore, for optimal overall system performance, set this setting as low as possible. Determine the correct settings empirically by adjusting them until you achieve the desired performance, or until increasing it further yields no performance improvement.

Out of the box protocols use a framework-level task execution service to minimize thread usage. For these protocols, a concurrent capacity of more than 2-3 per core is rarely useful, unless the VSM has been highly customized.

For extensions and any VSM that does not use an out of the box protocol, setting a long Think Time may consume a thread during the think time. In these cases, you can increase the concurrent capacity.

The following formula gives an approximate initial setting in these cases:

$$\text{Concurrent Capacity} = (\text{Desired transactions per second} / 1000) * \text{Average Think Time in ms} * (\text{Think Scale} / 100)$$

### Example:

Assume that you are using a custom protocol that does not use the framework task execution service to handle think times. You want an overall throughput of 100 transactions per second. The average think time across the service image is 200 ms, and the virtual service is deployed with a 100 percent think scale.

$$(100 \text{ Transactions per second} / 1000) * 200\text{ms} * (100 / 100) = 20$$

In this case, each thread blocks for an average of approximately 200 ms before responding, and during that time is unable to handle new requests. We therefore need a capacity of 20 to accommodate 100 transactions per second. A thread would become available, on average, every 10 ms, which would be sufficient to achieve 100 transactions per second.

**Default: 1**

#### Think time scale

Specifies the think time percentage for the recorded think time.

**Note:** A step subtracts its own processing time from the think time to have consistent pacing of test executions.

**Default: 100**

#### Examples:

- To double the think time, use 200.
- To halve the think time, use 50.

The screenshot shows the 'HTTP Transport Protocol' configuration window. At the top, a progress bar indicates three steps: 1. Record (completed), 2. Configure (active), and 3. Save. Below the progress bar, a message states: 'You have three options to choose to Save and Finish.' with three radio buttons: 'Save & Close', 'Save & Deploy Virtual Service' (selected), and 'Save & Edit Virtual Service'. Underneath, there is a 'Select Project:' dropdown menu currently showing 'Previous Project'. Two checkboxes are present: 'Treat all transactions as stateless' (checked) and 'Allow duplicate specific transactions' (unchecked). A section titled 'Options for Deploy Virtual Service' contains three input fields: 'Group Tag:' (empty), 'Concurrent capacity:' (set to 1), and 'Think Time Scale (%)' (set to 100). At the bottom right, there is a green 'Save' button with a right-pointing arrow.

## Save and Edit

Save & Edit Virtual Service saves the virtual service and opens it in edit mode in a new editor tab.

# Chapter 6: Editing Virtual Services

---

The DevTest Portal lets you open a virtual service and make changes to various components of the virtual service.

For basic information about the DevTest Portal, see *Getting Started*.

This section contains the following topics:

[Open a Virtual Service](#) (see page 75)

[Stateless Transactions View](#) (see page 76)

[Conversation View](#) (see page 81)

[Unknown Responses](#) (see page 82)

[Search for Text in a Virtual Service](#) (see page 83)

[Add a Note to a Signature](#) (see page 84)

[Add a Label to a Specific Transaction](#) (see page 84)

[Updating a Virtual Service Manually](#) (see page 85)

[Find the Transactions that Match a Request](#) (see page 94)

[Virtual Service URLs](#) (see page 95)

## Open a Virtual Service

You open a virtual service from the DevTest Portal.

**Follow these steps:**

1. Go to the home page of the DevTest Portal.
2. In the left pane, click Manage.
3. (Optional) Select a different project.
4. Expand the Virtual Services category.
5. Click the virtual service name.

## Stateless Transactions View

To view the stateless transactions, open the virtual service and select Stateless Transactions from the View drop-down list.

The Stateless Transactions view is divided into the following areas:

- Stateless Signatures pane
- Specifics tab
- Signature Definition tab

By default, only the basic features are displayed. Experienced users can access the advanced features by clicking Advanced in the upper right area.

The following graphic shows the Basic and Advanced links.





## Stateless Signatures

When VSE tries to find a match for an inbound request in the stateless transactions, VSE starts by searching for a *signature* that matches the inbound request.

A signature has the following components:

- Operation name
- (Optional) A set of argument names

In a virtual service, an operation represents an action to be performed. An example of an operation is **depositMoney**.

In a virtual service, an argument is a name/value pair that can be included with an operation. An example of an argument is the name **amount** and the value **100.00**.

**Note:** VSE does not consider the argument values at this stage of the matching process.

Each signature also has a unique identifier. The identifier is unique within the virtual service.

Multiple signatures can have the same operation name. For example:

- Signature A has the **depositMoney** operation and one argument: **amount**.
- Signature B has the **depositMoney** operation and two arguments: **amount** and **date**.

The unique identifiers can help you differentiate signatures that have the same operation name.

Another way to differentiate signatures that have the same operation name is by adding [notes](#) (see page 84).

The following graphic shows the Stateless Signatures pane. For each signature, the unique identifier and the operation name appear. The selected signature has the unique identifier **159** and the operation name **addUserObject**.

Stateless Signatures (7)	
<input type="checkbox"/>	^ v +
153	listUsers
159	addUserObject
165	addUserObject
173	deleteToken
179	GET /itkoExamples/TokenBean
185	GET /itkoExamples/EJB3UserControlBean
191	GET /itkoExamples/EJB3AccountControlBean

To display the argument names, select the signature, then select the Signature Definition tab.

## Request Data Arguments and Response Data

If VSE finds a matching signature, it then searches for a *specific transaction* that matches the inbound request.

A specific transaction has the following components:

- Operation name
- (Optional) A set of argument names and values, plus other match criteria such as an operator

Each specific transaction also has an identifier that is unique within the virtual service.

The Request Data Arguments pane displays the specific transactions for the selected signature.

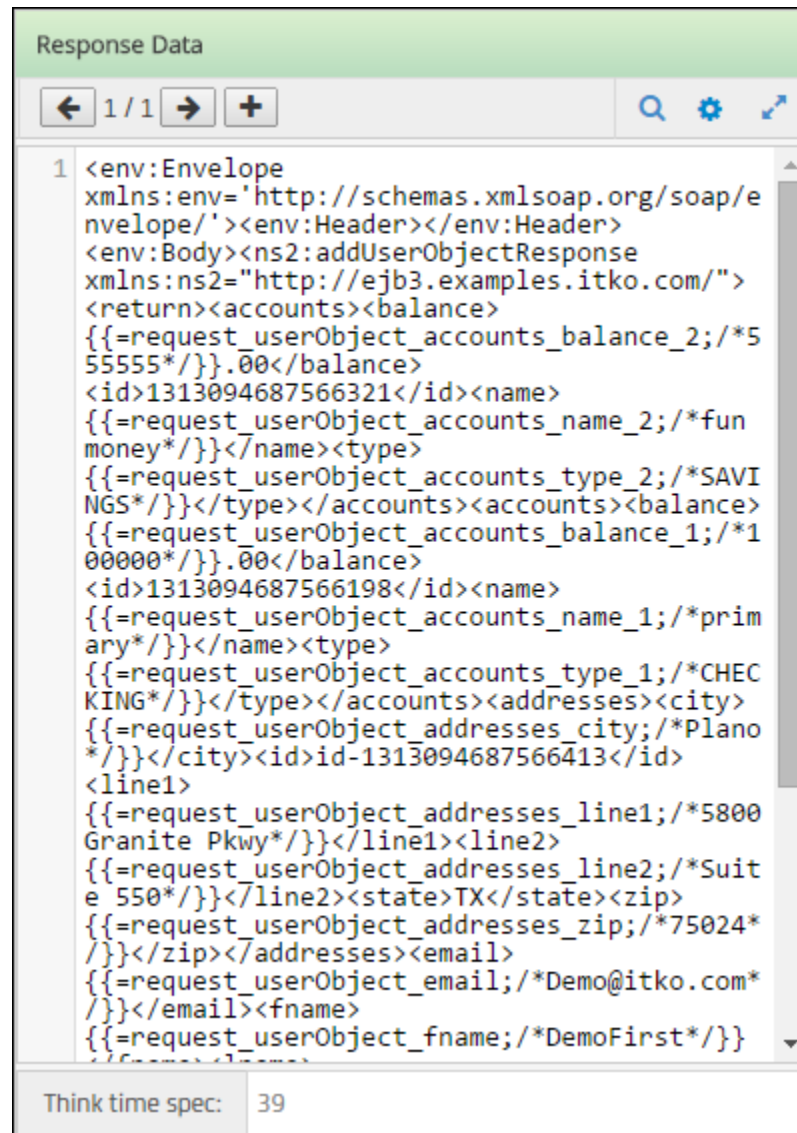
The following graphic shows a specific transaction. The unique identifier is **162**. The operation name is **addUserObject**. The arguments appear in a table.

Request Data Arguments			
162 addUserObject			
Name	Operator	Value	Magic String
userObject_accoun...	=	100000	<input type="checkbox"/>
userObject_accoun...	=	primary	<input type="checkbox"/>
userObject_accoun...	=	CHECKING	<input type="checkbox"/>
userObject_accoun...	=	555555	<input type="checkbox"/>
userObject_accoun...	=	fun money	<input type="checkbox"/>
userObject_accoun...	=	SAVINGS	<input type="checkbox"/>
userObject_addres...	=	Plano	<input type="checkbox"/>
userObject_addres...	=	5800 Granite Pkwy	<input type="checkbox"/>
userObject_addres...	=	Suite 550	<input type="checkbox"/>
userObject_addres...	=	TX	<input type="checkbox"/>
		10	20
« 1 2 »			

If VSE finds a matching specific transaction, it sends the response that is associated with the specific transaction.

The Response Data pane contains the response for the selected specific transaction.

The following graphic shows the Response Data pane.



You can search for text in a response. You can also change the syntax highlighting in a response.

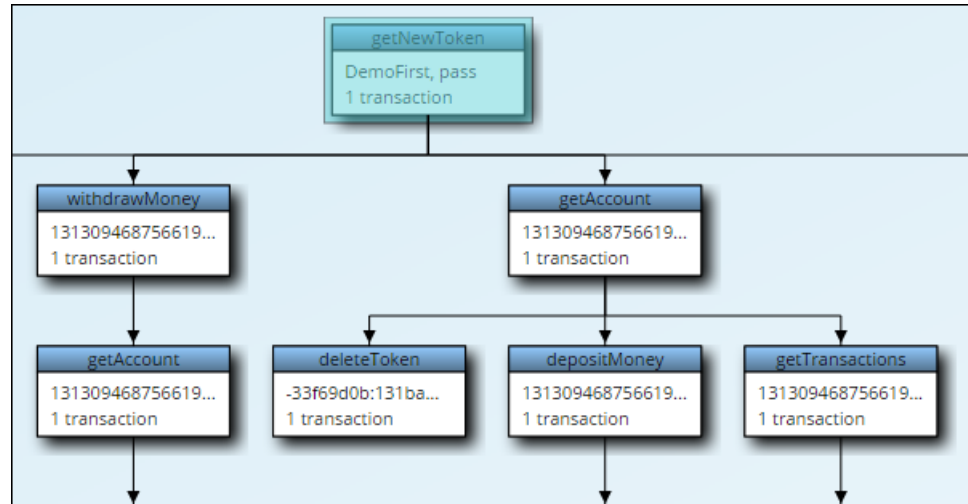
If VSE does not find a matching specific transaction, it sends the response that is associated with the *default transaction*.

The default transaction appears at the bottom of the Request Data Arguments pane. The Response Data pane contains the response for the selected default transaction.

## Conversation View

To view a conversation, open the virtual service and select the conversation name from the View drop-down list.

The following graphic shows part of a conversation. The starter transaction is selected.



Each node in a conversation is a [logical transaction](#) (see page 57).

The upper area of a node displays the operation name.

The main area of a node displays the following information:

- The argument values of the first specific transaction
- The number of specific transactions

To display a miniature view of the conversation, click Show Outline. This feature is helpful for viewing large conversations.

The Request Data Arguments pane and the Response Data pane contain detailed information about the selected node. The behavior of these panes is the [same as for stateless transactions](#) (see page 78).

## Unknown Responses

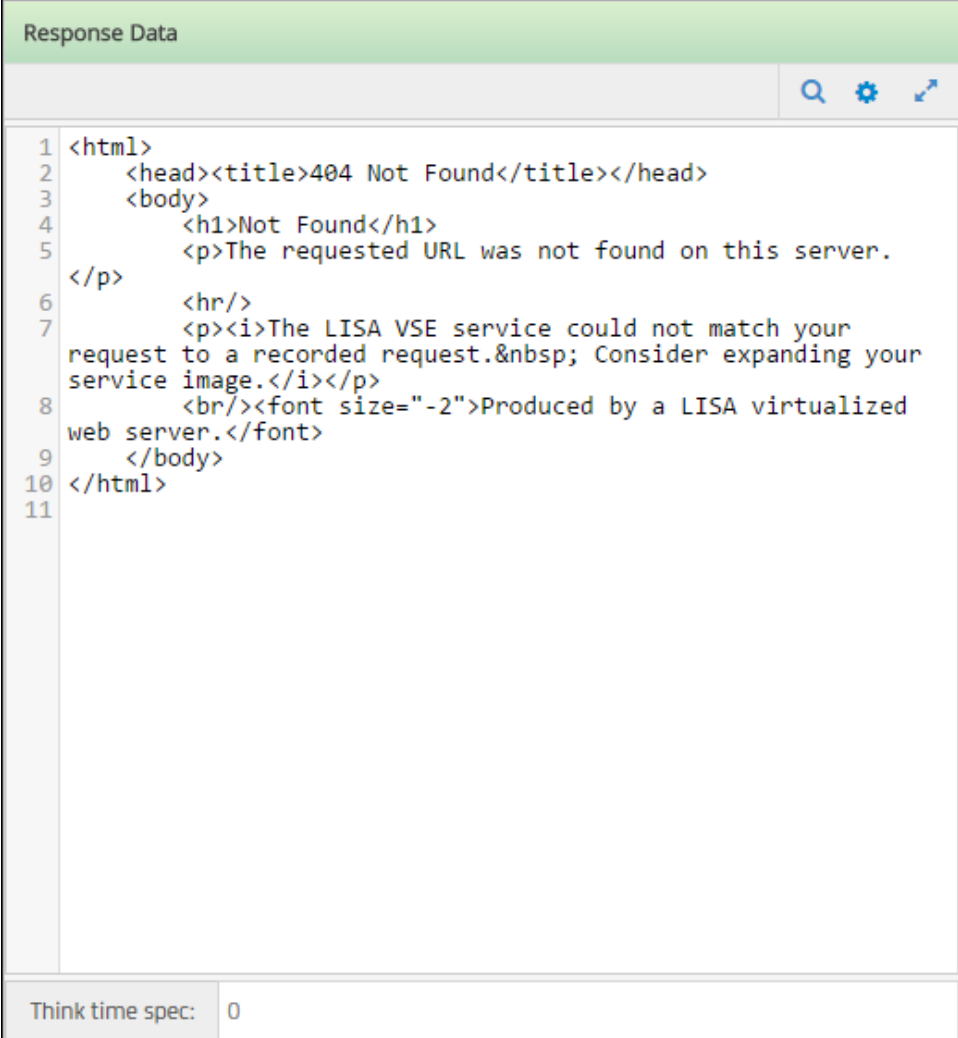
When VSE cannot find a match for an inbound request, it sends one of the following responses:

- Response for unknown conversational request
- Response for unknown stateless request

To view the unknown responses, open the virtual service and select Service Image Properties from the View drop-down list.

You can search for text in a response. You can also change the syntax highlighting in a response.

The following graphic shows an unknown response.



The screenshot shows a window titled "Response Data" with a search icon, a settings gear, and a share icon in the top right corner. The main area displays an HTML response with line numbers 1 through 11 on the left. The HTML content is as follows:

```
1 <html>
2   <head><title>404 Not Found</title></head>
3   <body>
4     <h1>Not Found</h1>
5     <p>The requested URL was not found on this server.
6   </p>
7     <hr/>
8     <p><i>The LISA VSE service could not match your
9     request to a recorded request.  Consider expanding your
10    service image.</i></p>
11    <br/><font size="-2">Produced by a LISA virtualized
    web server.</font>
  </body>
</html>
```

At the bottom of the window, there is a field labeled "Think time spec:" with the value "0".

## Search for Text in a Virtual Service

You can search for text across the following components of a virtual service:

- Signature names
- Specific transaction names
- Request data
- Response data
- Unique identifiers
- Notes

The following graphic shows the Search Results table.

Search Results for : <b>savings</b> found in <b>Response Body</b>			✕
Type	Operation	Snippet	
conversational	<a href="#">getAccount</a>	</balance><id>{{=request_accountId;/*1313094687566321*/}}</id><name>fun money</name><type> <b>SAVINGS</b> </type>	
stateless	<a href="#">listUsers</a>	money</name> <type> <b>SAVINGS</b> </type> </accounts>	

The Type column indicates whether the result is part of a stateless transaction or a conversation.

If the result is in a response body, the Snippet column highlights the location of the text.

### Follow these steps:

1. Open a virtual service.
2. In the search field, enter the text.  
As you type, suggestions are displayed below the search field.
3. Click a suggestion or press Enter.  
The Search Results table displays a set of results.
4. Click an entry in the table.  
The editor displays the component where the text is located.

## Add a Note to a Signature

You can annotate a signature by adding a note.

For example, assume that a signature represents a scenario that occurs infrequently. You can add a note that provides a description of the scenario.

You could also add a note to indicate that you changed the default settings of a signature.

**Follow these steps:**

1. Open a virtual service.
2. Right-click a signature and click Add Note.
3. Enter text. You can copy and paste within the note.
4. Click the Save icon.

## Add a Label to a Specific Transaction

You can annotate a specific transaction by adding a label.

The following graphic shows a specific transaction that has a label. The label is highlighted.

176 [deleteToken] UseCase42			
Name	Operator	Value	Magic String
token	=	-33f69d0b:131ba6957...	<input type="checkbox"/>

**Follow these steps:**

1. Open a virtual service.
2. Click the operation name of the specific transaction.  
A text field opens.
3. Type the label and click the check mark.  
The label is displayed to the right of the operation name.



## Updating a Virtual Service Manually

You can update a virtual service manually from the DevTest Portal.

The following reasons are some of the reasons to update a virtual service:

- During the recording of the virtual service, you forget to invoke an operation.
- The development team adds a new operation to the service that is being virtualized.
- You want to test a proposed change to a service before the change is implemented.

If you need to make extensive updates, consider an alternate approach:

- [Combine](#) (see page 102) the virtual service with another virtual service
- Run the virtual service with the [execution mode](#) (see page 364) set to Image Validation
- Rerecord the entire virtual service

## Add, Change, and Delete Signatures

The DevTest Portal lets you perform any of the following actions:

- Add a signature
- Move a signature up or down
- Change the definition of a signature
- Delete a signature

You can also add a signature by [importing request/response pairs](#) (see page 88).

When VSE searches for a signature that matches an inbound request, it starts at the top of the list of signatures and proceeds downward. Therefore, moving a signature up or down affects the order in which matching takes place.

Changes to a signature definition are automatically applied to all of the specific transactions for the signature. For example, if you add the **NewArg** argument, the specific transactions now have this new argument.

By default, matching is case-sensitive. To force the [comparison operators](#) (see page 90) to ignore case for an argument, go to the Signature Definition tab and clear the check box in the Case Sensitive column.

The following graphic shows the Signature Definition tab.

Specifics			
Signature Definition			
Request Data Arguments			
^ v + [trash icon]			[link icon]
Name	Date Pattern	Case Sensitive	Is Numeric
<a href="#">token</a>	Not set	<input checked="" type="checkbox"/>	<input type="checkbox"/>

By default, argument values are processed as strings. For example, "10000" is considered less than "9" because "1" comes alphabetically before "9". To force an argument value to be processed as a number, go to the Signature Definition tab and select the check box in the Is Numeric column.

Date patterns are used to parse date information. The Date Pattern column is read only.

**Follow these steps:**

1. Open a virtual service.
2. To add a signature:
  - a. Ensure that the advanced view is displayed.
  - b. In the Stateless Signatures pane, click the Add button and select Add New Signature.
  - c. Enter the operation name.
  - d. Click Continue.
  - e. Select the new signature.
  - f. (Optional) Go to the Signature Definition tab and add one or more arguments.
  - g. Go to the Specifics tab and add one or more specific transactions.
3. To move a signature up or down, select the signature and click the Up or Down icon. You can also drag a signature to the new location.
4. To change the definition of a signature:
  - a. Select the signature.
  - b. Go to the Signature Definition tab.
  - c. Make the changes. You can move an argument up or down. You can add or delete an argument. You can change the settings of the Case Sensitive and Is Numeric columns. You can change the operation name.
5. To delete a signature, select the signature and click the Delete icon. You can use the Ctrl key to select multiple signatures.

## Import Request/Response Pairs

A request/response pair consists of one request file and one or more response files.

When you import a request/response pair, the request is processed against the virtual service.

If the virtual service contains a signature with the same arguments, a specific transaction is added to the signature.

If the virtual service does not contain a signature with the same arguments, a signature is added. The details of the request and response are added as a specific transaction in the signature. A default transaction is also added.

The request/response pair must be in text or XML format.

The following list contains an example of the file names in a request/response pair:

- **depositMoney-req.xml**
- **depositMoney-rsp1.xml**
- **depositMoney-rsp2.xml**
- **depositMoney-rsp3.xml**

The request file name must include a prefix followed by the string **-req**.

The response file name must include the same prefix followed by the string **-rsp**. As the preceding example shows, you can provide multiple response files by adding a number after the string **-rsp**. Multiple response files are applicable to messaging scenarios.

You can specify request and response metadata by including one or more sidecar files. For more information, see [Sidecar Files with Request/Response Pairs](#) (see page 116).

**Note:** If the virtual service is based on a transport protocol other than HTTP, this feature has the following prerequisite: Open the corresponding virtual service model in DevTest Workstation and enter the path to the recording session file in the Documentation field. A recording session file is one of the following files:

### **VRS file**

Generated by the Virtual Service Image Recorder in DevTest Workstation.

### **VR2 file**

Generated by the CA Service Virtualization Recorder in the DevTest Portal.

**Follow these steps:**

1. Open a virtual service.
2. In the Stateless Signatures pane, click the Add button and select Import Request/Response Pairs.

The Import Request/Response Pairs dialog opens.

3. Specify the files that contain the request/response pairs. You can drag the files into the dialog, or you can click a button to select the files from a file system.
4. Click Done.

## Comparison Operators for Arguments

Each argument in a specific transaction has a comparison operator. The operator indicates how to match the argument value with an inbound request.

By default, the operator is set to the equal sign (=). Therefore, a match occurs if the argument value in the inbound request is the same as the argument value in the specific transaction.

The following mathematical symbols are also supported:

- Not equal to (!=)
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

You can also specify a regular expression or a property expression. You use these operators along with the Value column.

For example, assume that any five-digit postal code that begins with **750** is valid. You could set the operator to RegEx and set the value to the following text:

750\d\d

If you want to allow an argument to have any value, set the operator to Anything.

The following graphic shows the use of various operators. In this scenario, all of the argument values in the inbound request match the criteria in the specific transaction. For example, the account balance in the inbound request is 5000. The specific transaction indicates that the account balance must be greater than 100.

Inbound Request		Specific Transaction		
Name	Value	Name	Operator	Value
AccountType	Checking	AccountType	=	Checking
AccountBalance	5000	AccountBalance	>	100
AccountName	Name2	AccountName	Anything	Name1

## Disable and Enable Magic Strings

Each argument in a specific transaction can be classified as a [magic string](#) (see page 48).

When you view a specific transaction in the DevTest Portal, the Magic String column indicates whether each argument has been classified as a magic string.

The Magic String column is read only. DevTest Workstation provides a way to change the setting. For more information, see [Request Data Editor](#) (see page 281).

## Edit the Request Attributes

In a virtual service, an *attribute* is a name/value pair that contains information about the request body.

Assume that you record a virtual service with the HTTP transport protocol. The attributes in the virtual service include the XML namespace and the recorded raw request.

Request attributes are not used in matching.

### **Follow these steps:**

1. Open a virtual service.
2. Ensure that the advanced view is displayed.
3. Click the Attributes link to the right of the Request Data label.
4. Add, change, and delete attributes as necessary.

## Edit the Request Metadata

In a virtual service, the term *metadata* refers to name/value pairs that contain information that is not part of the request or response body.

Assume that you record a virtual service with the HTTP transport protocol. The request metadata in the virtual service includes the HTTP method and the content type.

Request metadata is not used in matching.

### **Follow these steps:**

1. Open a virtual service.
2. Ensure that the advanced view is displayed.
3. Click the Metadata link to the right of the Request Data label.
4. Add, change, and delete metadata as necessary.

## Edit the Response Metadata

In a virtual service, the term *metadata* refers to name/value pairs that contain information that is not part of the request or response body.

Assume that you record a virtual service with the HTTP transport protocol. The response metadata in the virtual service includes the HTTP response code and the content type.

Response metadata is not used in matching.

**Follow these steps:**

1. Open a virtual service.
2. Ensure that the advanced view is displayed.
3. Click the Metadata link to the right of the Response label.
4. Add, change, and delete metadata as necessary.



## Change the Think Time of a Response

Each response for a specific transaction includes a field named Think time spec. This field specifies how long to wait before sending the response. If the service image was created from recording, the initial value is based on the response behavior that was observed during the recording.

To simulate faster performance, decrease the value.

To simulate slower performance, increase the value.

By default, the value is in milliseconds. To specify the unit of time, add a suffix to the number. Case does not matter. The following suffixes are allowed:

- **t**: milliseconds
- **s**: seconds
- **m**: minutes
- **h**: hours

You can specify a number range. The think time is randomly selected from the range. For example, the value **100-1000** specifies a random think time from 100 to 1000 milliseconds.

If the think time is 10 milliseconds and it takes 5 milliseconds to process the request and find the response, you only incur an extra 5 milliseconds of delay.

If the think time is 0, or is less than the time that it took to process the request, VSE sends the response as fast as it can.

The unknown responses in a virtual service have the same field.

When you [deploy a virtual service](#) (see page 96), you can configure a related field named Think time scale. This field specifies the percentage by which to multiply all of the Think time spec fields in a service image.

### Follow these steps:

1. Open a virtual service.
2. Locate the response for the specific transaction or unknown response.
3. Change the value of the Think time spec field.

## Find the Transactions that Match a Request

Assume that when you send a request to a virtual service, the response is not what you expect.

You can open the virtual service and then enter the request. The DevTest Portal indicates which components in the virtual service are a match for the request. The results can help you determine how to fix the problem.

The request must be in text or XML format.

The DevTest Portal lets you enter the actual contents of the request, or specify a request file. The request file name must include a prefix followed by the string **-req** (for example, **depositMoney-req.txt**).

If the results include stateless signatures or stateless specific transactions, the order in which they appear is the same order in which they are matched during playback.

The order of conversational results is not significant.

**Note:** If the virtual service is based on a transport protocol other than HTTP, this feature has the following prerequisite: Open the corresponding virtual service model in DevTest Workstation and enter the path to the recording session file in the Documentation field. A recording session file is one of the following files:

### **VRS file**

Generated by the Virtual Service Image Recorder in DevTest Workstation.

### **VR2 file**

Generated by the CA Service Virtualization Recorder in the DevTest Portal.

### **Follow these steps:**

1. Open a virtual service.
2. Click Find a Match by Request.
3. Do one of the following actions:
  - Paste the request into the text area.
  - Drag the request file into the dialog.
  - Click Select Request File and select the request file from a file system.
4. Click Find a Match.

The Search Results table displays any matches that are found.
5. To display the original request, click View Request.
6. Click an entry in the Search Results table.

The matching component is displayed.

## Virtual Service URLs

When you open a virtual service in the DevTest Portal, the URL specifies the path to the virtual service component that is displayed.

The following example shows the URL format of a specific transaction in the Stateless Transactions view.

```
http://localhost:1507/devtest/#/main/serviceimageeditor/0/bmS18CcJu-b74PB4tcz/bmS18CcJu/b74PB4tcz/kioskV6/0?signatureId=153&specificId=156&view=stateless
```

You can bookmark a given URL. You can later open the virtual service to the same component by entering the URL in a browser.

## Deploy a Virtual Service

You can deploy a virtual service from the DevTest Portal.

**Follow these steps:**

1. Go to the home page of the DevTest Portal.
2. In the left navigation pane, click Manage.
3. (Optional) Select a different project.
4. Right-click the virtual service that you want to deploy and select Deploy.

The Deploy Virtual Service dialog opens.

5. Select the VSE Server where you want to deploy.
6. Modify the fields as necessary:

**Group Tag**

Specifies the name of the [virtual service group](#) (see page 402) for this virtual service. If deployed virtual services have group tags, they are available in the drop-down list. A group tag must start with an alphanumeric character and can contain alphanumeric characters and the following special characters:

- Period (.)
- Dash (-)
- Underscore (\_)
- Dollar sign (\$)

**Concurrent capacity**

Specifies a number that indicates the load capacity. *Capacity* is how many virtual users (instances) can simultaneously execute with the VSM. Capacity here indicates how many threads there are to service requests for this service model.

VSE allocates a number of threads equivalent to the total concurrent capacity. Each thread consumes some system resources, even when dormant. Therefore, for optimal overall system performance, set this setting as low as possible. Determine the correct settings empirically by adjusting them until you achieve the desired performance, or until increasing it further yields no performance improvement.

Out of the box protocols use a framework-level task execution service to minimize thread usage. For these protocols, a concurrent capacity of more than 2-3 per core is rarely useful, unless the VSM has been highly customized.

For extensions and any VSM that does not use an out of the box protocol, setting a long Think Time may consume a thread for the duration of the think time. In these cases, you may need to increase the concurrent capacity.

The following formula gives an approximate initial setting in these cases:

$$\text{Concurrent Capacity} = (\text{Desired transactions per second} / 1000) * \text{Average Think Time in ms} * (\text{Think Scale} / 100)$$

**Example:**

Assume that you are using a custom protocol that does not use the framework task execution service to handle think times. You want an overall throughput of 100 transactions per second. The average think time across the service image is 200 ms, and the virtual service is deployed with a 100 percent think scale.

$$(100 \text{ Transactions per second} / 1000) * 200\text{ms} * (100 / 100) = 20$$

In this case, each thread blocks for an average of approximately 200 ms before responding, and during that time is unable to handle new requests. We therefore need a capacity of 20 to accommodate 100 transactions per second. A thread would become available, on average, every 10 ms, which would be sufficient to achieve 100 transactions per second.

**Default:** 1

**Think Time Scale**

Specifies the think time percentage for the recorded think time.

**Note:** A step subtracts its own processing time from the think time to have consistent pacing of test executions.

**Default:** 100

**Examples:**

- To double the think time, use 200.
- To halve the think time, use 50.

7. Click Deploy.

**Note:** A virtual service can be in the following states:

**Deployed**

No service with the name you entered is already deployed. The service is deployed.

**Redeployed**

A service with the name you entered is deployed with the same .vsm file as the service you entered. The service is redeployed.

**Overridden**

A service with the name you entered is deployed with a .vsm file that is different from the one associated with the service you entered. The application prompts you to override the deployed service.



# Chapter 7: Using the Workstation and Console with CA Service Virtualization

---

This section contains the following topics:

[Creating Service Images](#) (see page 101)

[Editing Service Images](#) (see page 273)

[Editing a VSM](#) (see page 305)

[Desensitizing Data](#) (see page 351)

[Virtualizing a Service](#) (see page 353)





# Chapter 8: Creating Service Images

---

This section contains the following topics:

[Open a Service Image](#) (see page 101)  
[Combine Service Images](#) (see page 102)  
[Delete a Service Image](#) (see page 102)  
[Create a Service Image](#) (see page 103)  
[Create a Service Image from Scratch](#) (see page 104)  
[Create a Service Image from a WSDL](#) (see page 104)  
[Create a Service Image from WADL](#) (see page 108)  
[Create a Service Image from RAML](#) (see page 110)  
[Create a Service Image from Layer 7](#) (see page 112)  
[Create a Service Image from Request/Response Pair](#) (see page 113)  
[Create a Service Image from PCAP](#) (see page 117)  
[Create a Service Image by Recording](#) (see page 119)  
[Create and Deploy a Virtual Service with VSEasy](#) (see page 207)  
[Work with VSMs](#) (see page 208)  
[Using Data Protocols](#) (see page 209)

## Open a Service Image

The Virtual Service Image Recorder generates service images. Service images pretend to be what you recorded (a manipulated or altered version of your recorded raw traffic).

**Note:** If you have service images from releases of VSE earlier than LISA 6.0, export those service images. Exporting moves them from the database of earlier versions to the file system in which they are stored in the current release. For more information, see [Legacy Service Images](#) (see page 273).

Opening a service image allows you to change the image in the Service Image Editor.

**Follow these steps:**

1. Right-click the service image in the Project panel and select Open.  
The Service Image Editor opens.
2. Review the selected service image and make any changes.

## Combine Service Images

Combining service images is useful when you want to add functionality to an existing service image.

**Follow these steps:**

1. Right-click the service image on the Project panel and select Combine.  
The Combine Service Images dialog opens.
2. Select one or more service images to combine with the original image (the target).
3. To replace what is in the matching target data, select the Favor source image(s) check box.

When you combine service images, each stateless transaction (at the Meta level) from each source service image is matched against each one in the target service image. For each one that matches, the specific transactions from the source are matched against the ones in the target. When no matches are found, the source transactions are added to the target image. When they do match, the transactions must be merged.

You have the choice of having the source data (for example, response bodies) replace what is in the matching target transaction. You can also leave the target data as-is.

**Note:** The process for combining conversations is similar. For each conversation in each source service image, the starter transaction is matched against each starter in the conversations of the target. If no starter matches, new conversations are created in the target image. If they do match, the same process for the stateless list is applied to each Meta node in the source conversation tree. The process adds new transactions and merges matching transactions as appropriate.

## Delete a Service Image

**Follow these steps:**

1. Right-click the service image on the Project panel and select Delete.  
A confirmation dialog asks you to confirm that you want to delete the selected service image.
2. Click OK.

The selected service image is permanently removed from your project.

**Note:** As a best practice, think of service images as transient and remove unused or outdated service images.

## Create a Service Image

**Follow these steps:**

1. Right-click VirtualServices, Images on the Project panel.
2. Select Create New VS Image.

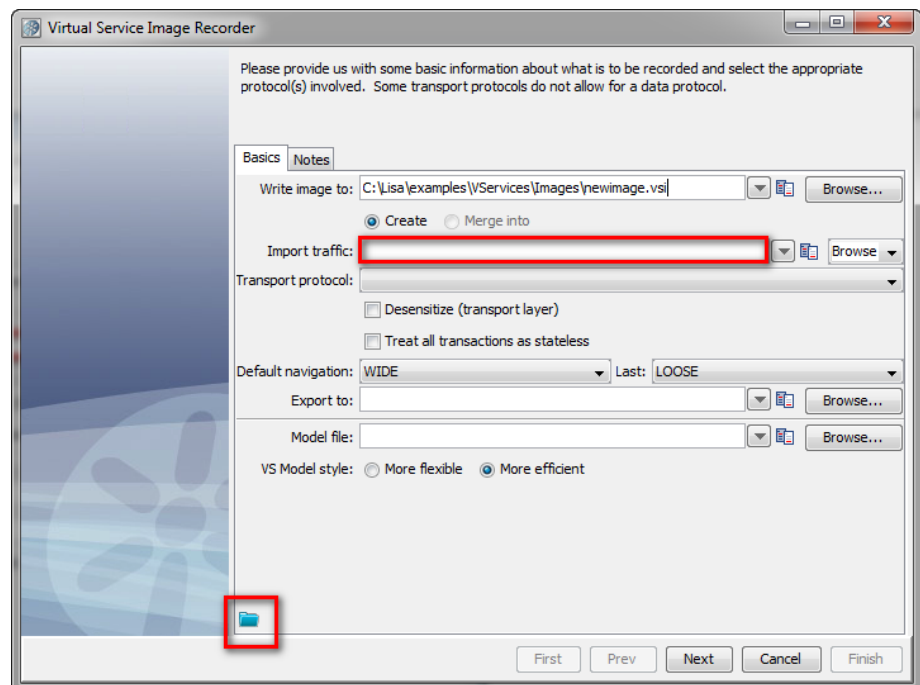
## Import Raw Traffic

You can only import raw traffic through the VSE Service Image Recorder.

Follow these steps:

1. Create a service image and select Virtual Service Image Recorder.
2. Enter the raw traffic file name or select it from the file browser.
3. To load the parameters for the service image from a previously saved recording session, click the blue folder at the bottom corner of the panel.

You are prompted to browse for a .vrs file with which to load the parameters.



4. To continue the recording process normally, select a transport protocol.

When you start the recording process, VSE imports the raw traffic to a new service image.

**More information:**

[Create a Service Image from Scratch](#) (see page 104)

[Create a Service Image from a WSDL](#) (see page 104)

[Create a Service Image from Request/Response Pair](#) (see page 113)

[Create a Service Image from PCAP](#) (see page 117)

[Work with VSMs](#) (see page 208)

[Using Data Protocols](#) (see page 209)

## Create a Service Image from Scratch

**Follow these steps:**

1. Right-click VirtualServices on the Project panel and select Create a New VS Image, From scratch.
2. Enter the identification and protocol information, and click OK.  
The Service Image Editor window opens.
3. Enter the specific parameters and information for the new service image.

## Create a Service Image from a WSDL

You can generate a virtual web service image from a WSDL in the following ways:

- [From the Quick Start menu](#) (see page 105)
- [From the Create New VS Image Option](#) (see page 106)


## From the Quick Start Menu

This procedure describes how to create a virtual service image from a WSDL using the UI.

**Follow these steps:**


1. Select Create an SI from a WSDL from the Quick Start menu.

2. Enter the name of a WSDL on the Connection tab.

Click Utilities  to find WSDLs on your system.

After you enter the WSDL name, the Service and Port fields are populated. The associated operations are listed on the Operations tab. You can select All or None, or you can use the check boxes to select specific operations to test.

3. Enter the name of the service image to create in the Save to field at the bottom of the window.

**Note:** If the default service image name is already in use, a warning icon  appears.

4. Click the green arrow  at the bottom of the window.

The Service Image Editor opens with your service image displayed.

## From Create New VS Image


This procedure describes how to create a virtual service image from a WSDL using the New VS Image option.

### Follow these steps:

1. Right-click VirtualServices, Images, on the Project panel and select Create New VS Image, From WSDL.

The Virtual Service From WSDL window opens.

2. Enter a service image name and the name of a VS model file.
3. Accept the default values for the remaining fields on this window.

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.

4. Click Next.

The Connection tab opens.

5. Enter the port number to which the virtual service listens in the Listen on port field at the bottom of the window.
6. Add the WSDL to be virtualized in the WSDL URL field.

This entry can be a local file or a URL.

7. Select the service in that WSDL that must be virtualized, in the Service field.

Usually, only one selection is available.

8. Select the operations in that service to virtualize.

By default, all the operations are selected.

9. Click Next.

The request/response side data protocols options open.

10. Select **Web Services (SOAP)**.


The Request Side Data Protocols list is prepopulated with Web Services (SOAP) and the XML data protocol handlers. The Response Side Data Protocols list is automatically populated with the Delimited Text data protocol.

11. Click Next.

12. See [Delimited Text Data Protocol](#) (see page 233) in *Using CA Service Virtualization* for information about how to configure the Delimited Text data protocol. When you have configured the Delimited Text data protocol, click Next.

On the next window, the service image is generated and the wizard is finished.

13. Click Finish.

**Note:** To save the settings on this recording to load to another service image recording, click Save  above the Finish button.

The blank virtual service model is populated with steps.

14. Save the virtual service model.

The service image that was generated is a stub service. The service image returns correctly formatted responses, but the values are default values.


The Service Model (VSM) that was saved is the model that is deployed to the Virtual Service Environment.

## Create a Service Image from WADL


This procedure describes how to create a virtual service image from Web Application Description Language (WADL).

**Follow these steps:**

1. Complete one of the following options:
  - Right-click VirtualServices on the Project panel and select Create New VS Image, From WADL.
  - From the File menu, select New, VS Image, From WADL.The Virtual Service From WADL window opens.
2. Enter a service image name and the name of a VS model file.

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.


For more information about field descriptions, see [Basics Tab](#) (see page 120).
3. Click Next.
4. In the Listen on port field at the bottom of the window, enter the port number to which the virtual service listens.
5. In the WADL URL field, add the WADL to virtualize.

This entry can be a WADL file on the file system or a URL.
6. Click Refresh WADL Cache .
7. In the Endpoint field, select the endpoint in the WADL that must be virtualized.

**Note:** Usually, only one selection is available.
8. In the Methods pane, select the methods in the endpoint to virtualize.

By default, all the methods are selected. You can click Select All or Select None, as appropriate.
9. Click Next.
10. Add or chain other Data Protocol Handlers as appropriate. By default, the Rest Data Protocol Handler is selected.
11. Click Next.

On the next window, the service image is generated and the wizard is finished.
12. Click Finish.

To save the settings on this recording to load to another service image recording, click Save  above the Finish button.



**Note:** The parameter `lisa.vse.rest.max.optionalqueryparams` specifies the maximum number of optional query parameters to process per method in a WADL file. The default is five; any optional parameters after the fifth one are ignored. We recommend that you do not change the value above five. This can result in the number of generated responses growing exponentially after the fifth one.

## Create a Service Image from RAML


This procedure describes how to create a virtual service image from RESTful API modeling language (RAML).

A RAML can define message bodies using a combination of Schema and Example properties. The Example property is used for the transaction body in DevTest. Make sure to specify the Example property in a message body, or else the Schema property is used instead.

**Important!** The Schema property is not interpreted in any way and appears exactly as specified in the transaction body within DevTest.


### Follow these steps:

1. Complete one of the following options:
  - Right-click VirtualServices on the Project panel and select Create New VS Image, From RAML.
  - From the File menu, select New, VS Image, From RAML.The Virtual Service From RAML window opens.
2. Enter a service image name and the name of a VS model file.

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.

For more information about field descriptions, see [Basics Tab](#) (see page 120).
3. Click Next.
4. In the Listen on port field at the bottom of the window, enter the port number to which the virtual service listens.
5. In the RAML URL field, add the RAML of the web service to virtualize.

You can also select the RAML from the drop-down list, or click the Browse button to locate the RAML from the file system.

This entry can be a RAML file on the file system or a URL.
6. Click Refresh RAML Cache .
- DevTest parses the RAML and populates the Endpoint field and the Methods pane.

**Note:** If DevTest fails to parse the RAML, a warning icon displays after the Endpoint field. To view the error message, click the warning icon.
7. In the Methods pane, select the methods to virtualize.


By default, all the methods are selected. You can click Select All or Select None, as appropriate. At least one method is required.
8. Click Next.

9. Add or chain other Data Protocol Handlers as appropriate. By default, the Rest Data Protocol Handler is selected.

10. Click Next.

On the next window, the service image is generated and the wizard is finished.

11. Click Finish.

To save the settings on this recording to load to another service image recording, click Save  above the Finish button.

**Note:** The parameter `lisa.vse.rest.max.optionalqueryparams` specifies the maximum number of optional query parameters to process per method in a RAML file. The default is five; any optional parameters after the fifth one are ignored. We recommend that you do not change the value above five. This can result in the number of generated responses growing exponentially after the fifth one.

## Create a Service Image from Layer 7

The Virtual Service From Layer7 generates a virtual service image from Layer 7.

### Prerequisites:

1. Download the Layer 7 Command-line Migration Tool 2.2 from the Layer 7 support website.
2. Extract the Jar file.

**Note:** The Jar file contains the cmt2.jar file that is required for step 3. This file can be located anywhere.

### Follow these steps:

1. Do one of the following:
  - Select File, New, VS Image, From Layer7
  - Right-click the root node of the project and select Create New VS Image, from Layer7.
  - Right-click the Virtual Services Images folder and select Create New VS Image, Layer7.

The Virtual Service From Layer7 window opens.

2. Enter the Layer7 Connection Info fields.
3. Click the folder icon to locate the cmt2.jar file and click Get Layer7 Services.

The Layer7 Services tab displays the available services.

4. Select a service and click Next.
5. Enter a service image name and the name of a VS model file.

Accept the default values for the remaining fields on this window.

**Note:** To load parameters from a previously saved service image, click the Load from File icon at the bottom of the window.

6. Click Next.

The Connection tab opens.

7. Select the operations in that service to virtualize.

By default, all the operations are selected.

8. Click Next.

The request/response side data protocols options open.


9. Select **Web Services (SOAP)**.

The Request Side Data Protocols list is prepopulated with Web Services (SOAP) and the XML Data Protocol data protocol handlers. The Response Side Data Protocols list is automatically populated with the Delimited Text Data Protocol.

10. Click Next.
11. See [Delimited Text Data Protocol](#) (see page 233) for information about how to configure the Delimited Text data protocol. When configured, click Next.

On the next window, the service image is generated and the wizard is finished.

12. Click Finish.

**Note:** To save the settings on this recording to load into another service image recording, click Save , above the Finish button.

The blank virtual service model is populated with steps.

13. Save the virtual service model.

The service image that was generated is a stub service. The service image returns correctly formatted responses, but the values are default values.

The service model (VSM) that was saved is the model that is deployed to the Virtual Service Environment.

## Create a Service Image from Request/Response Pair

You can generate a virtual service image from request/response pairs in the following ways:

- [From the UI](#) (see page 114)
- [From the Command Line](#) (see page 115)

You can use [sidecar files](#) (see page 116) to customize the metadata from your request/response pairs.

## Create a Service Image from Request/Response Pairs in the UI


This procedure describes how to create a virtual service image from request/response pairs using the UI.

### Follow these steps:

1. Right-click the VirtualServices, Images icon and select Create New VS Image, From Request/Response Pairs.

The Virtual Service From Request/Response Pairs page opens.

2. Enter a service image name and the name of a VS model file.
3. Accept the default values for the remaining fields on this window and click Next.

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.

4. Browse the file system for the directory that contains your request/response pairs.

The request/response pairs should be named with a unique identifier, then a "-req" on the request side and a "-rsp" on the response side, with an **.xml** or **.txt** extension. For example, **addUserObject-req.xml** and **addUserObject-rsp.xml**.

**Note:** A request can have multiple responses. Using the previous example, the following file names would create one request with three responses.

- addUserObject-req.xml
- addUserObject-rsp1.xml
- addUserObject-rsp2.xml
- addUserObject-rsp3.xml

VSE generates a transaction for each request/response pair in the directory you specify. For HTTP/S, the files must contain the entire SOAP envelope and the headers.

5. Specify the following information:
  - Transport protocol
  - Appropriate encoding
  - Whether to use binary request/response pairs

6. Click Configure.

The configuration window for the selected transport protocol opens.

7. Enter the configuration information for the request and click Finish.

The Virtual Service from Request/Response Pairs window opens.

8. Click Configure again to persist the values you entered.

If you select a different protocol, that protocol is presented. You can provide the configuration information for it.


9. Click Next.

The Data Protocols panel opens, where the request/response pairs were analyzed and appropriate data protocols for the request and response were defaulted.

10. Make any changes or add more data protocols and click Next.

The virtual service request/response pairs for token identification and conversations display. Only stateless transactions are supported from request/response pairs.

11. Click Next.

**Note:** To save the settings on this recording to load to another service image recording, click Save  above the Finish button.

After the processing of the VS image finishes, you can open the service image and the virtual service model.

## Create a Service Image from Request/Response Pairs from the Command Line

In addition to creating a service image from request/response pairs using the UI, you can create the image through the ServiceImageManager command line.

### Follow these steps:

1. Create a service image in the UI.
2. After the image is created, click Save  above the Finish button, then click Finish.

The settings are saved in a file with a **.vrs** extension.

3. Navigate to the LISA\_HOME\bin directory and enter the following command:

```
ServiceImageManager -vrs recording-session-file
vsi-file=vsi-file --vsm_file=vsm-file --record
```

#### **recording-session-file**

Defines the path of the **.vrs** file that you created previously, and

#### **vsi-file**

Defines the service image file that is created.

#### **vsm\_file**

Defines the virtual service model files that is created.

## Sidecar Files with Request/Response Pairs

When VSE creates a service image from request/response pairs, the system can use another file with the request/response pairs named a sidecar file. A sidecar file is a properties file in which you can add key/value pairs that need to be added to the metadata of certain requests, responses, or both. You can add some files that are generic for all requests and responses.

### An example:

A directory that is named **soap** has request/response pair files that are named **abc-req.xml** and **abc-rsp.xml**. You add the sidecar files with the naming convention of **abc-req-meta.properties**, **abc-rsp-meta.properties**, and so on. If you intend to add a property to the meta of all requests and responses in that directory, use the file names **meta-req.properties** and **meta-rsp.properties**. The entries in the transaction-specific sidecar files (**abc-req-meta.properties** and **abc-rsp-meta.properties**) override anything that is found in the global sidecar files (**meta-req.properties** and **meta-rsp.properties**). Therefore, you can have the set of all the defaults in the global files and then override the defaults for specific transactions with transaction-specific sidecars.

For example, if you have three requests/responses (**abc1-req.xml/abc1-rsp.xml**, **abc2-req.xml/abc2-rsp.xml**, and **abc3-req.xml/abc3-rsp.xml**), you can have a common sidecar file, **rsp-meta.properties**, that has:

`Content-type=text/plain`

and an abc3-specific sidecar file, **abc3-rsp-meta.properties**, that has:

`Content-type=text/html`

In the service image, the metadata for responses **abc1** and **abc2** has the Content-type meta property set to "text/plain", but the **abc3** response has the value "text/html".



## Create a Service Image from PCAP

If you use packet capture software such as Wireshark to create logs of traffic, VSE can use those logs to create a virtual service image from a packet capture file (PCAP).

### Prerequisites:

1. Download the appropriate binary package for your operating system from <http://jnetpcap.com/download>.
2. Add the **jnetpcap.jar** from the binary package to the **LISA\_HOME\lib** directory and the jnetpcap native library to the **LISA\_HOME\bin** directory.

The native library is different for different operating systems. For Windows, it is jnetpcap.dll.

The PCAP functionality is now configured.


3. Restart DevTest Workstation (if it is running) to pick up the configuration changes.

### Follow these steps:

1. Right-click the VirtualServices, Images folder and select Create New VS Image, From PCAP.

Enter a service image name and the name of a virtual service model file.

Accept the default values for the remaining fields on this window.

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.

2. (Optional) To add documentation about this virtual service, click the Notes tab.
3. Click Next.  
The data protocol window opens.
4. Click Next.
5. To select the packet capture file to be used for input, enter the file name or browse the file system.

6. Select HTTP/S as the transport protocol, and click Configure.

The Virtual Service from PCAP Transport Protocol Configuration window opens.

7. Enter the following configuration options:

#### Listen/Record on port

Defines the port on which the client communicates to DevTest.

#### Target host

Defines the name or IP address of the target host where the server runs.

#### Target port

Defines the target port number listened to by the server. Leave this field blank if you will select a Proxy passthrough style.

**Defaults:** 80 (HTTP) and 443 (HTTPS)

**Note:** Target host and Target port are important. They determine how DevTest matches packets. Select Gateway and enter the IP address and port of the server hosting the service to be virtualized. You could have some network traffic for every computer on the subnet where the capture was performed, depending on how the PCAP capture was done. Given an IP address and port, the data can be filtered from the file for all packets going to or from a specific IP and port. Then those packets are reconstructed to form valid TCP streams, which remove duplicates, and reorders packets in the correct order, and so forth. This work is all done for you by the operating system TCP stack during a real recording. These streams are replayed to the real protocol (HTTP) and as far as the protocol is concerned, the data arrives over a valid TCP stream.

#### Recorder passthru style

Specifies how VS Image Recorder acts during recording. Select **Gateway**.

#### Use SSL to server

Specifies whether DevTest uses HTTPS to send the request to the server.

- **Selected:** DevTest sends an HTTPS (secured layer) request to the server.

If you select Use SSL to server, but you do not select Use SSL to client, DevTest uses an HTTP connection for recording. DevTest then sends those requests to the server using HTTPS.

- **Cleared:** DevTest sends an HTTP request to the server.

#### Use SSL to client

Specifies whether to use a custom keystore to play back an SSL request from a client. This option is only enabled when Use SSL to server is selected.

##### Values:

- **Selected:** You can specify a custom client keystore and a passphrase. If these parameters are entered, they are used instead of the hard-coded defaults.
- **Cleared:** You cannot specify a custom client keystore and a passphrase.

#### SSL keystore file

Specifies the name of the keystore file.

#### Keystore password

Specifies the password associated with the specified keystore file.

**Note:** For more information about configuring VSE in a two-way SSL environment, see [Virtualizing Two-way SSL Connections](#) (see page 131).

#### Allow duplicate specific transactions (good for NTLM)

Specifies whether to record duplicate specific transactions.


8. Click Finish to return to the previous window.
9. Click Next to start recording.

## Create a Service Image by Recording

The Virtual Service Image Recorder generates service images, which pretend to be what you recorded.

### Follow these steps:

1. To start recording a new virtual service image, complete one of the following steps:

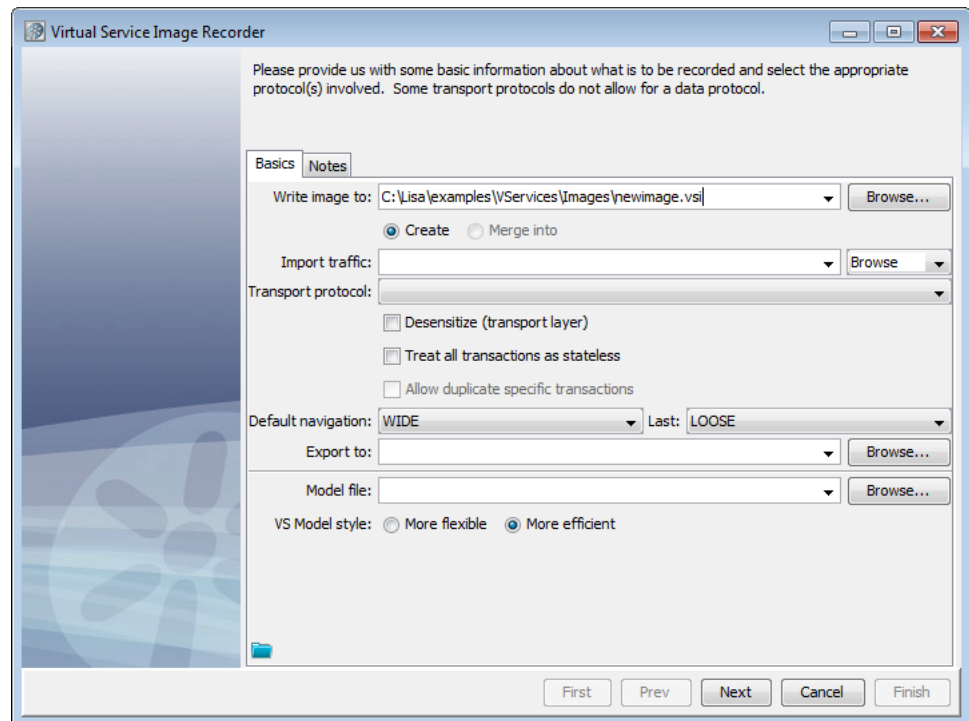
- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, By recording.

The Virtual Service Image Recorder opens.

2. Complete the following Virtual Service Image Recorder tabs, as appropriate:
  - [Basics](#) (see page 120)
  - [Data Protocols](#) (see page 123)
  - [Recording by Transport Protocol](#) (see page 125) (provides instructions for each method of recording virtual service images)

## Basics Tab

The Basics tab on the first window in the Virtual Service Image Recorder wizard provides the name, protocol, and navigation options for the image. Enter the information as needed in the fields, depending on the protocol you are using. All subsequent windows are protocol-specific. For more information about specific protocols, see "[Virtual Service Image Recorder - Transport Protocols](#) (see page 125)".



The Basics tab options are:

### Write image to

Specifies a unique service image name.

The VSI path defaults to the project that was open when DevTest Workstation started. If you change to another project, the VSI path still defaults to the project that was open when DevTest Workstation started.

### Import traffic

Specifies a raw or conversational XML traffic file to import. This field can be blank if no such file exists. If you specify a file, the transactions in the referenced XML document are merged with those resulting from the ensuing recording.

### Transport protocol

Specifies the transport protocol to use. For more information, see "[Virtual Service Image Recorder - Transport Protocols](#) (see page 125)".

### Desensitize

Specifies whether to try to recognize sensitive data and substitute random values during the recording. For more information, see "[Desensitizing Data](#) (see page 351)".

**Treat all transactions as stateless**

Specifies whether to treat all recorded transactions as stateless. In most cases, leave this option cleared.

**Allow duplicate specific transactions**

Specifies whether DevTest can respond more than once to the same call, choosing a different response. Round-robin matching only happens if this check box is selected. This option is disabled for transport protocols that do not allow duplicate specific transactions.

**Default navigation**

Specifies the navigation tolerance that determines where in the conversation tree a VSM searches for a transaction that follows the specified transaction. Select the default navigation tolerance for all except the last (leaf) transactions.

**Values:**

- CLOSE: The children of the current transaction are searched.
- WIDE: A close search, including the current transaction plus siblings and nieces/nephews of the current transactions.
- LOOSE: A wide search, plus the parent and siblings of the current transaction, followed by the children of the starting transaction. If both fail to find a match, then the starting transactions for all conversations are checked, resulting in searching the full conversation.

**Default:** WIDE

**Last**

Specifies the navigation tolerance that determines where in the conversation tree a VSM searches for a transaction that follows the last (leaf) transactions.

**Values:**

- CLOSE: The children of the current transaction are searched.
- WIDE: A close search, including the current transaction plus siblings and nieces/nephews of the current transactions.
- LOOSE: A wide search, plus the parent and siblings of the current transaction, followed by the children of the starting transaction. If both fail to find a match, then the starting transactions for all conversations are checked, resulting in searching the full conversation.

**Default:** LOOSE

**Export to**

Specifies the full path of a file where you want the raw traffic logged. If you specify one, every time a transaction is offered to the recorder (either from the transport protocol during actual recording or from the import process), it is written to this file. A recording session can be captured for importing later and can be reused while data protocol details are being refined.

#### **Model file**

Defines the full path of your virtual service model file for this service image. If you provide a file name in this field, the recorder generates a VSM automatically. The model style is in effect only if you request a VSM.

#### **VS Model style**


Specifies whether to generate a VSM including the prepare steps.

##### **Values:**

**More flexible:** Includes prepare steps (leading to a five-step model in case of HTTP/S protocol).

**More efficient:** The prepare steps are absent (leading to a three-step model in case of HTTP/S protocol).

**Default:** More flexible

**Note:** To load parameters from a previously saved service image, click Load from File  at the bottom of the window.

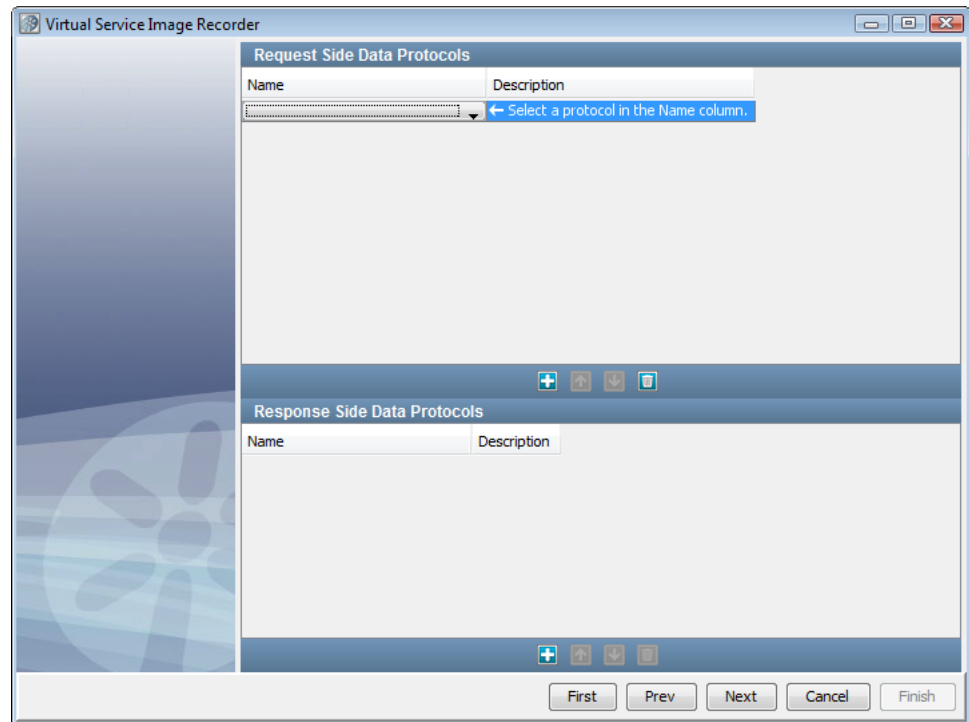
The Notes tab lets you document this service image.

**Note:** If you import a raw traffic file with the VSE Recorder and then click the back button to return to the first panel, the VSE Recorder reimports the traffic file and processes the transactions again, creating twice the number of transactions.

Additionally, if you complete the process but you do not initially designate a traffic file and progress to the recording panel, then go back to the first panel and select the traffic file; when you go through the VSE Recorder again, the recorder processes no transactions.

## Data Protocols

The second window in the Virtual Service Image Recorder wizard lets you enter information about data protocols for the virtual service.



The recorder can use the following data protocol handlers. Choosing the appropriate data protocol helps the recorder to analyze the information it records to differentiate the conversations and identify transactions belonging to the conversations. You can chain these data protocol handlers to use them with each other.

### Auto Hash Transaction Discovery

Identifies a message by the hash code of the data. The hash code changes with even a slight change in the data, which effectively makes all requests unique. This protocol is useful if you run the same small set of requests against the service.

### CICS Copybook Data Protocol

Splits the recorded request into its respective container chunks. It then sends each chunk to the Copybook data protocol and aggregates the corresponding XML.

### Copybook Data Protocol

Converts copybook text to XML.

### CTG Copybook Data Protocol

Splits the recorded request into its respective container chunks. It then sends each chunk to the Copybook data protocol and aggregates the corresponding XML.

### Data Desensitizer

Tries to recognize sensitive data and substitute random values during the recording. For more information, see [Desensitizing Data](#) (see page 351).

**Delimited Text Data Protocol**

Converts delimited strings to XML.

**DRDA Data Protocol**

Converts binary DRDA payloads to XML during recording to facilitate alignment with native DevTest functionality, readability, and dynamic data support. Converts responses back to their native format on playback.

**EDI X12 Data Protocol**

Transforms ANSI X12 EDI documents to an XML representation in the body of the request.

**Generic XML Payload Parser**

Identifies whether the requests and responses are XML strings. If you use this protocol, you can identify variables in the XML messages that the recorder uses.

**JSON Data Protocol**

Converts JSON data to an XML equivalent and converts XML data to JSON format.

**Request Data Copier**

Copies data from the current inbound request into the current testing context.

**Request Data Manager**

Manipulates VSE requests as they are recorded or played back.

**REST Data Protocol**

Analyzes HTTP requests that follow the REST architectural style

**Scriptable Data Protocol**

Provides scripts on the request side, the response side, or both, to process the request or response.

**SWIFT Data Protocol**

Converts SWIFT messages to an XML equivalent and converts XML data to SWIFT messages.

**Web Services Bridge**

Applies only to the DevTest Travel example. You can ignore this data protocol because it is specific to the example and is not useful in a general case.

**Web Services (SOAP)**

Applies to use by a web service client.

**Web Services (SOAP Headers)**

Converts SOAP header elements into request arguments.



**WS-Security Request**

Strips security from the SOAP Request before sending it along the Virtualize framework and applies security to outgoing SOAP responses.

**XML Data Protocol**

Converts an XML document into a proper operation/arguments type of request.

**Note:** JDBC does not allow a data protocol.

For more information about these data protocols, see "[Using Data Protocols](#) (see page 209)".

For more information about using a dynamic data protocol, see "[Generic XML Payload Parser](#) (see page 240)".

## Transport Protocols

**Each available transport protocol is described in the following sections.**

[HTTPS](#) (see page 126)

[IBM WebSphere MQ](#) (see page 134)

[JMS](#) (see page 139)

[Standard JMS](#) (see page 147)

[Java](#) (see page 152)

[JDBC](#) (see page 155)

[TCP](#) (see page 164)

[Record CICS \(LINK DTP MRO\)](#) (see page 169)

[Record CICS Transaction Gateway \(ECI\) Images](#) (see page 187)

[Record IMS Connect Service Images](#) (see page 189)

[Record SAP RFC via JCo](#) (see page 193)


[Record JCo IDoc](#) (see page 195)

[Opaque Data Processing](#) (see page 198)

## HTTPS

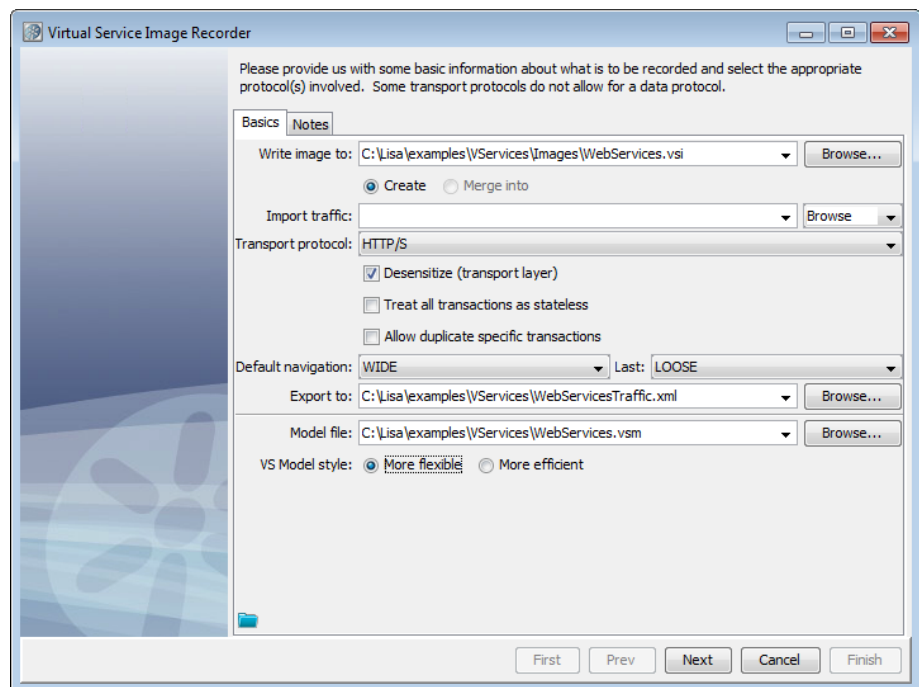
**Follow these steps:**

1. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

2. Complete the [Basics](#) (see page 120) tab as in the following graphic:



3. Click Next.

The next wizard window opens.

4. Enter the port and host information for this step.

**Listen/Record on port**

Defines the port on which the client communicates to DevTest. It is typical to select 8001, but you can use another port number.

**Target host**

Defines the name or IP address of the target host where the server runs. Leave blank if you are going to select a Proxy pass through style.

**Target port**

Defines the target port number listened to by the server. Leave this field blank if you will select a Proxy passthrough style.

**Defaults:** 80 (HTTP) and 443 (HTTPS)

#### **Recorder pass-through style**

Specifies how the VS Image Recorder acts during recording. The choices are **Gateway** and **Proxy**. If you select **Proxy**, the contents in Target host and Target port fields are cleared and the fields become disabled. This choice affects how the client connects in the recording mode.

- If VS Image Recorder listens in a gateway mode, the client must send HTTP requests directly to the recorder and not to the server. If the client is a browser, the URL contains the host and port of the recorder instead of the host and port of the server.
- If VS Image Recorder listens in a proxy mode, the client must specify the recorder host and port as the proxy. If the client is a browser, then the URL contains the host and port of the server. The proxy settings must be set to route the request through the recorder.

**Most of the HTTP clients have a setting for NOT using proxy for localhost. If your VS Image Recorder is running on localhost in proxy mode, disable this setting for the traffic to get correctly passed through the recorder.**

#### **Do not modify host header parameter received from client**

Specifies whether to pass through the value of the Host parameter. This option is only available when recording in Gateway mode. The pass-through option instructs the recorder not to rewrite the Host header parameter when resending traffic to the target endpoint.

#### **Use SSL to server**

Specifies whether DevTest uses HTTPS to send the request to the server.

- **Selected:** DevTest sends an HTTPS (secured layer) request to the server.

If you select Use SSL to server, but you do not select Use SSL to client, DevTest uses an HTTP connection for recording. DevTest then sends those requests to the server using HTTPS.

**Cleared:** DevTest sends an HTTP request to the server.

#### **Use SSL to client**

Specifies whether to use a custom keystore to play back an SSL request from a client. This option is only enabled when Use SSL to server is selected.

##### **Values:**

- **Selected:** You can specify a custom client keystore and a passphrase. If these parameters are entered, they are used instead of the hard-coded defaults.

**Cleared:** You cannot specify a custom client keystore and a passphrase.

**SSL keystore file**

Specifies the name of the keystore file.

**Keystore password**

Specifies the password associated with the specified keystore file.

**Note:** For more information about configuring VSE in a two-way SSL environment, see [Virtualizing Two-way SSL Connections](#) (see page 131).

5. Click Next.

The VS Image Recorder starts recording the traffic. The assigned port and service target display on this window.

6. To send the requests to the server routed through the VS Image Recorder to start recording traffic, use your HTTP client.

As the VS Image Recorder records transactions, the dynamic display statistics on the lower portion of the window increase. The options and dynamic display statistics include:

**Total conversations**

Displays the number of conversations recorded.

**Total transactions**

Displays the number of transactions recorded.

**Clear**

Clears the list of currently recorded transactions.

7. When you have completed the recording, click Next to move to the next step.

If you click Next and no transactions were recorded, an error message appears. Click OK to continue recording.

**Note:** If the transactions are not recorded, you could have a port conflict. The client sends transactions to the application instead of the Virtual Service Recorder. If another service is using that port, either stop that service or change the port setting so there is no longer a conflict.

The Transactions tab displays a list of the most recent transactions recorded. On this list of transactions, you can double-click a transaction and can see a dialog showing the content of the transaction.

8. Verify the base path and update it if necessary.
9. To require a bind-to-port step before processing requests, select the A separate bind-to-port step is required check box.
10. Click Next.
11. Do not select any value for the data protocol on the next window and click Next.

12. If no conversations were detected during the recording process, select transactions that start conversations. For token-based conversations, specify where tokens can be found. Use the Token Identification area in the VS Image Recorder. Select the transactions that start conversations and identify the session tokens. To designate the **getNewToken** Conversation Starter Transaction listed as a conversation starter, select it and click the blue arrow.

The step components include:

**Conversation Starter Transactions**

Lists the transactions that you have selected as conversation starters. To move the transaction to the Remaining Transactions list (if you do not want it to be a conversation starter), select a transaction and click the arrow.

**Remaining Transactions**

Lists the recorded transactions. To move the transaction to the Conversation Starter Transactions list, select a transaction and click the arrow.

**Plus icon**

Selects all transactions (either Conversation Starters or Remaining) in the list that are like a selected transaction. To move all the selected transactions, use the appropriate arrow button.

**Conversation count**

Displays the number of conversations in the recording. As you build conversations, the number increases.

**Force stateless**

From the Remaining Transactions list, select any transactions that should stay stateless and select the check box. For example, you could decide that a transaction that includes an image should stay stateless even though it contains a conversation starter token.

**Stateless Transactions**

Click to see a list of all transactions that remain stateless, assuming the identified conversations on this panel. You can use the list to verify that you have identified all the conversation starter transactions.

**Save**

Click to save the raw recorded transactions. Click Browse to navigate to the location in which to save the file. You can import the raw traffic recording in the Basics tab before beginning a new recording.

**Response**

For the currently selected transaction, this field identifies which of its responses to look in. In general, 1 is the only option.

**Look in**

This field identifies the piece of the response you want to see when looking for conversation tokens. The drop-down list contains an entry for each of Meta data entries in the response, plus one for the body of the response.

#### Token Identification area


Based on the selected transaction and response, the content of the selected Look in section of the response is displayed here.

- To mark a piece of the text as a conversation token, select the text and click the red rubber stamp icon. The text is then highlighted in yellow.
- To mark the text as no longer a conversation token, either mark a different piece of the text or click Erase.
- After you mark a token, you can use the Search icon to find similar transactions and mark their tokens. To open a dialog where you can select text (such as XML tags) that bound the conversation token, click the Search icon. To specify the leading and trailing text to search for, use this method.

#### 13. Click Next.

During postprocessing, the VS Image Recorder displays the processing status. As part of the preparation for writing the .vsi file, the recorder verifies request and response bodies to ensure that, if they are marked as text, that they are text. If they are not, the type is switched to binary.

The recorder completes postprocessing the recording.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

#### 14. Click Finish to store the image.

#### 15. Review and save the virtual service model in DevTest Workstation.

## Virtualize Two-way SSL Connections

To virtualize a two-way SSL connection, DevTest must have one of the following:

- Both client keystore and server keystore.
- Client keystore and DevTest keystore (see **webreckeys.ks** in the installation directory). Extract the DevTest certificate from the DevTest keystore and add it to the client truststore. The DevTest certificate is a self-signed certificate and not a certificate that a CA authority issues. This workaround only works when the client accepts self-signed certificates.

Both cases result in two keystores: one client keystore and one server keystore (or DevTest keystore).

Configure the SSL properties in the local.properties file, (in the installation directory) to use the client keystore as follows:

**ssl.client.cert.path**

Defines the path to your keystore; for example:

c:/mykeystore.jks.

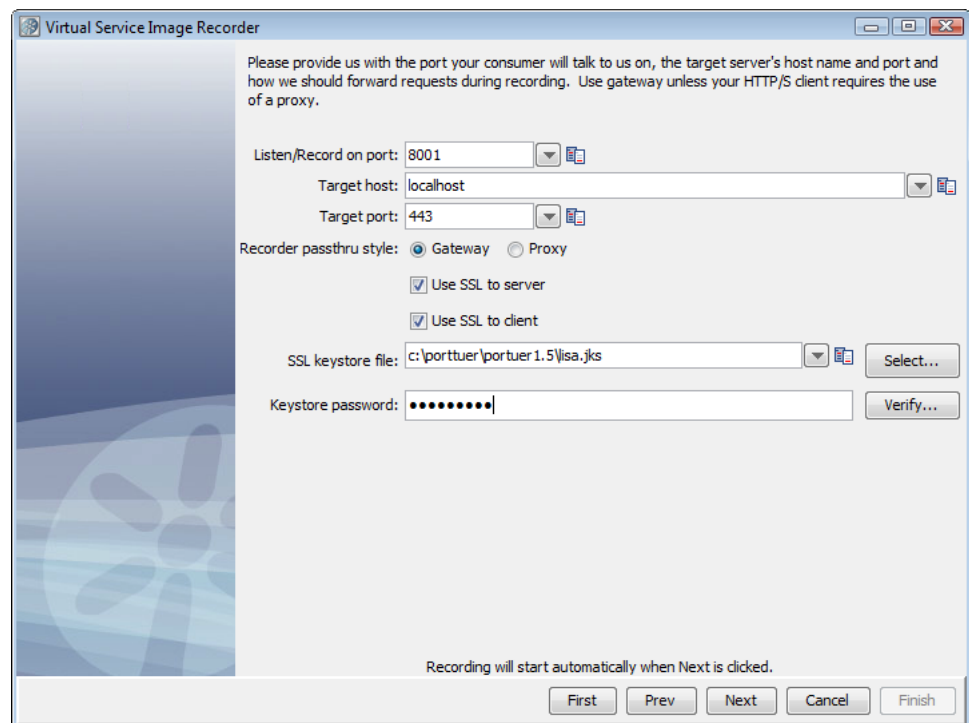
**ssl.client.cert.pass**

Defines your keystore password.

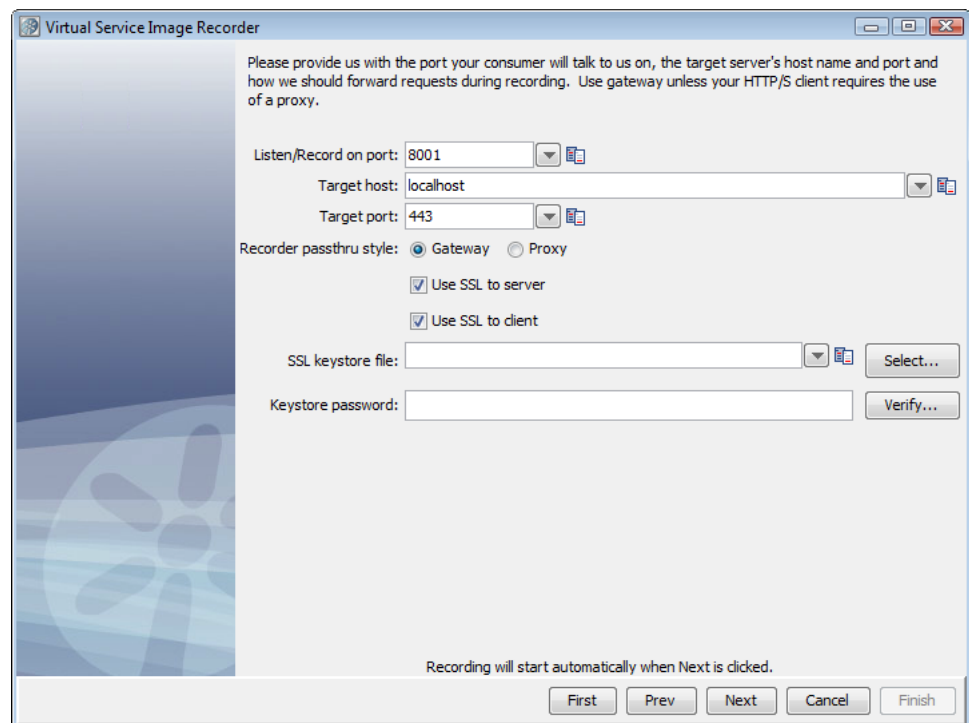
**ssl.client.key.pass**

Defines your certificate password.

Start the VSE recorder and configure it to use two-way SSL. If you use a client and a server keystore, your recorder resembles the following graphic:



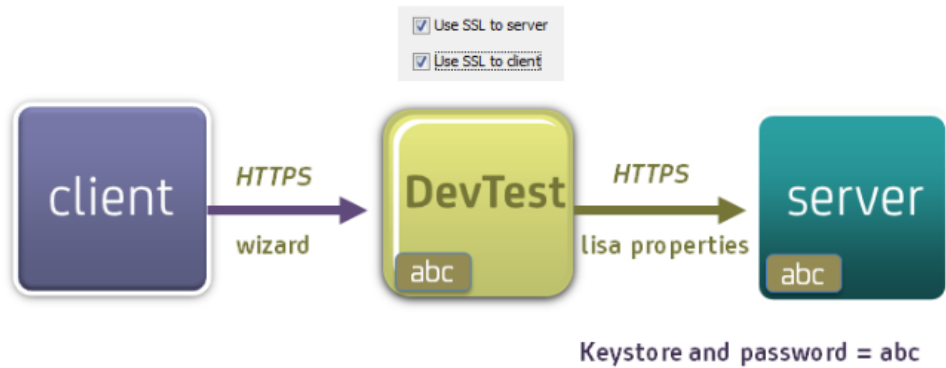
If you use a DevTest keystore instead of the actual server keystore, you do not need to provide the path to it. The DevTest keystore is used by default and you must configure your recorder similar to the following graphic:



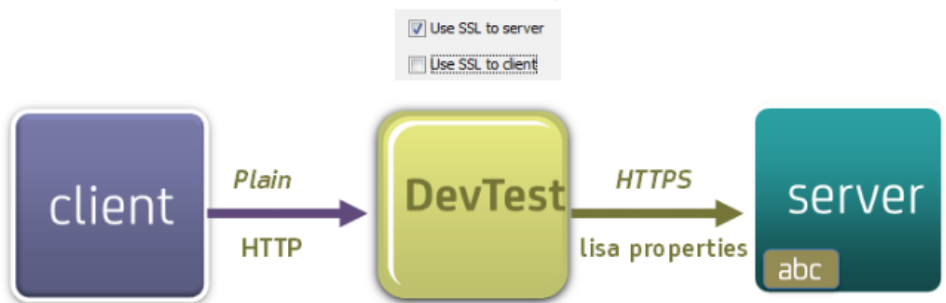


The following diagrams illustrate one-way and two-way SSL virtualization.

## DevTest 2-way SSL



## LISA 1-way SSL




## IBM WebSphere MQ

**Prerequisites:** Using DevTest with this application requires that you make one or more files available to DevTest. For more information, see Third-Party File Requirements in *Administering*.

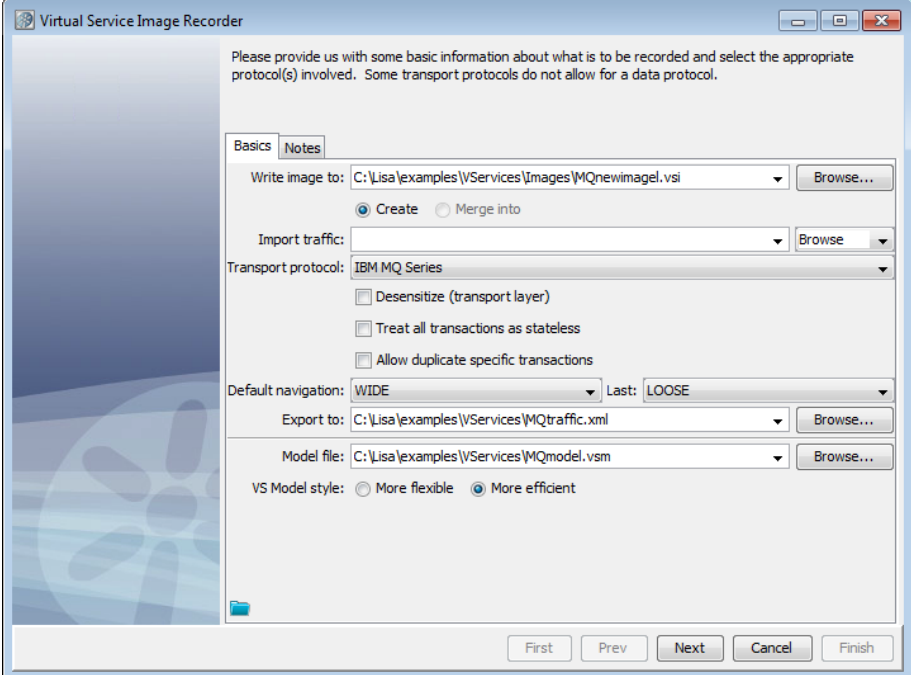
**Follow these steps:**

1. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

2. Complete the [Basics](#) (see page 120) tab as in the following graphic:



The screenshot shows the 'Virtual Service Image Recorder' dialog box with the 'Basics' tab selected. The 'Notes' tab is also visible. The dialog contains the following fields and options:

- Write image to:** C:\Lisa\examples\VSservices\Images\MQnewimagel.vsi (with a 'Browse...' button)
- Import traffic:** (empty field with a 'Browse...' button)
- Transport protocol:** IBM MQ Series (dropdown menu)
- Options (checkboxes):**
  - ☐ Desensitize (transport layer)
  - ☐ Treat all transactions as stateless
  - ☐ Allow duplicate specific transactions
- Default navigation:** WIDE (dropdown menu)
- Last:** LOOSE (dropdown menu)
- Export to:** C:\Lisa\examples\VSservices\MQtraffic.xml (with a 'Browse...' button)
- Model file:** C:\Lisa\examples\VSservices\MQmodel.vsm (with a 'Browse...' button)
- VS Model style:**
  - ☐ More flexible
  - ☒ More efficient

At the bottom of the dialog are buttons: First, Prev, Next, Cancel, and Finish.

3. Click Next.

The recording mode selection step opens, with one of the following options selected:

**Proxy Mode**

Proxy mode is the only true recording mode that VSE supports. Proxy mode provides the following options:

- Generate the service from the Live queues
- Generate the service using the Proxy queues

Proxy mode allows the use of proxy destinations to be set up on the message bus. The client application is configured to put messages on those proxy destinations and DevTest records them and forwards to the real destination. The same thing happens on the response side. DevTest is configured to listen on the real response destination, get the message, and forward it to the response proxy destination. This mode can behave differently if you enable temporary destinations, making the response side automated.

### **Import Mode**

This mode for recording virtual services is available if you specify a raw traffic file in the Import traffic field on the initial recording window. In import mode, you can specify extra information about which request and response queues were detected in the raw traffic file. You can also select to skip the request response steps.

4. If you are recording in proxy mode, complete the following fields:

#### **Generate service using the Live queues / Proxy queues**

Specifies whether to use the live queues or the proxy queues when generating the final virtual service model and image.

#### **Max Pending Transactions**

Defines the maximum number of running response listeners on each response queue. These response listeners run as long as a transaction is pending. When the number of pending transactions exceeds the maximum, the oldest transaction closes automatically. If you enter 0 in this field, there is no maximum.

#### **Disable Multiple Responses**

Specifies whether to support multiple responses in each transaction. By default, a transaction stays pending after the first response, waiting for more responses to arrive for the same transaction. When you select this option, the transaction automatically closes after the first response. If there are many transactions coming in quickly, this option can boost performance, especially for MQ.

### **Correlation**

Contains the possible schemes for correlating the request and response sides of individual transactions. JMS is asynchronous, which means the requests and responses are received separately. This drop-down list lets you define to the VSE Recorder which request to associate with which response. The Correlation field has the following options:

- **Sequential:** Associates every response with the last request received, chronologically. There is no correlation scheme, so the VSE MQ recorder acquires an exclusive read lock on the live response queue to ensure that another MQ listener cannot take response messages from it.

When you specify a correlation scheme, such as Correlation ID or Message ID to Correlation ID, the VSE MQ recorder can assume that any other listener on the live response queue will also use a correlation scheme. If all listeners on the live response queue use correlation schemes, the VSE MQ recorder can keep its responses separate without resorting to an exclusive read lock, so it opens the queue with the shared input flag.

- Correlation ID: The request and the response must have the same Correlation ID.
- Message ID to Correlation ID: The request Message ID must be the same as the response Correlation ID.
- Message ID: The request and the response have the same Message ID.

5. If you are recording in import mode, complete the following fields:

#### **Client Mode**

Specifies how to interact with the WebSphere MQ server.

##### **Values:**

- Native Client: A pure Java implementation using IBM-specific APIs
- JMS: A pure Java implementation that is based on the JMS specification. If you want this implementation, the best practice is to use the JMS transport protocol instead of MQ.
- Bindings: This option requires access to the native libraries from a WebSphere MQ client installation. Ensure that the DevTest application run time can access these libraries. In most cases, having these libraries available in the PATH environment is adequate.

#### **Review the queues and transaction tracking mode**

Specifies whether to skip the request and response steps. If the raw traffic file comes from CAI, this option is cleared. CAI autodetects queues and transactions. If the raw traffic file does not come from CAI, this option is selected by default to let you review destination and tracking settings.

#### **Correlation**

Contains the possible schemes for correlating the request and response sides of individual transactions. JMS is asynchronous, which means the requests and responses are received separately. This drop-down list lets you define to the VSE Recorder which request to associate with which response. The Correlation field has the following options:

- Sequential: Associates every response with the last request received, chronologically. There is no correlation scheme, so the VSE MQ recorder acquires an exclusive read lock on the live response queue to ensure that another MQ listener cannot take response messages from it.

When you specify a correlation scheme, such as Correlation ID or Message ID to Correlation ID, the VSE MQ recorder can assume that any other listener on the live response queue will also use a correlation scheme. If all listeners on the live response queue use correlation schemes, the VSE MQ recorder can keep its responses separate without resorting to an exclusive read lock, so it opens the queue with the shared input flag.

- Correlation ID: The request and the response must have the same Correlation ID.
- Message ID to Correlation ID: The request Message ID must be the same as the response Correlation ID.
- Message ID: The request and the response have the same Message ID.

6. Click Next.

The Destination Info tab opens.

7. Enter your proxy and live queue names, and select the queue type.

The Create Proxy Queue check box lets you create a temporary queue on the fly to be used as a proxy queue. Select this option if you did not manually create the proxy queue on WebSphere MQ.

8. Click the Connection setUp tab.

9. Enter the connection parameters used to connect to the MOM.

These connection parameters are internally saved.

The Advanced tab at the bottom of the Connection setUp tab includes the following subtabs:

#### **Environment**

Lets you add, delete, and specify values for more MQ environment settings.


#### **MQ Exits**

Lets you point to MQ exits for Security, Send, and Receive.

10. To define an optional separate set of connection information for when your proxy queues are on a different queue manager from your live queues, click the Proxy Connection setUp tab.

11. Define the connection details for the response destinations on which to listen for messages.

12. Define the proxy queue on which the client application will receive these responses.

- Remember that multiple responses are possible for a request. To register a set of response proxy queues, click Add .
- Select the Use for Unknown Responses check box before registering the response queue to use for unknown requests.

- To define a temporary destination, select the Use temporary queue/topic check box. Selecting this option disables the destination name and proxy destination name fields and marks the response listener you are adding as a temporary response. The destination name is blank.

A "temporary" destination in messaging is a destination that is created on demand for a messaging client. A temporary destination is typically used in request/response scenarios. DevTest supports using a temporary queue for the responses, but only supports one concurrent transaction with a temporary queue at a time.

13. Click Next.

The Destination List tab opens.

14. Click the Current Connection Info tab and verify that the connection information is correct.

15. The information that is shown on the Current Connection Info tab is copied from the connection information that you gave earlier. You might want to change it in a rare case when the response connection information is different.

16. Click Next to start recording.

The names of the queues to which VSE is listening display, and the status is Waiting.

If you connect to WebSphere MQ and you chose to create the proxy queues on the fly, those proxy queues are created.

17. Run the client that adds messages to the request proxy queue.

VSE copies the messages to the real request queue. The server acquires those messages from there and sends responses to the response queue or queues. Again, VSE acquires those messages and copies them to the response proxy queue, where the client is listening.


As the transactions are recorded, the message count increases and the Total sessions and Total transactions counts increase on the Virtual Service Image Recorder window. At the end of recording, all the requests have gone through the same request queue. About half of the responses have returned through temporary queues, and the other half have returned through the nontemporary response queue.

When the run is complete, you may see the count of messages on the response queues one fewer than you would expect. Because a single request can have multiple responses, VSE will not yet have recognized the last transactions complete. The messages corresponding to the last transaction are therefore not counted.

18. Click Next to trigger closure of the last transaction and complete the necessary cleaning. (You would see an intermediate window if you were using a dynamic data protocol.) As part of preparing the recorder for writing out the .vsi file, the recorder verifies request and response bodies to ensure that, if they are text, if so marked. If they are not, the type is switched to binary.

A Request Data Manager data protocol has been added on the request side. For more information about configuring this protocol, see "[Request Data Manager](#) (see page 248)". You can change or add more data protocols.

19. Click Next.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

20. Click Finish to close the window and store the image.

21. Review and save the VSM that was generated in the main window.

## JMS

The following topics contain detailed instructions for recording a service image that uses the JMS transport protocol:

- [Basic Proxy Recording](#) (see page 140)
- [TIBCO Monitor Recorder](#) (see page 145)

**Prerequisites:** Using DevTest with this application requires that you make one or more files available to DevTest. For more information, see Third-Party File Requirements in *Administering*.

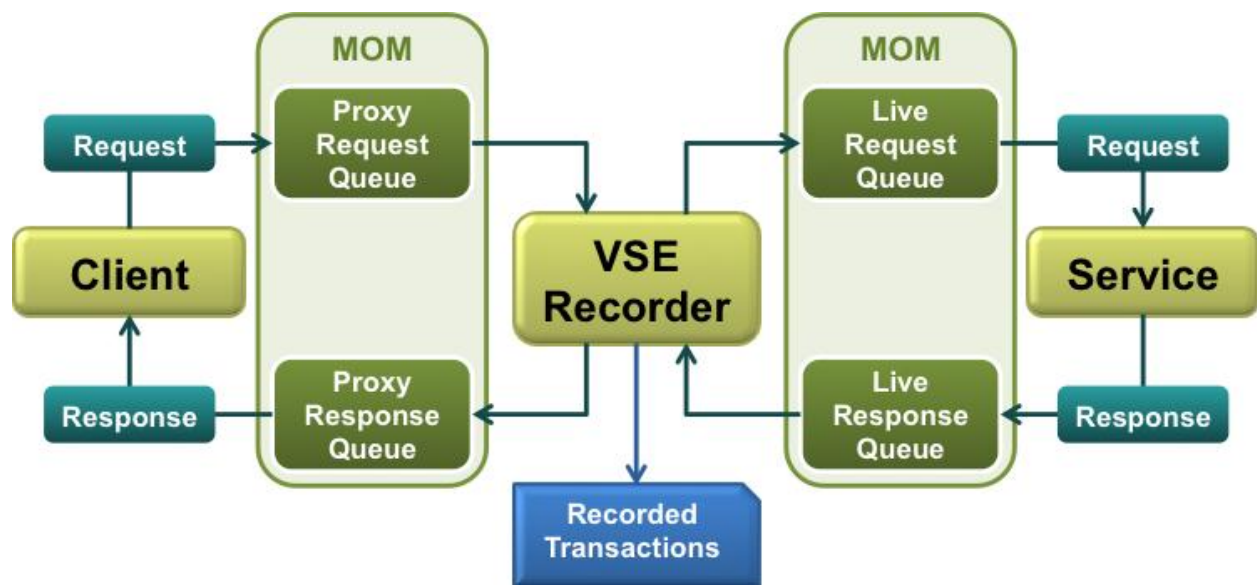
## Basic Proxy Recording

*Proxy recording* is a common method for recording a messaging application.

The following graphic shows the main components of proxy recording:

- The client
- The service
- The proxy queues
- The live queues
- The VSE recorder.

The message-oriented middleware (MOM) is the platform on which messages are exchanged.



The VSE recorder is inserted into the message flow between the client and the service. Each request message goes through the recorder before it reaches the service. Each response message goes through the recorder before it reaches the client.

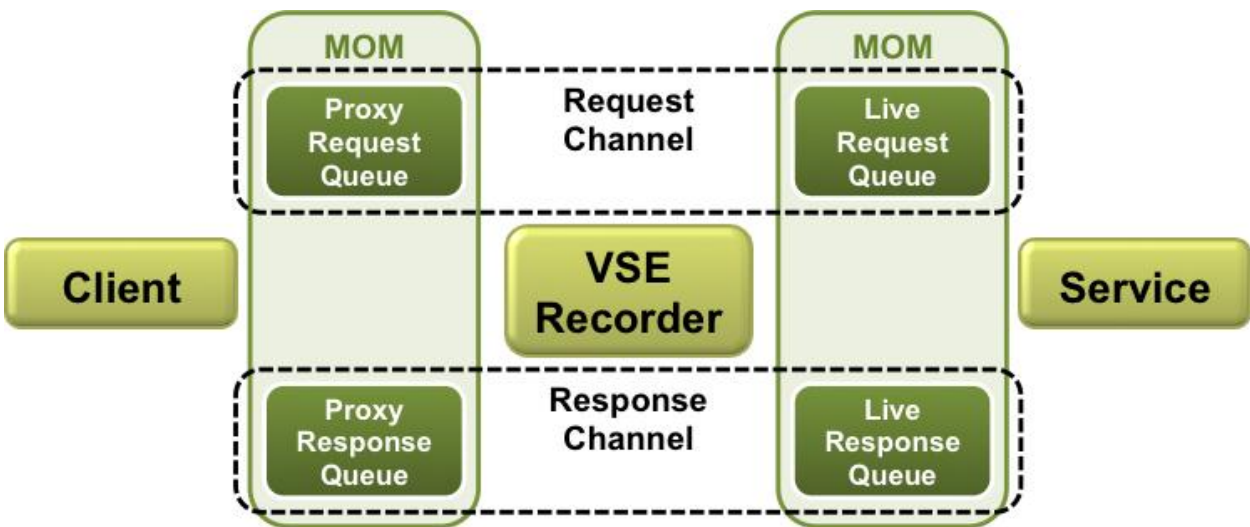
A *channel* is the pairing of a proxy queue and a live queue.

The proxy request queue and the live request queue form a request channel.

The proxy response queue and the live response queue form a response channel.

The following graphic shows the request channel and the response channel in a proxy recording.





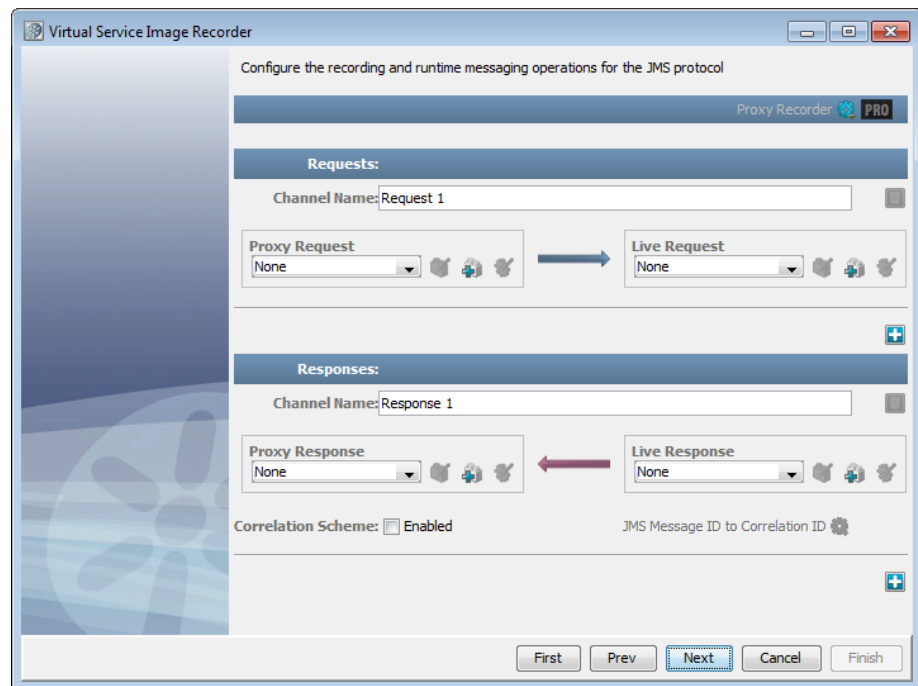
A proxy recording can have multiple request channels and multiple response channels.

In this procedure, you select an asset for each queue. An *asset* is a set of configuration properties that are grouped into a logical unit. For more information about assets, see *Using CA Application Test*.

**Follow these steps:**

1. In the Basics tab of the Virtual Service Image Recorder, set the transport protocol to JMS. Be sure to configure the service image and virtual service model.
2. Click Next.

The proxy recorder setup page opens.

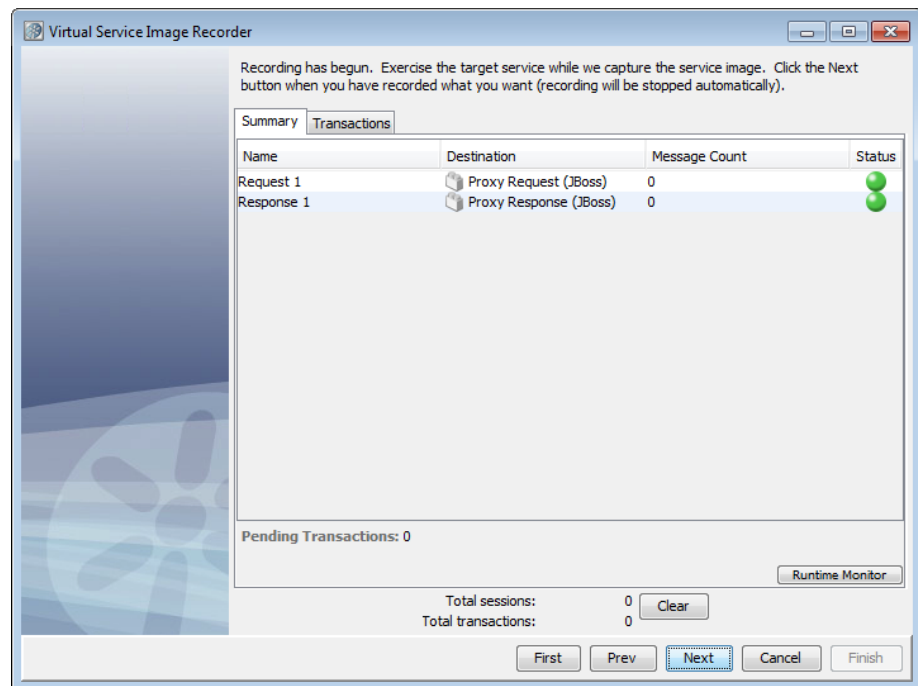


The proxy recorder setup page has basic and advanced parameters. To display the advanced parameters, click PRO at the top of the editor.

Each channel must have a unique name. This channel name is the means by which the virtual service model and the service image communicate about which queues are being used in the transaction.

3. Select the queue assets for the Proxy Request, Live Request, Proxy Response, and Live Response lists. If the assets have not been created, you can create them from this page. You can also edit assets from this page.
4. (Optional) Click the plus ( + ) icons to add request channels and response channels.
5. (Optional) Select the Correlation Scheme check box and specify which scheme you want. Each response channel can have a different correlation scheme. For a description of the correlation schemes, see the documentation for the JMS Send Receive step in *Using CA Application Test*.
6. Click Next to start the recording session.

The recording feedback page opens.



The table in the Summary tab lists each request channel and response channel. Each row includes the channel name, the selected destination asset, the number of messages that have flowed through the channel, and the status.

You can click a cell in the Destination column to see the proxy destination that is associated with the channel.

The bottom of the page shows the number of pending transactions, the number of sessions, and the total number of transactions.

If an error prevents the recording session from starting, a dialog opens and you return to the proxy recorder setup page.

Errors can also occur during the recording session. Errors that are associated with a specific request channel or response channel appear in the Status column. The status icon turns red, and a tooltip displays the error message. More general errors appear at the top of the recording feedback page.

You can use the run-time monitor to view the assets that are used in real time. For information about how the run-time monitor works, see the documentation for the JMS Send Receive step in *Using CA Application Test*.

7. Exercise the target service.
8. When the service completes, click Next to finish the recording.
9. Specify any data protocols and click Next.
10. (Optional) Save a recording session file.
11. Click Finish.

The [service image](#) (see page 304) and the [virtual service model](#) (see page 341) are created.

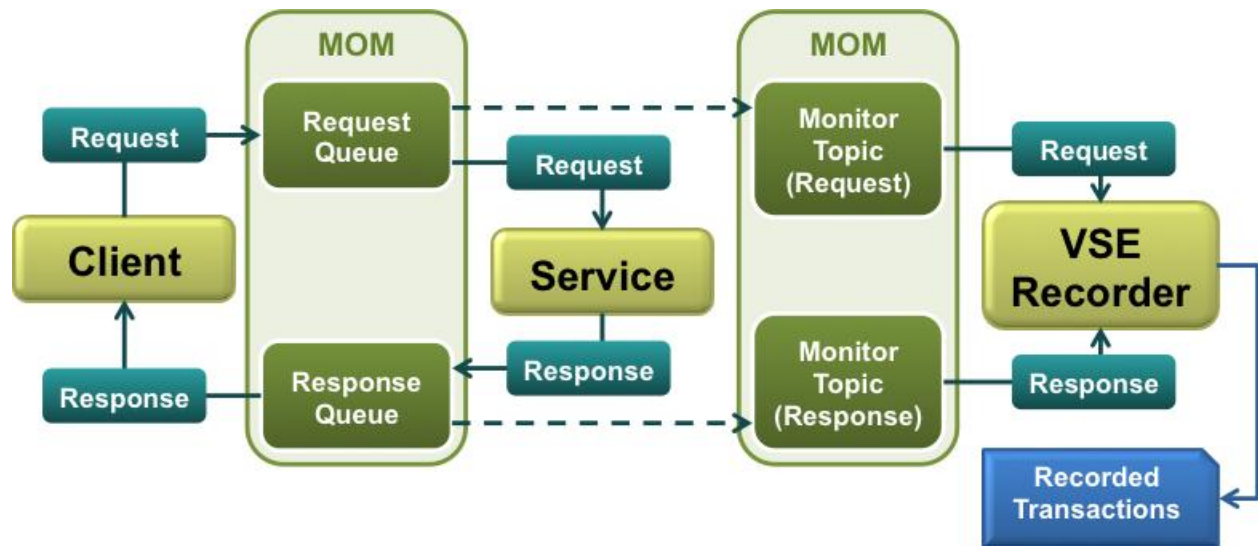
## TIBCO Monitor Recorder

With every queue on a TIBCO EMS server, a user with administrator privileges can connect to a special monitor topic. While subscribed to this monitor topic, an administrator can receive a copy of every message that is sent over the corresponding queue. This feature allows the VSE recorder to be outside of the message flow of the application entirely and simply monitor the message traffic.

The following graphic shows the main components of TIBCO Monitor recording:

- The client
- The service
- The request and response queues
- The request and response monitor topics
- The VSE recorder.

The message-oriented middleware (MOM) is the platform on which messages are exchanged.



### Monitor Topic (Request)

The TIBCO monitor topic that is associated with the request queue for the application. The TIBCO server forwards each message that is sent on the request queue to this topic automatically.

### Monitor Topic (Response)

The TIBCO monitor topic that is associated with the response queue for the application. The TIBCO server forwards each message that is sent on the response queue to this topic automatically.

The TIBCO Monitor Recorder supports TIBCO EMS 6.3.

You cannot use monitor topics during playback to virtualize the service. You can do either of the following actions:

- Create a separate set of queues and configure the client and VSE service to use them.
- Shut down the live service so the VSE service can replace it.

In the JMS connection asset, the admin user must have administrative privileges.

**Follow these steps:**

1. In the Basics tab of the Virtual Service Image Recorder, set the transport protocol to JMS. Be sure to configure the service image and virtual service model.

2. Click Next.

The proxy recorder setup page opens.

3. Change the recorder type from Proxy Recorder to TIBCO Monitor Recorder.

The proxy recorder setup page includes basic and advanced parameters. To display the advanced parameters, click PRO at the top of the editor.

4. Select the queue assets for the Receive Destination list and the Send Destination list. If the assets have not been created, you can create them from this page. You can also edit assets from this page.

5. (Optional) Click the plus ( + ) icons to add request queues and response queues.

6. (Optional) Select the Correlation Scheme check box and specify which scheme you want to use.

7. Click Next to start the recording session.

The recording feedback page opens.

The table in the Summary tab contains a list of each request channel and response channel. Each row includes the channel name, the selected destination asset, the number of messages that have flowed through the channel, and the status.

8. Exercise the target service.

9. When the service is complete, click Next to finish the recording.

10. Specify any data protocols and click Next.

11. (Optional) Save a recording session file.

12. Click Finish.

The service image and the virtual service model are created.


## Standard JMS

This topic contains the detailed instructions for recording a virtual service image using the Standard JMS transport protocol.

**Prerequisites:** Using DevTest with this application requires that you make one or more files available to DevTest. For more information, see *Third-Party File Requirements in Administering*.

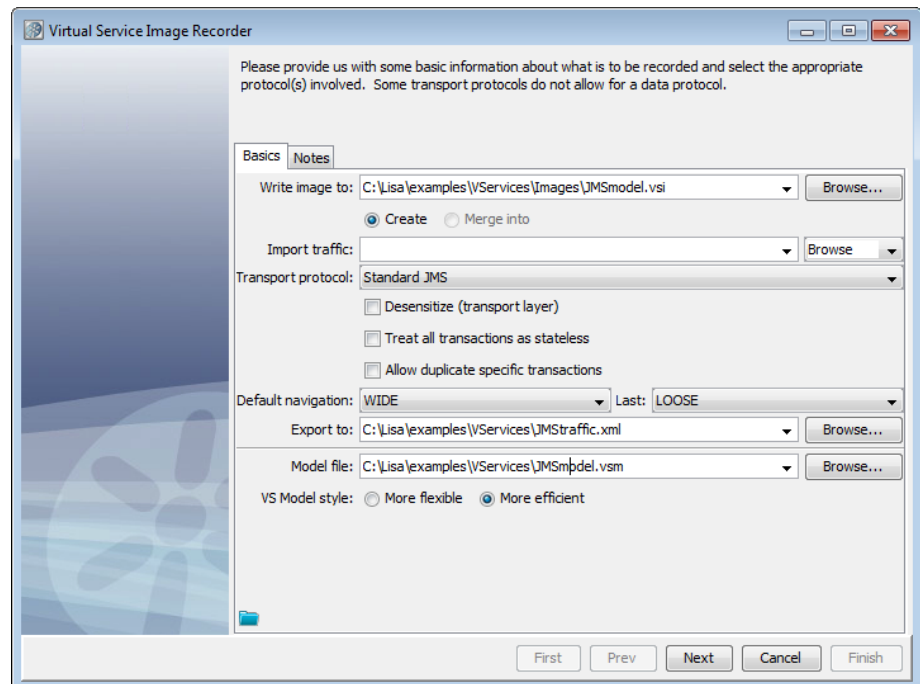
### Follow these steps:

1. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

2. Complete the [Basics](#) (see page 120) tab as in the following graphic:



3. Click Next.

The recording mode selection step opens with one of the following options selected:

### Proxy Mode

Proxy mode is the only true recording mode that VSE supports. Proxy mode provides the following options:

- Generate the service from the Live queues
- Generate the service using the Proxy queues

Proxy mode allows the use of proxy destinations to be set up on the message bus. The client application is configured to put messages on those proxy destinations and DevTest records them and forwards to the real destination. The same thing happens on the response side. DevTest is configured to listen on the real response destination, get the message, and forward it to the response proxy destination. This mode can behave differently if you enable temporary destinations, making the response side automated.

#### **Import Mode**

This mode for recording virtual services is available if you specify a raw traffic file in the Import traffic field on the initial recording window. In import mode, you can specify extra information about which request and response queues were detected in the raw traffic file. You can also select to skip the request response steps.

4. To record in proxy mode, complete the following fields:

#### **Generate service using the Live queues / Proxy queues**

Specifies whether to use the live queues or the proxy queues when generating the final virtual service model and image.

#### **Max Pending Transactions**

Defines the maximum number of running response listeners on each response queue. These response listeners run as long as a transaction is pending. When the number of pending transactions exceeds the maximum, the oldest transaction closes automatically. If you enter 0 in this field, there is no maximum.

#### **Disable Multiple Responses**

Specifies whether to support multiple responses in each transaction. By default, a transaction stays pending after the first response, waiting for more responses to arrive for the same transaction. When you select this option, the transaction automatically closes after the first response. If there are many transactions coming in quickly, this option can boost performance, especially for MQ.

5. To record in import mode, complete the following fields:

#### **Review the queues and transaction tracking mode**

Specifies whether to skip the request and response steps. If the raw traffic file comes from CAI, this option is cleared. CAI autodetects queues and transactions. If the raw traffic file does not come from CAI, this option is selected by default to let you review destination and tracking settings.

#### **Correlation**



Contains the possible schemes for correlating the request and response sides of individual transactions. JMS is asynchronous, which means the requests and responses are received separately. This drop-down list lets you define to the VSE Recorder which request to associate with which response. The Correlation field has the following options:

- Sequential: Associates every response with the last request received, chronologically. There is no correlation scheme, so the VSE MQ recorder acquires an exclusive read lock on the live response queue to ensure that another MQ listener cannot take response messages from it.

When you specify a correlation scheme, such as Correlation ID or Message ID to Correlation ID, the VSE MQ recorder can assume that any other listener on the live response queue will also use a correlation scheme. If all listeners on the live response queue use correlation schemes, the VSE MQ recorder can keep its responses separate without resorting to an exclusive read lock, so it opens the queue with the shared input flag.

- Correlation ID: The request and the response must have the same Correlation ID.
- Message ID to Correlation ID: The request Message ID must be the same as the response Correlation ID.
- Message ID: The request and the response have the same Message ID.

6. Click Next.

The Destination Info tab opens.

7. Enter your proxy and live destination names, and select the destination type.

8. Click the Connection setUp tab.

9. Enter the connection parameters used to connect to the MOM.

These connection parameters are internally saved.


The Advanced tab at the bottom of the Connection setUp tab lets you define custom connection properties for the service image.

10. Click Next.

The Destination List tab opens.

11. Define the connection details for the response destinations on which to listen for messages.

12. Define the proxy queue on which the client application will receive these responses.

- Remember that multiple responses are possible for a request. To register a set of response proxy queues, click Add .
- Select the Use for Unknown Responses check box before registering the response queue to use for unknown requests.

- To define a temporary destination, select the Use temporary queue/topic check box. Selecting this option disables the destination name and proxy destination name fields and marks the response listener you are adding as a temporary response. The destination name is blank.

A "temporary" destination in messaging is a destination that is created on demand for a messaging client. A temporary destination is typically used in request/response scenarios. DevTest supports using a temporary queue for the responses, but only supports one concurrent transaction with a temporary queue at a time.

13. Click the Current Connection Info tab and verify that the connection information is correct.

The information that appears on this tab is copied from the connection information that was given earlier. You could change it in a rare case when the response connection information is different.

14. Click Next to start recording.

The names of the destinations to which VSE is listening display.

The Pending Transactions field displays the number of transactions that are stored in the pending transaction buffer waiting for more responses. The transactions are closed either by reaching the maximum pending transaction count or by using the Disable Multiple Responses option. As the transactions close, they are moved from the pending transaction buffer to the total transaction buffer.

15. Run the client that adds messages to the request proxy queue.

VSE copies the messages to the real request queue. The server acquires those messages from there and sends responses to the response queue. Again, VSE acquires those messages and copies them to the response proxy queue, where the client is listens.


As the transactions are recorded, the message count increases and Total sessions and Total transactions counts increase on the Virtual Service Image Recorder window. At the end of recording, all the requests have gone through the same request queue. About half of the responses have returned through temporary queues, and the other half have returned through the nontemporary response queue.

When the run is complete, you may see the count of messages on the response queues one less than you expect. Because a single request can have multiple responses, VSE will not yet have recognized the last transactions to complete. Therefore, the messages corresponding to the last transaction are not counted.

Recommended data protocols are defaulted on this panel. You can add or edit both request and response side data protocols here. If there are configuration panels for the data protocols you have chosen, they open next. For more information, see [Using Data Protocols](#) (see page 209).

16. Click Next.

The last transaction closes and the necessary cleanup completes.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

17. Click Finish to close the window and store the image.
18. Review and save the VSM that is generated in the main window.

## Java


The DevTest Java Agent is required to record Java service images. In the LISA Bank application, the agent is installed by default. For any other application, install the agent and start the registry.

For testing, start the demo server, where the agent is installed, or run the LISA Bank application. For running with any other Java application, configure the agent. For more information about the agent, see *Agents*.

When using multiple recorders in the same workstation to record multiple EJB remote calls concurrently from different classes, the service image and VSM created by each recorder display an aggregated list of classes.

### Follow these steps:

1. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, By recording.

The Virtual Service Image Recorder opens.

2. Select Java as the transport protocol on the [Basics](#) (see page 120) tab.
3. Click Next.

The Select Java classes to virtualize window opens.

4. To move selected agents between the following columns, use the provided arrows:

#### Available Online Agents


Lists the online agents that are available to be connected.

#### Connected Agents

Lists contains the online or offline agents that are connected for the virtual service model. The offline agents display in a gray italic font. You can select the agents from the Available Online Agents list.


When you select an agent from either list, the host name and the main class appear.

5. To add an agent that is not in the list, type the agent name in the field above the list

of connected agents and click Add Agent .

This mechanism is provided for adding any offline agents that were not previously in the connected list. The agent name cannot be empty or already present in the connected agents list. If the agent name entered is present in the online agents list, it is moved from the available online agents to the connected agents list.

6. To select a class by searching, complete the following steps:

- a. Double-click Search for Classes.
  - b. Type a search string as a fully qualified name (including the package), using regular expressions.
  - c. Click the search icon.
  - d. Select a class. Some classes could show up more than once. Select a class only once.
  - e. Click the right arrow.
7. To select a class by manually entering the name, complete the following steps:
  - a. Double-click Manually Enter a Class Name.
  - b. Type the fully qualified class name.
  - c. Click the right arrow.
8. To select a class from a list of classes that the agent suggests, complete the following steps:
  - a. Double-click Agent Suggestions.
  - b. Click Retrieve .
  - c. Select a class.
  - d. Click the right arrow.
9. To add a protocol to the recording, double-click Protocols and select the protocol to record.
10. Shut down the system under test.
11. Click Next.


The Recording Has Begun page opens.
12. Restart the system under test.

**Note:** Shutting down and restarting the system under test is optional. Restarting helps to ensure that VSE captures any initial traffic that occurs when the system under test initializes.

After the system under test starts, a list of the most recent transactions displays. To see the content of a selected transaction, double-click it.
13. Click Next when the recording completes.

The Data Protocols panel opens with appropriate data protocols prepopulated. You can change or add data protocols. There may be additional panels to configure the data protocols. For more information about configuring data protocols, see [Using Data Protocols](#) (see page 209).
14. When you have configured the data protocols, click Next.

While it prepared to write the .vsi file, the recorder verifies request and response bodies to ensure that (if they are marked as text) they are text. If they are not text, the type is switched to binary.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

15. Click Finish.

## JDBC


This topic contains the detailed instructions for recording a virtual service image using the JDBC (Driver Based) transport protocol.

**Note:** The JDBC (Driver Based) transport protocol is scheduled for end of life in a future release. This functionality is replaced by agent-based JDBC virtualization.

For more information about prerequisites and preparatory steps, see [Preparing to Virtualize JDBC](#) (see page 159).

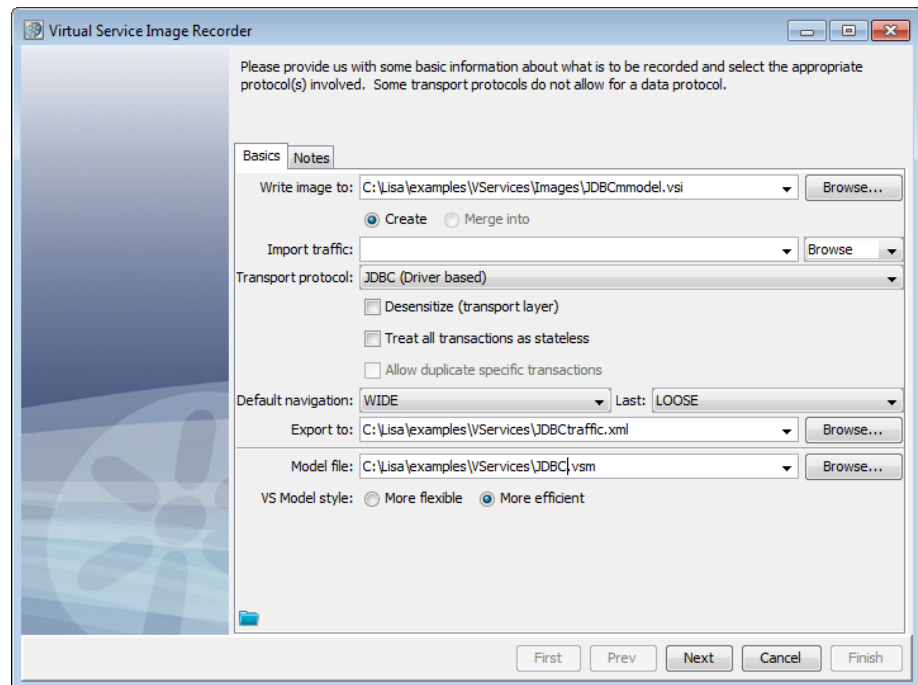
### Follow these steps:

- To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

- Complete the [Basics](#) (see page 120) tab as in the following graphic:



- Click Next.

The endpoint configuration window opens.

- Set up the simulation host and range of ports (default is 2999), as appropriate.

JDBC VSE supports multiple endpoints during recording and playback. In the recorder and the JDBC listen step editor, there is a table of Driver Host, Base Port, and Max Port. When Base Port and Max Port differ, a unique endpoint is created for the base port and the max port and each port between.

5. Click Next to connect to the JDBC Simulation server.

To stop trying to connect, click Cancel in the status window.

6. Select the connection URLs and users to be recorded in the SQL Activity tab.

The SQL Activity tab shows the ten most recent statements executed (and how many times they ran) for every unique connection URL and database user combination. This display refreshes approximately every five seconds.

- To put the selected connection string and user in the URL and user fields at the bottom of the panel, click To URL.
- To put the URL in the URLs to Record list, click Add.

7. Click the Loaded Drivers tab.

The Loaded Drivers tab lists the JDBC drivers that are installed and registered in the system under test.

- Select a driver and click To URL to put a regular expression that matches any connection through that driver in the URL field at the bottom of the panel.
- Click Add to add the pattern to the URLs to Record list.

#### URLs to Record

Lists the URLs added from the SQL Activity list or the URL/User entry fields. To delete an entry, select it and click Remove.

#### User

An unnamed area below the URLs to Record field where you can include a database user with the URL. If you leave this field blank, the recording applies to all users connecting to the URL.

**Note:** To enable the Add button, supply a connection URL in the first field and a user name in the next field.

The bottom section shows the list of connection URL and database user combinations that are recorded. The URL can be a regular expression. This list is typically initially empty unless the state attribute is set to RECORD on a simulation connection URL for DataSource style systems under test.

If a driver is selected from the drivers list, you can use the To URL button to copy a generic regular expression that matches the connection URLs for that driver to the URL entry field. If the connection URL (top level) node in the activity tree is selected, you can use the To URL button to copy the URL and user to the URL and user entry fields. Also, to add the connection URL and user directly to the recording list, use the Add button to the right of the activity tree.



Connection URLs can be either exact or a regular expression that matches connection requests and can be added to the recording list with or without a database user. The absence of a database user matches any user attempting to connect. If the Add button to the right of the URL and user fields is disabled, the recording list probably already covers the URL and user combination.

8. Click Next when the recording list contains all the URLs and users to record.

The Recording has begun window opens.

The options and dynamic display statistics include:

#### **Total conversations**

Displays the number of conversations recorded.

#### **Total transactions**

Displays the number of transactions recorded.

#### **Clear**

Clears the list of currently recorded transactions.

The number of Total conversations and Total transactions increases with the number of transactions recorded.

**Note:** If no transactions are recorded, you could have a port conflict. The client sends transactions to the application instead of the Virtual Service Recorder. If another service is using that port, either stop that service or change the port setting so there is no longer a conflict.

If you have performance issues, it is possible that the target database is responding slowly. Check the `user.home\lisatmp\tmanager.log` file for debug messages that report the query execution time.

For example:

```
2009-07-01 15:35:39,248 [AWT-EventQueue-0] DEBUG
com.itko.lisa.vse.stateful.model.SqlTimer - Exec time 72ms :
SELECT TRAFFIC_ID, LAST_MODIFIED, SERVICE_INFO, CREATED_ON,
NOTES, UNK_CONV_RESPONSE, UNK_STLS_RESPONSE FROM SVSE_TRAFFIC
```


If the query time exceeds a threshold, warning messages like the following are generated:

```
2009-07-01 15:17:11,161 [AWT-EventQueue-0] WARN
com.itko.lisa.vse.stateful.model.SqlTimer - SQL query took a long
time (112 ms) : SELECT TRAFFIC_ID, LAST_MODIFIED, SERVICE_INFO,
CREATED_ON, NOTES, UNK_CONV_RESPONSE, UNK_STLS_RESPONSE FROM
SVSE_TRAFFIC
```

9. Click Next when the recording is complete.

The recorder prepares to write the .vsi file by verifying request and response bodies. The recorder ensures that, if they are marked as text, that they are text. If they are not text, the type is switched to binary.

The recorder finishes postprocessing the recording.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

10. Click Finish to store the image.
11. Review and save the virtual service model in DevTest Workstation.

## Virtualize JDBC

This section provides more detail about how to prepare to use the VSE framework to virtualize JDBC-based database traffic. Information about JDBC configuration is also available in [Install the Database Simulator](#) (see page 21). This section documents the various ways that driver-based JDBC virtualization can be combined, and the configuration options supported. This section does not include any detail about how to implement the configuration on any specific app container.

**Note:** JDBC virtualization only works with Java 1.6; it does not work with Java 1.5.

### System-Level Properties

You can specify the following properties in one of the following ways:

- By using system variables (adding a **-Dvar=val** flag to the startup for the system under test, or through another mechanism)
- In a properties file named **rules.properties**, which has to be on the classpath (probably in the same directory as **lisajdbcsim.jar**).

If a property exists in both locations, the system property overrides the configuration file.

#### **lisa.jdbc.sim.require.remote**

Determines whether the driver waits on a connection to VSE before processing commands.

**Default:** false

#### **lisa.jdbc.hijack.drivermanager**

Determines whether the driver attempts to take over all the database connections from this application.

**Default:** false

#### **lisa.jdbc.sim.port**

Defines the default listen port.

**Default:** 2999

#### **lisa.log.level**

Specifies the default log level.

**Values:**

- OFF
- FATAL
- ERROR

- WARN
- INFO
- DEBUG
- DEV

**Default:** WARN

**`lisa.log.target`**

Specifies where the logs are written.

**Values:**

- stderr
- stdout
- any file location

**Default:** stderr

**VSE Driver (stand-alone)**

The system under test can use the VSE driver directly or the VSE driver can be wrapped in a data source such as the Apache BasicDataSource. In such cases, the configuration (other than the username and password) is passed through the URL. The driver properties include:

**URL**

Specifies the actual URL that is passed to the underlying driver. This URL must be the last element passed.

**State**

Specifies the initial state for the driver.

**Values:**

- watch
- record
- playback

**JdbcSimPort**

Designates the port on which to listen for connections from VSE. This value overrides the system-level property.

**Driver**

Designates the class name of the real driver to use.

**User**

## Password

Specify the classname of the driver as **com.itko.lisa.vse.jdbc.driver.Driver**.

The following example shows the format of the URL. Everything is optional except the driver and the URL.

```
jdbc:lisasim:driver=<real driver class>;state=<initial state>;jdbcSimPort=<starting port>;url=<real url>
```

**Note:** The URL element must come last. Everything following "url=" is passed unchanged to the underlying driver. DevTest supports passing extra information to the underlying driver by embedding it in the real URL.

For example, the following URL is appropriate for a Derby connection:

```
jdbc:lisasim:driver=org.apache.derby.jdbc.ClientDriver;state=watch;jdbcSimPort=4000;url=jdbc:derby://localhost:1527/sample;create=true
```

This URL instructs the VSE driver to:

- Use the Derby client driver
- Start up in "watch" mode
- Use port 4000 to communicate with VSE
- Pass the URL jdbc:derby://localhost:1527/sample;create=true to the real Derby driver

## VSE Datasource Wrapping a Driver

VSE Datasource Wrapping a Driver could be a more typical configuration for app containers than using a VSE driver directly. VSE provides a data source that wraps a real driver. Most application containers provide a mechanism for directly specifying properties for the data source.

Specify the classname of the driver as **com.itko.lisa.vse.jdbc.driver.VSEDataSource**.

## Datasource Properties

### Driver

Specifies the name of the real driver to use (the normal configuration).

### URL

Specifies the real URL to connect with (cannot start with **jdbc:lisasim**).

**User**

**Password**

**JdbcSimPort**

Specifies the port on which to listen for connections from VSE. This value overrides the system-level property.

**CustomProperties**

Defines a semicolon-separated list of name=value pairs. These properties are parsed and delegated to the wrapped data source, if any. If the first character is not alphanumeric, it is used as the delimiter instead of a semicolon.

**VSE Data source Wrapping a Datasource**

Wrapping a data source is an unusual configuration, when an application container does not permit a driver to be instantiated directly with `Class.forName()`. To use it, instead of specifying **driver=<classname of real driver>** (for example, **oracle.jdbc.OracleDriver**), specify **datasource=<classname of real datasource>** (for example, **oracle.jdbc.pool.OracleDataSource**).

Specify the class name for the driver as **com.itko.lisa.vse.jdbc.driver.VSEDataSource**.

**Supporting Multiple Endpoints**

If one application has multiple JDBC connections you want to virtualize (or if one appserver has multiple applications with different JDBC connections you want to virtualize), specify a different **JdbcSimPort** for each connection. You can specify this endpoint either as a property to the VSE Data source, or as a parameter on the URL for the driver.

**Example:**

Assume an application uses two JDBC connections, and you want to virtualize both of them. Also assume the application uses drivers directly.

Specify **com.itko.lisa.vse.jdbc.driver.Driver** for each connection, with the appropriate underlying parameters such as driver and URL for each. For the first connection, you could specify **jdbcSimPort=3000** and for the second you could use **jdbcSimPort=3001**. Your configuration could look like:

```
connection1.url=jdbc:lisasim:driver=org.apache.derby.jdbc.ClientDriver;jdbcSimPort=3000;url=jdbc:derby://localhost:1527/sample;create=true
```

```
connection2.url=jdbc:lisasim:driver=org.apache.derby.jdbc.ClientDriver;jdbcSimPort=3001;url=jdbc:derby://localhost:1527/sample2;create=true
```

Although you must still record separately, you can deploy both services simultaneously, and can start and stop them independently.


## TCP

This topic contains the detailed instructions for recording a virtual service image using the TCP transport protocol.

For more information about prerequisites and preparatory steps, see [Preparing to Virtualize TCP](#) (see page 167).

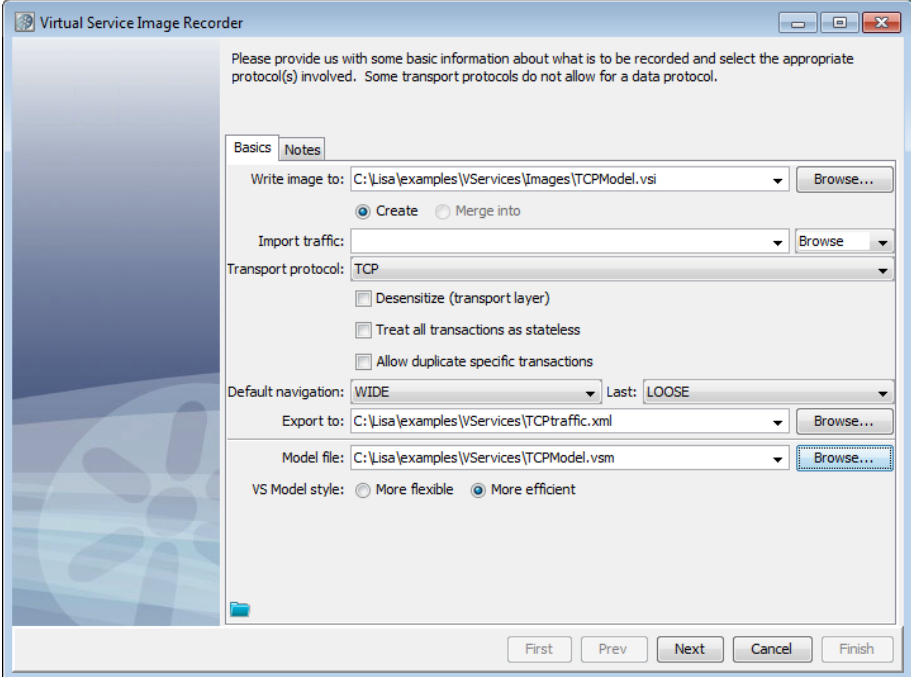
**Follow these steps:**

1. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

2. Complete the [Basics](#) (see page 120) tab as in the following graphic:



The screenshot shows the 'Virtual Service Image Recorder' dialog box with the 'Basics' tab selected. The dialog contains the following fields and options:

- Write image to:** C:\Lisa\examples\VSservices\Images\TCPModel.vsi (with a 'Browse...' button)
- Options:** ☒ Create, ☐ Merge into
- Import traffic:** (empty dropdown with a 'Browse...' button)
- Transport protocol:** TCP (dropdown menu)
- Checkboxes:** ☐ Desensitize (transport layer), ☐ Treat all transactions as stateless, ☐ Allow duplicate specific transactions
- Default navigation:** WIDE (dropdown), **Last:** LOOSE (dropdown)
- Export to:** C:\Lisa\examples\VSservices\TCPtraffic.xml (with a 'Browse...' button)
- Model file:** C:\Lisa\examples\VSservices\TCPModel.vsm (with a 'Browse...' button)
- VS Model style:** ☐ More flexible, ☒ More efficient

At the bottom are navigation buttons: First, Prev, Next, Cancel, and Finish.

3. Click Next.
4. Enter the information for both the client and the server, select your delimiters, and add SSL parameters.

**Listen/Record on port**

Defines the port on which the client communicates to DevTest.

**Target host**



Defines the name or IP address of the target host where the server runs.

**Target port**

Defines the port number of the target host where the server runs.

**Treat request as text**

Specifies whether the request is treated as text. For more information, see [Preparing to Virtualize TCP](#) (see page 167).

**Request Encoding**

Lists the available request encodings on the machine where DevTest Workstation is running. The default is UTF8.

**Treat response as text**

Specifies whether the response is treated as text. For more information, see [Preparing to Virtualize TCP](#) (see page 167).

**Response Encoding**

Lists the available response encodings on the machine where DevTest Workstation is running. The default is UTF8.

**Use SSL to server**

Specifies whether DevTest uses HTTPS to send the request to the server.

- **Selected:** DevTest sends an HTTPS (secured layer) request to the server.

If you select Use SSL to server, but you do not select Use SSL to client, DevTest uses an HTTP connection for recording. DevTest then sends those requests to the server using HTTPS.

**Cleared:** DevTest sends an HTTP request to the server.

**Use SSL to client**

Specifies whether to use a custom keystore to play back an SSL request from a client. This option is only enabled when Use SSL to server is selected.

**Values:**

- **Selected:** You can specify a custom client keystore and a passphrase. If these parameters are entered, they are used instead of the hard-coded defaults.

**Cleared:** You cannot specify a custom client keystore and a passphrase.

**SSL keystore file**

Specifies the name of the keystore file.

**Keystore password**


Specifies the password associated with the specified keystore file.

5. Click Next to start recording.
6. When your recording is complete, click Next.

7. Select a response data protocol if the response is encrypted, compressed, or otherwise encoded.
8. The recorder attempts to detect message delimiters that tell DevTest when it has read a complete request or response. Confirm, or correct, these delimiters on this screen.

**Note:** A Request delimiter is mandatory. A Response delimiter must be selected for Live Invocation to be available.

9. The recorder verifies request and response bodies to ensure that, if they are marked as text, that they are text. If they are not, the type is switched to binary.

**Note:** To save the settings on this recording to load into another service image recording, click Save  above the Finish button.

The TCP/IP protocol supports a live invocation step. To enable the live invocation step, select a response protocol. If you do not select a response protocol, no live invocation step is included in the VSM.

## Virtualize TCP

The TCP transport protocol lets you virtualize raw TCP/IP traffic between one or more clients and a server. This transport resembles the HTTP protocol, and uses many of the same features.

During recording, you configure the TCP protocol exactly as you would configure HTTP in Endpoint mode. You give it a socket on which to listen, the target server, and the destination. The protocol forwards data between the client and server, and "listens in" on that data to create VSE transactions.

### Converting Bytes to VSE Transactions

It can be difficult to know what comprises a complete request or response, and how to correlate those requests and responses. The TCP data has no inherent structure. A request delimiter is required to identify requests in the incoming data stream. You can also select a response delimiter, although a response delimiter is not strictly required.

The recorder uses the following rules to determine requests and responses, and to correlate requests and responses together into transactions:

1. Zero or one responses follow a request.
2. Each response is paired with the latest complete request.
3. A request is considered complete when response data arrives, even if the request delimiter says it is not complete.
4. A response is considered complete when request data arrives, even if the response delimiter (if any) says it is not complete.
5. If the response delimiter finds a complete response when response data is received, any extra response data is discarded.

For example:.

Request 1 - Request 2 - Response A - Partial Request 3 - Response B  
- Request 4 - Partial Response C - Request 5 - Response D With Trailing  
Data

This data is parsed into transactions as follows:

- Request 1 with no response
- Request 2 with Response A (from rule 2)
- Partial Request 3 with Response B (from rule 3)
- Request 4 with Partial Response C (from rule 4)
- Request 5 with Response D, but with the "trailing data" dropped (from rule 5)

This system works if the client and server follow a synchronous request/response paradigm, and it handles requests with no responses. The system does not handle asynchronous communication, or one request with multiple responses.

### Creating Conversations

By default, it records TCP service images, VSE creates conversations based on your machine name (or IP address) and the outbound socket. Any time your outbound socket changes, a new conversation is started. To change this behavior, select the Treat all conversations as stateless check box.

### Out-of-the-Box Delimiters

DevTest has the following delimiters:

#### Records are terminated with line endings

One or more newline characters terminates each request/response. The delimiter itself is not included in the request/response.

##### Values:

- \n
- \r
- Unicode characters 0085, 2028, and 2029

#### Records are of fixed length

Each request/response is a specific number of bytes in length.

#### Records are delimited by specific characters

A specific character or sequence of characters, such as a comma, terminates each request/response. The entire sequence must be matched. The delimiter itself is not included in the request/response. This option is similar to importing a CSV file.

#### Delimiters match a regular expression

A set of bytes that matches the regular expression terminates each request/response. The delimiter itself is not included in the request/response.

If the delimiter characters are not included in the request/response (as in most cases), back-to-back sets of delimiters are ignored.

**Note:** You cannot edit the selected delimiter after the recording is done. You cannot change the selected delimiter in the model editor.

### Treat Request as Text and Treat Response as Text

The Treat Request as Text and Treat Response as Text check boxes control how the request and response data is parsed.

#### Treat Request as Text

Specifies whether to use the request as the body of the VSE request.

##### Values:

- **Selected:** Sets the entire request as the body text of the VSE request. The first "word" (up to the first space character) is used as the operation name.
- **Cleared:** Leaves the request body as binary and sets the operation name to OPERATION.

#### Treat Request as Text

Specifies whether to use the request as the body of the VSE request.

##### Values:

- **Selected:** Treats the response body as text and converts any embedded nulls (\0) to spaces.
- **Cleared:** Leaves the response body as binary.

### Why is the Transaction Count Incorrect?

The transaction count always lags by at least one. When the TCP protocol finds a complete transaction, it caches that transaction until the next transaction starts, because the server may intentionally close the socket after responding. If this happens, the TCP protocol detects it and sets **lisa.vse.close\_socket\_after\_response=TRUE** on the response. This property causes VSE to close the server-side socket after sending that response during playback.

If you do not use a response delimiter, the transaction count lags by another transaction. The TCP protocol does not know the response is finished until a new request starts. So, the total count can be off by two. The count is correct when the recording finishes.

## Record CICS (LINK DTP MRO)

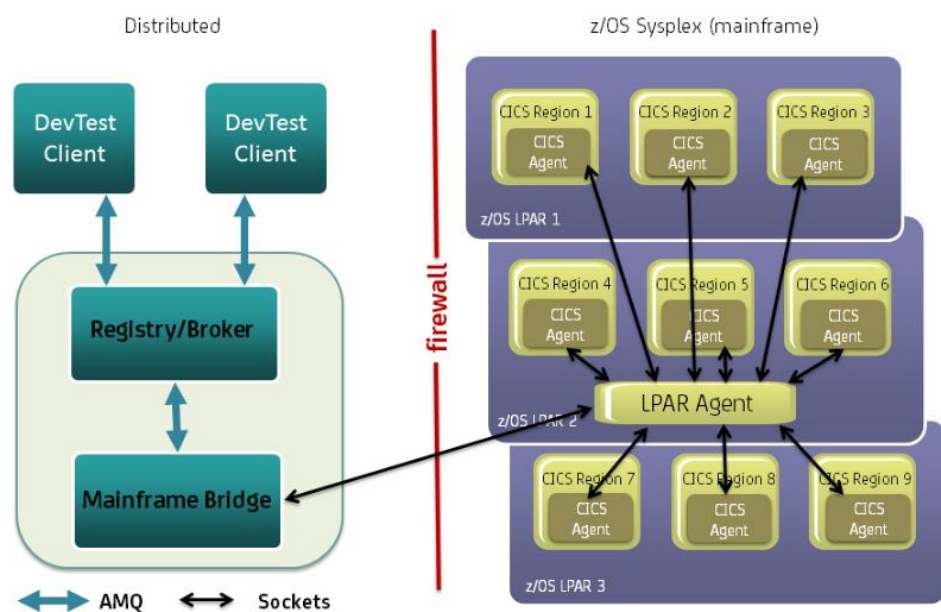
This section contains the detailed instructions for recording a virtual service image using the CICS transport protocol. The section contains the following topics:

- [Recording IBM CICS Overview](#) (see page 170)
- [Recording IBM CICS Example](#) (see page 171)
- [Using the CICS Programs to Virtualize Panel](#) (see page 179)

## IBM CICS Overview

To allow recording IBM CICS service images, DevTest has a mainframe bridge that is a part of the registry. That mainframe bridge supplies a single point of contact for all of the DevTest Workstations to the mainframe. The mainframe has an LPAR agent that runs on a z/OS LPAR. The LPAR agent provides a single point of contact to all of the CICS agents.

Each CICS region has a DevTest CICS agent that provides the virtualization capability. The CICS agents communicate with the LPAR agent over sockets, and the LPAR agent communicates with the mainframe bridge over sockets. The mainframe bridge provides an ActiveMQ interface through the registry to all of the DevTest clients.



The mainframe bridge can run in the following modes:

### Client mode

The mainframe bridge initiates the connection to the LPAR agent.

### Server mode

The mainframe bridge waits for the LPAR agent to initiate the connection.

The **lisa.properties** file contains properties to enable the mainframe bridge. For more information, see "Mainframe Properties".

## IBM CICS Example

This example uses the VSE recorder to record a CICS LINK service image.

When the registry is started, it shows that the mainframe bridge has started in client mode and the bridge is connected to the LPAR agent.

This example demonstrates eliminating a resource dependency. In the example, a CICS application requires a VSAM file for input to fields for account balance information, but the VSAM file is unavailable. This example shows how to virtualize the program that maintains the missing VSAM file, eliminating the resource dependency.

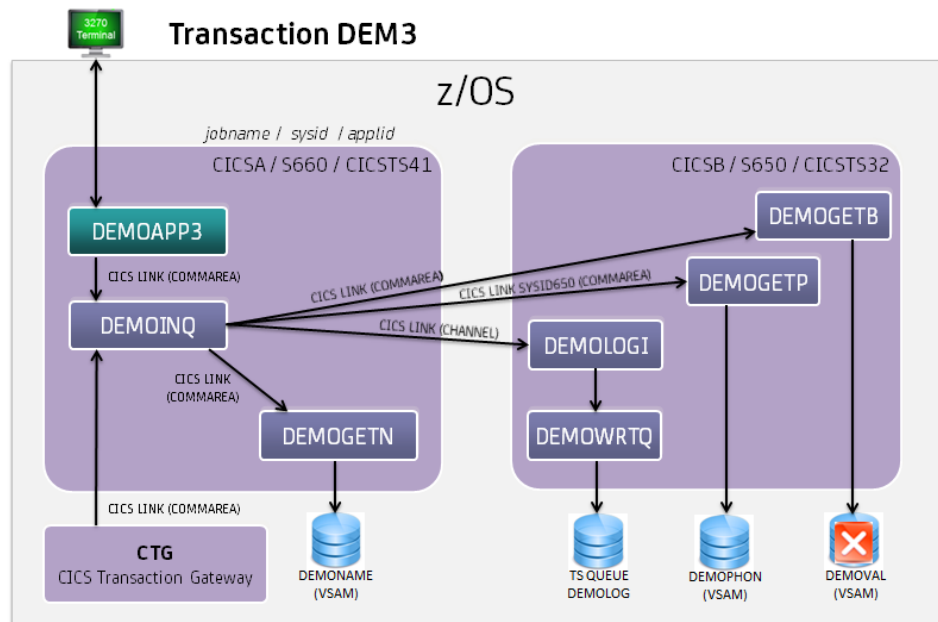
The following graphic shows the CICS transaction (DEM3) that is running. The transaction runs from a 3270 terminal, but it could run from the CICS Transaction Gateway.

Running the transaction initiates DEMOAPP3, which performs a CICS LINK to DEMOINQ. DEMOINQ then runs the following CICS LINKs:

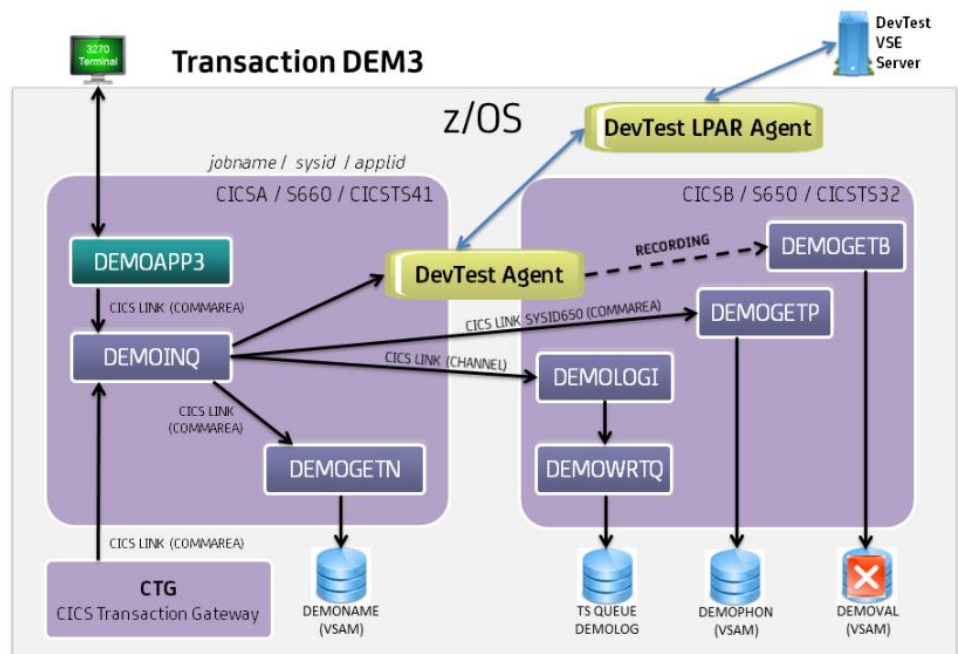
1. A local LINK passes a COMMAREA to DEMOGETN to get the customer name and address from a VSAM.
2. A CICS LINK to CICSB passes a COMMAREA to DEMOGETB. DEMOGETB reads the VSAM file DEMOGETBAL, which is not available in the example.
3. A CICS LINK passes a COMMAREA to DEMOGETP to get the telephone number information.
4. A CICS LINK passes a CHANNEL with CONTAINERS to DEMOLOGI.

The following CICS regions are involved:

- Jobname CICS A, with a system ID of S660 and an applid of CICSTS41
- Jobname CICS B, with a system ID of S650 and an applid of CICSTS32



This example virtualizes DEMOGETB on CICSTS41. The example uses the DevTest agent, running in CICS, communicating with the LPAR agent, communicating with the VSE server. We first run the CICS transaction to record the CICS LINK to DEMOGETB. We then use that recording to virtualize the CICS LINK.

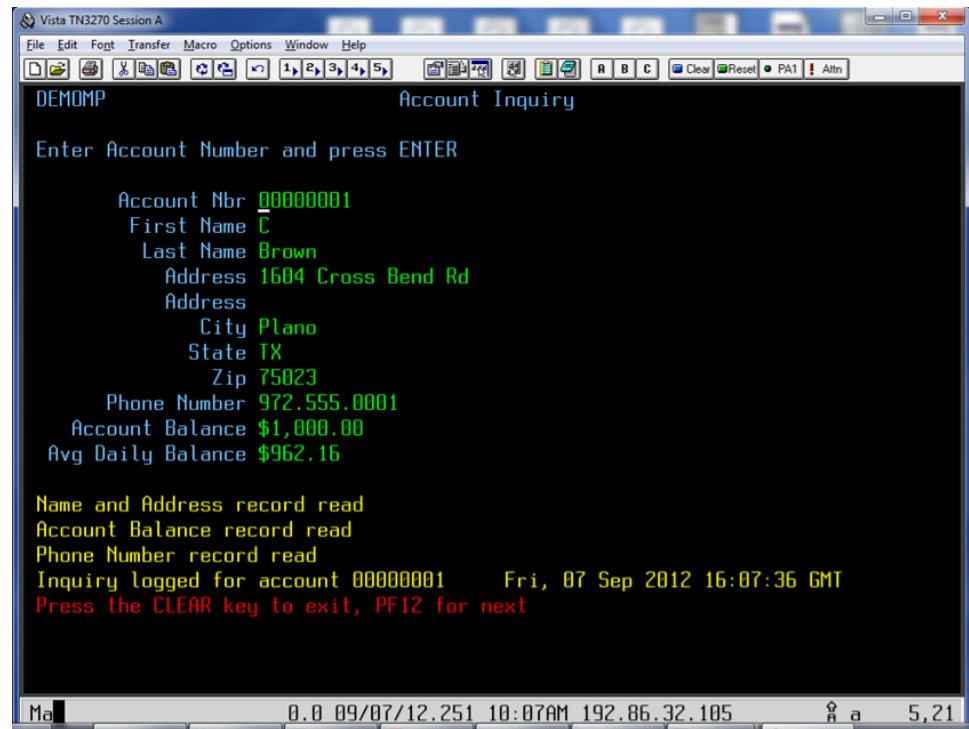




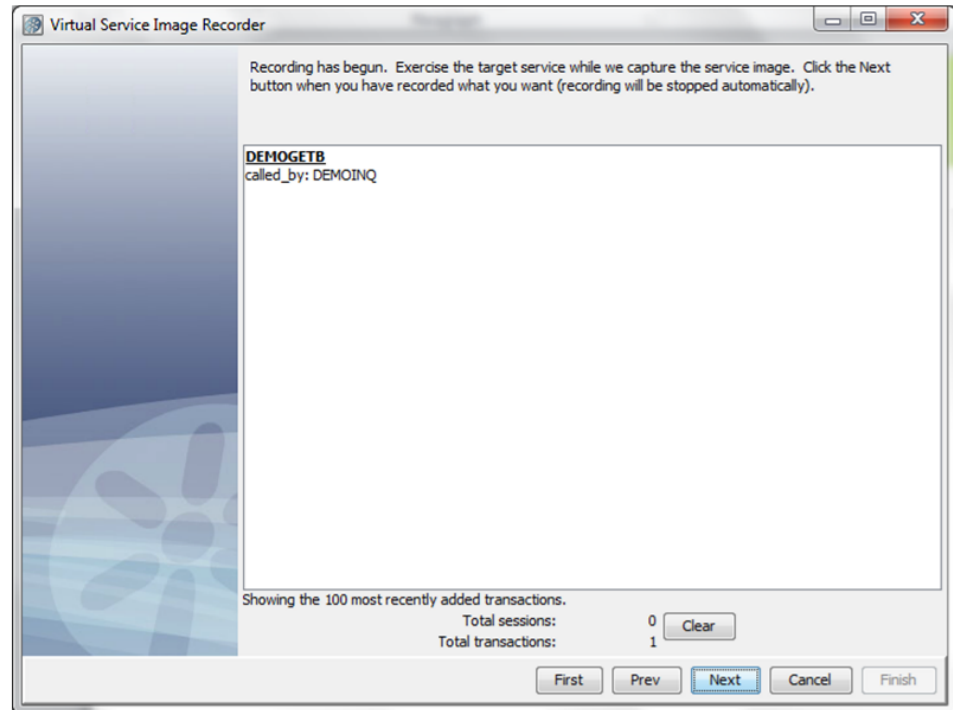
The Virtual Service Image Recorder Basics tab contains the name of the service image (DEMOGETB.vsi), the VSM (DEMOGETB.vsm), and the transport protocol of CICS LINK.

The CICS Programs to Virtualize panel displays next. For detailed information about this panel, see [Using the CICS Programs to Virtualize Panel](#) (see page 179).

Recording starts when the Next button is clicked. During recording, we run transaction DEM3 to invoke DEMOGETB with a CICS LINK.



As the following graphic shows, after it runs once, VSE records one transaction.



On the next panel, VSE adds the CICS Copybook data protocol to both the request and response sides.

The following graphic shows the copybook mapping XML file to use in this example. This file defines the mapping for DEMOGETB and it indicates to use DEMOGETB.txt as the copybook if a request for the program DEMOGETB exists.

### payload-copybook-mapping-CICS.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<payloads>
  <!-- DEMOGETB -->
  <payload name="DEMOGETB" type="request" matchType="metaData"
    key="DEMOGETB" value="DEMOGETB" >
    <section name="Body">
      <copybook key="">DEMOGETB.txt</copybook>
    </section>
  </payload>
  <payload name="DEMOGETB" type="response" key="DEMOGETB" value="DEMOGETB">
    <section name="Body">
      <copybook key="">DEMOGETB.txt</copybook>
    </section>
  </payload>
</payloads>
```

As the following example shows, DEMOGETB.txt contains the COBOL copybook definition:

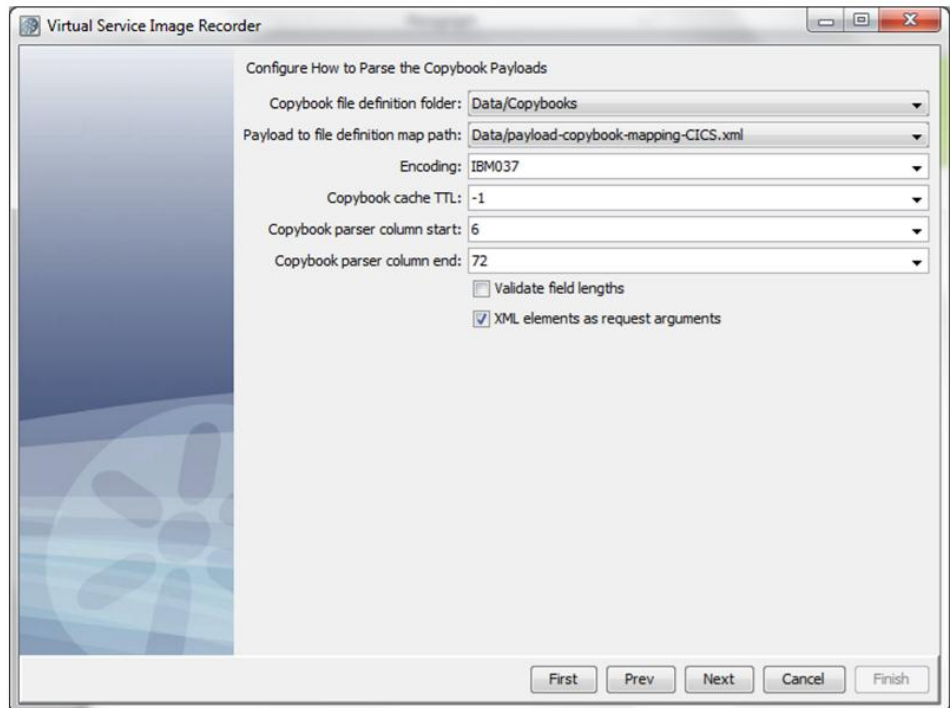
Copybooks/DEMOGETB.txt

```

05 DEMOGETB-COMMAREA.
    10 DEMOGETB-RETURN          PIC X.
    10 DEMOGETB-MESSAGE        PIC X(70).
    10 BALANCE-REC.
        15 BALANCE-ACCOUNT-NBR  PIC X(8).
        15 FILLER                PIC X.
        15 BALANCE-BALANCE      PIC X(14).
        15 FILLER                PIC X.
        15 BALANCE-AVERAGE     PIC X(14).

```

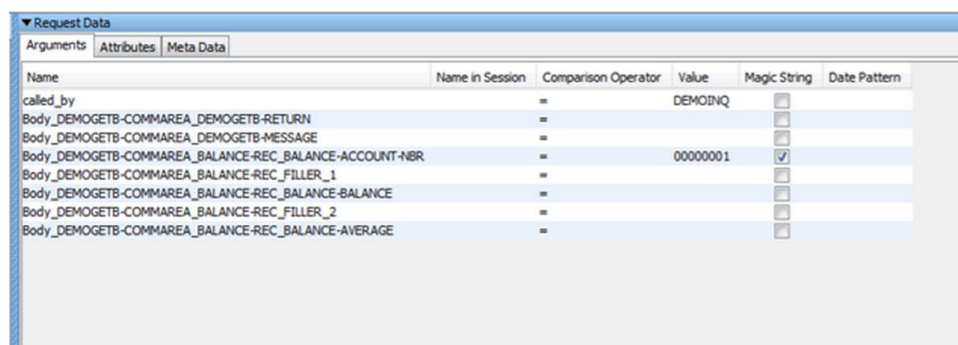
On the next panel, the folder that has DEMOGETB.txt is specified, and the mapping XML. The codepage is changed to IBM037.



Click Next. The copybook processes the data.

Click Next to stop processing and complete the service image recording.

When you open the VSI in the Service Image Editor, you can see that the copybook mapped the request data, then converted it to attributes. For example:

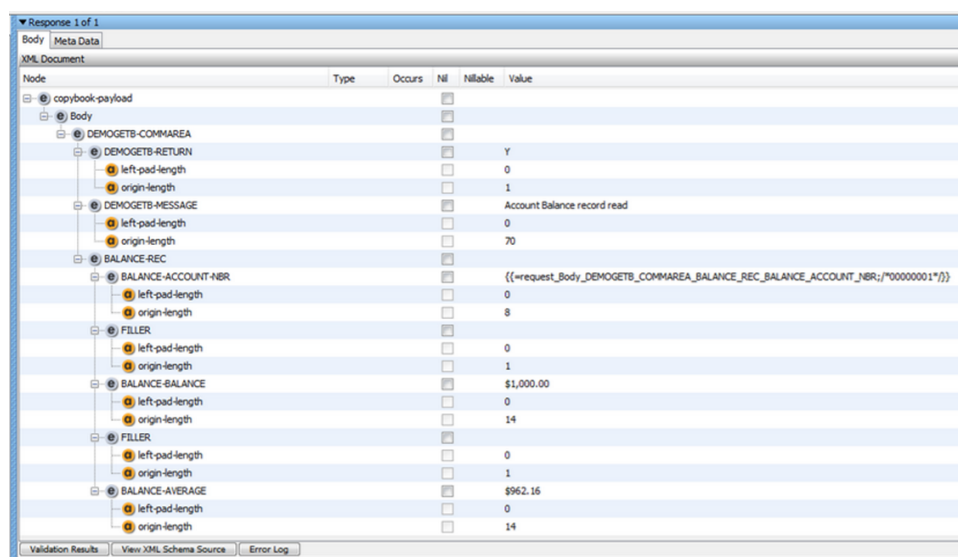


▼ Request Data

Arguments    Attributes    Meta Data

Name	Name in Session	Comparison Operator	Value	Magic String	Date Pattern
called_by		=	DEMOINQ	<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_DEMOGETB-RETURN		=		<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_DEMOGETB-MESSAGE		=		<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_BALANCE-REC_BALANCE-ACCOUNT-NBR		=	00000001	<input checked="" type="checkbox"/>	
Body_DEMOGETB-COMMAREA_BALANCE-REC_FILLER_1		=		<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_BALANCE-REC_BALANCE-BALANCE		=		<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_BALANCE-REC_FILLER_2		=		<input type="checkbox"/>	
Body_DEMOGETB-COMMAREA_BALANCE-REC_BALANCE-AVERAGE		=		<input type="checkbox"/>	

The copybook has also formatted the response. The following graphic shows the response to return when using virtualization:



▼ Response 1 of 1

Body    Meta Data

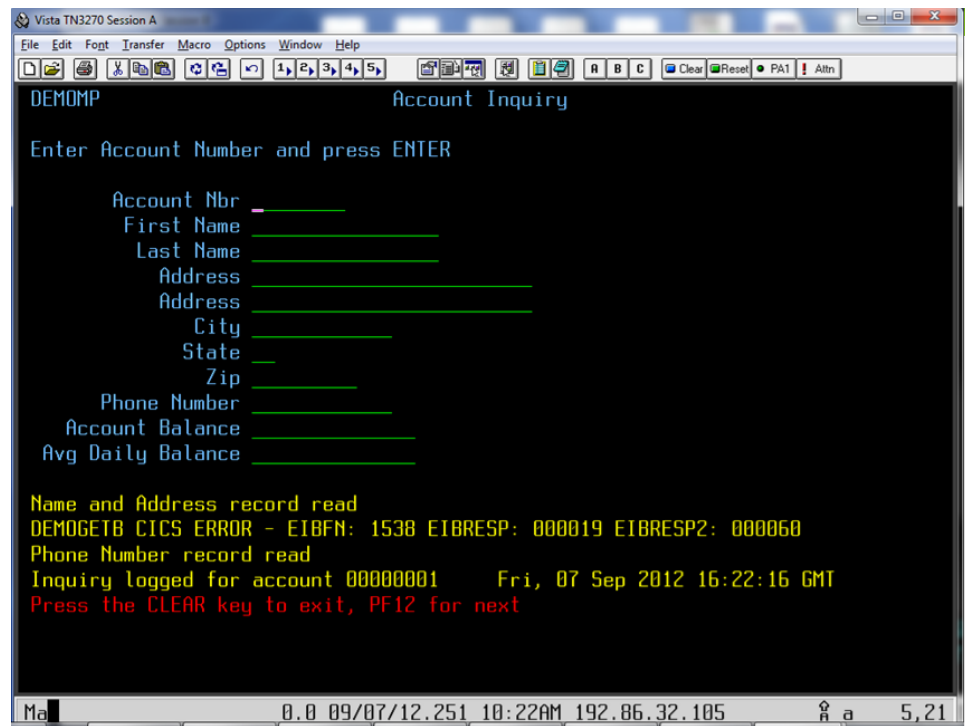
XML Document

Node	Type	Occurs	Nil	Nilable	Value
copybook-payload			<input type="checkbox"/>		
Body			<input type="checkbox"/>		
DEMOGETB-COMMAREA			<input type="checkbox"/>		
DEMOGETB-RETURN			<input type="checkbox"/>	Y	
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	1	
DEMOGETB-MESSAGE			<input type="checkbox"/>	Account Balance record read	
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	70	
BALANCE-REC			<input type="checkbox"/>		
BALANCE-ACCOUNT-NBR			<input type="checkbox"/>	{{=request_Body_DEMOGETB_COMMAREA_BALANCE_REC_BALANCE_ACCOUNT_NBR;/"00000001"/}}	
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	8	
FILLER			<input type="checkbox"/>		
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	1	
BALANCE-BALANCE			<input type="checkbox"/>	\$1,000.00	
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	14	
FILLER			<input type="checkbox"/>		
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	1	
BALANCE-AVERAGE			<input type="checkbox"/>	\$962.16	
left-pad-length			<input type="checkbox"/>	0	
origin-length			<input type="checkbox"/>	14	

Validation Results    View XML Schema Source    Error Log

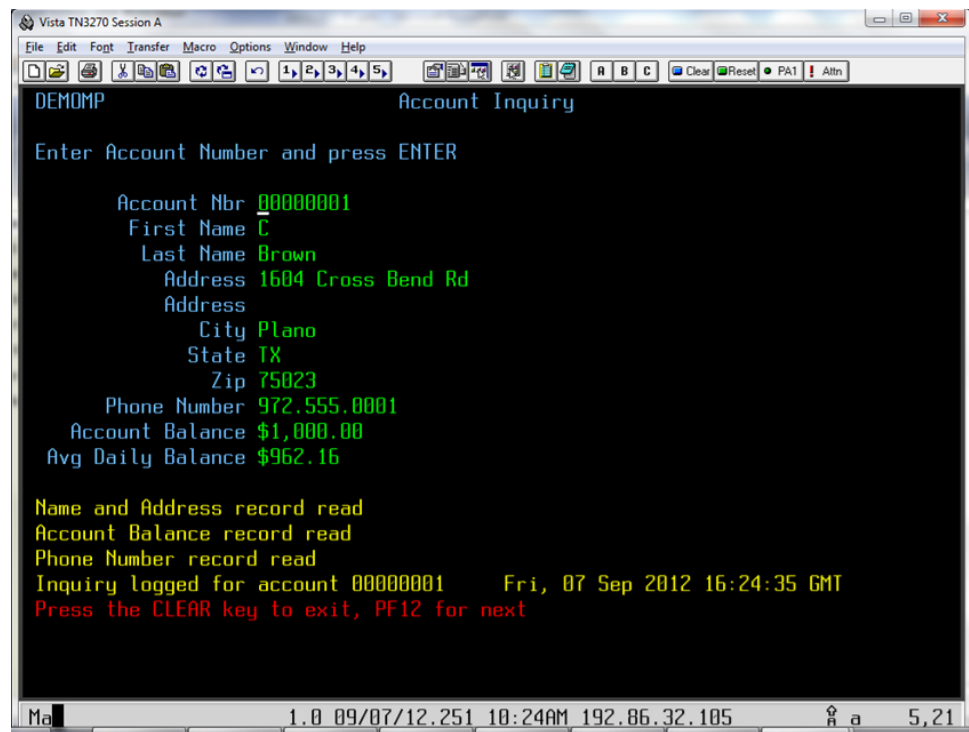
### Example

In this example, the VSAM file DEMOBAL is made unavailable, and program DEMO3 is run. DEMO3 fails because of the unavailability of DEMOBAL.



Next, the virtual service is deployed.

When the program DEMO3 is run again, it works, even though the VSAM file is still unavailable.

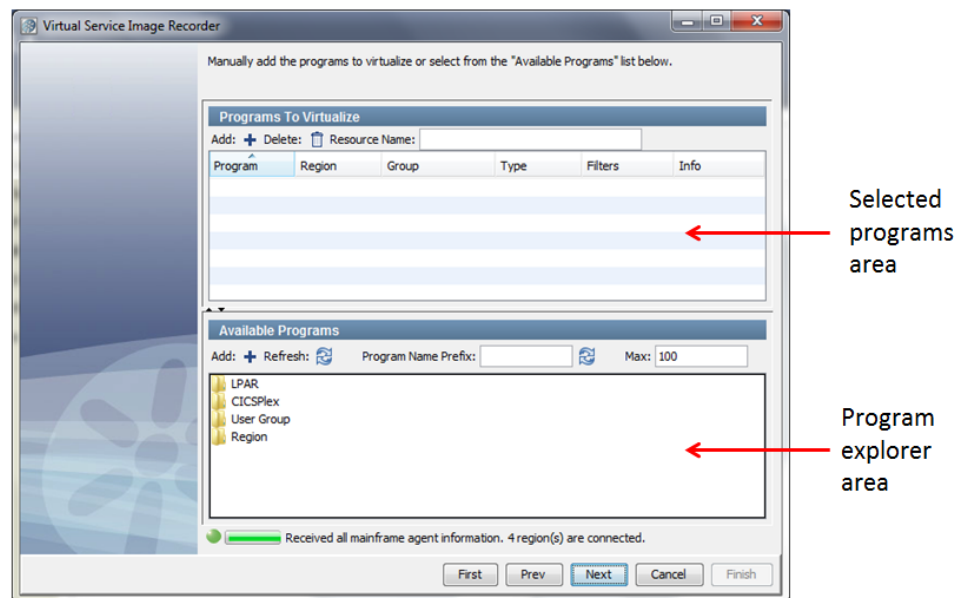


## CICS Programs to Virtualize Panel

Use the CICS Programs to Virtualize panel to:

- Explore the CICS regions that are connected to DevTest LPAR Agents
- Explore the programs that are defined to these CICS regions
- Select the programs to virtualize
- Manually add programs to virtualize
- Set the filters to refine the CICS LINK statements, transactions, and CICS users that are virtualized
- Establish a group membership to virtualize CICS programs in a group of CICS regions

The following graphic shows the main areas of the CICS Programs to Virtualize panel:



### Available Programs

The Available Programs area lets you explore CICS regions that are connected to DevTest with the DevTest CICS Agent and see the programs that are defined there.

You can explore the known CICS regions and programs by:

- LPAR (z/OS Logical Partition)
- CICSplex
- User-defined grouping
- Region

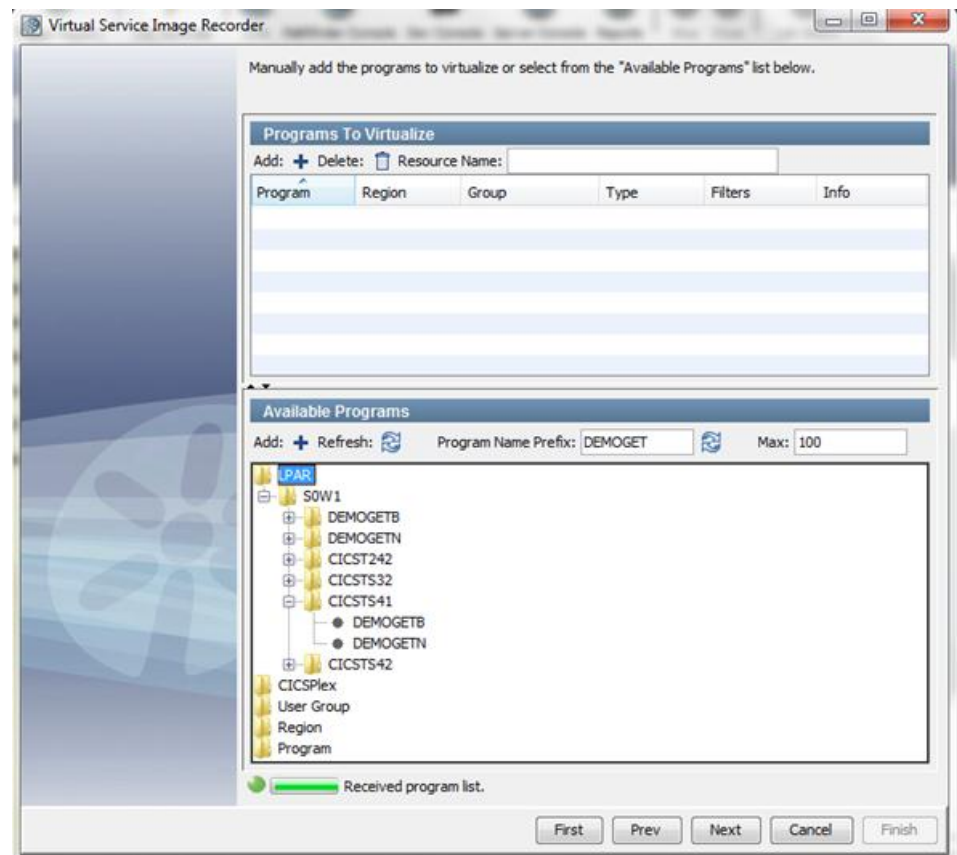
VTAM APPLIDs (application IDs) identify the CICS regions.

### Program Name Prefix

Defines the prefix that the recorder uses to filter which programs to display. The Available Programs area only displays programs that begin with the specified prefix. Always add a program prefix.

### Max

Defines the maximum number of programs to display from one CICS region.



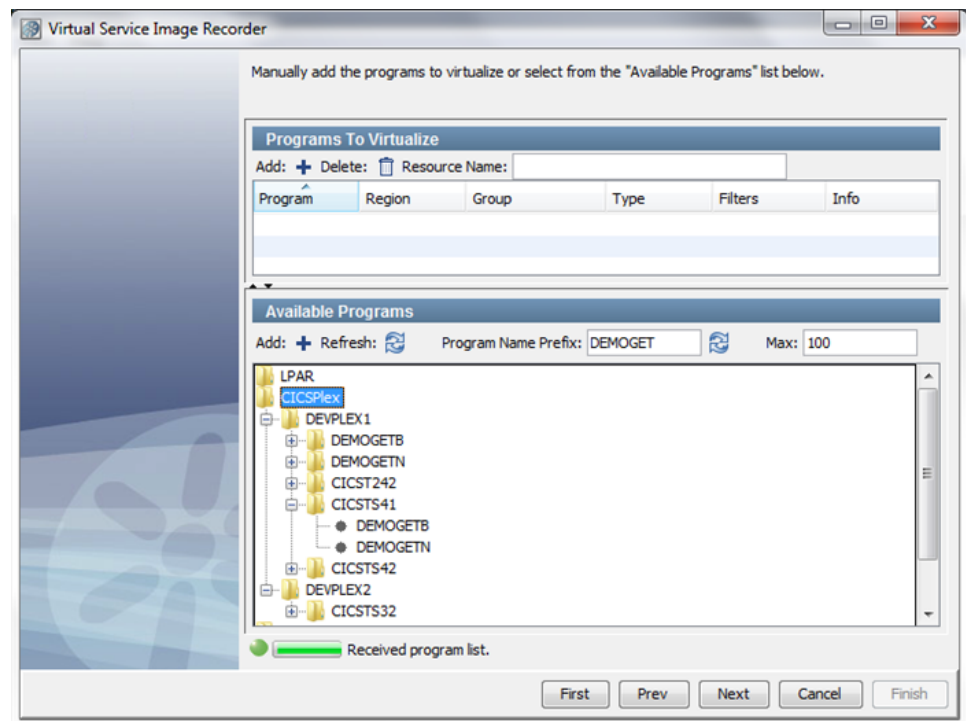
Clicking LPAR shows the known LPARs (in this case, SOW1).

Clicking the plus sign next to SOW1 shows the known CICS regions in SOW1: CICST242, CICSTS32, CICSTS41, and CICSTS42.

Clicking the plus sign next to CICSTS41 shows the known programs that match the prefix DEMOGET on CICSTS41 (DEMOGETB and DEMOGETN).

The list under SOW1 also contains DEMOGETB and DEMOGETN because they are programs that were found in the LPAR.





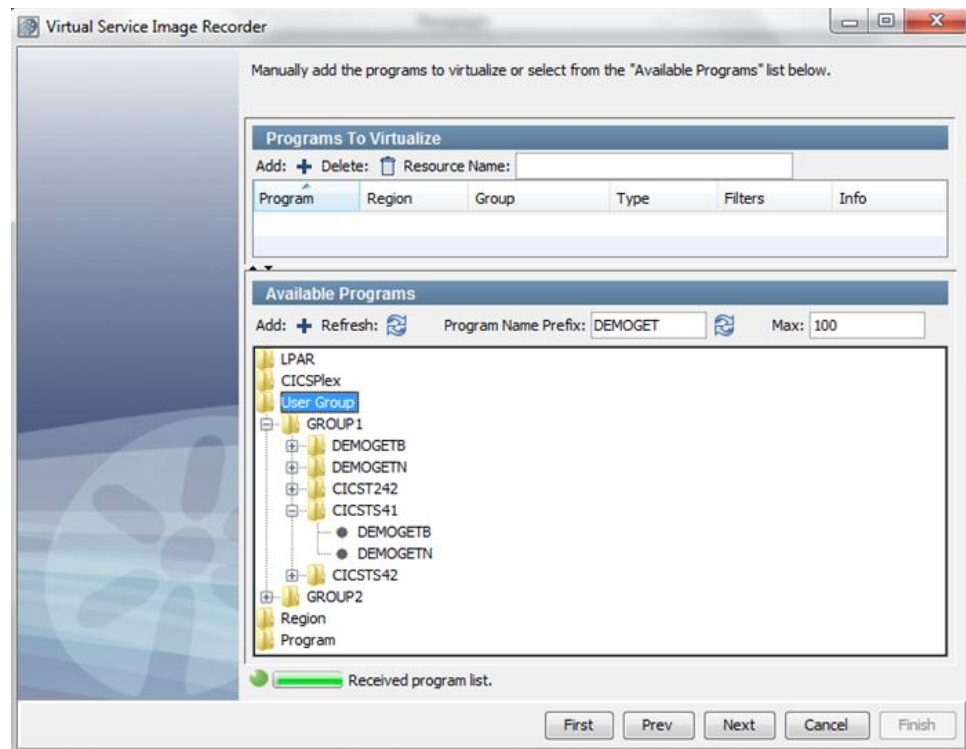
Clicking CICSplex shows the known CICSplexes and reveals DEVPLEX1 and DEVPLEX2.

Clicking the plus sign next to DEVPLEX1 shows the known CICS regions: CICST242, CICSTS41, and CICSTS42.

Clicking the plus sign next to CICSTS41 shows the known programs that match the prefix on CICSTS41 (DEMOGETB and DEMOGETN).

The list under DEVPLEX1 also contains DEMOGETB and DEMOGETN because they are programs that were found in the CICSplex.

**Note:** For a CICS region to be a member of a CICSplex group, you must modify a DevTest CICS agent user exit. For more information, see "CICS Agent User Exits" in *Agents*.



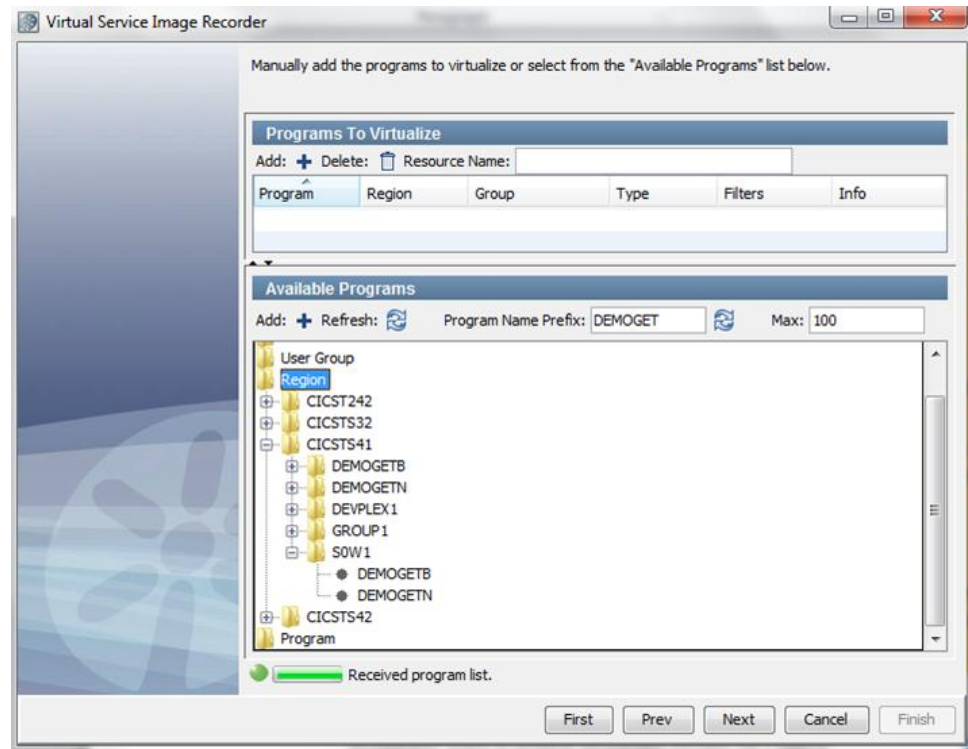
Clicking User Group shows the known user groups and reveals GROUP1 and GROUP2.

Clicking the plus sign next to GROUP1 shows the known CICS regions: CICST242, CICSTS41, and CICSTS42.

Clicking the plus sign next to CICSTS41 shows the known programs that match the prefix on CICSTS41 (DEMOGETB and DEMOGETN).

The list under GROUP1 also contains DEMOGETB and DEMOGETN because they are programs that were found in the user group.

**Note:** For a CICS region to be a member of a user group, you must modify a DevTest CICS agent user exit. For more information, see "CICS Agent User Exits" in *Agents*.

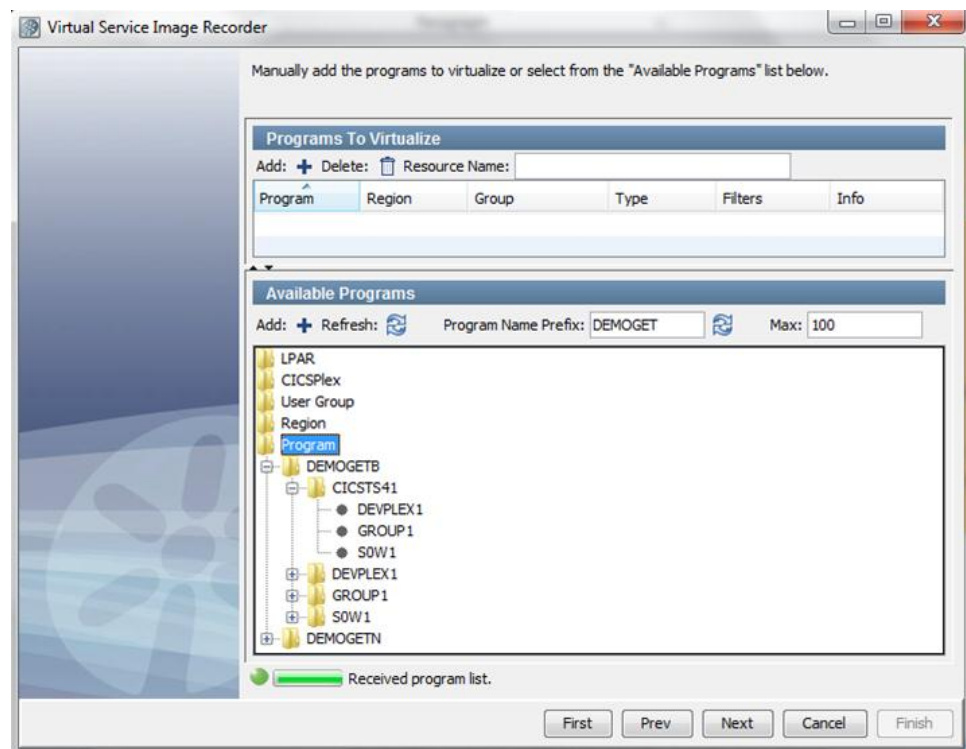


Clicking Region shows the known CICS Regions and reveals CICST242, CICSTS32, CICSTS41, and CICSTS42.

Clicking the plus sign next to CICSTS41 shows the groups that contain this CICS region.

Clicking the plus sign next to SOW1 (the LPAR group of which this region is a member) shows the known programs that match the prefix on CICSTS41 (DEMOGETB and DEMOGETN).

The list under GROUP1 also contains DEMOGETB and DEMOGETN because they are programs that were found in the user group.




When the program information is retrieved by clicking a CICS region, the grouping by Program folder appears.

Clicking Program reveals all known CICS programs; DEMOGETB and DEMOGETN in this case.

Clicking the plus sign next to DEMOGETB expands the known CICS regions and groups containing DEMOGETB.

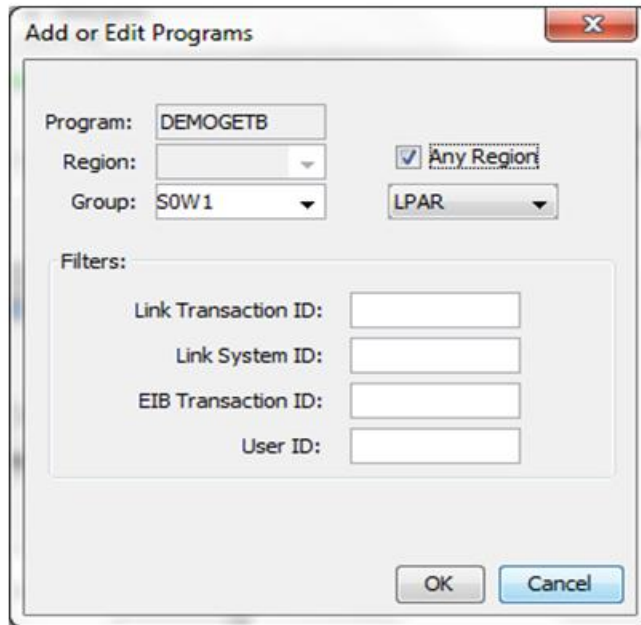
#### Programs to Virtualize

To add programs, double-click them and they appear in the Programs to Virtualize panel. Or, click Add  on the Programs to Virtualize panel to add a program manually.

When a program is listed in the Programs to Virtualize panel, double-click it to add filters and groupings.

In this example, program DEMOGETB is virtualized in CICS region CICSTS41. Group SOW1 and LPAR are ignored unless the Any Region check box is selected.

Filters allow you to specify which of the CICS LINKs to DEMOGETB are virtualized.

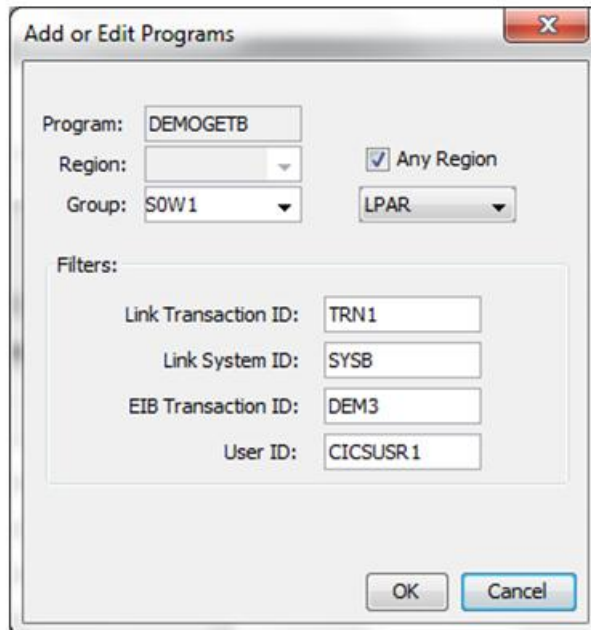


The 'Add or Edit Programs' dialog box is shown. It has a title bar with a close button (X). The main area contains the following fields:

- Program: DEMOGETB
- Region: (empty dropdown)
- Group: SOW1
- LPAR: LPAR
- Any Region: ☒ (checked)
- Filters section with four input fields:
  - Link Transaction ID: (empty)
  - Link System ID: (empty)
  - EIB Transaction ID: (empty)
  - User ID: (empty)

At the bottom right are 'OK' and 'Cancel' buttons.

Selecting the Any Region check box clears the Region text box and indicates that DEMOGETB will be virtualized in ALL CICS regions in LPAR group SOW1 (all CICS regions residing in LPAR SOW1).



The 'Add or Edit Programs' dialog box is shown again, but with the filter fields populated:

- Program: DEMOGETB
- Region: (empty dropdown)
- Group: SOW1
- LPAR: LPAR
- Any Region: ☒ (checked)
- Filters section with four input fields:
  - Link Transaction ID: TRN1
  - Link System ID: SYSB
  - EIB Transaction ID: DEM3
  - User ID: CICSUSR1

At the bottom right are 'OK' and 'Cancel' buttons.

In the previous graphic, filters are used to narrow the programs to virtualize.

#### Link Transaction ID

Indicates that CICS LINKs to DEMOGETB are virtualized only if they are coded with TRANSID('TRN1').

**Link System ID**

Indicated that CICS LINKs to DEMOGETB are virtualized only if coded with SYSID('SYSB').

**EIB Transaction ID**

Indicates that CICS LINKs to DEMOGETB are virtualized only if executed while running under the transaction DEM3.

**User ID**

Indicates that CICS LINKs to DEMOGETB are virtualized only if executed while running under user ID CICSUSR1.

## Record CICS Transaction Gateway (ECI) Images

### To record CICS Transaction Gateway (ECI) images:

1. Select CICS Transaction Gateway (ECI) as the transport protocol on the Basics tab of the Virtual Service Image Recorder.
2. Complete the fields on the Basics tab and click Next.  
The next step in the recorder opens.
3. Enter the port and host information for this step.

#### **Listen/Record on port**

Defines the port on which the client communicates to DevTest.

#### **Target host**

Defines the name or IP address of the target host where the server runs.

#### **Target port**

Defines the target port number on which the server listens.

#### **Expect SSL From Clients**

Specifies whether the recorder expects clients to connect to it using SSL. The related keystore and password, if provided, are used to obtain security material (such as certificates).

##### **Values:**

- **Selected:** The recorder expects clients to use SSL to connect.
- **Cleared:** The recorder does not expect clients to use SSL to connect.

#### **Initiate SSL To Server**

Specifies whether the recorder connects to the real system using SSL. The related keystore and password, if provided, are used to obtain security material (such as certificates).

##### **Values:**

- **Selected:** The recorder uses SSL to connect.
- **Cleared:** The recorder does not use SSL to connect.

#### **SSL keystore file**

Specifies the name of the keystore file.

#### **Keystore password**

Specifies the password associated with the specified keystore file.

4. Click Next to display the recording window.
5. Start your application that communicates with the CTG server and perform the activity you want to record.

6. When you see transactions are recorded on the Virtual Service Image Recorder window, click Next.

The data protocols window opens.

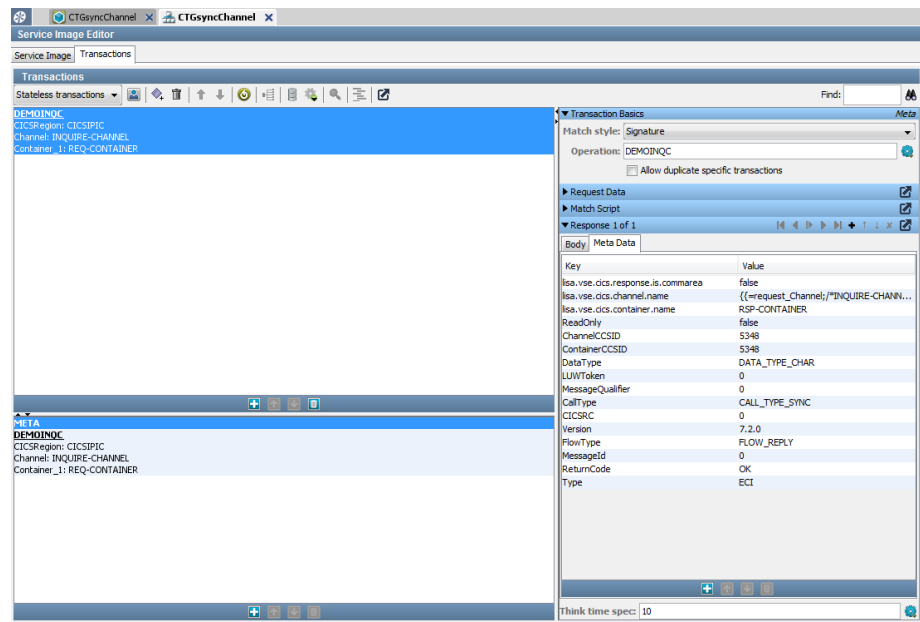
7. Click Next.

The conversation starter window opens.

8. Click Next.

9. Select the options to display the VSM and VSI, and click Finish.

The Service Image Editor displays the CTG service image.





## Record IMS Connect Service Images

### To record IMS Connect service images:

- Before starting the recorder, add the following properties in **local.properties** as necessary:
  - If the system under test sends responses that have a four-byte LLLL length field at the beginning of the response message, set the following property:
 

```
lisa.vse.protocol.ims.response.includes.llll=true
```
  - If the system under test receives requests that have a four-byte LLZZ length field at the beginning of the request message, set the following property:
 

```
lisa.vse.ims.connect.llzz.request=true
```
  - If the system under test sends responses that have a four-byte LLZZ length field at the beginning of the response message, set the following property:
 

```
lisa.vse.ims.connect.llzz.response=true
```
  - If the system under test uses copybooks for parsing the payloads, add the following property:
 

```
lisa.vse.copybook.unknown.passthrough=true
```

This ensures that the binary responses that represent acknowledgment messages (they do not include any payload data) are bypassed from copybook processing.

- Select IMS Connect as the transport protocol on the Basics tab of the Virtual Service Image Recorder.
- Complete the fields on the Basics tab, then click Next.  
The next step in the recorder opens.
- Enter the port and host information:

#### Listen/Record on port

Defines the port on which the client communicates to DevTest.

#### Target host

Defines the name or IP address of the target host where the server runs.

#### Target port

Defines the target port number listened to by the server. Leave this field blank if you will select a Proxy passthrough style.

**Defaults:** 80 (HTTP) and 443 (HTTPS)

#### IMS Format File

Defines one of the following:

- The name of the file that supports IMS connect requests that use the 160-byte header that the IBM sample app uses

- An 80-byte header.

The field definitions for these supported 160-byte and 80-byte headers are provided in the **ims.format** file in the LISA\_HOME directory.

To use the IMS Connect support that is included in DevTest by default, leave this field blank.

#### **Character Set**

Defines how the data is encoded by the application to be virtualized.

##### **Values:**

- ASCII - CP1252
- EBCDIC - CP037

5. Click Next to display the recording window.
6. Start your application that communicates with the IMS Connect server and perform the activity that you want to record.

Ensure that the host and port for this application are the host and port for the recorder (typically localhost:8001).

7. When you see transactions have been recorded on the Virtual Service Image Recorder window, click Next.

The data protocols window opens.

By default, the TransactionCode header field (if it exists) is set as the request operation. Also, TransactionCode, DestinationId, and ClientId are added as arguments, if these fields exist in the format file definition.

8. Add data protocols, including the Copybook data protocol if needed, and click Next.

The conversation starter window opens.

9. Click Next.

10. Select the options to display the VSM and VSI, then click Finish.

The Service Image Editor displays the IMS Connect service image.

## IMS Connect Customized Format Files

If the system under test uses request headers that differ from the ones DevTest supports by default, follow these steps before recording:

1. Create a file **<custom-format>.format** in the Data folder of the DevTest project.
2. Edit the file in a text editor and add the following text to it. Provide your field definitions under the RequestUserHeader area.

```
#-----
# The IMS™ request message (IRM) header contains a 28-byte fixed-format
# section that is common to all messages from all IMS Connect client applications
# that communicate with IMS TM.
#-----
RequestHeaderCommon {
    LLLL                int;                #total message length IRM + user data
    IRMFixedHeader {
        LL                short;            #IRM_LEN, total length of the header segment
including user header portion
        ZZ                byte;              #IRM_ARCH
        Flag0              byte;              #IRM_F0
        UserExitId          string(8);        #IRM_ID
        NakReasonCode       byte(2);          #IRM_NAK_RSNCDE
        Reserved1           byte(2);          #IRM_RES1
        MessageType         byte;              #IRM_MessageType
        WaitTime             byte;              #IRM_TIMER
        SocketConnectionType byte;              #IRM_SOCT
        EncodingSchema       byte;              #IRM_ES
        ClientId             byte(8);          #IRM_CLIENTID
    }
}

#-----
# Format of user portion of IRM some custom header
#
# Following the 4-byte length field and the 28-byte fixed portion of the
# IMS™ request message (IRM) header in IMS Connect client input messages,
# user-written client applications can include a user-defined section in the IRM.
#
#-----
RequestUserHeader {
    MyFlag1                byte;
    MyFlag2                byte;
    MyFlag3                byte;
    MyFlag4                byte;
    TransactionCode         string(8);
    DestinationId           string(8);
    LogicalTerminal         string(8);
    Miscellaneous           byte(20);
}
```

```
}

#-----
# Format of data segments of request message
#-----
RequestPayloadSegment {
    LL          short;
    ZZ          byte(2);
    Data        byte(LL:inclusive);
}

#-----
# Format of header for response message. Some responses will have it, some won't
#-----
ResponseMessageHeader {
    LLLL          int;          # followed by multiple ResponsePayloadSegment
}

#-----
# Format of data segments of request message
#-----
ResponsePayloadSegment {
    LL          short;
    ZZ          byte(2);
    Data        byte(LL:inclusive);
}
```

When the file is created and saved in the Data folder, it appears in the IMS Format file field in the recorder. You can select the format file for applications that use nondefault request headers.

## Record SAP RFC via JCo

### Follow these steps:

1. Select SAP RFC via JCo as the transport protocol on the Basics tab of the Virtual Service Image Recorder.
2. Complete the fields on the Basics tab, then click Next.  
The next step in the recorder opens.
3. Complete the connection details fields.

#### **Client System Name**

Specifies a unique name to identify the client SAP system.

#### **Client System Connection Properties**

The Client System Connection properties file contains connection properties to connect to the destination on the client system. This must be a .properties file in the Data directory of your project and contains properties that are typically found in a .jcoServer file. This file MUST NOT specify jco.server.repository\_destination. For more information about supported properties, see the JavaDocs for **com.sap.conn.jco.ext.ServerDataProvider** in the **doc** folder of your installation directory.

#### **RFC Repository Name**

Specifies a unique name to identify the SAP system that has the repository for the RFC template. This name is often the same as the Destination System.

#### **RFC Repository Connection Properties**

Defines the Repository Connection properties file that contains connection properties to connect to the system that has the repository. This must be a .properties file in the Data directory of your project and contains properties that are typically found in a .jcoDestination file. For more information about supported properties, see the JavaDocs for **com.sap.conn.jco.ext.DestinationDataProvider** in the **doc** folder of your installation directory.

#### **Destination System Name**

Defines a unique name that identifies the SAP system on which the RFC executes. This is often the same as the Repository Name.

#### **Destination System Connection Properties**

Specifies the Destination System Connection properties file that contains connection properties with which to connect to the system that has the repository. This must be a .properties file in the Data directory of your project and contains properties that are typically found in a .jcoDestination file. For more information about supported properties, see the JavaDocs for **com.sap.conn.jco.ext.DestinationDataProvider** in the **doc** folder of your installation directory. This can be the same file as the Repository Connection Properties.

**Function Name**

Specifies the name of the RFC that DevTest should intercept.

4. Click Next to display the recording window.
5. Start the application that communicates with the SAP server and perform the activity that you want to record.
6. When you see transactions are recorded on the Virtual Service Image Recorder window, click Next.

The data protocols window opens.

7. Click Next.

The conversation starter window opens.

8. Click Next.

9. Select the options to display the VSM and VSI, then click Finish.

The Service Image Editor displays the SAP RFC service image.

## Record JCo IDoc

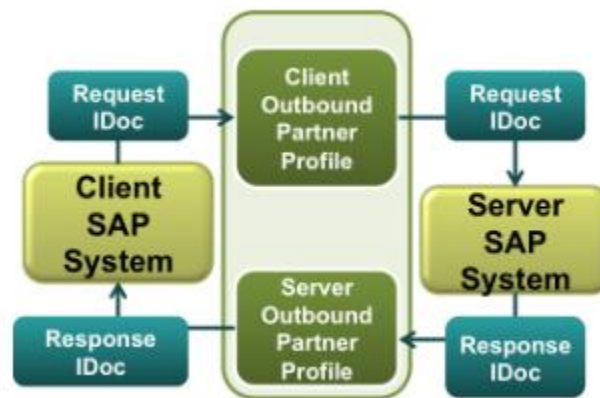
The SAP JCo IDoc transport protocol allows you to create virtual services that use the SAP JCo IDoc protocol.

The JCo IDoc protocol supports the creation of a virtual service that records and plays back IDocs sent from one SAP system to another using an RFC destination.

The SAP system that initiates communication and sends IDocs is referred to as the Client SAP system. The IDocs sent are referred to as the Request IDocs.

The SAP system that receives the Request IDocs is referred to as the Server SAP system. The IDocs that the Server SAP system returns are referred to as the Response IDocs.

The following graphic shows the regular flow of IDocs between two SAP systems.



**Prerequisites:** Using DevTest with this application requires that you make one or more files available to DevTest. For more information, see *Third-Party File Requirements in Administering*.

### To record JCo IDoc service images:

1. Select JCo IDoc Protocol as the transport protocol on the Basics tab of the Virtual Service Image Recorder.
2. Complete the fields on the Basics tab and click Next.  
The next step in the recorder opens.
3. Complete the connection details fields.

#### Client RFC Connection Properties

Defines the Client RFC Connection properties file that contains connection properties that VSE uses to register itself under a program ID to an SAP gateway and receive IDocs. The properties should be the same as those specified in a .jcoServer file.

**Client RFC Destination Name**

Specifies a unique name that identifies the RFC destination.

**Client System Connection Properties**

Specifies the Client System Connection properties file that contains connection properties to return IDocs to the client SAP system. These properties should be the same as those specified in a .jcoDestination file that can be used to connect to the client SAP system.

**Client System Name**

Specifies a unique name to identify the client SAP system.

**Request Identifier XPath Expressions**

Specifies the XPath expressions that the protocol uses with the request IDoc XML to generate an identifier. The request identifier XPath expressions can be a single XPath expression. This identifier is used to correlate a request IDoc to a response IDoc. XPath expressions can also be a comma-separated list of XPath expressions, in which case the resulting values from the multiple expressions are concatenated (separated by dashes) and used as an identifier.

**Server RFC Connection Properties**

Specifies a properties file that contains connection properties that VSE uses to register itself under a program ID to an SAP gateway and receive IDocs. The properties should be the same as those specified in a .jcoServer file to start a JCo server program that receives IDocs from the server SAP system.

**Server RFC Destination Name**

Specifies a unique name to identify the Server RFC destination.

**Server System Connection Properties**

The Server System Connection properties file contains connection properties to return IDocs to the client SAP system. These properties should be the same as those specified in a .jcoDestination file that can be used to connect to the SAP server system.

**Server System Name**

Specifies a unique name to identify the Server SAP system.

**Response Identifier XPath Expressions**

Defines the XPath expressions that the protocol uses with the response IDoc XML to generate an identifier. The Response Identifier XPath Expressions can be a single XPath expression. This identifier is used to correlate a response IDoc to a request IDoc that was received earlier. XPath expressions can also be a comma-separated list of XPath expressions, in which case the resulting values from the multiple expressions are concatenated (separated by dashes) and used as an identifier.

4. Click Next to open the recording window.



5. Start the application that communicates with the SAP server and perform the activity that you want to record.
6. When you see that transactions have been recorded on the Virtual Service Image Recorder window, click Next.

The data protocols window opens.

The XML data protocol is selected on the request-side data protocols, because VSE stores the IDocs as XML. Remove the XML data protocol, as it overrides the operation name that the JCo IDoc protocol sets.

7. Click Next.

The conversation starter window opens.

8. Click Next.

9. Select the options to display the VSM and VSI, and click Finish.

The Service Image Editor displays the SAP JCo IDoc service image.

## Opaque Data Processing

Opaque Data Processing (ODP) allows CA Service Virtualization to virtualize data in sufficient detail when the format of the requests and responses is not known. ODP eliminates the need for a new data handler every time you encounter a new message format.

By recording several requests and corresponding responses, CA Service Virtualization can infer the message structure. This means CA Service Virtualization is able to correlate bytes in a request to corresponding bytes in a response, giving the same "magic string" behavior that is available in other protocols. CA Service Virtualization is also able to sufficiently understand the request structure to intelligently match new requests that are encountered when the virtual service is played back.

Underneath, ODP uses a patented algorithm to compare an incoming request to all of the requests in the recorded ODP service image. The closest matching request is selected, and the corresponding response is returned, after performing a dynamic magic string substitution.

The ODP matching algorithm applies entropy-derived weights during the matching process. The entropy weighting process infers which bytes in the message are more important. For example, which correspond to the operation type, as opposed to the rest of the payload, and gives those bytes a greater importance during the matching process. The entropy weighting process works best with larger samples of recorded messages (100 or more) and with a diverse sampling of parameter values.

The ODP matching and response algorithm has been shown to work on both binary and textual message protocols. The best results have been obtained with protocols that use fixed width fields (such as IMS) or delimited fields (such as XML-based protocols). Reasonable accuracy has also been obtained with length-encoded protocol formats (for example, ASN.1), but these pose the greatest challenge.

ODP is supported for traffic that can be captured using raw TCP/IP sockets. The data can also be imported from a PCAP file.

## How to Use ODP

Use the following instructions for recording a virtual service image using ODP.

For more information about prerequisites and preparatory steps, see [Preparing to Virtualize TCP](#) (see page 167).

### Follow these steps:

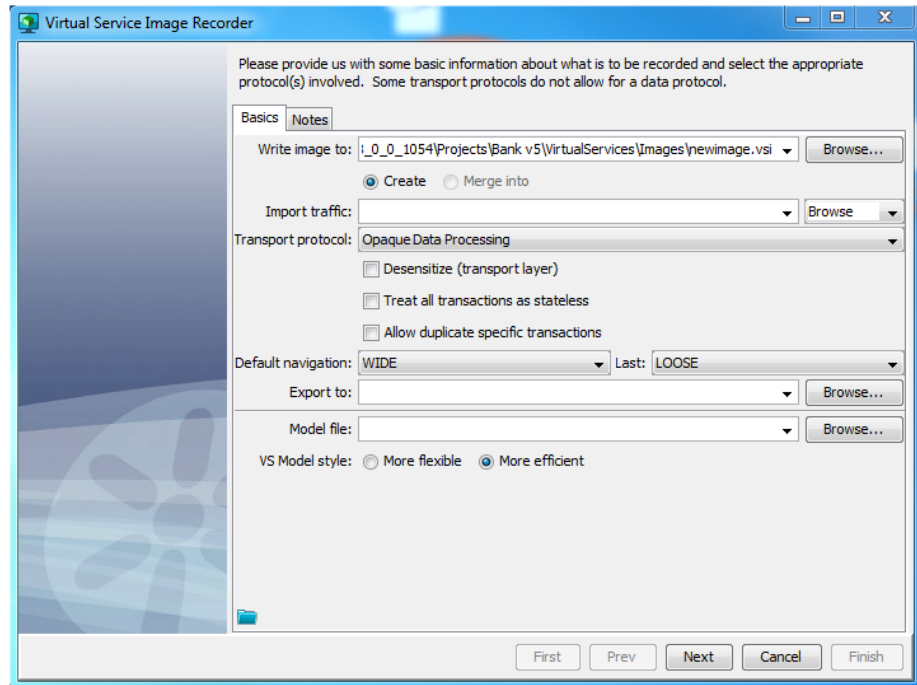
1. Open DevTest Workstation.
2. To start recording a new virtual service image, complete one of the following steps:

- Click VSE Recorder  on the main toolbar.

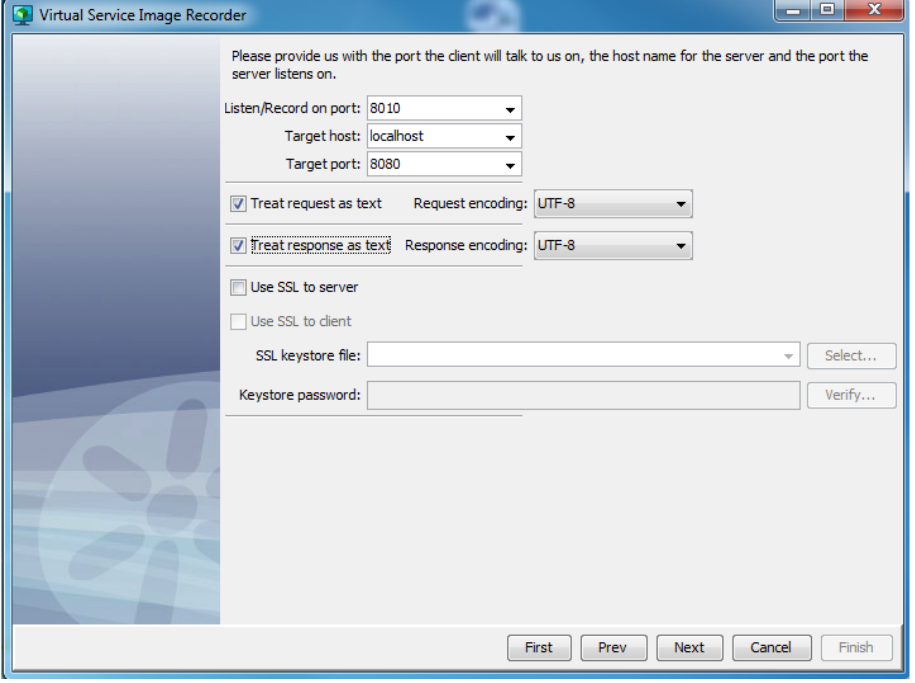
- Right-click the VirtualServices node on the Project panel and select Create a VS Image, by Recording.

The Virtual Service Image Recorder opens.

3. Complete the [Basics](#) (see page 120) tab as in the following graphic:



4. Click Next.
5. Enter the information for both the client and the server, select your encoding, and add SSL parameters.



The screenshot shows the 'Virtual Service Image Recorder' dialog box. It contains the following fields and options:

- Listen/Record on port:** 8010
- Target host:** localhost
- Target port:** 8080
- ☒ **Treat request as text**    Request encoding: UTF-8
- ☒ **Treat response as text**    Response encoding: UTF-8
- ☐ **Use SSL to server**
- ☐ **Use SSL to client**
- SSL keystore file:** [Empty field]    Select...
- Keystore password:** [Empty field]    Verify...

At the bottom, there are navigation buttons: First, Prev, Next, Cancel, and Finish.

**Listen/Record on port**

Defines the port on which the client communicates to DevTest.

**Target host**

Defines the name or IP address of the target host where the server runs.

**Target port**

Defines the port number that the server is listening on.

**Treat request as text**

Specifies whether the request is treated as text. For more information, see [Preparing to Virtualize TCP](#) (see page 167).

**Request Encoding**

Lists the available request encodings on the machine where DevTest Workstation is running. The default is UTF8.

**Treat response as text**

Specifies whether the response is treated as text. For more information, see [Preparing to Virtualize TCP](#) (see page 167).

**Response Encoding**

Lists the available response encodings on the machine where DevTest Workstation is running. The default is UTF8.

**Use SSL to server**

Specifies whether DevTest uses HTTPS to send the request to the server.

- **Selected:** DevTest sends an HTTPS (secured layer) request to the server.

If you select Use SSL to server, but you do not select Use SSL to client, DevTest uses an HTTP connection for recording. DevTest then sends those requests to the server using HTTPS.

- **Cleared:** DevTest sends an HTTP request to the server.

**Use SSL to client**

Specifies whether to use a custom keystore to play back an SSL request from a client. This option is only enabled when Use SSL to server is selected.

**Values:**

- **Selected:** You can specify a custom client keystore and a passphrase. If these parameters are entered, they are used instead of the hard-coded defaults.
- **Cleared:** You cannot specify a custom client keystore and a passphrase.

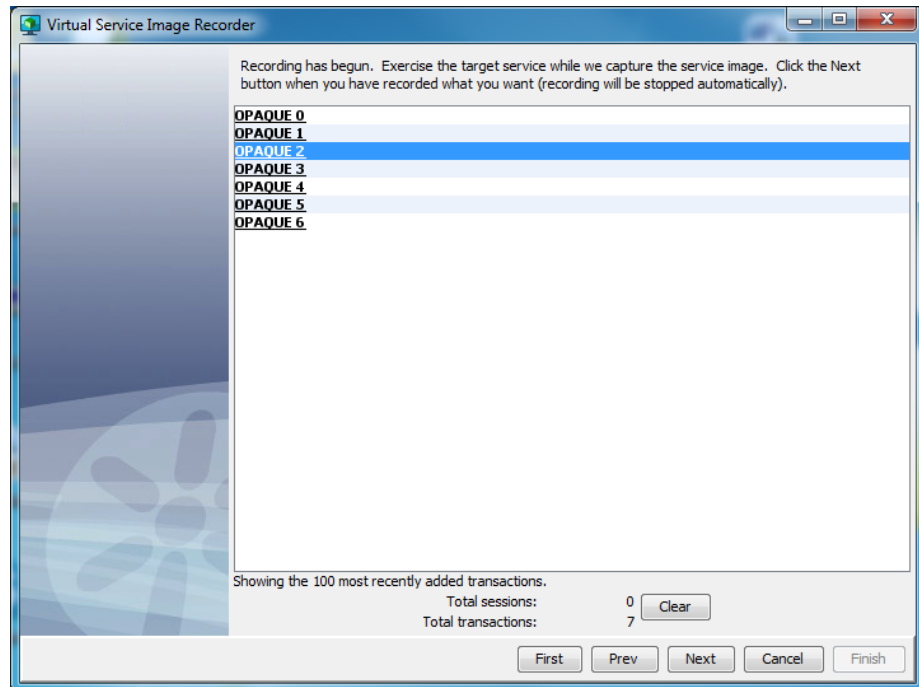
**SSL keystore file**

Specifies the name of the keystore file.

**Keystore password**

Specifies the password associated with the specified keystore file.

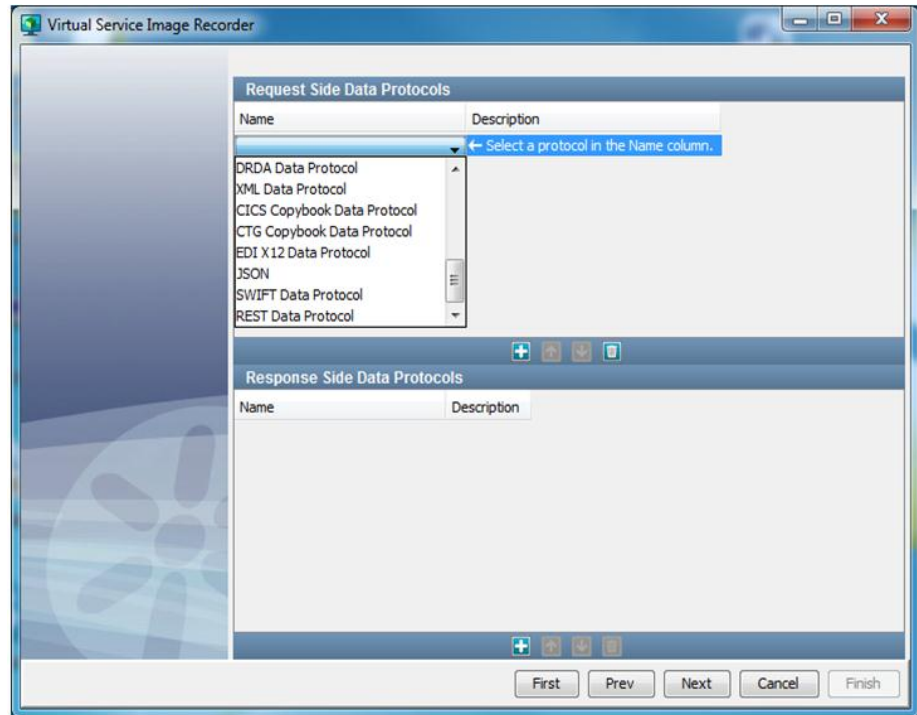
6. Click Next to start recording.



7. When your recording is complete, click Next.

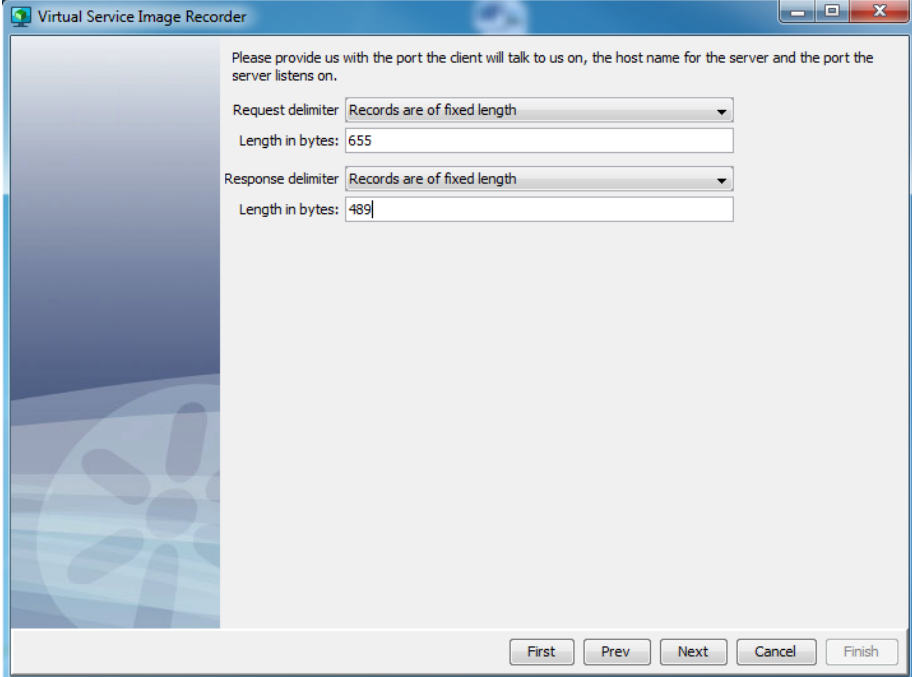
ODP relies on the request body, rather than operation and arguments, for matching. Therefore, most request data protocols are not appropriate.

Response data protocols, on the other hand, are appropriate if the response is encrypted, compressed, or otherwise encoded.



8. The recorder attempts to detect message delimiters that tell DevTest when it has read a complete request or response. Confirm, or correct, these delimiters on this screen.

**Note:** A Request delimiter is mandatory. A Response delimiter must be selected for Live Invocation to be available.



The screenshot shows a window titled "Virtual Service Image Recorder". Inside the window, there is a text area on the left with a blue background and a white wheel-like pattern. To the right of the text area, there is a form with the following fields:

- A text prompt: "Please provide us with the port the client will talk to us on, the host name for the server and the port the server listens on."
- A "Request delimiter" dropdown menu set to "Records are of fixed length".
- A "Length in bytes" text box containing the value "655".
- A "Response delimiter" dropdown menu set to "Records are of fixed length".
- A "Length in bytes" text box containing the value "489".

At the bottom of the window, there are five buttons: "First", "Prev", "Next", "Cancel", and "Finish".

9. After recording, the recorder verifies request and response bodies to ensure that, if they are marked as text, that they are actually text. If they are not, the type is switched to binary.



## SSL with VSE Recording

During VSE recording, the SSL Server application is the System Under Test.

### Use SSL to the Server

If Use SSL to the Server only is selected during recorder setup (and Use SSL to Client is not selected) the client sends a plain HTTP connection to DevTest but the recorder sends an HTTPS (SSL secured socket layer) request to the server application. In this case, *client* denotes the application or test case sending the request.

The certificate public key for the SSL Server application must be in the DevTest truststore to authenticate the Server certificate (or provide one-way authentication).

If the SSL Server requests Client Authentication (two-way authentication), LISA v6.0.8 and later uses the keystore identified in **ssl.client.cert.path** to send a client certificate from the recorder to the SSL Server application. The recorder simulates a client to complete the two-way client authentication, when Use SSL to the Server is selected.

### Use SSL to the Server and Use SSL to Client

To record, the client or test case sends to the configured listen/record on port. The recorder Target Host is the real SSL server and the Target port is the SSL port that the SSL Server uses (typically, 443 or 8443).

During recording, the SSL handshake is between the client (recorder) and the SSL server. The server sends its certificate and DevTest authenticates it. If the server requests client authentication, the certificate that is in **local.properties** is used. If there is not a valid keystore in **ssl.client.cert.path** and the server requests client authentication, then a **bad\_certificate** situation is returned because is not a certificate for the client recorder to return to the server.

During recording, the SSL handshake is between the client (application or test case) and the server (recorder). The recorder sends the certificate that is specified in the Use SSL to Client keystore. If SSL Keystore File is blank, the default keystore (`{LISA_HOME}\webreckeyks.ks`) is used. The recorder server does not request client authentication. The handshake is one-way authentication.

### Playback of the VSM

If you select Use SSL to Client, an SSL handshake occurs between the client application or test case client and the VSM. The keystore that is provided in the VSM Listen step is used as the Server certificate for one-way authentication. There is no SSL handshake between the client and the VSM. The handshake is straight HTTP.

If the Live Invocation step was executed, an SSL handshake occurs between the VSM client to the real server. If the real server requests client authentication, the keystore in the HTTP/S Protocol Live Invocation step is used.

If the keystore contains multiple certificates, VSE uses the first one.

## Create and Deploy a Virtual Service with VSEasy

You can quickly create and deploy a virtual service using the VSEasy console.

VSEasy supports the following modes:

- HTTP protocol
- From VRS file

You can use VSEasy to create stateless virtual services for HTTP with SOAP, JSON, XML, or HTML payloads.

To configure non-HTTP virtual services, use VRS files.

When using a VRS file, you can chain data protocols. For example, you can use the Generic XML Payload Parser data protocol or the Request Data Manager data protocol to modify your service image.

The VRS file cannot reference external data. For example, the VRS file cannot reference copybooks and certificates for signing.

In the VSEasy console, you must assign the virtual service to a virtual service group. When you select a VSE server, the available groups appear in the **Service Group** list. The default group is named **VSEasy**. To change the name of the default group, update the **lisa.vseasy.default.group.tag** property in the **lisa.properties** file.

**Note:** If you deploy a virtual service that has the same name and port as a virtual service that is already deployed in the VSE server, the existing virtual service is overwritten.

The following browsers are supported:

- Internet Explorer 10 or 11
- Mozilla Firefox 24
- Google Chrome
- Safari 5

### Follow these steps:

1. Ensure that a VSE server is running.
2. From DevTest Workstation, select the Server Console icon on the toolbar.
3. From the Server Console, select VSEasy.

The VSEasy console opens.

4. Follow the instructions in the Help on the right panel. The Help also includes the following sample files:

- HTTP request/response pair

- JMS request/response pairs and VRS file

## Work with VSMs

A Virtual Service Model (VSM) is a specialized type of test case that becomes the endpoint of a virtualized service. To start the Virtual Service Image Recorder, create a VSM.

### Create a VSM

**Follow these steps:**

1. Open an existing project or create a project.  
The project opens, and the left Project panel shows the project folder and its subfolders.
2. Right-click the VirtualServices subfolder node and select Create New VS Model.  
Although you can also create a VS Model in another folder, it is a good practice to create it under the VirtualServices folder.  
The VS Model Editor window opens.
3. Browse to the location for the new VSM.
4. Enter a unique name for the VSM in the File name field. The extension is **.vsm**.
5. Click Save.  
The new VSM opens.  
When a VSM is open, the Commands menu shows menu items relevant to a VSM.

### Open a VSM

**Follow these steps:**

1. Click Open on the toolbar, or select a project from the Open Recent project list on the Quick Start window.
2. Select the virtual service model from its folder in the Project panel.  
Virtual service models typically reside in the VServices or VirtualServices folder.

## Using Data Protocols

A *data protocol* is also known as a *data handler* or a *data protocol handler*. A data protocol is responsible for parsing requests. Some transport protocols allow (or require) a data protocol to which the job of creating requests is delegated. As a result, the data protocol has to know the payload of the request. One transport protocol can have many data protocols. Some transport protocols (for example, JDBC) do not allow a data protocol.

During recording, all transport protocols try to detect the request or response payload and default appropriate data protocols on either the request or response side. The indicated data protocol handlers are added for the following payloads:

- **SOAP:** Web Services (SOAP) data protocol handler
- **XML:** XML data protocol handler
- **HTML:** No data protocol handler
- **Signed SOAP:** A request-side WS-Security Request and a response-side WS-Security Response data protocol handler
- **Encrypted SOAP:** A request-side WS-Security Request and a response-side WS-Security Response data protocol handler
- **SOAP Security:** A request-side WS-Security Request and a response-side WS-Security Response data protocol handler

You can add more data protocols, reorder the defaulted ones, or delete the defaults. If you are creating a service image from request/response pairs, a raw traffic file, or means other than recording and you have specified data protocol handlers, the defaults are not added.

You can chain data protocol data handlers to work with each other. For example, on the request side, you can use the Delimited Text data protocol, the Generic XML Payload Parser, and the Request Data Manager together.

**Each available data protocol is described in the following sections.**

[Auto Hash Transaction Discovery](#) (see page 210)  
[CICS Copybook](#) (see page 211)  
[Copybook](#) (see page 214)  
[CTG Copybook](#) (see page 230)  
[Data Desensitizer](#) (see page 232)  
[Delimited Text](#) (see page 233)  
[DRDA](#) (see page 235)  
[EDI X12 Data Protocol](#) (see page 236)  
[Generic XML Payload Parser Data Protocol](#) (see page 240)  
[JSON Data Protocol](#) (see page 245)  
[Request Data Copier Data Protocol](#) (see page 247)  
[Request Data Manager Data Protocol](#) (see page 248)  
[REST Data Protocol](#) (see page 251)  
[Scriptable Data Protocol](#) (see page 259)  
[SWIFT Data Protocol](#) (see page 262)  
[Web Services Bridge Data Protocol](#) (see page 264)  
[Web Services \(SOAP\) Data Protocol](#) (see page 265)  
[Web Services \(SOAP Headers\)](#) (see page 266)  
[WS - Security Request Data Protocol](#) (see page 268)  
[XML Data Protocol](#) (see page 272)

## Auto Hash Transaction Discovery

The Auto Hash Transaction Discovery data protocol uses the hash code of the data to identify a message. The hash code changes with even a slight change in the data, which effectively makes all requests unique. This data protocol is useful if you run the same small set of requests against the service.

The Auto Hash Transaction Discovery data protocol has a simple purpose. As it is given requests to handle, the standard Java hash code function is applied to the text version of the request body. The resulting hash code value is then added as a new argument in the request named **lisa.vse.auto.hashDiscovery**.

This ability can be helpful, especially in virtualizing messaging, for creating a reasonably unique value to use for identifying conversations.

Because this protocol requires no configuration information, it does not present a window in the recording wizard.

## CICS Copybook

The CICS Copybook data protocol splits the recorded request into its respective container chunks. Each chunk is then sent to the Copybook data protocol and aggregated with the corresponding XML. The resulting XML is the aggregated XML of the containers.

This example demonstrates the CICS Copybook data protocol usage.

### Follow these steps:

1. Complete the required fields on the Virtual Service Image Recorder [Basics](#) (see page 120) tab.

2. Click Next.

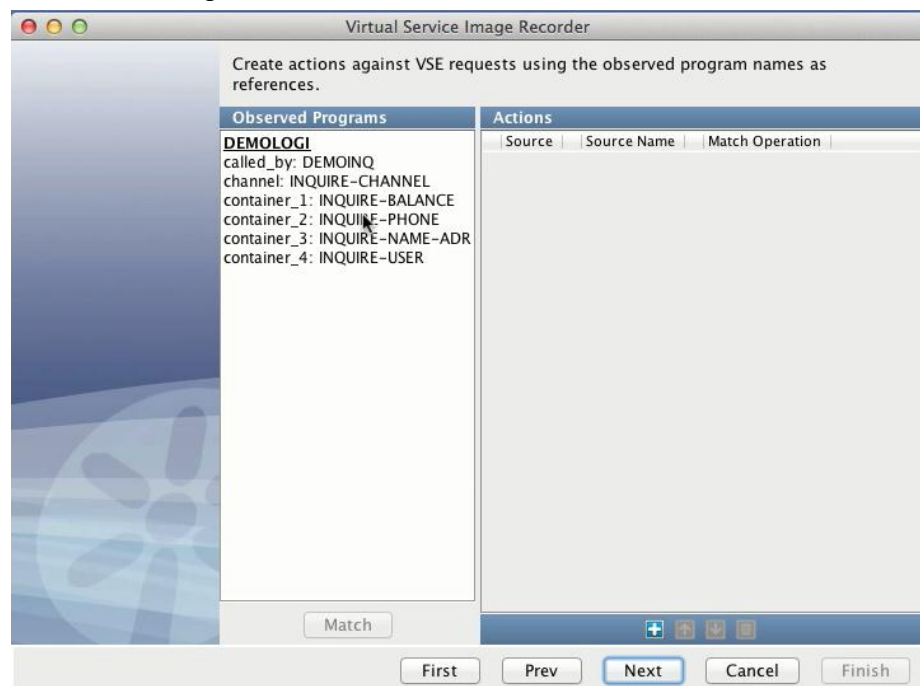
3. Enter an IP address and port, then click Next.

The Data Protocol selection window opens.

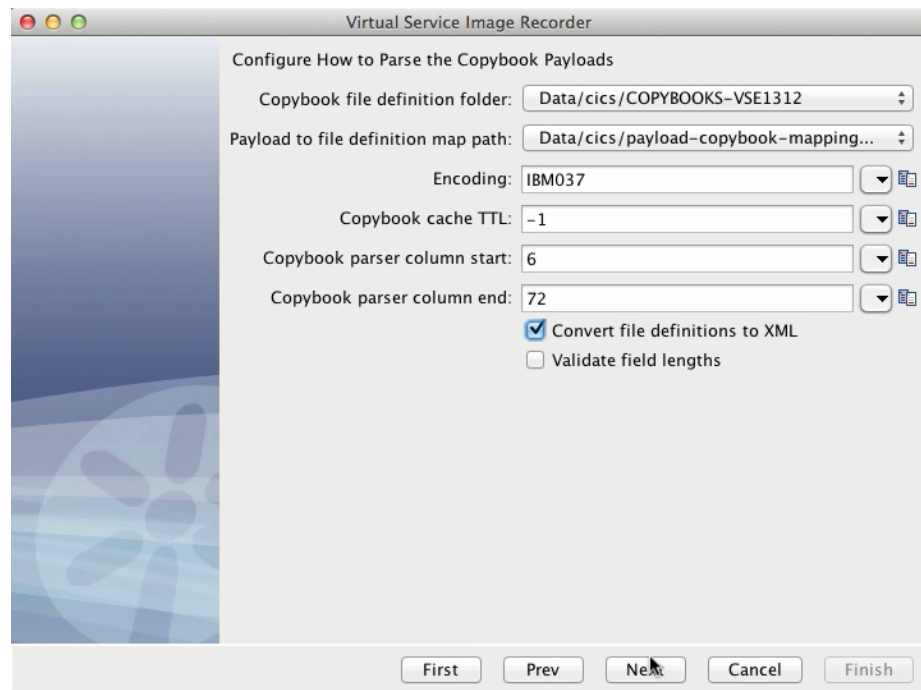
4. Select the CICS Request Data Access data protocol on the request side.

5. Select the CICS Container Copybook data protocol for both the request and response sides, then click Next.

The Observed Programs list contains the CICS containers:



6. Click Next.



7. Enter the following parameters on the request side definitions:

**Copybook file definition folder**

Specifies the folder for the copybook file definition.

**Payload to file definition map path**

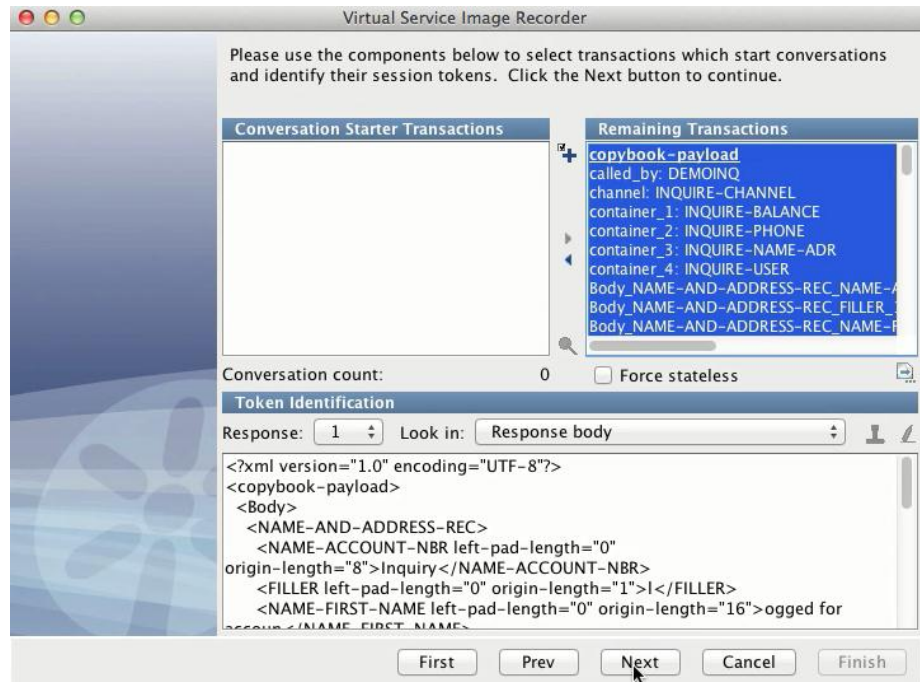
Specifies the path for the payload to file definition map.

**Encoding**

Specifies the encoding information.

8. Select Convert file definitions to XML and click Next.

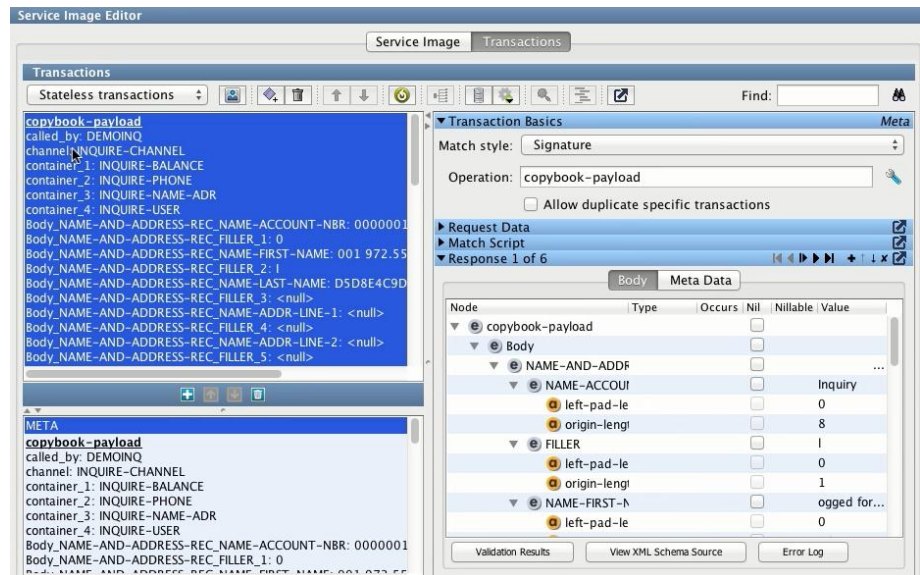




9. Click Next to complete the service image.

10. To view the service images, select View Service Image and click Finish.

The completed service image opens.



## Copybook

The Copybook data protocol allows VSE to convert copybook payloads to and from XML at run time in a way that is transparent to the caller. The ability to display these payloads in XML allows other (standard) VSE mechanisms to provide value. You can use the Generic XML Payload Parser to identify and select request arguments and operations for matching and transaction organization in the service image. Using a standard XML structure allows for magic strings to be processed in the responses so that dynamic data is created in a more usable way.

This section contains the following topics:

- [Terms](#) (see page 215)
- [Setup](#) (see page 215)
- [Usage](#) (see page 216)
- [Payload Mapping File](#) (see page 219)
- [Matching Logic](#) (see page 227)

## Terms

To better understand the Copybook data protocol, it is important to understand the following terms:

### **Copybook**

A hierarchical structure that is used to define the layout of data. The copybook identifies the field names, their lengths, and their relationships to each other.

### **Copybook File Definition**

A file in plain text format containing a copybook.

### **Field**

A single element in a copybook. A field encapsulates a name for, the length of, and the data type for a single logical piece of data in a record.

### **Payload**

A collection of one or more records that VSE identifies in either a single request or a single response.

### **Payload Mapping File**

Also known as a *Payload Copybook Mapping* or *Payload to File Definition Map*, a payload mapping file is an XML document describing how to identify the copybook or copybooks required to parse a payload. The payload mapping file provides the basic structure for the resulting XML.

### **Record**

A collection of data with no inherent structural elements. One or more fields making up a single copybook defines a record. A record may not contain data for every field a copybook defines, but the relationship between records and copybooks is always one-to-one. A specific record is defined only by a single copybook, and a single copybook defines only one record.

## Prerequisites for Creating a Copybook Virtual Service

To prepare for creating a copybook virtual service, complete the following tasks:

- Gather all of the copybooks that define the records in your payloads. These copybooks must be in a directory in your DevTest project. You can include whatever hierarchy you find helpful inside the directory you use. Typically, you would place the copybook file definition folder somewhere under the Data folder of the project.
- Create a [payload mapping file](#) (see page 219).
- Prepare your client for recording, gather request/response pairs, capture a PCAP file of the traffic, or use some other means of gathering the transactions to use for your virtual service.

## How to Use the Copybook Data Protocol

You can use the Copybook data protocol with any of the transport protocols that VSE supports. Commonly, the Copybook data protocol handler is used with a messaging protocol such as JMS or MQ or with the CICS transport protocol. A sample copybook application that uses HTTP is available in the demo server.

### Selection

Keep the following in mind when selecting the Copybook data protocol:

- Consider using the data protocol handler on both the Request and Response sides. Although you can use the data protocol on only one side, it is not common. This documentation does not cover the single-sided usage in detail.
- The Copybook data protocol changes the Request (or Response) body into XML. The Copybook data protocol does not automatically add arguments to the Request for every field. To do useful things with the Copybook data protocol, you also must include another data protocol handler on the Request side. Most commonly, the Generic XML Payload Parser is the second data protocol handler. However, any data protocol handler that does useful things with XML payloads is acceptable.

### Configuration

When you have captured your traffic (or imported from Request/Response pairs, PCAP, or raw traffic files), the Copybook DPH configuration window opens.

Enter the following parameters:

#### Copybook file definition folder

Designates the folder in your project where you store your copybook file definitions. This folder becomes the base path for relative paths in the Payload Mapping File.

#### Payload to file definition map path

Specifies the XML document that serves as the Payload Mapping File for this virtual service.

#### Encoding

Specifies a valid [Java charset](#). If provided, this value is used to try to convert the bytes in the payload into text for use in the output XML.

**Default:** UTF-8

To configure the default charset, set `lisa.vse.default.charset` in `local.properties`.

#### Copybook cache TTL

VSE must reference a specific copybook at run time and potentially convert it to XML. The Copybook cache TTL parameter defines how long a cached version of the converted copybook can be kept in memory. When the specified interval expires, the converted copybook is removed from the cache. If the file is needed again, it is read again and reconverted.

**Values:**

- To define the timeout, set Copybook cache TTL to a positive number (in seconds).

To disable caching, set Copybook cache TTL to 0 or a negative number. VSE reads and parses the files each time that they are needed.

**Copybook parser column start**

Copybooks frequently start each line with a line number. This parameter defines the column on which the parser starts when trying to parse a copybook file definition.

**Value:** A zero-based inclusive index. However, you can think of it as a "normal" one-based exclusive index.

**Default:** 6

**Example:** If you set this value to 6, the parser skips the first six characters in a line and starts with the seventh character.

**Copybook parser column end**

Occasionally, copybooks contain other reference data at the end of each line. When that happens, the parser must know on which column to stop. If there is no "extra" data at the end of the lines in the file, you can set this number to something greater than the length of the longest line in the file. If this number is greater than the length of a line, the parser stops at the end of the line.

**Value:** A zero-based exclusive index. However, you can think of it as a "normal" one-based inclusive index.

**Example:** If you set this value to 72, the parser reads the 72nd character in the line and then it stops (without trying to read the 73rd).

**Validate field lengths**

Specifies whether to enforce the values set in the Copybook parser column start and Copybook parser column end values. This option is only used in VSE, on the Response side. During recording, the payload is converted to XML and then back to bytes to ensure that it can convert symmetrically. Also, during playback the XML responses are converted back to records/payloads before responding to the caller. In both of those operations, VSE can validate that the value in each field is exactly the length that is specified in the copybook. However, it is not always desirable to have this validation.

**Values:**

- **Selected:** The records in your data align exactly with the length defined in your copybook.

- **Cleared:** The records in your data do not align exactly with the length defined in your copybook.

**Example:** if your record does not contain any data for some fields, they are seen as 0 length, whereas the copybook defines that field for a length greater than 0. If you select this option in that case, VSE fails the validation and reports it as an error.

#### XML elements as request arguments

Specifies whether to verify that the requests and responses are XML strings.

##### Values:

- **Selected:** Identifies variables from the XML messages that the recorder uses. This verifies that the requests and responses are XML strings.
- **Cleared:** Does not verify that the requests and responses are XML strings.

Verifies that the requests and responses are XML strings. Selecting this option lets you identify variables out of the XML messages that the recorder uses. For more detailed information about identifying variables, see [Generic XML Payload Parser](#) (see page 240).

The same editor is also available on the data protocol filters in the VSM (one on the Request side and one on the Response side). This editor lets you change the configuration after recording, as necessary.

## Payload Mapping File

The *payload mapping file* is an XML document that describes how incoming payloads can be matched to corresponding copybooks. The payload mapping file is also known as *payload copybook mapping* and *payload to copybook file definition map*. This file also provides hints for how to structure the resulting XML to provide clarity to the user.

### Sample

A sample mapping file can be found [here](#). The sample file contains examples of various configurations and comments describing each of the attributes on each node. This sample file is useful as a reference while creating your copybook mapping files.

### Structure

The following example shows the basic payload mapping file structure. For simplicity, this example contains no attributes or values.

```
<payloads>
  <payload>
    <key></key>
    <section>
      <copybook></copybook>
      ...
    </section>
    ...
  </payload>
  <payload>
    ...
  </payload>
</payloads>
```

<payloads>

The root XML node for the document. The node cannot be repeated.

<payload>

This element, in its entirety, fully describes a payload that matches it. When a payload is determined to match a <payload> element in this document, the <payload> element is used to describe the incoming payload.

<key>

(Optional) This option makes the XML document more readable. Everything that you specify on this element you can also specify as attributes on the <payload> element.

<section>

A logical grouping of copybooks that defines a portion of the payload. If there are multiple <section>s, they are processed in order with no repetition.

<copybook>

A single copybook that may or may not describe a record in the payload. A <section> can contain multiple <copybook> elements.

<payload>

A <payload> element can contain the following attributes:

```
<payload name="TEST" type="request" matchType="all" key="reqKey"
value="reqVal" keyStart="3" keyEnd="6" headerBytes="0"
footerBytes="0" saveHeaderFooter="false" definesResponse="false">
  ...
</payload>
```

Attribute	Required/Optional	Description
<b>name</b>	Required	Defines a unique name to identify the type of request described here. <b>Name</b> must be unique among the set of payloads of the same type (that is, request or response).
<b>type</b>	Required	Specifies the payload type. <b>Value:</b> One of the following: <ul style="list-style-type: none"><li>■ request</li><li>■ response.</li></ul>



<b>matchType</b>	Optional	<p>The matchType attribute applies differently to payload definitions of type "response." For example, responses do not contain arguments or operation names. If a payload definition of type "response" sets this attribute to <b>argument</b>, <b>attribute</b>, or <b>operation</b>, it will never match anything (unless the matching request sets the definesResponse attribute to <b>true</b>, which overrides this attribute). If a payload definition of type "response" sets this attribute to <b>all</b>, the argument, attribute, and operation phases of matching are skipped.</p> <p><b>Value:</b> One of the following:</p> <ul style="list-style-type: none"> <li>■ <b>argument:</b> Try to match on only the specified argument.</li> <li>■ <b>attribute:</b> Try to match only the specified attribute.</li> <li>■ <b>metaData:</b> Try to match only the specified metaData.</li> <li>■ <b>operation:</b> Try to match on only the operation name.</li> <li>■ <b>payload:</b> Try to match on the body of the request.</li> <li>■ <b>all:</b> Try to match in the following order: argument, attribute, metaData, operation, then payload.</li> </ul> <p><b>Default:</b> payload</p>
<b>key</b>	Required	<p>The behavior of this attribute depends as follows on the matchType:</p> <ul style="list-style-type: none"> <li>■ If matchType is <b>operation</b> or <b>payload</b>, the value of this attribute is used for matching.</li> <li>■ If matchType is <b>argument</b>, <b>attribute</b>, <b>metaData</b>, or <b>all</b>, the value of this attribute is assumed to be the key of an argument, attribute, or metaData entry. If the matchType value is <b>all</b>, the key value is NOT used for matching against the operationName or the payload body. The value attribute is used instead.</li> </ul>

<b>value</b>	Optional/Required	<p>The behavior of this attribute (and whether it is required) changes depends as follows on the matchType:</p> <ul style="list-style-type: none"> <li>■ If the matchType is <b>operation</b> or <b>payload</b>, the value of this attribute is ignored and can be excluded.</li> <li>■ If the matchType is <b>argument</b>, <b>attribute</b>, or <b>metaData</b>, this attribute is assumed to be the value of an argument, attribute, or metaData entry and is required for matching.</li> <li>■ If the matchType is <b>all</b>, this attribute is required and behaves as described previously during the argument, attribute, and metaData matching phases. During the operation and payload matching phases, however, the value of this attribute is used for matching (as opposed to using the value of the <b>key</b> attribute as is the case if the matchType is <b>operation</b> or <b>payload</b>).</li> </ul>
<b>keyStart</b>	Optional	<p>Designates the position where the search for the key should start in the payload (1-based index). The search is <b>inclusive</b> of this value.</p> <ul style="list-style-type: none"> <li>■ If you set keyStart to 1, the search starts with the first byte.</li> <li>■ If you do not set keyStart, the entire payload is searched and keyEnd is ignored.</li> <li>■ If you set keyStart to a number less than 1, the value is treated as if the value was 1.</li> <li>■ If you set keyStart to a number greater than the length of the payload, the entire payload is searched and keyEnd is ignored.</li> <li>■ If you set keyStart equal to keyEnd or greater than keyEnd, the entire payload is searched.</li> <li>■ If keyEnd minus keyStart is less than the length of the key, the entire payload is searched.</li> </ul>
<b>keyEnd</b>	Optional	<p>Designates the position where the search for the key should end in the payload (1-based index). The search is <b>exclusive</b> of this value.</p> <ul style="list-style-type: none"> <li>■ If you set keyEnd to 3, it searches only bytes 1 and 2.</li> <li>■ If you do not set keyEnd, the search starts at keyStart and ends at the end of the payload.</li> <li>■ If this value exceeds the length of the payload, the search stops at the end of the payload.</li> <li>■ If this value is equal to or less than keyStart, the entire payload is searched.</li> <li>■ If keyEnd minus keyStart is less than the length of the key, the entire payload is searched.</li> </ul>

<b>headerBytes</b>	Optional	Designates the number of bytes to strip from the beginning of the payload. This defaults to 0 if you do not provide a value. If the attribute is present, the value must be a valid integer.
<b>footerBytes</b>	Optional	Designates the number of bytes to strip from the end of the payload. This defaults to 0 if you do not provide a value. If the attribute is present, the value must be a valid integer.
<b>saveHeaderFooter</b>	Optional	<p>Specifies whether the header and footer bytes that were stripped are persisted in the XML version of the request as hex-encoded strings under the <b>rawHeader</b> and <b>rawFooter</b> tags.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"><li>■ <b>true:</b> Persists the stripped header and footer bytes.</li><li>■ <b>false:</b> Does not persist the stripped header and footer bytes.</li></ul> <p><b>Default value:</b> false.</p>
<b>definesResponse</b>	Optional	<p>If <b>true</b> then the response for this request will look for a payload element with an identical name but of type <b>response</b>. This attribute is ignored when the type is <b>response</b>.</p> <p><b>Default value:</b> false.</p>

You can use the optional `<key>` element to replace the key-related attributes. The example `<payload>` element that is shown previously could, optionally, be written as follows for readability:

```
<payload name="TEST" type="request" headerBytes="0" footerBytes="0"
saveHeaderFooter="false" definesResponse="false">
  <key matchType="all" value="reqVal" keyStart="3"
    keyEnd="6">reqKey</key>
  ...
</payload>
```

`<section>`

**Example:** The following example is a sample section element with all attributes on it and a description of the attributes.

```
...
<section name="Body">
  ...
</section>
...
```

Attribute	Required	Description
<b>name</b>	Required	Defines the name of a grouping XML element in the XML output version of the payload. One or more converted copybook elements will be present beneath it.

<copybook>

**Example:** The following example is a sample copybook element with all attributes on it and a description of the attributes.

```
...
<copybook key="cpk" order="1" max="1" name="TESTRECORD"
length-field="SOME-ID">TESTIN.CPY.TXT</copybook>
...
```

Attribute	Required	Description
<b>key</b>	Required	Defines the unique string in the record that identifies the copybook. Technically, this attribute is optional. However, if a key is not provided, it means that that copybook is the only one that will ever get applied to the payload. It will be applied over and over until the payload runs out of bytes. If multiple copybook elements have no key, then the first one will always be used, unless the max attribute is specified.
<b>order</b>	Optional	Defines a hint as to the order in which the records are found in the payload. The numbers used are irrelevant, but "later" records in the payload should use a larger integer. Multiple copybooks can be tagged with the same order number, meaning that those records could be in any order. When a record has been found with a specific order number, subsequent searches only search for copybooks with that order number and greater. You can include copybooks in a group that will never match against the payload. They are just ignored. However, this affects performance because each copybook has to be checked. <b>Default:</b> 0
<b>max</b>	Optional	Defines the maximum times that a copybook can be applied to the payload. Blank values, 0, negative numbers, non-numbers, and non-existent values all mean "no limit".
<b>name</b>	Optional	Defines a value to override the record name (that is, the root level in the copybook). <ul style="list-style-type: none"> <li>■ If you set this value, the generated node in the XML for this copybook uses this name instead of the record name from the copybook definition.</li> <li>■ If you do not set this value, the default is to look up the record name from the copybook definition and use that.</li> <li>■ If you set this value to the record name from the copybook definition, the only effect is on readability of this file.</li> </ul>

<b>length-field</b>	Optional	<p>Defines how to split the payload so that the next record search begins in the correct place.</p> <p>If this attribute is not present, the processor attempts to determine the length of the copybook from the definition. If, for some reason, it cannot figure out the length, the processor assumes that the rest of the payload applies to this copybook, and ends processing after applying this copybook. The processor ignores this field if it is not an unsigned Display numeric field.</p>
---------------------	----------	--

## Matching Logic

You can determine how the matching logic works based on the descriptions of the XML elements and arguments. However, to provide a clearer picture, this section explores matching fully.

When VSE receives a payload, matching occurs in the following phases:

- Payload
- Section
- Copybook
- Finalizing

### Payload

Payload is the first phase of matching that happens when VSE receives a new payload. This phase determines which <payload> element from the payload mapping file corresponds to the payload received. VSE tries to match a <payload> element by processing the elements in the payload mapping file in order, from top to bottom. The first match wins.

**Note:** If no payload elements match the payload, it is treated as an "unknown payload." The content is HEX encoded, and it is wrapped in a generic XML structure. If necessary, unknown payloads are automatically converted back to bytes during playback.

For the Request payloads:

1. If the type attribute of this element is "request," proceed. Otherwise, this element does not match.
2. If the matchType is argument, look for a request argument with a key that matches the key attribute from this element and a value that matches the value attribute from this element. If one is found, this payload element matches.
3. If the matchType is attribute, look for a request attribute with a key that matches the key attribute from this element and with a value that matches the value attribute from this element. If one is found, this payload element matches.
4. If the matchType is metaData, look for a request metaData entry with a key that matches the key attribute from this element and with a value that matches the value attribute from this element. If one is found, this payload element matches.
5. If the matchType is operation, verify whether the operation name for the request matches the key attribute from this element. If so, this payload element matches.
6. If the matchType is payload, search in the boundary that is specified by this element's keyStart and keyEnd attributes for the value in the key attribute. If the key is found in those bounds, the payload element matches.

7. If the matchType is all, Steps 2 through 6 are processed in order, stopping when a match is found. The only variation is that in Steps 5 and 6, the value attribute is used in place of the key attribute.

For the Response payloads (during recording):

1. If the previously seen request payload element set the definesResponse attribute to **true**, immediately return the payload element with the same name as the request element, but with a type of response. If no such element is found, there is no match.
2. If the previously seen request payload element did *not* set the definesResponse attribute to true:
  - a. If the type attribute of this element is response, proceed. Otherwise, this element does not match.
  - b. If the matchType is metaData, look for a response metaData entry with a key that matches the key attribute from this element and with a value that matches the value attribute from this element. If one is found, this payload element matches.
  - c. If the matchType is payload, search in the boundary that is specified by this element's keyStart and keyEnd attributes for the value in the key attribute. If the key is found in those bounds, the payload element matches.
  - d. If the matchType is all, complete Steps b and c in order, stopping when a match is found. The only variation is that in Step c, the value attribute is used in place of the key attribute.

For the Response payloads (during playback):

During playback, no matching is done for responses. Instead, during recording, whatever match was determined is saved in the Metadata of the Response object. Then, during playback, when that Response object is returned, the processor picks up the value from the Response Metadata, finds the payload element with that name (and type="response") and uses it. If no such element exists, it is considered an unrecoverable error.

## Section

After the payload element is identified, the processor reviews each section element in order, from top to bottom. No matching is done at this level. After the processor has fully processed a section (that is, when none of the copybooks in the section match), it proceeds to the next section and does not revisit previous sections.

## Copybook



The final phase of matching is to determine which copybook in this section applies first, second, and so on, until all the records in the payload are processed. This phase is the most complex matching phase. The process is:

1. The processor reviews all of the copybook elements in the section and sorts them by the number that is specified in the order attributes.
2. The list of copybook elements with the lowest order number is checked first (in the order the file specifies them). The remaining payload is searched for the value in the key attribute for this copybook element and the index where it is found (if at all) is saved. The remaining payload is searched for the value in the key attribute for this copybook element and the index where it is found (if at all) is saved.
3. After all copybooks in the lowest order list have been checked, if any matched, the one found earliest in the payload is used.
4. If a copybook matched:
  - a. If the matching copybook has matched this payload previously (an earlier record in the payload), the max attribute is checked to see if it can be applied again. If the max has been reached, this is not considered a match and the next matching copybook is used.
  - b. If the max has not been reached, this copybook is applied to the payload and the number of bytes specified by the copybook is consumed. If there is a length-field attribute on the copybook, the value of that field in the record is used to determine how many bytes to consume and the processor returns to Step 2.
5. If a copybook did not match, the list of copybooks with the next lowest order number is checked using the same rules as Steps 2, 3, and 4. After the processor decides that no copybooks in an order number list match, it does not try to process that order number list (for this section) again for this payload.
6. After all lists are checked and no matches are found (or the payload runs out of bytes), the processor proceeds to the next section.

### Finalizing

If bytes are left over in the payload after all sections for a payload have been processed, they are truncated.

## CTG Copybook

The CTG Copybook data protocol splits the recorded request into its respective container chunks. The processor then sends each chunk to the Copybook data protocol and aggregates it with the corresponding XML. The resulting XML is the aggregated XML of the containers.

This example demonstrates the CTG Copybook data protocol usage.

**Follow these steps:**

1. Complete the required fields on the Virtual Service Image Recorder [Basics](#) (see page 120) tab.

2. Click Next.

3. Enter an IP address and port, then click Next.

The Data Protocol selection window opens.

4. Select the CTG Copybook data protocol for both the request and response sides and click Next.

5. Click Next.

The screenshot shows a window titled "Virtual Service Image Recorder" with a sub-header "Configure How to Parse the Copybook Payloads". The window contains several configuration fields:

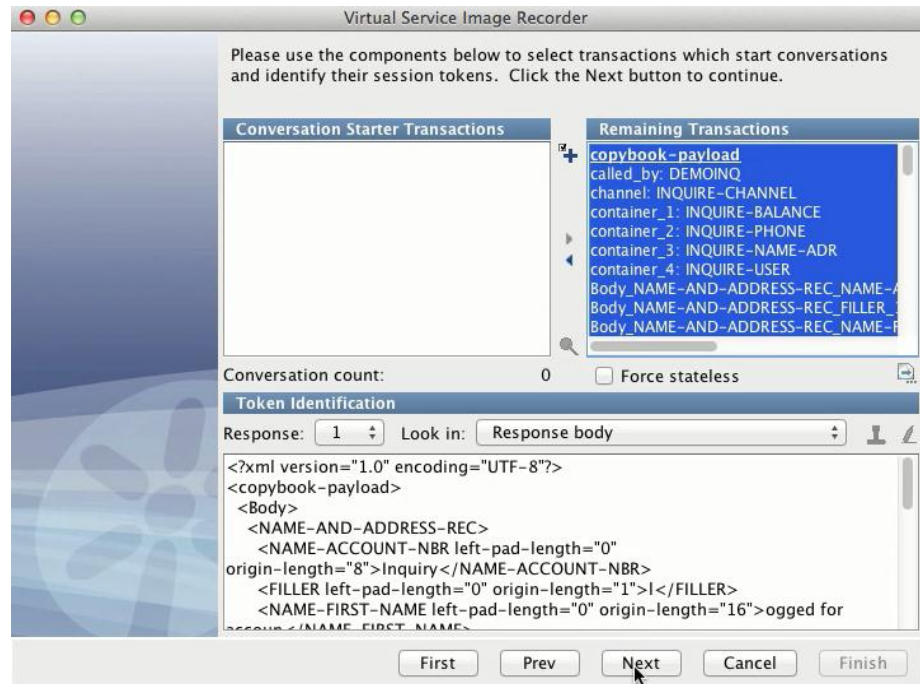
- Copybook file definition folder: Data/cics/COPYBOOKS-VSE1312
- Payload to file definition map path: Data/cics/payload-copybook-mapping...
- Encoding: IBM037
- Copybook cache TTL: -1
- Copybook parser column start: 6
- Copybook parser column end: 72
- ☒ Convert file definitions to XML
- ☐ Validate field lengths

At the bottom of the window, there are five buttons: "First", "Prev", "Next", "Cancel", and "Finish". The "Next" button is highlighted with a mouse cursor.

6. Enter the following parameters on the request-side definitions:

- The folder for the copybook file definition.
- The path for the payload to file definition map.
- The encoding information.

7. Select Convert file definitions to XML and click Next.



8. Click Next to complete the service image.
  9. To view the service images, select View Service Image and click Finish.
- The completed service image opens.

## Data Desensitizer

In cases where the transport layer cannot "see" the data to desensitize, VSE provides a data protocol handler. For example, consider the use of gzip in SOAP over HTTP or messaging. If VSE is given a gzipped SOAP request (or response), then data desensitization cannot be done at this level. In that case, the recorder can be configured with a data protocol to perform the necessary gunzip operation. To handle desensitization after the SOAP body has been converted to plain text, add the data desensitization protocol.

VSE typically tries to desensitize data at the lowest possible level. The Desensitize check box on the first window of the VSE recording wizard requests that the transport protocol try to desensitize as soon as possible. However, the payload for a request is sometimes opaque to the transport protocol. Consider HTTP messages in which the payload is compressed, such as with gzip or FastInfoSet. In this case, the transport protocol cannot apply the desensitization rules. This data desensitizer data protocol is built to address such situations.

Add the data desensitizer data protocol to the request side, response side, or both of a data protocol chain. Add the protocol after another protocol that converts the opaque payload into something that is readable. This data protocol is only intended for recording, not playback. As each request or response is presented to the protocol handler, all desensitization rules that are specified in **LISA\_HOME\desensitize.xml** are applied.

This protocol requires no configuration information and so does not present a window in the recording wizard.

## Delimited Text

The Delimited Text data protocol lets you parse delimited textual data into arguments and values. This protocol takes the body of the request or response, parses it, and then replaces the body with an XML representation of the parsed data. For example:

```
"name1=val1;name2=val2"
```

becomes:

```
<name1>val1</name1><name2>val2</name2>
```

### Follow these steps:

1. Complete the [Basics](#) (see page 120) tab of the Virtual Service Recorder.
2. Select the Delimited Text Data Protocol on both the request and response sides.
3. After the recording completes, select the format for your delimited text request:

#### Name/Value Pairs

Defines the delimiter between the name/value pairs and the delimiter between the name and value. For example, for the following data:

```
"name1=val1,name2=val2,name3=val3"
```

the delimiter between the pairs is a comma, and the delimiter between the name and value is an equal sign.

#### List of Values

Defines the delimiter between the values. You can use two types of lists:

- List of values separated by a textual delimiter.

For example, for the following data:

```
"val1,val2,val3,val4"
```

the delimiter is a comma. The parameters are named positionally.

- List of values separated by new line characters, such as a carriage return or line feed.

The parameters are named positionally.

#### Fixed Width

Defines (as a whole number) the width of the data field. The parameters are named positionally.

#### RegEx Delimited

Defines a regular expression that locates the values. For example, for the following data:

```
"xxxx123xxx456xx789"
```

a RegEx of `\d\d\d` finds “123”, “456”, and “789” as values and discard the rest. The parameters are named positionally.

#### Line Delimited

4. Complete the following fields.

#### Field Names Path

Defines the location of a field names document, which is a line-delimited document that specifies an ordered list of the names of fields. By default, the fields are named **value1**, **value2**, **value3**, and so on. To specify different names for those XML elements, use a field names document.

#### Delimiter Type

Defines the delimiter type used to separate name/value pairs and lists of values. Text or hex delimiters are automatically selected based the specified delimiter.

**Values:** Any alphanumeric characters, and the following:

- `\r` (carriage return)
- `\n` (newline)
- `\t` (tab)

You can also use hexadecimal notation to specify delimiters.

**Note:** You can only use delimiters that are valid XML 1.0. Specifying non-printable control characters makes the service image unusable.

#### XML Elements as request arguments

Specifies whether to add parameters and associated values to requests as arguments.

**Values:**

- **Selected:** Automatically adds the parameters and associated values to the request as arguments.
- **Cleared:** You must use the [Generic XML Payload Parser](#) (see page 240) (or a similar protocol) to select which values become arguments.

**Note:** This check box has no effect when the Delimited Text data protocol is used on the response side.

5. Click Next.

The name/value pairs are represented in XML. Here, you can double-click a transaction to show the contents of that transaction.

6. Configure the delimiters for the response side in the same way you did for the request side.

You can see the virtual service image and the payload that is converted to XML when the processing completes.

**More Information:**

[Generic XML Payload Parser Data Protocol](#) (see page 240)

**DRDA**

The DRDA data protocol converts binary Distributed Relational Database Architecture (DRDA) payloads to XML during recording. This protocol facilitates alignment with native DevTest functionality, readability, and dynamic data support. Responses are converted back to their native format on playback.

When you select the DRDA data protocol, the Request-side and Response-side data protocols fields are automatically populated with the DRDA data protocol selections.

The next window of the recorder lets you specify communications information.

Enter the communications parameters for the service image.

**Note:** If the payload contains non-ASCII characters, the payload is displayed in binary in the Service Image Editor.

## EDI X12 Data Protocol

The EDI X12 data protocol transforms ANSI X12 EDI documents into an XML representation in the body of the request. The DPH also creates a VSE request operation consisting of the EDI document type (for example, 835) combined with the specific document version (for example, 004010) and a series of VSE request arguments. These request arguments are the flattened representation of the XML body. They are formed by combining the XML elements, which are separated by `_` for tags or `@` for each tag attribute for the argument name and the element value for its value.

For example, if the following ANSI X12 EDI 850 document:

```
ISA*00*  *00*  *ZZ*0011223456  *ZZ*999999999
*990320*0157*U*00300*000000015*0*P*~$
GS*PO*0011223456*999999999*950120*0147*5*X*003040$
ST*850*000000001$
BEG*00*SA*95018017***950118$
N1*SE*UNIVERSAL WIDGETS$
N3*375 PLYMOUTH PARK*SUITE 205$
N4*IRVING*TX*75061$
N1*ST*JIT MANUFACTURING$
N3*BUILDING 3B*2001 ENTERPRISE PARK$
N4*JUAREZ*CH**MEX$
N1*AK*JIT MANUFACTURING$
N3*400 INDUSTRIAL PARKWAY$
N4*INDUSTRIAL AIRPORT*KS*66030$
N1*BT*JIT MANUFACTURING$
N2*ACCOUNTS PAYABLE DEPARTMENT$
N3*400 INDUSTRIAL PARKWAY$
N4*INDUSTRIAL AIRPORT*KS*66030$
P01*001*4*EA*330*TE*IN*525*VN*X357-W2$
PID*F****HIGH PERFORMANCE WIDGET$
SCH*4*EA****002*950322$
CTT*1*1$
SE*20*000000001$
GE*1*5$
```

the resulting VSE request object contains:

**Operation:** 850-00340

**Arguments:** A subset of these name/value pairs follows:

```
interchange_sender_address <null>
interchange_sender_address@Id    0011223456
interchange_sender_address@Qual  ZZ
interchange_receiver_address <null>
interchange_receiver_address@Id  999999999
interchange_receiver_address@Qual ZZ
```



```
interchange_group_transaction_segment_element_1 00
...
```

**Body:** Structured as follows. The XML document has been formatted in this document for readability. The request body does not contain formatting elements such as line ends and indentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<ediroot>
  <interchange Standard="ANSI X.12" Date="990320" Time="0157"
StandardsId="U" Version="00300"
  Control="000000015">
    <sender>
      <address Id="0011223456 " Qual="ZZ"/>
    </sender>
    <receiver>
      <address Id="999999999 " Qual="ZZ"/>
    </receiver>
    <group GroupType="P0" ApplSender="0011223456"
ApplReceiver="999999999" Date="950120"
      Time="0147" Control="5" StandardCode="X"
StandardVersion="003040">
      <transaction DocType="850" Name="Purchase Order"
Control="000000001">
        <segment Id="BEG">
          <element Id="BEG01">00</element>
          <element Id="BEG02">SA</element>
          <element Id="BEG03">95018017</element>
          <element Id="BEG06">950118</element>
        </segment>
        <loop Id="N1">
          <segment Id="N1">
            <element Id="N101">SE</element>
            <element Id="N102">UNIVERSAL
WIDGETS</element>
          </segment>
          <segment Id="N3">
            <element Id="N301">375 PLYMOUTH
PARK</element>
            <element Id="N302">SUITE 205</element>
          </segment>
          <segment Id="N4">
            <element Id="N401">IRVING</element>
            <element Id="N402">TX</element>
            <element Id="N403">75061</element>
          </segment>
        </loop>
        <loop Id="N1">
          <segment Id="N1">
            <element Id="N101">ST</element>
```

```

                                <element Id="N102">JIT
MANUFACTURING</element>
                                </segment>
                                <segment Id="N3">
                                    <element Id="N301">BUILDING 3B</element>
                                    <element Id="N302">2001 ENTERPRISE
PARK</element>
                                </segment>
                                <segment Id="N4">
                                    <element Id="N401">JUAREZ</element>
                                    <element Id="N402">CH</element>
                                    <element Id="N404">MEX</element>
                                </segment>
                                </loop>
                                <loop Id="N1">
                                    <segment Id="N1">
                                        <element Id="N101">AK</element>
                                        <element Id="N102">JIT
MANUFACTURING</element>
                                    </segment>
                                    <segment Id="N3">
                                        <element Id="N301">400 INDUSTRIAL
PARKWAY</element>
                                    </segment>
                                    <segment Id="N4">
                                        <element Id="N401">INDUSTRIAL
AIRPORT</element>
                                        <element Id="N402">KS</element>
                                        <element Id="N403">66030</element>
                                    </segment>
                                </loop>
                                <loop Id="N1">
                                    <segment Id="N1">
                                        <element Id="N101">BT</element>
                                        <element Id="N102">JIT
MANUFACTURING</element>
                                    </segment>
                                    <segment Id="N2">
                                        <element Id="N201">ACCOUNTS PAYABLE
DEPARTMENT</element>
                                    </segment>
                                    <segment Id="N3">
                                        <element Id="N301">400 INDUSTRIAL
PARKWAY</element>
                                    </segment>
                                    <segment Id="N4">
                                        <element Id="N401">INDUSTRIAL
AIRPORT</element>
                                        <element Id="N402">KS</element>

```

```

        <element Id="N403">66030</element>
    </segment>
</loop>
<loop Id="P01">
    <segment Id="P01">
        <element Id="P0101">001</element>
        <element Id="P0102">4</element>
        <element Id="P0103">EA</element>
        <element Id="P0104">330</element>
        <element Id="P0105">TE</element>
        <element Id="P0106">IN</element>
        <element Id="P0107">525</element>
        <element Id="P0108">VN</element>
        <element Id="P0109">X357-W2</element>
    </segment>
    <loop Id="PID">
        <segment Id="PID">
            <element Id="PID01">F</element>
            <element Id="PID05">HIGH PERFORMANCE
WIDGET</element>
        </segment>
    </loop>
    <loop Id="SCH">
        <segment Id="SCH">
            <element Id="SCH01">4</element>
            <element Id="SCH02">EA</element>
            <element Id="SCH06">002</element>
            <element Id="SCH07">950322</element>
        </segment>
    </loop>
</loop>
<loop Id="CTT">
    <segment Id="CTT">
        <element Id="CTT01">1</element>
        <element Id="CTT02">1</element>
    </segment>
</loop>
</transaction>
</group>
</interchange>
</ediroot>

```

## Generic XML Payload Parser Data Protocol

The Generic XML Payload Parser identifies that the requests and responses are XML strings. Using this protocol, you can identify variables out of the XML messages that the recorder uses.

**Note:** For the [Delimited Text Data Protocol](#) (see page 233), [Copybook Data Protocol](#) (see page 214), and [DRDA Data Protocol](#) (see page 235), this functionality is enabled by selecting the XML Elements as request arguments check box on the data protocol configuration windows.

### Identifying Conversations and Transactions

The quality of the recorded service image is directly dependent on how much information VSE has to identify the conversations and the individual transactions in them. Specifically, VSE needs help differentiating the transactions from each other:

- **As part of a conversation:** Identifying some unique ID representing a session.
- **As an individual operation:** Identifying some information specific to the semantics of the operation. For example, distinguishing an "order creation" operation from an "order listing" operation.

VSE internally knows many patterns for which to search. For example, for HTTP virtualization (if the server is a Java server) VSE sets a cookie that contains the value of a special variable that is named as sessionid. The sessionid variable uniquely identifies the session. VSE can use the variable to distinguish different sessions.

However, VSE sometimes needs further help, which can be supplied as follows:

- In the HTTP recording, VSE lets you identify tokens.
- For the JMS protocol, you could identify whether VSE uses a JMS correlation ID, custom JMS headers, or all JMS headers for this identification.
- VSE lets you specify a dynamic data protocol that lets you get meaningful information from the message itself. The following section describes this technique.

Using the Generic XML Payload Parser is a technique to help VSE inspect the body of the recorded messages (payload) and extract meaningful information from them to help identify the transactions. This technique can be the only way to get meaningful conversation information, especially for opaque protocols like the WebSphere MQ native protocol.

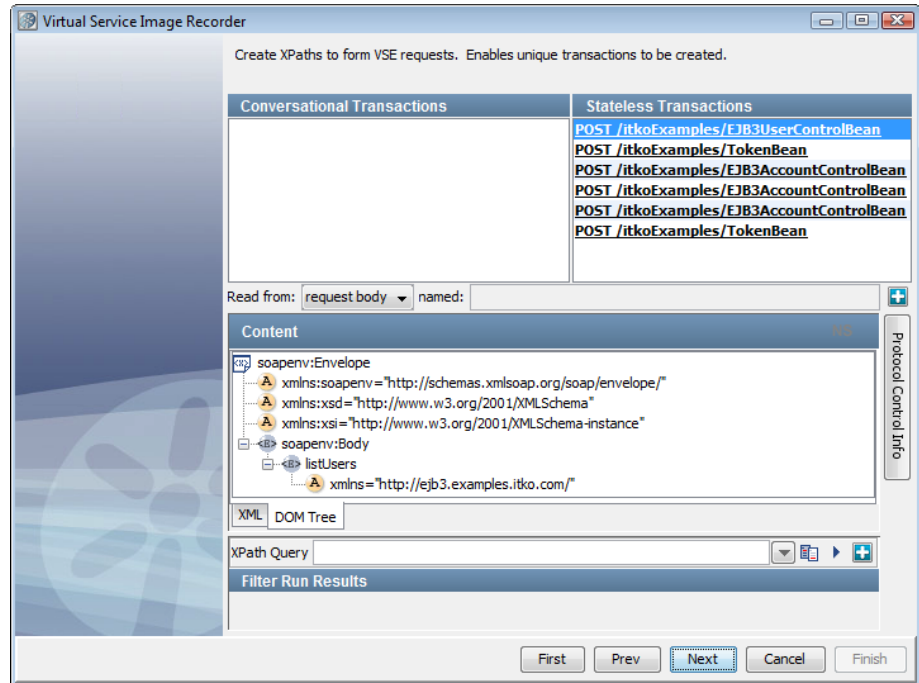
**Note:** When using both the Generic XML Payload Parser and the Delimited Text data protocols, add the Delimited Text data protocol before the Generic XML Payload Parser. Otherwise, the request never appears as parsed in the recorder.

#### Follow these steps:

1. To use a dynamic data protocol, in the Data Protocols tab of the VSI Recorder, select Generic XML Payload Parser, if the payload is a well-formed XML.

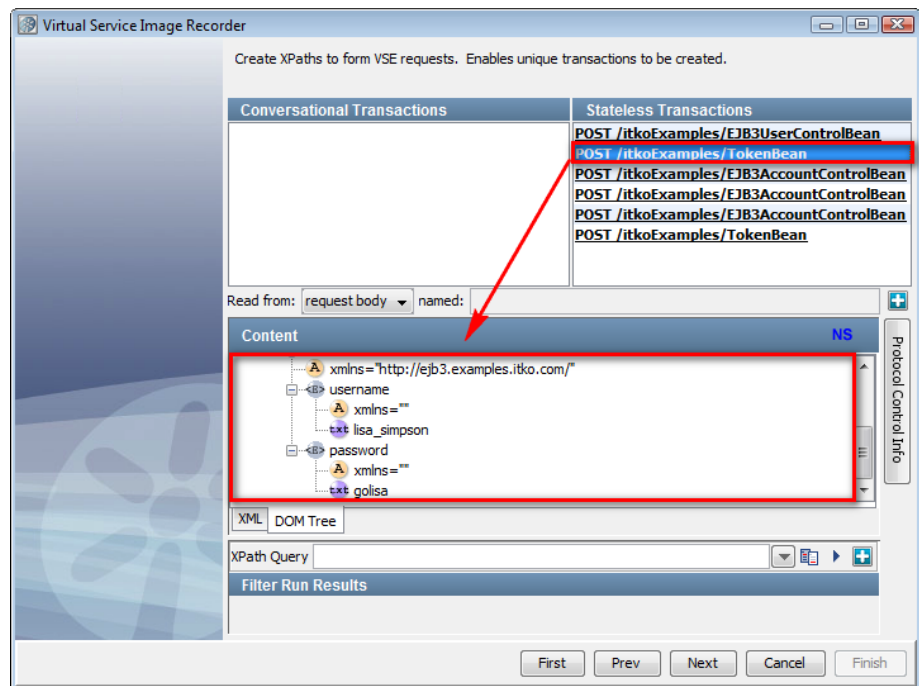
- Go through the rest of the steps, up to the cleanup step.

After the cleanup step, the following window opens:




By default, no starter transactions are identified, so DevTest does not know which data to review to identify the conversations. However, the Other Transactions list shows the recorded transactions.

- To see the XML payload in the Content area, click the first transaction.




The XML tab in the Content area shows the classic XML view. The DOM Tree tab shows the payload in the tree format.

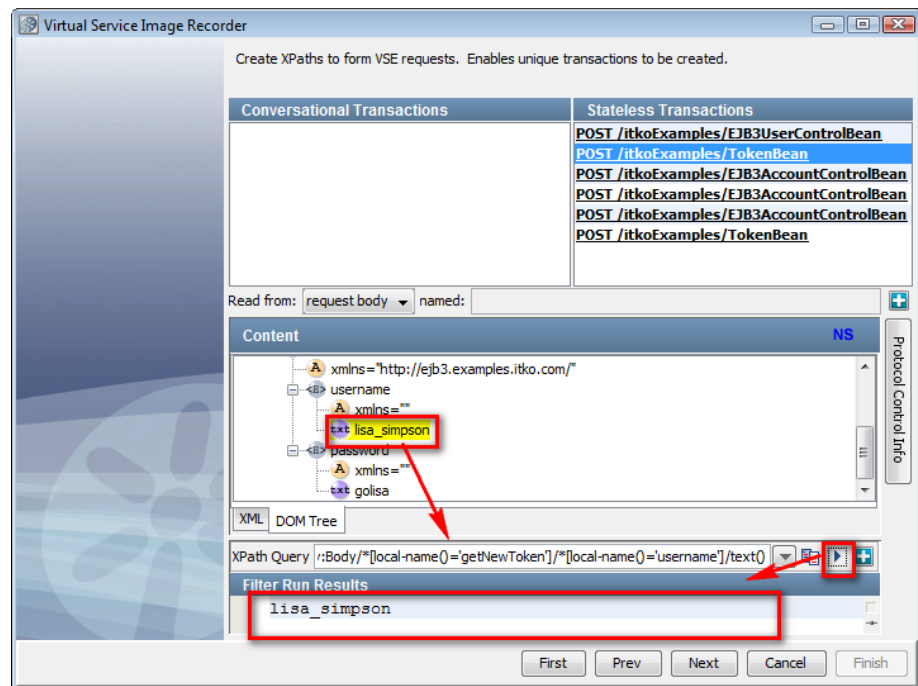
**Note:** This portion of the window is similar to the panel that appears when you create an XML XPath filter. As described in "XML XPath Filter" in *Using*, you can use the `lisa.xml.xpath.computeXPath.alwaysUseLocalName` property to ignore the namespace.



4. To identify a specific node as a parameter for VSE, click the node on the DOM Tree tab. The XPath Query box displays the query that corresponds to the selected node. To evaluate the XPath query on the current payload, click .

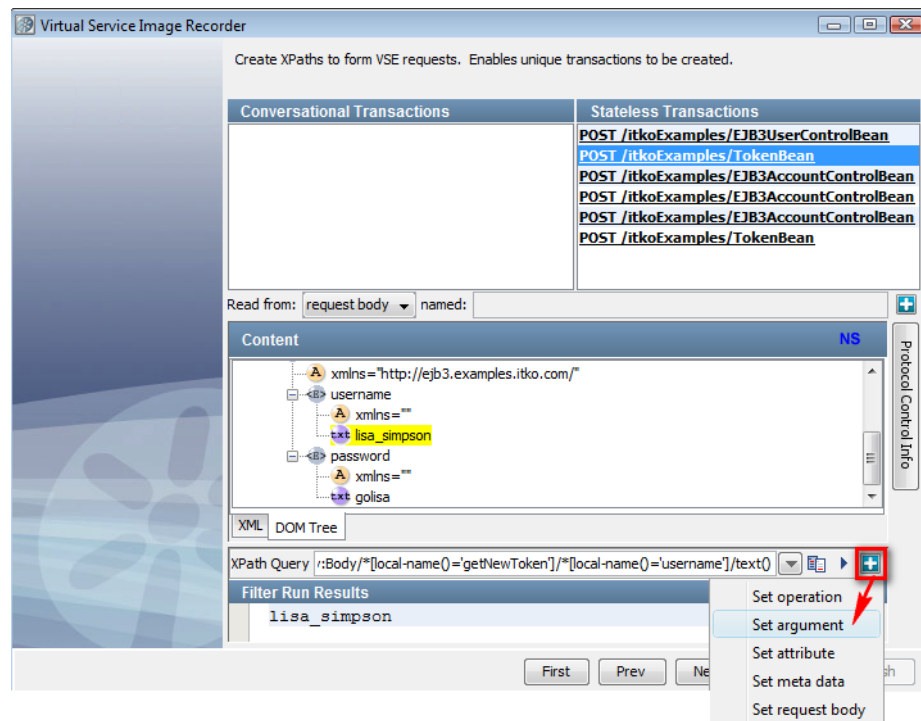
The result of the evaluation is shown under the heading Filter Run Results.

5. To add this XPath query to the parameter list, click Add .

A menu displays where you can set an operation, an argument, a meta parameter, or the request body. For example, use the menu to select an embedded SOAP document to the request body so that the WS SOAP protocol can be used to parse the document.



6. Click Protocol Control Info. DevTest remembered the XPath string that you identified and gave it an argument name. You can rename the argument. Also from this panel, use the Save  and Restore  buttons to save your list of XPath strings to a file, or to load a list of XPath strings from a file. By saving and loading, you can easily copy and paste from one XML Payload Parser to another.



Likewise, you can pull other variables from this transaction or others. It is not required for all transactions to have the specific argument. At the time of processing, if a specific argument is not present in the payload then it is ignored.

If you scroll to the right on the Protocol Control Info panel, you see an NS column in which you can add or edit namespace definitions. To locate and import an XML file from the file system and use the namespace definitions from that file, use the Browse button.

7. Double-click a transaction in the XML tab and you can see a dialog showing the content of the transaction.
8. Click Next when you are satisfied with your choice of variables. You are then directed to the post-processing window.



## JSON Data Protocol

The JSON data protocol is used to convert JSON data to an XML equivalent and to convert XML data to JSON format.

For example, the data protocol converts the following JSON data:

```
[
  {
    "organizationType": "W",
    "parentOrganizationType": "S",
    "parentUnitNbr": 777,
    "positionName": "S",
    "positionType": "S",
    "positionTypeId": 56,
    "unitNbr": 433,
    "l": [
      {
        "Roles": [
          "P", "P1", "P2", "S", "C", "AC", "ACF", "ACM"
        ]
      }
    ],
    "groupKey": "P",
    "groupName": "P"
  }
]
```

to the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <element class="object">
    <groupKey type="string">P</groupKey>
    <groupName type="string">P</groupName>
    <l class="array">
      <element class="object">
        <Roles class="array">
          <element
type="string">P</element>
          <element
type="string">P1</element>
          <element
type="string">P2</element>
          <element
type="string">S</element>
          <element
type="string">C</element>
          <element
type="string">AC</element>
          <element
type="string">ACF</element>
          <element
type="string">ACM</element>
        </Roles>
      </element>
    </l>
  </element>
</root>
```

```
        </element>
    </l>
    <organizationType type="string">W</organizationType>
    <parentOrganizationType
type="string">S</parentOrganizationType>
    <parentUnitNbr type="number">777</parentUnitNbr>
    <positionName type="string">S</positionName>
    <positionType type="string">S</positionType>
    <positionTypeId type="number">56</positionTypeId>
    <unitNbr type="number">433</unitNbr>
    </element>
</root>
```

If the JSON contains a one-element array, the data protocol converts it to an object with "element" as the key. For example, the data protocol converts the following JSON data:

```
[{"question1":"answer1","question2":"answer2"}]
```

into the following XML:

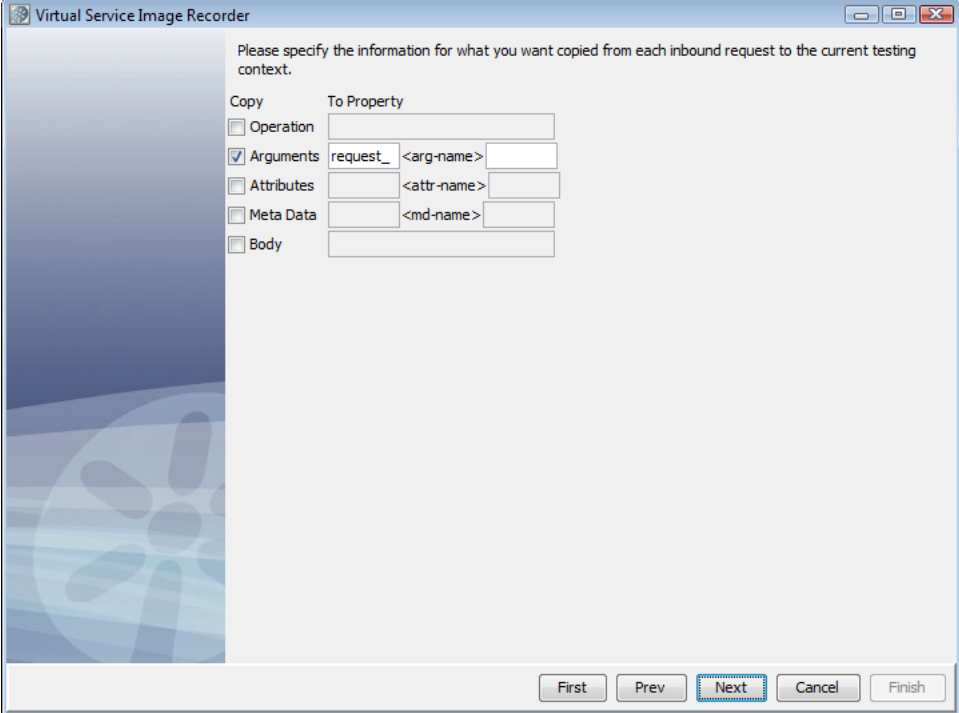
```
{"element":{"question1":"answer1","question2":"answer2"}}
```

The JSON data protocol handler reorders the key/value pairs in an object so they are alphabetical by key name.

**Note:** Recording JSON works properly if the application type is set correctly to application/json, text/json, or text/javascript. If the wrong application type is set, recording fails.

## Request Data Copier Data Protocol

The Request Data Copier lets you copy data from the current inbound request into the current testing context. This capability is useful to more easily examine request information when custom behavior is required before the VSE response lookup step in the VS model.



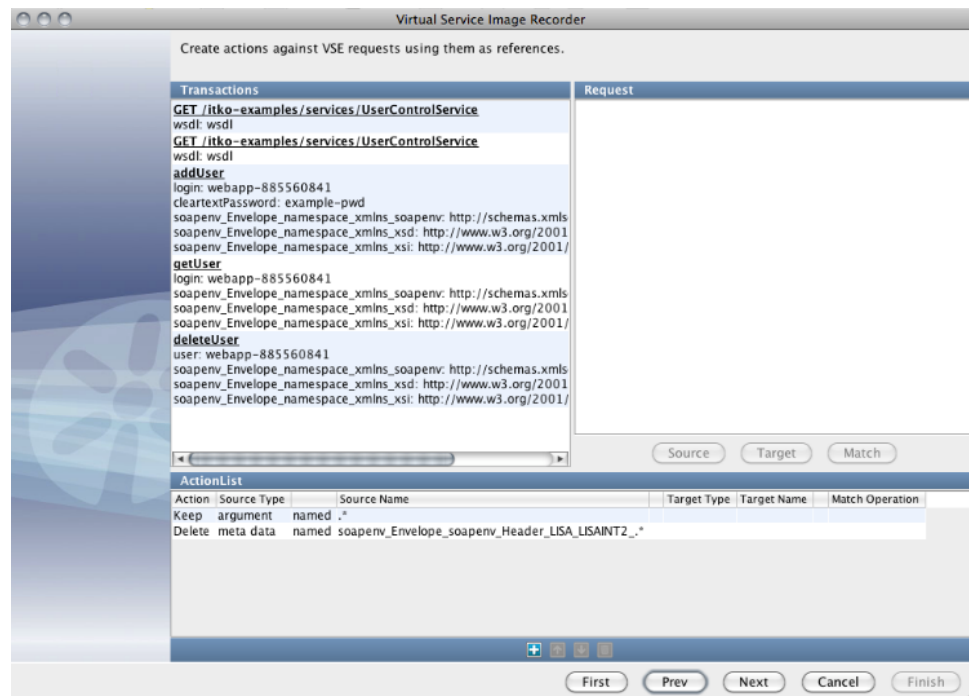
The screenshot shows a window titled "Virtual Service Image Recorder". Inside, there is a text prompt: "Please specify the information for what you want copied from each inbound request to the current testing context." Below this, there is a table with two columns: "Copy" and "To Property".

Copy	To Property
<input type="checkbox"/> Operation	
<input checked="" type="checkbox"/> Arguments	request_ <arg-name> <input type="text"/>
<input type="checkbox"/> Attributes	<attr-name> <input type="text"/>
<input type="checkbox"/> Meta Data	<md-name> <input type="text"/>
<input type="checkbox"/> Body	<input type="text"/>

At the bottom of the window, there are five buttons: "First", "Prev", "Next", "Cancel", and "Finish". The "Next" button is highlighted with a blue border.

## Request Data Manager Data Protocol

On the Data Protocols window, select Request Data Manager for a data protocol. When your recording is complete, the following window opens:



The Request Data Manager protocol lets you alter VSE requests during recording or playback.

Fundamentally, this protocol lets you apply a list of actions against a request. You can add the following actions in the ActionList section of the window:

**Copy**

Copies a piece of data in the request to another part of the request.

**Move**

Moves a piece of data in the request to another part of the request.

**Delete**

Deletes (or clears) a piece of data from the request.

**Keep**

Keeps a piece of data in a request while deleting anything else in that group for the data value.

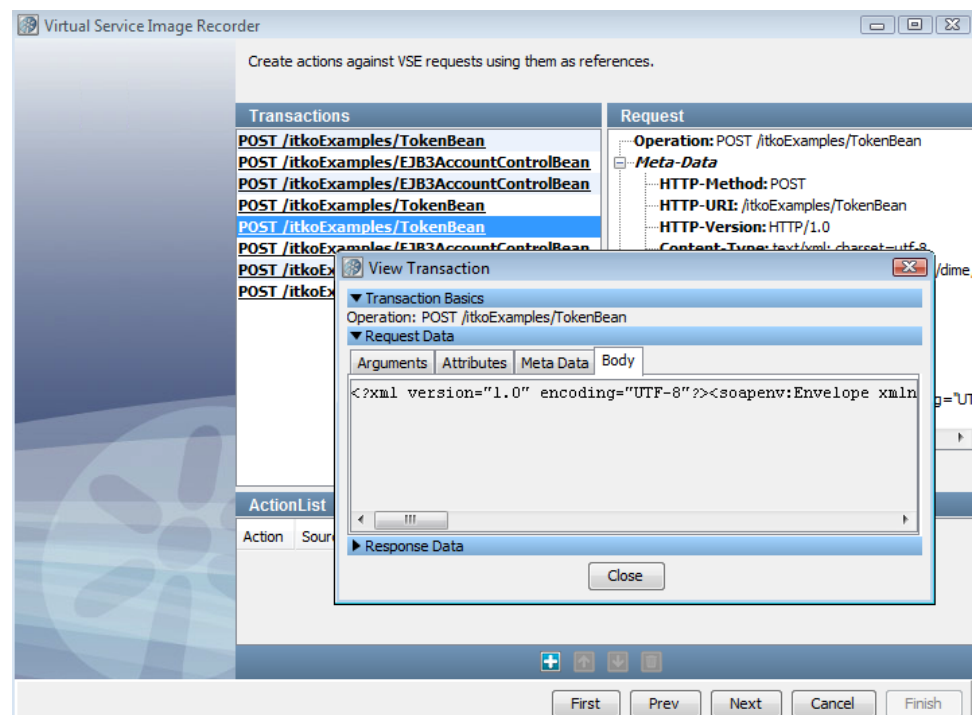
All actions can be applied to the request operation, any argument, attribute, or metadata entry, or the request body. For example, when virtualizing Java (which ends up with XML documents as arguments) you can move or copy the value of an argument into the request body, so other data protocols can process the argument.

**Note:** The Keep action is most meaningful for arguments, attributes, and metadata. If you specifically keep a value from one of these three groups, any other value in that group that is not referenced by any action in the list for the data protocol is deleted. If you keep one argument, other arguments are removed unless they were the target of a move or copy. This technique is a good way to remove meaningless arguments.

Each action can also be limited to apply only to requests whose operations match a specified regular expression.

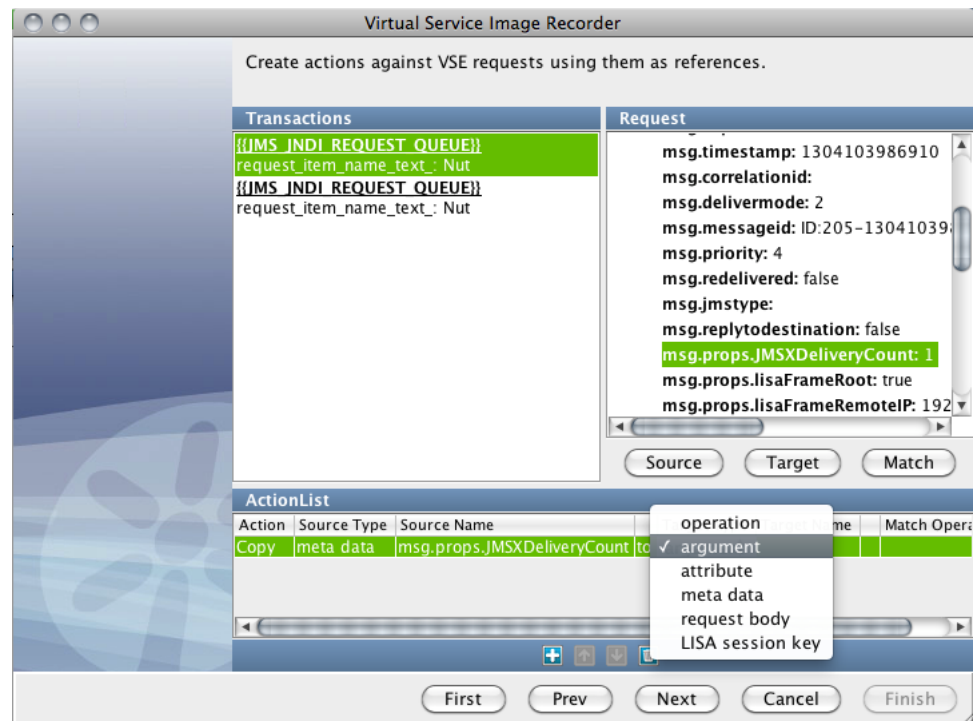
In the recording wizard or the model editor containing a Request Data Manager DPH, add a Keep or Delete action, or both. Select argument/attribute/meta data and specify a regular expression to match as the name. You must also change the cell that reads *named* to *matches*. When the DPH is run, it keeps or deletes all items in the argument, attribute, or metadata list with a name that matches the pattern. Leaving the operation matching pattern for an action empty affects all requests.

From this list of transactions, double-click a transaction to open a dialog showing the content of the transaction.



Use the Request Data Manager data protocol to set JMS and MQ Message Properties

After the recording is finished, use the Request Data Manager window to add the targeted JMS message properties to the request arguments. The JMS message properties can be found under the request Meta Data with a **msg.** prefix for standard JMS properties, like correlation ID, and a **msg.props.** prefix for custom message properties. To copy a property to the request arguments, select argument from the drop-down list:



MQ works the same way.

To set the stateful session key instead of an argument, select session key from the drop-down instead.

You can use a single Request Data Manager data protocol to set any number of arguments and the session key simultaneously.

## REST Data Protocol

REST is a way of calling web services using the HTTP protocol.

The REST data protocol handler analyzes HTTP requests that follow the REST architectural style. The data protocol identifies the dynamic parts of the URI strings. The result is a set of rules. During the playback phase, VSE uses the rules to virtualize HTTP requests that invoke the same operations.

If VSE does not automatically select the REST data protocol, you can select it manually in the data protocols page of the Virtual Service Image Recorder.

You can include the HTTP requests in live traffic or in request/response pairs.

You must use the HTTP/S transport protocol for request/response pairs.

You can configure some aspects of the analysis process.

Each rule contains one or more parameters, which represent the dynamic parts. By default, the parameters begin with the text **URLPARAM**.

The following sample rules include the parameters **URLPARAM0** and **URLPARAM1**. The parameters must be inside curly brackets in the rule. After the analyzer generates the rule, you can change the **URLPARAM** identifier.

```
GET /Service/rest/user/{URLPARAM0}
GET /Service/rest/customer/{URLPARAM0}
GET /Service/rest/customer/{URLPARAM0}/order/{URLPARAM1}
```

At playback, the following URI strings match the first rule:

```
GET /Service/rest/user/100
GET /Service/rest/user/101
```

At playback, the following URI string matches the third rule:

```
GET /Service/rest/customer/1234/order/5678
```

Parameters can also be mixed with other constant text if required. For example:

```
GET /Service/rest/user/{USERNAME}/format.{type}
```

In this example, the last token could be “format.json” or “format.xml”.

There can be any number of parameters in a token in any position. For example:

```
GET
/Service/rest/users/format.{type}.sortorder.{order}.filter.{filter
}
```

You can use the REST rules editor that appears after a recording to add or modify rules to look like this depending on your requirements. Rules like this can also be generated from WADL or RAML files.



## Request/Response Pair Format

This topic describes the format of request/response pairs for the REST data protocol.

### Request

The request file must include a valid HTTP header. The URL is the first line of the header. The other header lines are optional.

The REST data protocol handler supports all the HTTP methods/verbs: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, and PATCH.

The format of the URL line is:

<METHOD><a space character><REST API path><space><HTTP-VERSION>

For example:

```
PUT /rest-example/control/users/save HTTP/1.1
```

If the request includes a body, separate the body from the header with a blank line.

The following example shows a request that contains a body in JSON format.

```
PUT /rest-example/control/users/save HTTP/1.1
accept: application/json
content-Type: application/json
Connection: Keep-Alive
User-Agent: LISA
```

```
{
  "user": {
    "emailAddress": "test@test.com",
    "firstName": "first-9",
    "lastName": "last-9",
    "password": "aaaaaaaa",
    "username": "dmxxx-009"
  }
}
```

You can condense the JSON or XML body on a single line.

```
{ "user": { "emailAddress": "test@test.com", "firstName": "first-9",
"lastName": "last-9", "password": "aaaaaaaa", "username":
"dmxxx-009"  }}
```

### Response

The response file contains an HTTP response code. The file can contain a header and a body. The response code must be the first line in the file, in the following format:

<HTTP-VERSION><a space character><HTTP-RESPONSE-CODE>

The following example shows a response that has a body.

HTTP/1.1 200

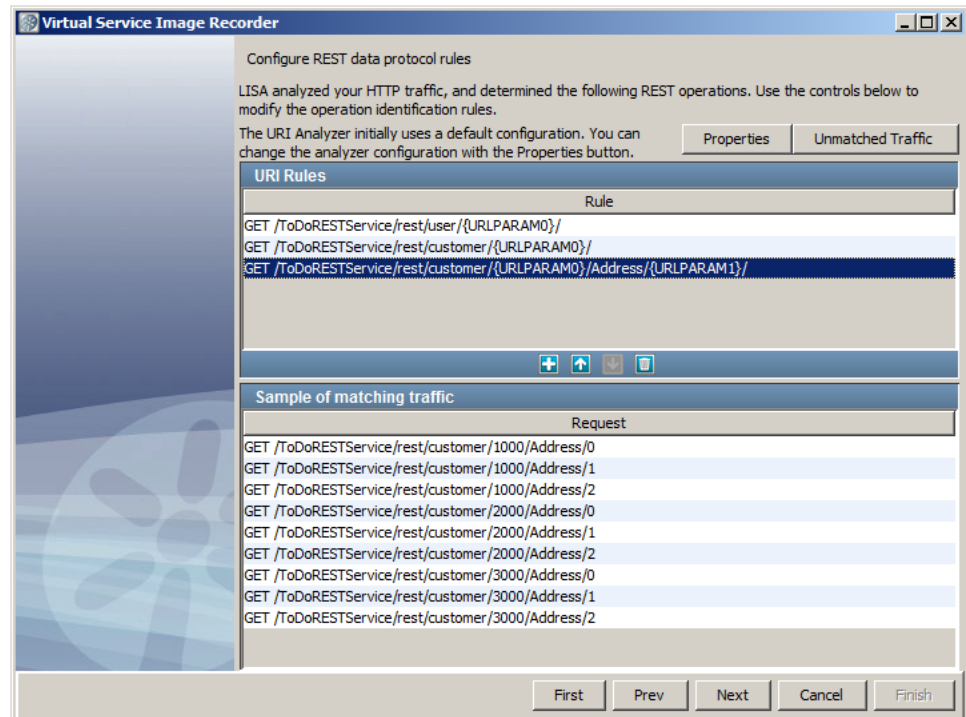
```
"user":{"emailAddress":"lisa.simpson@itko.com","firstName":"lisa",  
"lastName":"simpson","password":"60fAFoq+W0R4HrLgsfPodkWRw9I=",  
"phoneNumber":"","username":"lisa_simpson"}}
```

If the response does not include a response code line, the response code defaults to **200 (OK)**.

## Rules Review and Modification

The Virtual Service Image Recorder and the Virtual Service from request/response pairs interface include a page where you can review and modify the rules that the REST data protocol created.

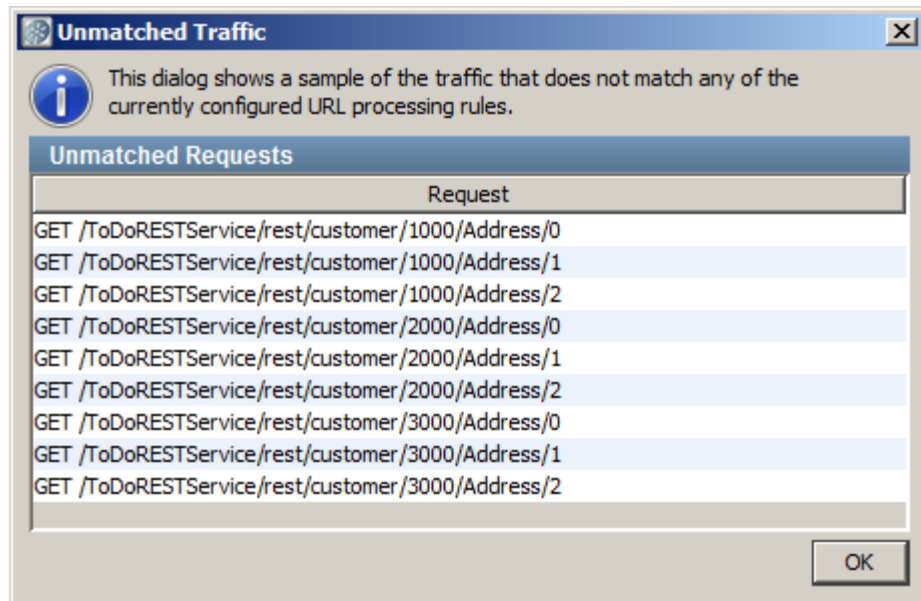
The following graphic shows the window in the Virtual Service Image Recorder.



The upper pane (URI Rules) displays the rules. When you select a rule, the lower pane (Sample of matching traffic) displays a sample of traffic that matches the rule. To configure the maximum number of rows that display, set the **lisa.protocol.rest.editor.observedtraffic.max** property in the **lisa.properties** file.

To display a list of HTTP requests that do not match the rules, click Unmatched Traffic. This list is empty if the rules match all the collected traffic. To configure the maximum number of requests that display, set the **lisa.protocol.rest.editor.unmatchedtraffic.max** property in the **lisa.properties** file.

The URI Rules pane lets you add, update, reorder, and delete the rules.



You can replace the parameters in the rules with more meaningful names. For example:

```
GET /Service/rest/customer/{customerid}/order/{orderid}
```

You can create multiple rules that match the same operation. The rules are matched in the order shown. As a result, a rule that is higher in the list can match when you expect a lower rule to match. To change the order, use the reordering buttons.

If you delete a rule, click Unmatched Traffic to see the effect of removing the rule on the captured traffic.

To change the values of the following configuration properties, click Properties.

You can perform re-analysis of the traffic by clicking Properties; the Properties page is displayed, then click OK without changing any values. For example, you may want to do this after you chain Scriptable data protocol and REST data protocol.

#### **Max Changes**

Defines the maximum number of changes that are allowed for a token before the variability is considered significant enough to generate a rule.

Think of a URI as a list of "tokens" separated by the "/" character. For example, the URI "GET /rest/user/1234" contains the tokens:

- GET
- rest
- user
- 1234

To change the default value, set the **`lisa.protocol.rest.maxChanges`** property in the **`lisa.properties`** file.

#### Start Position

Defines the position in the URL at which the REST data protocol starts looking for variable tokens.

The start position is the index of a token in the list of tokens of which a URI is composed. For example, in the URI:

GET /service/rest/customers"

"GET" is at position 0 and "customers" is at position 3.

To change the default value, set the **`lisa.protocol.rest.startPosition`** property in the **`lisa.properties`** file.

#### Id Identification Regular Expression

Defines a regular expression string that the REST data protocol uses to detect resource identifiers in the HTTP requests. To change the default value, set the **`lisa.protocol.rest.idPattern`** property in the **`lisa.properties`** file.

#### URL Parameter Prefix

Defines the prefix that the REST data protocol uses for the parameters in rules.

The purpose of this setting is to help the analyzer spot tokens that follow a specific pattern that you know is variable, but which the analyzer may not automatically detect.

For example, in the URI:

GET /rest/user/person-1234-dev

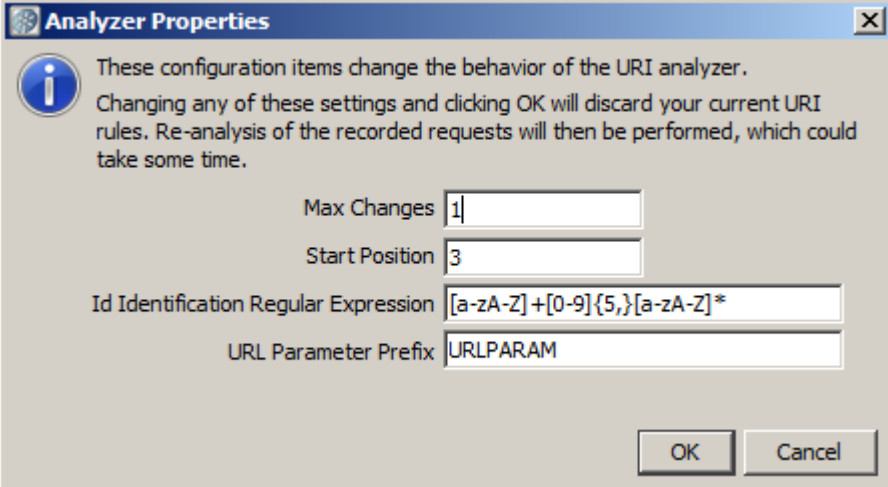
You may know that a user ID in the format *person-nnnn-nnn* always follows "user". In this case, you can define a regular expression to detect this pattern directly. In this example, the regular expression would be:

person-[0-9]{4}-[a-z]{3}

To change the default value, set the **`lisa.protocol.rest.parameterBaseName`** property in the **`lisa.properties`** file.


If you change one or more values for these properties, the data protocol reanalyzes the recorded traffic.

The following graphic shows the Analyzer Properties window.



The image shows a Windows-style dialog box titled "Analyzer Properties". It contains an information icon and a warning message. Below the message are four input fields: "Max Changes" with the value "1", "Start Position" with the value "3", "Id Identification Regular Expression" with the value "[a-zA-Z] + [0-9]{5,}[a-zA-Z]\*", and "URL Parameter Prefix" with the value "URLPARAM". At the bottom right are "OK" and "Cancel" buttons.

**Analyzer Properties**

 These configuration items change the behavior of the URI analyzer. Changing any of these settings and clicking OK will discard your current URI rules. Re-analysis of the recorded requests will then be performed, which could take some time.

Max Changes

Start Position

Id Identification Regular Expression

URL Parameter Prefix

## Scriptable Data Protocol

The Scriptable data protocol is available for situations where you need a small amount of processing on the request, the response, or both. Sample scripts are written in BeanShell. Your script can be written in any scripting language that CA Application Test supports.

To specify a language other than BeanShell for the script, enter the language on the first line of the script.

**Values:**

- applescript (for OS X)
- beanshell
- freemarker
- groovy
- javaScript
- velocity

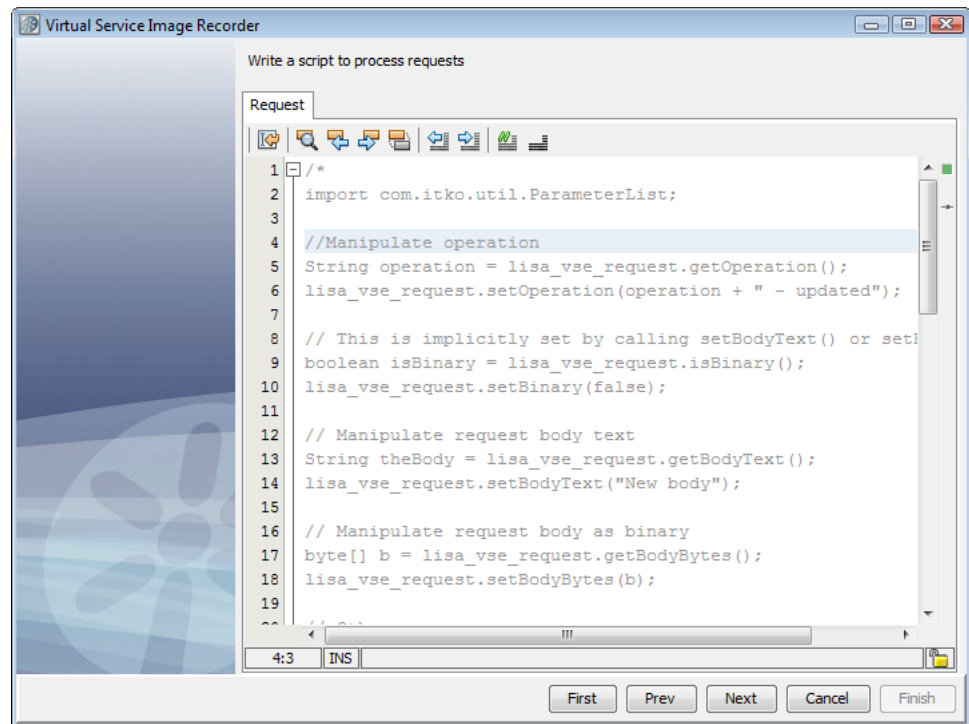
**Format:**

**%Language%**

**Default:** beanshell

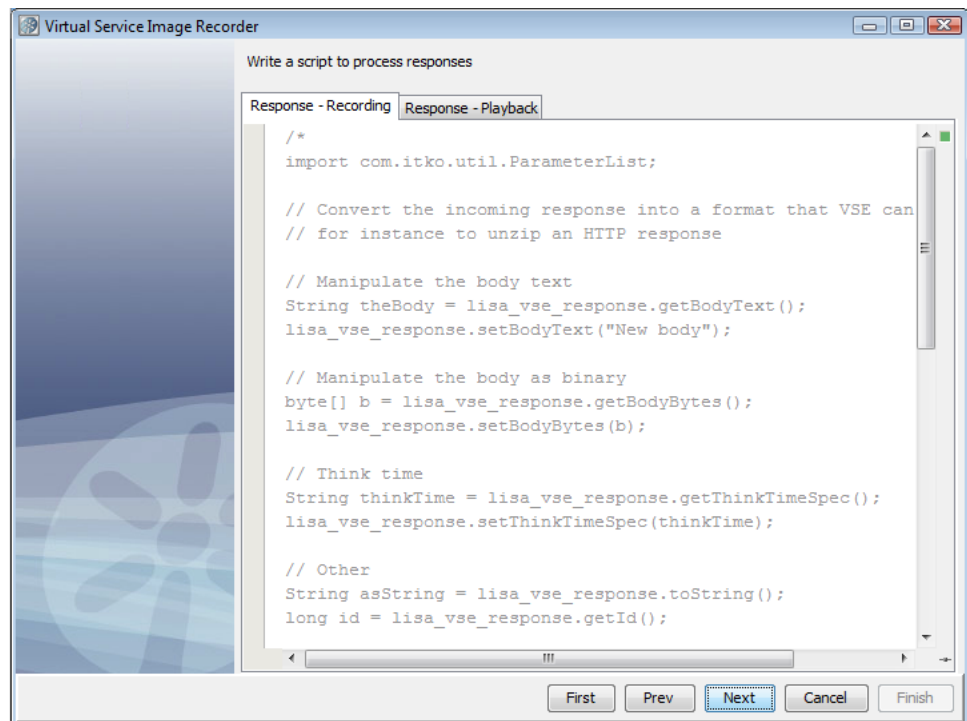
To use additional scripting languages, see "Enabling Additional Scripting Languages."

When you specify a Scriptable data protocol on the request side, the following window opens.



Two scripts are available on the response side: one for recording and one for playback. You can use the Response - Recording script to convert the recorded response into a format that VSE can process. Then, after it is processed, you can use the Response - Playback script to return it to the format that the System Under Test expects.





You can add your own script to perform any actions you want on the request, the response, or both.

## SWIFT Data Protocol

The SWIFT data protocol performs the following actions:

- Converts a SWIFT message to an XML representation
- Converts the XML representation of a SWIFT message back to its native format

For example, the data protocol converts the following SWIFT messages:

```
{1:F01BANKDEFMAXXX2039063581}{2:01031609050901BANKDEFAXXX89549829458949811609N}{3:{108:00750532785315}}{4:
```

```
:16R:GENL
```

```
:20C::SEME//YOUR REFERENCE
```

```
:16S:GENL
```

```
:16R:SETDET
```

```
:22F::SETR//TRAD
```

```
:16R:SETPRTY
```

```
:97A::SAFE//YYYY
```

```
:16S:SETPRTY
```

```
:16S:SETDET
```

```
-}
```

to the following XML:

```
<message>
<block1>
  <applicationId>F</applicationId><serviceId>01</serviceId>
  <logicalTerminal>BANKDEFMAXXX</logicalTerminal>
  <sessionNumber>2039</sessionNumber>
  <sequenceNumber>063581</sequenceNumber>
</block1>
<block2 type=\"output\">
  <messageType>103</messageType>
  <senderInputTime>1609</senderInputTime>
  <MIRDate>050901</MIRDate>
  <MIRLogicalTerminal>BANKDEFAXXX</MIRLogicalTerminal>
  <MIRSessionNumber>8954</MIRSessionNumber>
  <MIRSequenceNumber>982945</MIRSequenceNumber>
  <receiverOutputDate>894981</receiverOutputDate>
  <receiverOutputTime>1609</receiverOutputTime>
  <messagePriority>N</messagePriority>
</block2>
```

```

<block3>
  <BLOCK3_108>00750532785315</BLOCK3_108>
</block3>
<block4>
  <GENL>
    <GENL_20C>:SEME//YOUR REFERENCE</GENL_20C>
  </GENL>
  <SETDET>
    <SETDET_22F>:SETR//TRAD</SETDET_22F>
    <SETPRTY>
      <SETPRTY_97A>:SAFE//YYYY</SETPRTY_97A>
    </SETPRTY>
  </SETDET>
</block4>
</message>

```

If some fields in the SWIFT message include a date, the data protocol reformats the date in a format that DevTest can identify as a magic date candidate.

The following example shows the fields that the data protocol checks for dates:

```

:98A::SETT//19911130

:98C::TRAD//20140117125901

:98E::PREP//20091107093238,02/N0230

:32A:870902JPY3520000,

:30:640123

```

The data protocol converts these lines to the following XML:

```

<BLOCK4_98A>:SETT//1991-11-30</BLOCK4_98A>

<BLOCK4_98C>:TRAD//2014-01-17 12:59:01</BLOCK4_98C>

<BLOCK4_98E>:PREP//2009-11-07T09:32:38.020-0230</BLOCK4_98E>

<BLOCK4_32A>1987-09-02 JPY3520000,</BLOCK4_32A>

<BLOCK4_30>2064-01-23</BLOCK4_30>

```

The service image contains the dates it represents as magic dates.

## SWIFT Conversations

To generate a session key for SWIFT transactions, the SWIFT data protocol extracts information from the message body. The protocol uses the following rules to extract the SWIFT message reference (`mesg_ref`) and deal reference (`deal_ref`), then combines them to form the session key.

1. If field 70E is found in the form `:SPRO///xxxREF/<mesg_ref>/<deal_ref>`, they are extracted from here.
2. If field 20C is found in the form `:RELA//<mesg_ref>`, and a second field 20C is found in the form `:TRRF//<deal_ref>`, they are extracted from here.
3. If field 26H is found in the form `<msg_ref>/<deal_ref>`, they are extracted from here.

The data protocol applies these rules in order. If one of them is satisfied, it creates the session key in the form `<message ref>/<deal ref>`. If none of the rules is satisfied, the data protocol treats the transaction as stateless and it does not create a session key.

## Web Services Bridge Data Protocol

The Web Services Bridge data protocol exists specifically for the DevTest Travel example. This protocol is specific to the example and is not useful in a general case. Outside of the DevTest Travel example, you can ignore this protocol.

## Web Services (SOAP) Data Protocol

The Web Services SOAP data protocol converts a SOAP document in XML form to a proper operation/arguments type of request.

For each request presented to the data protocol, it tries to parse the text form of the request body as an XML document. If the body is a valid XML document and the top-level element is **Envelope**, then the data protocol looks for both **Header** and **Body** child elements.

If the SOAP envelope contains a header element, the data protocol looks for a **ReplyTo** element in the header. Then, under the **ReplyTo** element, the data protocol checks for an **Address** element. If an address element is present, the value is set in the metadata list for the current VSE request under the name **lisa.vse.reply.to**.

If the SOAP envelope contains a body element, then the tag name for the first child of the body element becomes the operation name for the VSE request. If the operation cannot be determined, then nothing occurs for the current request.

If the operation element contains any XML attributes, these attributes are added to the attribute list for the current VSE request.

After it adds the XML attributes to the list, the data protocol examines the entire tree of XML elements under the operation element. All elements that do not have child elements are added to the VSE request as arguments. The name of each argument is constructed by using all parent element tags (up to the operation element) to ensure uniqueness. If the same name appears more than once, then a numeric suffix is added. The numeric suffix indicates an array style structure and ensures the uniqueness of the specific argument.

Finally, the original XML document (the request body) is copied to a new attribute in the VSE request named **recorded\_raw\_request**.

This data protocol handler requires no configuration information and so does not present a page in the recording wizard.

## Web Services (SOAP Headers)

The SOAP Headers data protocol converts elements from the header of a SOAP message into arguments for requests in a virtual service image. This data protocol is compatible with any transport protocol that generates SOAP messages (typically HTTP/S).

To use the SOAP Headers data protocol, generate a VS Image either by recording or from request/response pairs. You typically use the HTTP/S transport protocol. Add the SOAP Headers data protocol as a Request Side data protocol. This data protocol does not require any extra configuration.

This data protocol can be used with the SOAP data protocol to process both SOAP headers and the SOAP body.

Arguments are named based on the structure of the XML.

### Example

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsse:Security
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken>
        <wsse:Username>username</wsse:Username>
        <wsse:Password>password</wsse:Password>
      </wsse:UsernameToken>
      </wsse:Security>
    <n1:ServiceControl xmlns:n1="http://localhost:8080/examples.xsd">
      <n1:VersionID>2.0</n1:VersionID>
      <n1:Asynchronous>
        <n1:ReplyRequiredIndicator>false</n1:ReplyRequiredIndicator>
        <n1:PassThroughData>
          <n1:Key>InteractionID</n1:Key>
          <n1:Value>444831</n1:Value>
        </n1:PassThroughData>
      </n1:Asynchronous>
    </n1:ServiceControl>
  </soapenv:Header>
  ...
```

This example would be parsed into elements that are named as follows:

- Security\_UsernameToken\_Username
- Security\_UsernameToken\_Password
- ServiceControl\_VersionID
- ServiceControl\_Asynchronous\_ReplyRequiredIndicator

- ServiceControl\_Asynchronous\_PassThroughData\_Key
- ServiceControl\_Asynchronous\_PassThroughData\_Value

Duplicate elements are named with \_1, \_2, and so on, appended to the name.

## WS - Security Request Data Protocol

The WS-Security Request data protocol supports SOAP messages that include WS-Security headers. This data protocol can strip any security from the SOAP Request before sending it along the Virtualize framework. The WS-Security Request data protocol then applies security to outgoing SOAP responses.

When recording a web service with WS-Security headers, add a WS-Security Request (Request Side) data protocol (typically before a Web Service SOAP data protocol) and a WS-Security Response (Response Side) data protocol.

Before you record, you are presented with a set of configuration panels:

- One for the WS-Security Request data protocol
- Two for the WS-Security Response data protocol.

### Request Data Protocol

For the Request data protocol, configure the handler to process a Request message that the client sends. Fill in the Receive actions that are used to decode and validate the headers. This configuration is used both for recording and playback.

Options available for Receive (Response) messages are:

- Decryption
- Signature Verification
- SAML Verifier
- Username Token Verifier
- Timestamp Receipt
- Signature Confirmation

Options available for Send (Request) messages are:

- Timestamp
- Username Token
- SAML Token
- Signature Token
- Encryption

To use encryption, enter the following information:

#### Keystore file

Specifies the keystore file to use for encryption.

#### Keystore Type



Specifies the keystore type.

**Values:**

- Java Key Store
- Personal Information Exchange (PKCS #12)

**Keystore password**

Defines the password associated with the specified keystore file.

**Keystore alias**

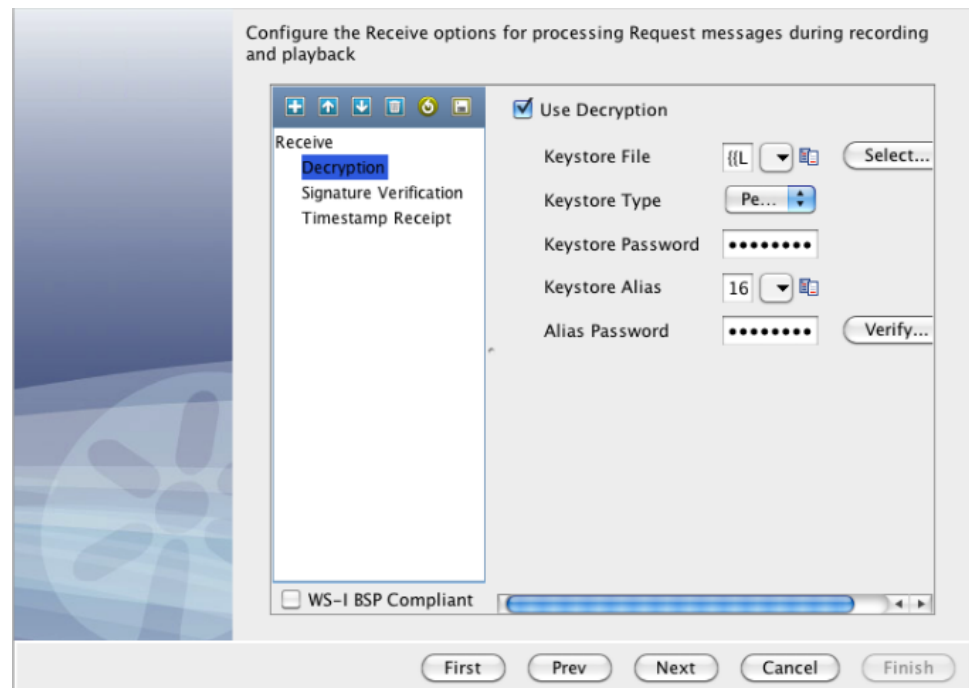
Designates an alias for a public key.

**Alias Password**

Defines an alias password for PKCS#12 files. Leave this value blank or enter the same value you specified for Keystore Password.

The WS-I BSP Compliant check box indicates whether to verify that you comply with the WS-I Basic Security Profile (including using InclusiveNamespaces and CanonicalizationMethod in SignedInfo).

To validate your keystore information, click the Verify button.



**Response Data Protocol**

For the Response data protocol, configure the handler to process Response messages returned from the live service during recording and Response messages that are returned from the VSM. During the recording phase, you must process the Response message as the client would.

Select the Add Timestamp check box.

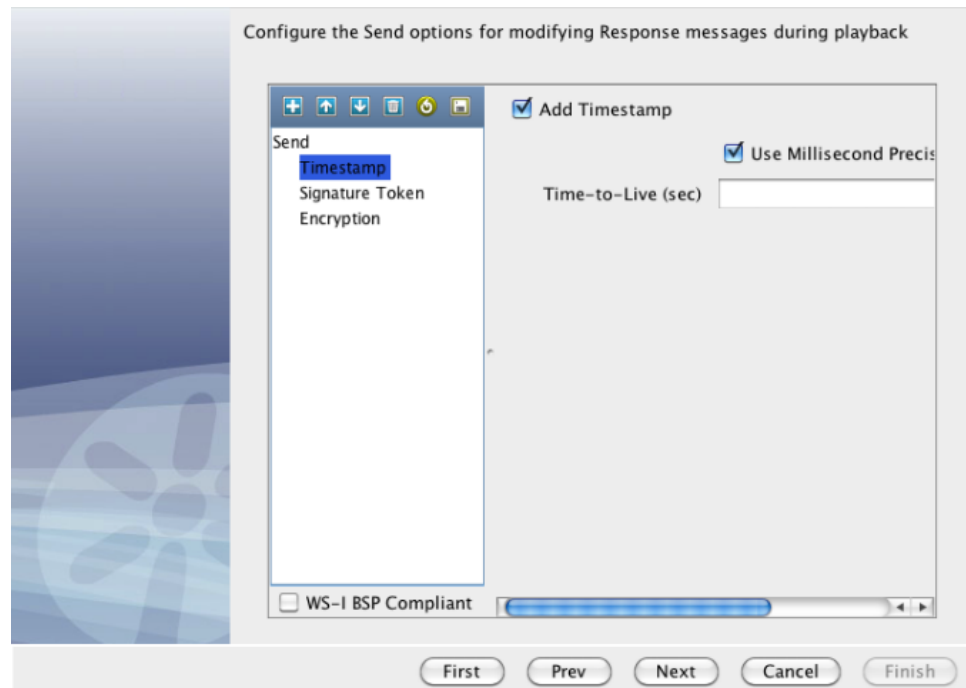
#### Time-To-Live (sec)

Defines the lifetime of the message in seconds. To avoid including an Expires element, enter 0.

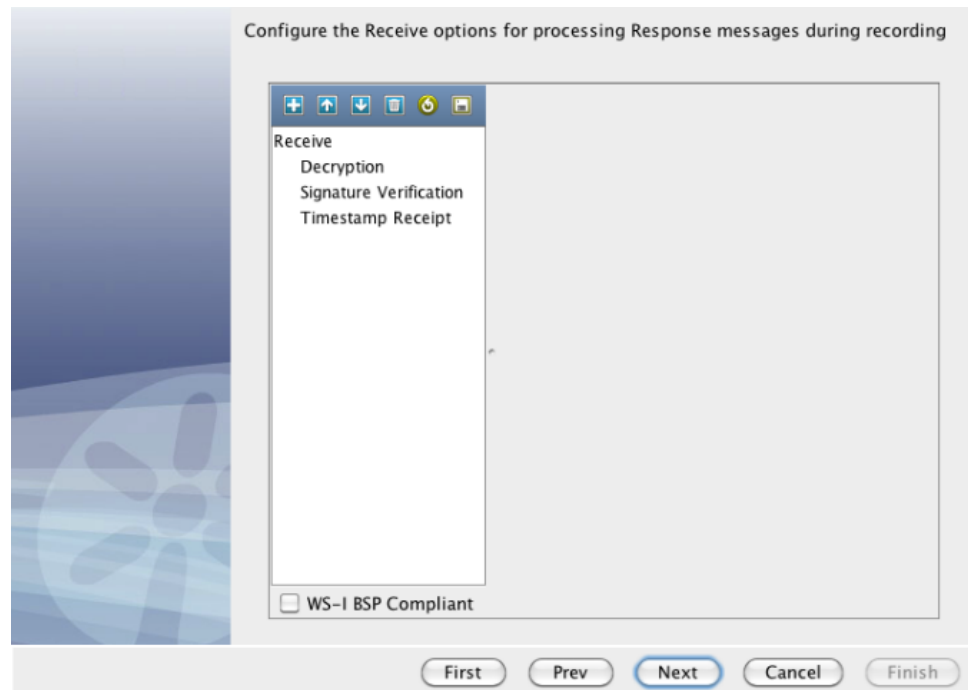
#### Use Millisecond Precision in Timestamp

Specifies whether to output the timestamp in milliseconds.

**Note:** Some web services (for example .NET 1.x/2.0 with WSE 2.0) do not comply with standard timestamp date formatting, and do not allow milliseconds. For these web services, clear the Use Millisecond Precision in Timestamp check box.

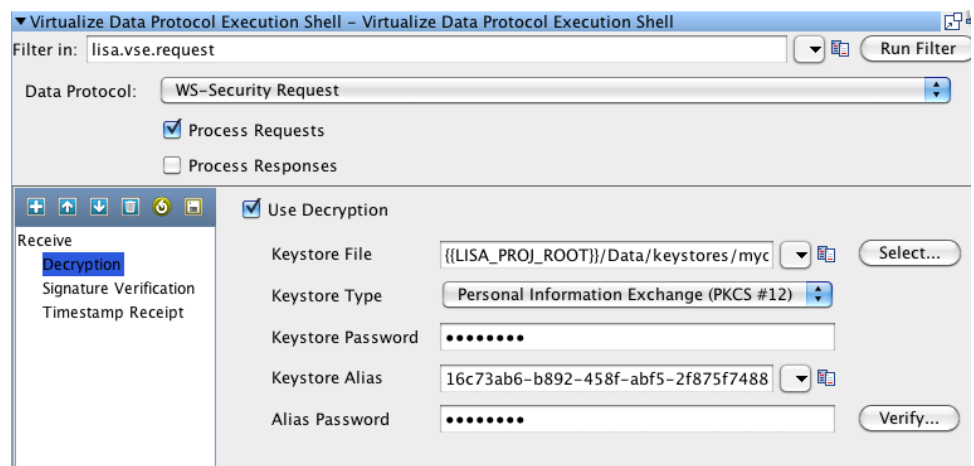


During playback, you must process the message as the server would send it. The SOAP message from the VSM has no Security headers and this configuration applies the Security header.





After the recording completes, a virtual service model is created. In that model, a data protocol filter is attached to the HTTP/S Listen step for the WS-Security Request data protocol.

Update any security configuration information for playback. For example, if your WS-Security settings change on the service, you can update them here instead of rerecording the virtual service.



In the VSM, a filter is also added for the WS-Security Response data protocol onto the HTTP/S Response step.

Any security configuration information for playback can be updated for the response message.

To save your security settings to a file, or to load a saved file containing security settings, use Load  and Save .

### XML Data Protocol

The XML data protocol converts an XML document into a proper operation/arguments type of request.

This data protocol handler works exactly like the [Web Services \(SOAP\)](#) (see page 265) data protocol. The data protocol requires no configuration information and so does not present a window in the recording wizard.

# Chapter 9: Editing Service Images

---

This section contains the following topics:

[Legacy Service Images](#) (see page 273)

[Open a Service Image to Edit](#) (see page 274)

[Service Image Tab](#) (see page 275)

[Transactions Tab](#) (see page 279)

[Transactions Tab for Stateless Transactions](#) (see page 280)

[Transactions Tab for Conversations](#) (see page 291)

[Conversation Editor](#) (see page 293)

[Service Images for JMS Transport Protocol](#) (see page 304)

## Legacy Service Images

Beginning with LISA 6.0, the product does not store service images in a database. If you must use LISA 5.0 service images in LISA 6.0 and later:

1. Export them using LISA 5.0.
2. Import them using the Import item on the project tree shortcut menu in the later LISA version.

**Note:** A service image exported from LISA 5.0 has an extension of **.xml**. To make this exported service image into a valid LISA 6.0 or later service image, change the extension to **.vsi**.

## Open a Service Image to Edit

The Virtual Service Image Recorder generates service images. Service images pretend to be what you recorded (a manipulated or altered version of your recorded raw traffic).

If you have service images from releases of VSE earlier than LISA 6.0, export those service images. Exporting moves them from the database of earlier versions to the file system where they are stored in the current release. For more information, see "[Legacy Service Images](#) (see page 273)".

Opening a service image allows you to change the image in the Service Image Editor.

**Follow these steps:**

1. Right-click the service image in the project panel and select Open.  
The Service Image Editor opens.
2. Review the selected service image and make any changes.

**Note:** You can also access the editor from the Response Selection step in the VSM by clicking Open next to the name of the service image.

## Service Image Tab

The following fields are available on the Service Image tab of the Service Image Editor:

**Image name**

Identifies the name of the current service image.

**Created on**

Identifies the date and time when the service image was created.

**Last modified**

Identifies the date and time when the service image was last modified.

**Notes**

Displays documentation about the service image.

**Approximate memory usage**

Identifies the estimated memory that the service image requires.

**Response for Unknown Conversational Request**

Displays the body, metadata, and think time for a response to an unknown conversational request in playback.

**Response for Unknown Stateless Request**


Displays the body, metadata, and think time for a response to an unknown stateless request in playback.

For more information about specific elements on this window, see:

- [Edit Responses for Unknown Requests](#) (see page 276)
- [Customize the Response Editor](#) (see page 277)

## Edit Responses for Unknown Requests



Use the arrow  to zoom the panels to their largest size. To return to the original size, use the same icon.

Complete the following Response panel fields as appropriate:

### Body

Contains the response to be returned for unknown stateless requests during playback.

### Response Meta Data

Displays a toolbar from which you can add, move, or delete key/value pairs as necessary.

### Think time spec

Defines the amount of think time that is required, in milliseconds. The *think time* is the time that is taken before sending the response to a request.

**Values:** A number or number range, in milliseconds.

If you enter a range, the application selects the think time randomly selected from that range.

You can specify time measurements by adding a suffix to the numbers. Case does not matter. The valid suffixes include:

- **t:** milliseconds
- **s:** seconds
- **m:** minutes
- **h:** hours

**Default:** 0

### Examples:

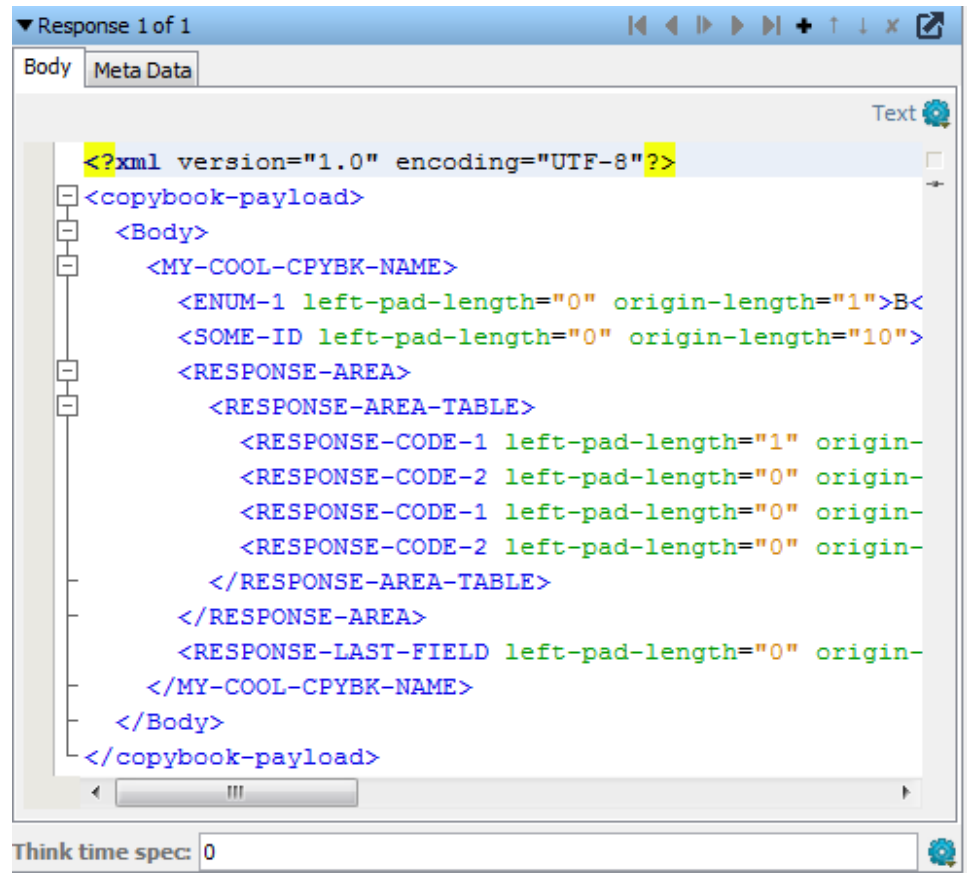
- 100 specifies a think time of 100 milliseconds.
- 100s specifies a think time of 100 seconds.
- 100-1000 specifies a random think time from 100 to 1000 milliseconds.
- 10t-5s indicates a random think time between 10 milliseconds and 5 seconds.

**Note:** A step subtracts its own processing time from the think time to have consistent pacing of test executions.



## Customize the Response Editor

To customize the response editor, click  at the lower right corner of the panel.



Options on the response editor menu include:

### Set reference protocol to

No Specific Protocol or JDBC (Driver based).

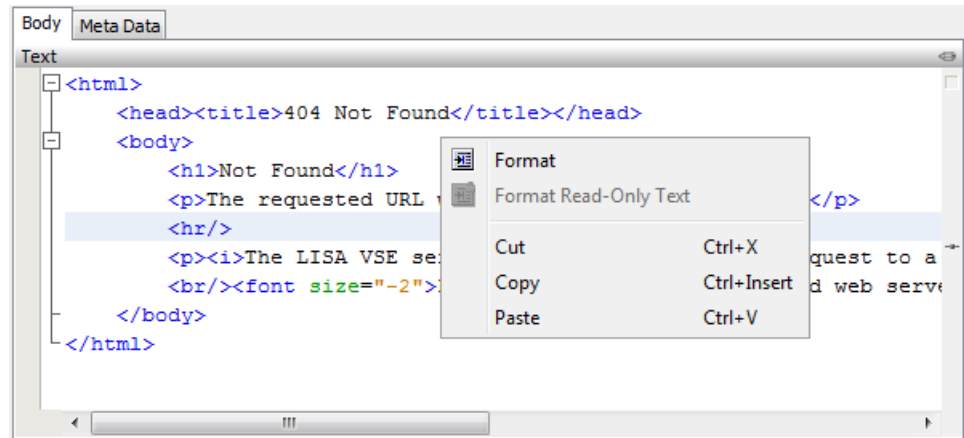
The title bar on the Body tab of the Response Editor shows what category of response has been returned. In the previous graphic, it indicates a Text response. Other categories of response payloads are:

- XML
- JSON
- String
- Large String
- Huge String
- Graphic Image

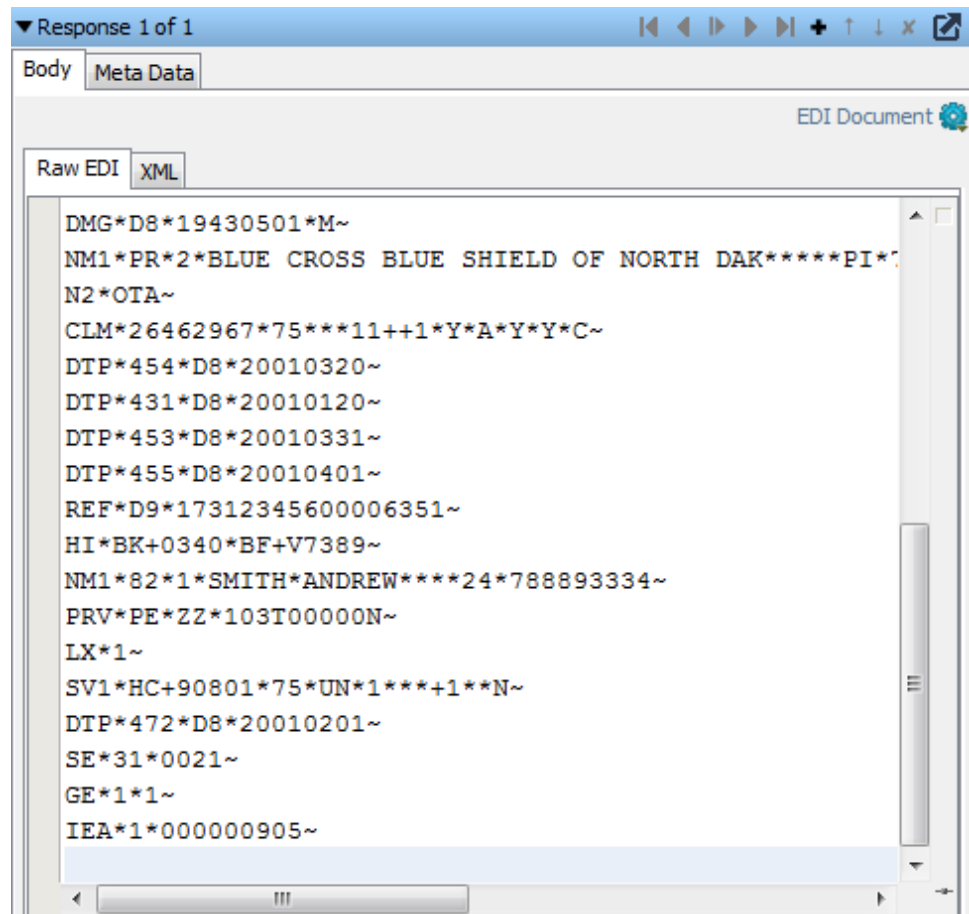
- Raw Bytes

To use a different editor or to change the type of payload, click the gear icon on the upper right corner of the panel. The menu builds dynamically to only show other potentially valid editors, based on the editor you currently have selected.

When the text body editor is set to Text, right-click the Response Body XML to format the response text.



If the response payload is detected to be EDI data, it is converted to XML on the XML tab.



## Transactions Tab

You can access the Transactions tab from the Service Image Editor. This tab gives information about both stateless and stateful (conversations) transactions, with slightly different components for each. This section describes viewing the general components of the Transactions tab that are the same for both types of transactions.

To select between viewing conversations or stateless transactions, select either Conversation by number or Stateless Transactions from the drop-down list.

### More information

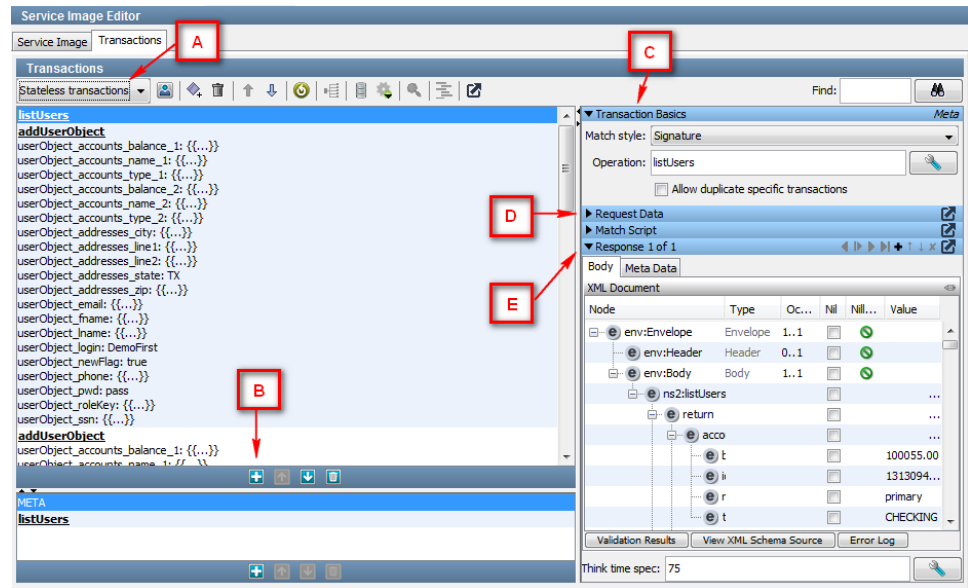
[Transactions Tab for Stateless Transactions](#) (see page 280)

[Transactions Tab for Conversations](#) (see page 291)


[Conversation Editor](#) (see page 293)

## Transactions Tab for Stateless Transactions

When viewing a stateless transaction, you can see the components that are shown in the following graphic.



- **A:** To view and edit stateless transactions, use the Stateless Transactions list. To add, move, or delete stateless transactions, use the toolbar at the bottom of the pane.
- **B:** One logical transaction (in the stateless transaction list) contains exactly one Meta transaction and any number of specific transactions. The Transactions list shows these transactions under the logical transaction. To add, move, or delete stateless transactions, use the toolbar at the bottom of the pane.
- **C:** To view and edit transaction requests and response data for either Specific or Meta transactions, which are selected from the Transactions area, use the Transaction Basics area. You can select a match style for the specific transaction here.
- **D:** The Request Data panel shows the stateless requests.
- **E:** The Response panel shows the response to the stateless requests.

**Note:** If you add or change several transactions, click . The magic string and date variables are created for you. Existing magic strings and variables are not modified.

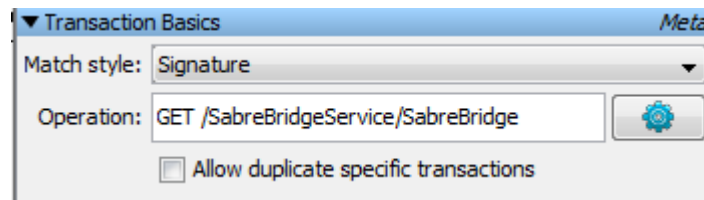
For more information about specific elements on this window, see:

- [Transaction Basics Editor](#) (see page 281)
- [Request Data Editor](#) (see page 281)
- [Match Script Editor](#) (see page 285)

- [Match Script Editor Toolbar](#) (see page 288)
- [Response Data Editor](#) (see page 289)

## Transaction Basics Editor

To view and edit transaction data for specific or meta transactions, use the Transaction Basics editor. Select a specific transaction or META from the Transactions list.



The Transaction Basics editor lets you specify the following information:

### Match Style

Values:

- **Signature**
- **Operation**

### Operation

Defines the operation that is selected.

### Allow duplicate specific transactions

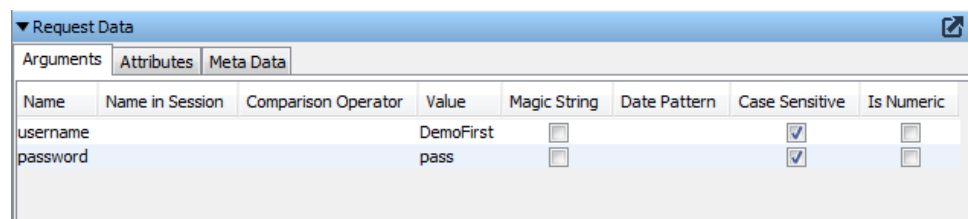
Specifies whether to allow DevTest to respond more than once to the same call, choosing a different response.

Values:

- **Selected:** DevTest can respond multiple times to the same call, with different responses. Round-robin matching only happens if this check box is selected.
- **Cleared:** DevTest can respond only once to a specific call.

## Request Data Editor

The Request Data editor allows you to update the data that is associated with a request.



## Arguments

VSE uses the operation name and arguments to look up a matching response for an incoming transaction.

### Adding and Removing Arguments

The META transaction is a template for the specific transactions. For more information, see [Logical Transactions](#). (see page 57) Therefore, arguments can be added to and removed from the META transaction, and these changes apply to all specific transactions in that logical group. Arguments cannot be directly added to or removed from specific transactions.

### Modifying Arguments

#### Name

Defines the name of the argument, which is parsed from the request. In most cases, this field should not be modified. It can only be changed at the META transaction level, and these changes propagate to the specific transactions.

#### Name in Session

Defines a value that the application automatically generates when magic strings are identified. The value can be referenced in the current (or later) responses using the `{{ }}` notation. If this field is not empty, the incoming value is stored in the session using the specified name. This value should not typically be modified.

#### Comparison Operator

Defines an operator to use in the matching logic. By default, for a specific transaction, all arguments are expected to match exactly. To change this and create more flexible matching logic, modify the comparison operator. See [Argument Match Operators](#) (see page 59) for the definitions of the comparison operators.

#### Magic String

Specifies whether to include the specified value as a candidate for magic strings.

#### Values:

- **Selected:** If this check box is selected and you click Regenerate magic strings, the specified value is a candidate for magic strings. Selecting this check box does not override the rules for identifying magic strings. You cannot use it to force something to be a magic string; it is still subject to the VSE magic string properties in the **`lisa.properties`** file.

- **Cleared:** If this check box is not checked and you click Regenerate magic strings, the specified value is excluded as a candidate for magic strings. If something was used as a magic string and you did not want the magic string substitution to happen, clear this check box and select Regenerate magic strings.

#### Date Pattern

Defines the pattern by which the application interprets incoming and specified values as dates. This value is automatically generated and should not typically be modified.

This pattern is a Java date and time pattern. It is a strict interpretation, so the values must match the pattern precisely. If both (or either) values cannot be parsed as dates, then the argument is deemed not to have matched. If both values can be parsed as dates, they can then be compared as dates using the following operators:

- =
- !=
- <
- <=
- >
- >=

You can still use "Anything," "Regular Expression," and "Property Expression." If you use "Regular Expression," the incoming value is treated as a string.

#### Case Sensitive

Specifies whether matching is case-sensitive.

**Values:**

- **Selected:** All matching is case-sensitive.
- **Cleared:** The comparison operators ignore case.

**Default:** Selected.

#### Is Numeric


Specifies whether argument values are processed as strings or numbers.

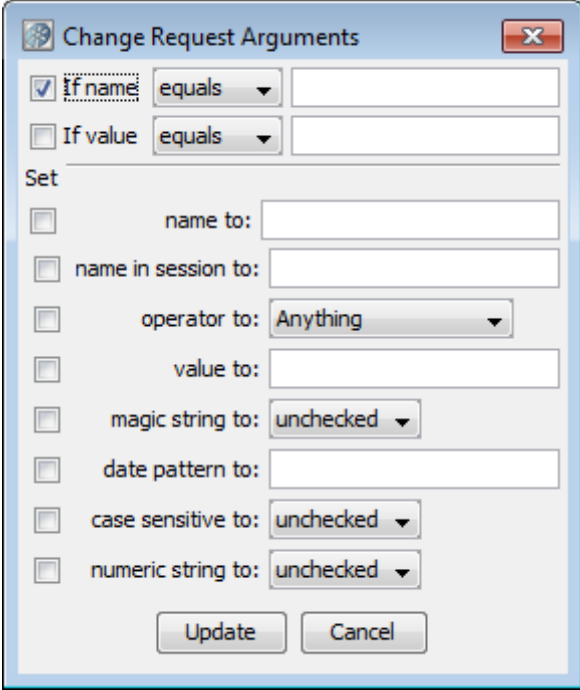
**Values:**

- **Selected:** The application processes argument values as numbers.
- **Cleared:** The application processes argument values as strings. That means "10000" is considered less than "9" because "1" comes alphabetically before "9".

**Default:** Cleared.

### Mass Change

To perform a mass change of request arguments, click Mass Change . The Change Request Arguments dialog appears.



The dialog box is titled "Change Request Arguments" and has a close button (X) in the top right corner. It contains two sections for conditional logic: "If name" and "If value". The "If name" section is checked, and both "If name" and "If value" are set to "equals". Below these is a "Set" section with several options, each with a checkbox and a corresponding field or dropdown menu: "name to:" (text field), "name in session to:" (text field), "operator to:" (dropdown menu set to "Anything"), "value to:" (text field), "magic string to:" (dropdown menu set to "unchecked"), "date pattern to:" (text field), "case sensitive to:" (dropdown menu set to "unchecked"), and "numeric string to:" (dropdown menu set to "unchecked"). At the bottom are "Update" and "Cancel" buttons.

To specify mass changes, complete the fields on the Change Request Arguments dialog as appropriate, and click Update.

### Attributes

To add, edit, move, and delete key/value pairs, use the Attributes tab.

### Meta Data

To add, edit, move, and delete Meta data key/value pairs, use the Meta Data tab.



## Match Script Editor

To insert a sample match script for your information, right-click the Match Script panel. You can also switch the match script on or off by selecting or clearing the Do Not Use the Script check box.

To designate the scripting language, use the language drop-down on the lower right of the pane.

### Language

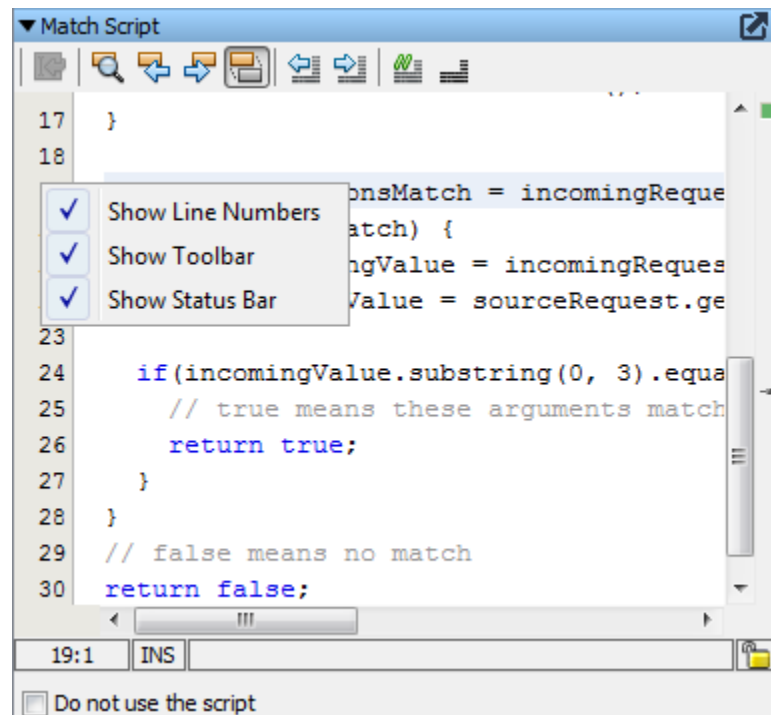
Designates the scripting language to use.

#### Values:

- Applescript (for OS X)
- Beanshell
- Freemarker
- Groovy
- JavaScript
- Velocity

**Default:** Beanshell

To hide or display line numbers, the editor toolbar, and the editor status bar, right-click on the left side of the Match Script panel, then select the appropriate options from the short-cut menu. The following graphic shows all options that are displayed.



A match script defines how VSE decides whether a specific transaction matches the incoming one. To receive a match that is based on the specific condition, write BeanShell scripts performing appropriate actions.

For example:

```
/* always match name=joe */  
  
ParameterList args = incomingRequest.getArguments();  
  
if ("joe".equals(args.get("name"))) return true else return  
  
defaultMatcher.matches();
```

You do not need to specify a match tolerance level or match operator for the match script to work. The match is found based on the condition in the match script.

By default (with no match script) an inbound request is matched against a service image request by comparing operations, arguments, or both to come to a true/false "Do they match?" answer. A match script simply replaces this logic with whatever logic makes sense and must still come to the true/false "Do they match?" answer.

The script can use the default matching logic. Inside the script, use the expression, "defaultMatcher.matches()". This expression returns a true or false using the VSE default matching logic.

The match script is similar to a scripted assertion. Basically, it is a regular BeanShell script but with the following additional variables preloaded for you (and the usual properties and testExec variable):

- `com.itko.lisa.vse.stateful.model.Request sourceRequest` (the recorded request)
- `com.itko.lisa.vse.stateful.model.Request incomingRequest` (the live request coming in)
- `com.itko.lisa.vse.RequestMatcher defaultMatcher` (you can default to this variable)

Return a Boolean value from the script; true means a match was found.

If there is an error evaluating the script, VSE deliberately ignores the error and defaults to the regular matching logic. If you do not think your script is being run, review the VSE log file.

A good way to add logging and tracing into your match scripts is to embed calls to the VSE matching logger. The VSE matching logger produces the messages in the `vse_XXX.log` file, where `XXX` is the service image name. For example:

```
import com.itko.lisa.VSE;

VSE.info(testExec, "short msg", "a longer message");

VSE.debug(testExec, "", "I got here!!");

VSE.error(testExec, "Error!", "Some unexpected condition");

return defaultMatcher.matches();
```

If you log messages at *INFO*, later when the production settings are applied to the `logging.properties` file, the log level is *WARN* and your messages appear as a DevTest test event (a "Log Message" event).

Tips from **logging.properties**:

- To simplify debugging, keep a separate log for VSE transaction match/no-match events.
- Change *INFO* to *WARN* or comment out the following line for production systems:  
`log4j.logger.VSE=INFO, VSEAPP`
- The *INFO* value typically reports every failure to match.

## Match Script Editor Toolbar



The Match Script editor toolbar lets you perform the following functions:



Returns you to the last edit that was made



Finds the next occurrence of the selected text



Finds previous occurrence



Finds next occurrence



Toggles the highlight search



Shifts the current line to the left four spaces



Shifts the current line to the right four spaces





Inserts comments slashes (//) at the cursor position

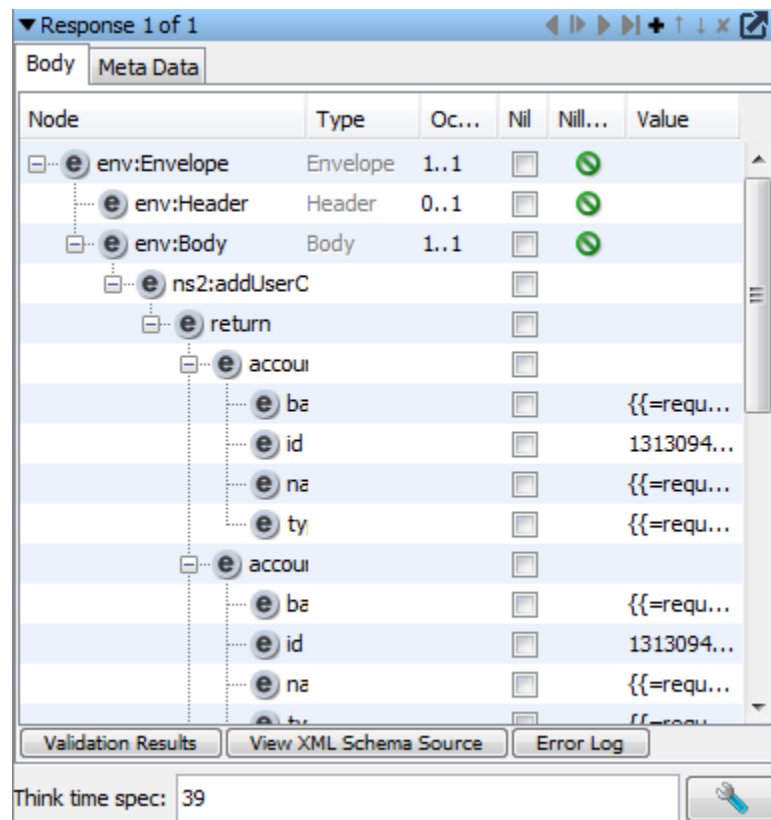


Removes the comments slashes (//)

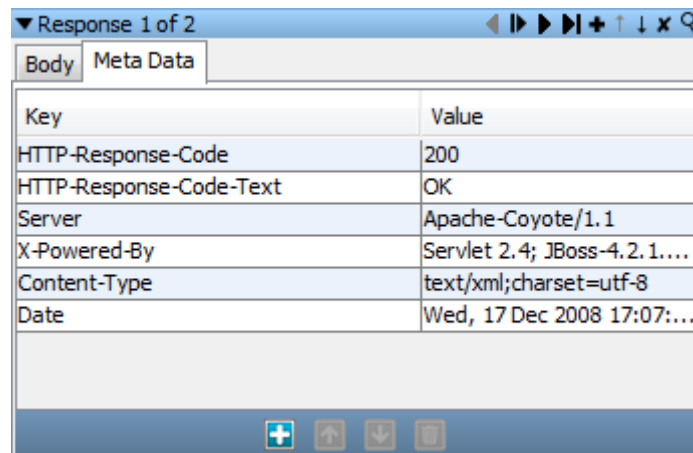
## Response Data Editor

To view and edit the response information, use the Response Data editor.

- To edit the expected response for a transaction, use the Response Body area.
- To add, order, delete, and navigate through responses, use the toolbar as described in the [Match Script Editor Toolbar](#) (see page 288).
- To enlarge the Response Data Editor panel, click .
- To customize the Response Data editor, click , as described in [Customize the Response Editor](#) (see page 277).
- To inspect the Validation Results, view the XML schema source, and see the error log, use the buttons at the bottom of the panel.



Edit the Think Time Spec field as necessary.



▼ Response 1 of 2

Body Meta Data

Key	Value
HTTP-Response-Code	200
HTTP-Response-Code-Text	OK
Server	Apache-Coyote/1.1
X-Powered-By	Servlet 2.4; JBoss-4.2.1....
Content-Type	text/xml; charset=utf-8
Date	Wed, 17 Dec 2008 17:07:...

+

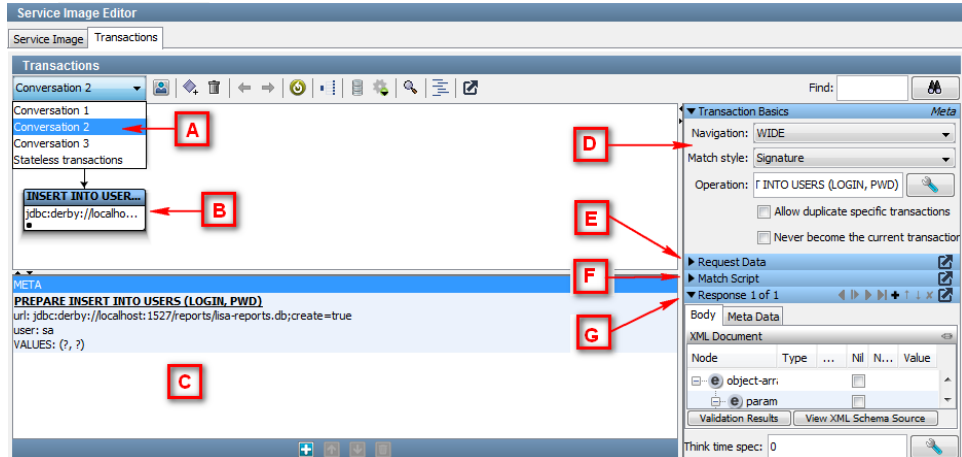
↑ ↓

✖

To add, edit, move, and delete key/value pairs, use the Meta Data tab.

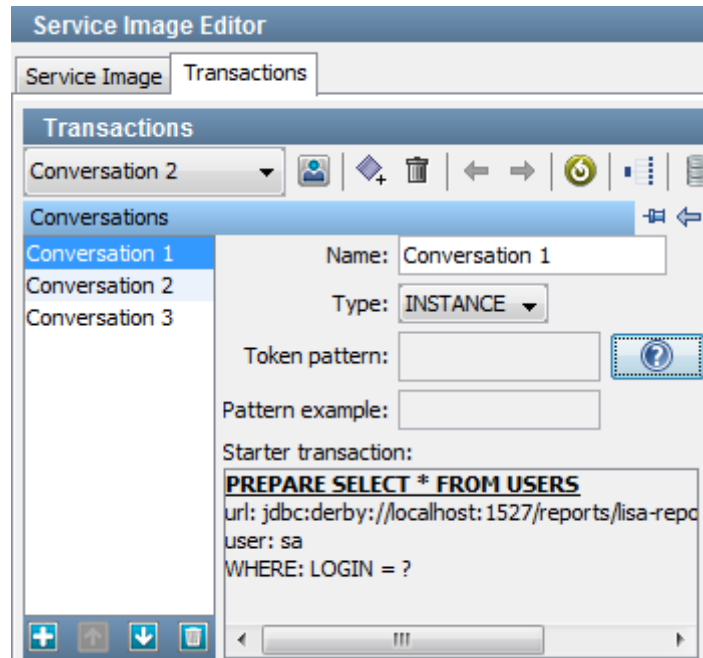
## Transactions Tab for Conversations


A stateful transaction (a transaction with conversations) has the following components.



- **A:** The Conversations list shows all the conversations in the service image. To view and edit a conversation, select it. A conversation consists of one or more logical transactions.
- **B:** The [Conversation Tree editor](#) (see page 297) displays the logical conversation that is selected in the Conversations list. The conversation is displayed in either a graph node tree view or a standard tree view.
- **C:** To view and edit specific transactions or the Meta data for transactions in a selected logical transaction, use the Transactions list. To add, move, or delete stateless transactions, use the toolbar at the bottom of the pane.
- **D:** To view and edit transaction requests and response data for the Specific or Meta transactions, selected in the Transactions area, use the [Transaction Basics](#) (see page 281) area. The fields are dependent on the selected transport and data protocols. For more information about the Transaction Basics area, see the [Service Image Editor Transactions Tab for Stateless Transactions](#) (see page 280).
- **E:** To enter the data for conversational requests during playback, use the [Transaction Request Data](#) (see page 281) pane.
- **F:** To enter and edit a script to return actions that are based on specified matching conditions, use the [Match Script Editor](#) (see page 285).
- **G:** To view and edit the response content, think time, and key/value pairs for the specific or Meta transaction, use the [Transaction Response Data](#) (see page 280) pane.

## Toggle Display Pane



You can see details about a conversation by clicking Toggle Display  on the Transactions tab toolbar. From this pane, you can display and edit the following values:

**Type**

Specifies the transaction type.

**Values:**

- INSTANCE
- TOKEN

**Token Pattern**

Required for token-based conversations. Clicking the question icon provides examples of string generator patterns.

**Pattern Example**

Displays an example of the specified Token Pattern.

**Starter Transaction**

Defines the starter transaction for the conversation.

**Note:** If you add or change several transactions, return to the Basic Info tab and click Regenerate Magic Strings and Data Variables. The magic string and date variables are created for you. Existing magic strings and variables are not modified.



## Conversation Editor

To view and edit recorded transactions or create transactions manually, use the Conversation editor. You can view the navigation trees in two display modes:

- [Graph View](#) (see page 295)
- [Tree View](#) (see page 297)

When you switch between views, the selected node remains selected. In both the Graph and Tree views, you can perform the following actions from the editor toolbar, shortcut menus, or the view itself.

### Conversation Editor Toolbar

The Conversation Tree editor toolbar varies slightly, depending on the display.

#### Graph View Tools













#### Tree View Tools



**Note:** If a tool appears dimmed, it cannot be used with the selected transaction.

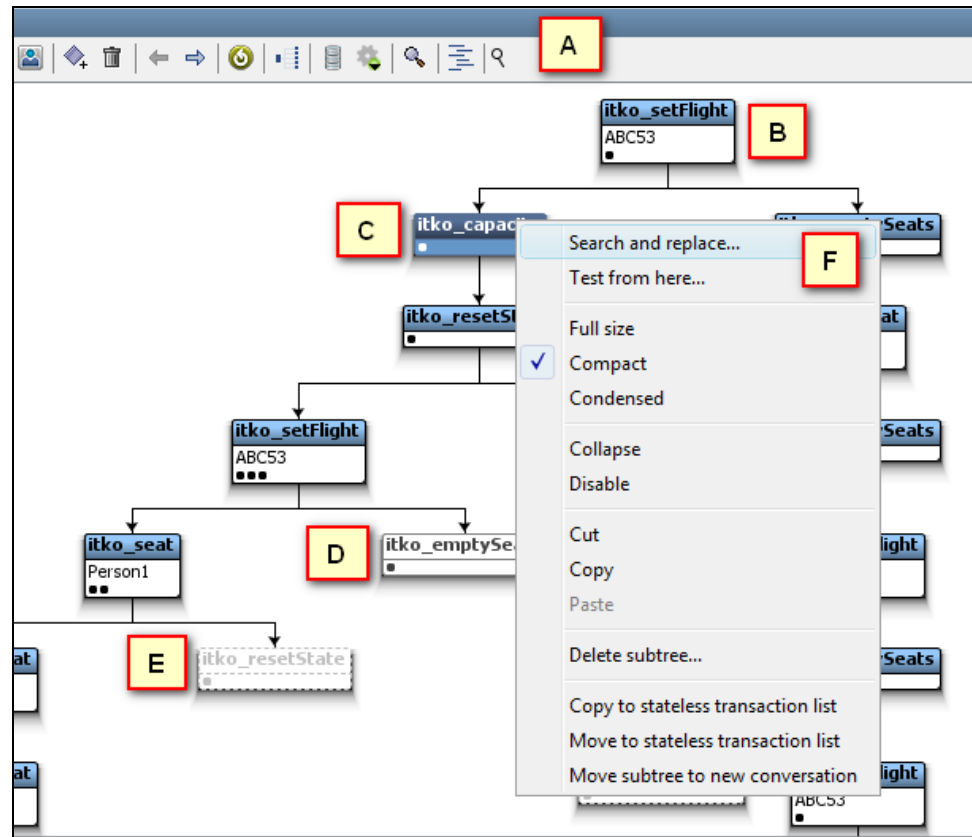
The Conversation Editor toolbar contains the following commands:

Tool	Icon	Description
Toggle Display		Toggles the display of the panel for managing a list of conversations. You can specify name, type, token pattern, pattern example, or starter transaction.
Create New Transaction		Adds a new transaction.
Delete Selected Transaction		Deletes a selected transaction.
Up Arrow		Tree View: moves the selected node earlier in the sibling list.
Down Arrow		Tree View: moves the selected node later in the sibling list.

Right <b>Arrow</b>		Graph View: moves the selected node later in the sibling list.
Left <b>Arrow</b>		Graph View: moves the selected node earlier in the sibling list.
Regenerate		Regenerates magic strings and date variables for all transactions.
View Navigation		Opens a menu to select menu navigation highlights for stateful transactions (conversations). You can select the following: <ul style="list-style-type: none"><li>■ No navigation highlight</li><li>■ Highlight on transaction's tolerance</li><li>■ Highlight as if close</li><li>■ Highlight as if wide</li><li>■ Highlight as if loose</li></ul>
Toggle		Toggles the display of transaction IDs for debugging.
Match		Pulls a match description from the clipboard and highlights the relevant information in the service image.
Zoom		Opens a zoom menu.
Display		Toggles the conversation display to a tree display.
Display		Toggles the conversation display to a graph display.
Zoom Panel		Zooms this panel to its largest size.

## Conversation Editor Graph View

The Conversation Editor displays nodes according to status. It has the following components:



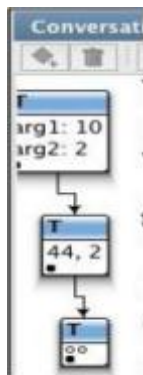
- **A:** Toolbar. For more information, see [Conversation Editor Toolbar](#) (see page 293).
- **B:** A standard node.
- **C:** A selected node.
- **D:** A collapsed node. Child nodes are not displayed. To view child nodes, expand the node.
- **E:** A disabled node. This node and related child nodes (not displayed) are ignored during run time.
- **F:** The shortcut menu.

## Node Display Status

You can display nodes in the following styles:

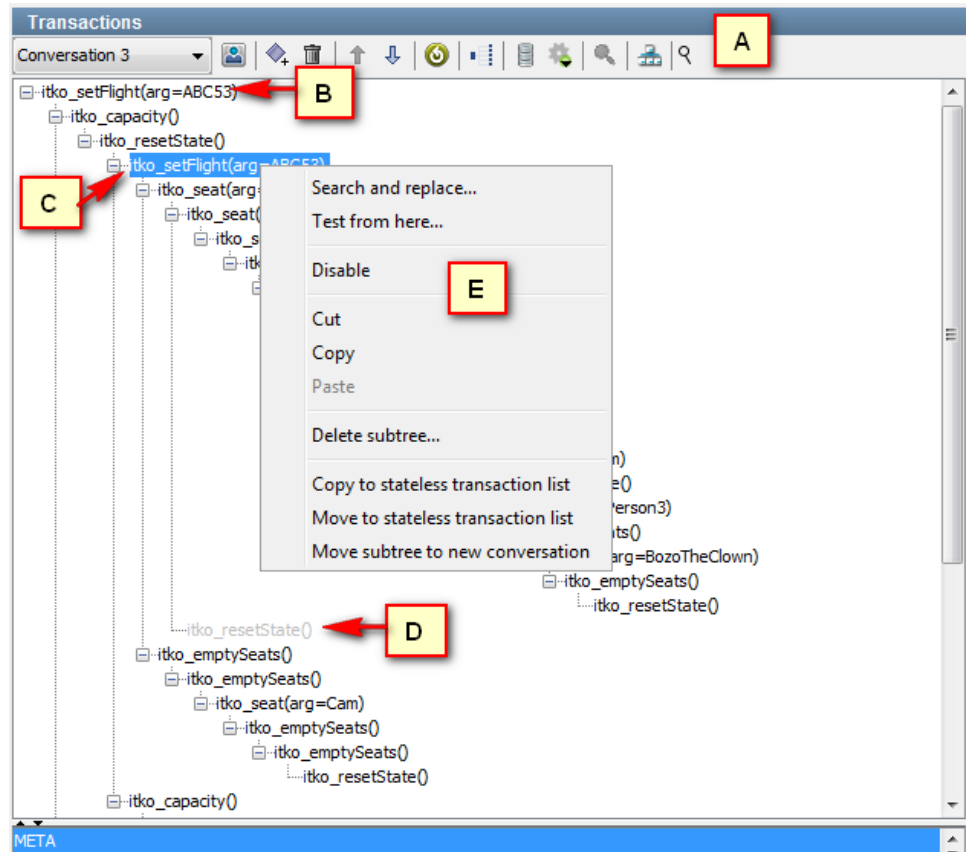
- Full size
- Compact (default)
- Condensed

In each style, a row of black dots at the bottom shows how many specific transactions belong to the node. In the condensed style, hollow dots indicate the number of arguments to the request. In the following graphic, the first node is displayed full size, the second is compact, and the third is condensed.



## Conversation Editor Tree View

The tree view displays the same information as the graph view more compactly. It displays nodes according to status, and has the following components:



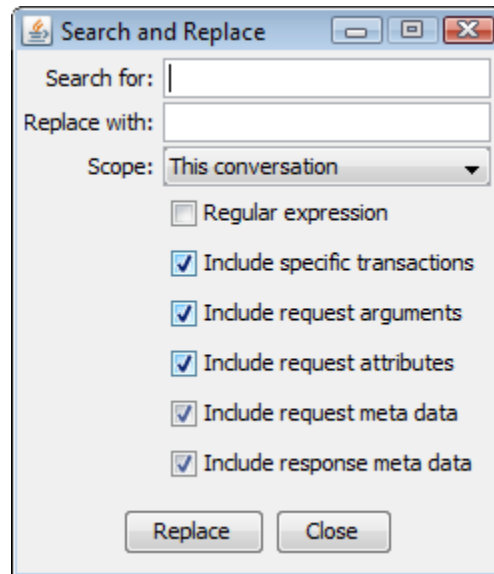
- **A:** Toolbar. For more information, see the [Conversation Editor Toolbar](#) (see page 293).
- **B:** A standard node.
- **C:** A selected node.
- **D:** A disabled node. This node and related child nodes (not displayed) are ignored during run time.
- **E:** The shortcut menu.

For more information about specific elements on this window, see:

- [Search and Replace Action](#) (see page 298)
- [Test from Here Action](#) (see page 299)
- [Highlight Navigation Possibilities](#) (see page 301)
- [Restructure a Conversation](#) (see page 303)

## Search and Replace Action

From the right-click menu, you can select the Search and Replace action to change values in the conversation or the transaction.



Scope lets you specify whether the search and replace extends to:

- This conversation (the default)
- This transaction
- This transaction and children
- The entire service image.

You can also use the check boxes to specify which pieces of the transactions to include.

## Test from Here Action

To test for an expected response from a selected transaction in both graph and tree views, use the Conversation Editor.

**Follow these steps:**

1. In the Conversation Tree editor, right-click a transaction.
2. Select Test from here from the shortcut menu.
3. The Create Test Request window opens.

**Create Test Request**

Previous:

Name:

Operation:

**Arguments**

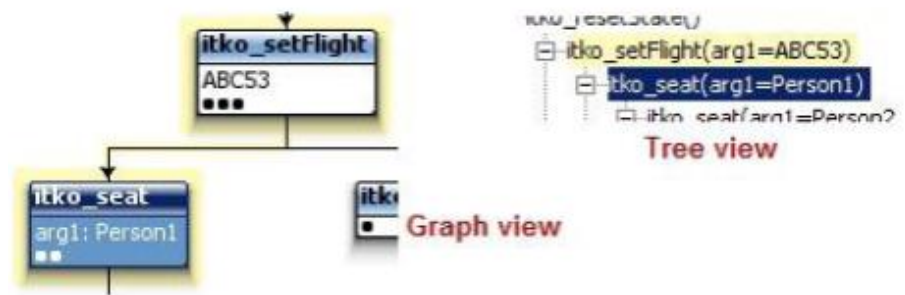
Key	Value
arg1	
arg2	

**VSE runtime properties (testExec)**

Key	Value
-----	-------

4. Enter the unique operation name and add argument key/value pairs, as appropriate.
5. Click Test.

As the following graphic shows, the result displays in blue with the path in yellow.



**Note:** If the test is not successful, the application displays the following error:

"No transaction matching the request follows the selected one in this conversation."

6. Click OK to continue.
7. Click Close.



## Highlight Navigation

To view navigation possibilities in a conversation that is based on the selected navigation tolerance, use the Conversation Tree.

Select a transaction and click View Navigation.

The default menu selection is No Navigation Highlight, which means no transactions are highlighted.

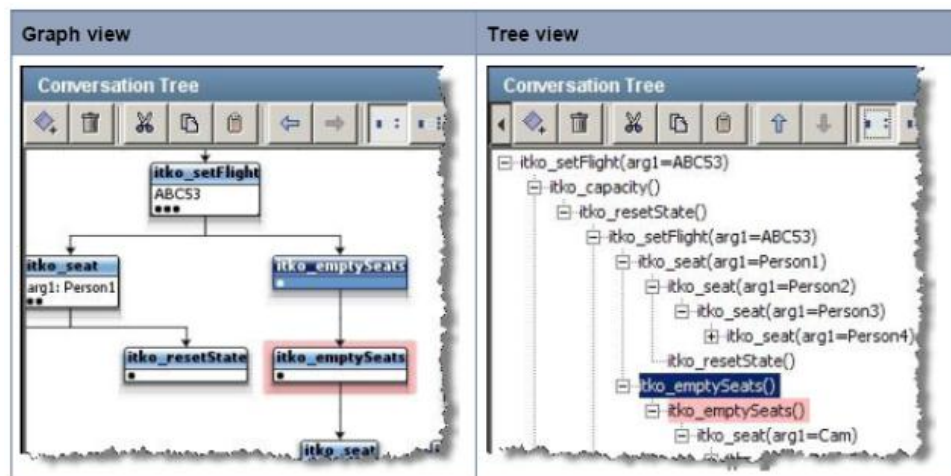
If you select a transaction and click the Highlight on transaction's tolerance option, the transaction and its associated transactions are highlighted according to the navigation tolerance defined for the selected transaction. Options are:

- Close
- Wide
- Loose

**Note:** If you use the drop-down list on the top right of the editor to change the navigation tolerance for the current transaction, that selection overrides the menu. To return the highlighting to the "correct" state in this case, reselect that navigation highlighting option.

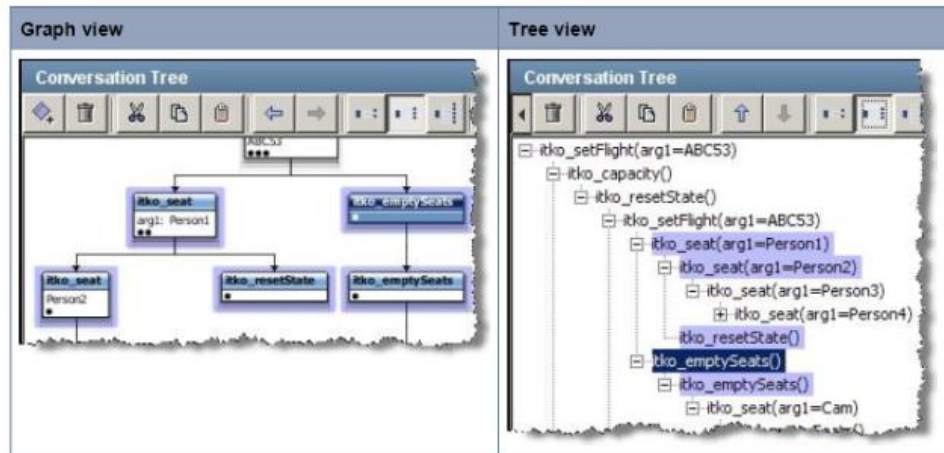
### Viewing Close Navigation

Select a transaction and click View Navigation. The transactions searched in the selected transaction subtree are highlighted in red.



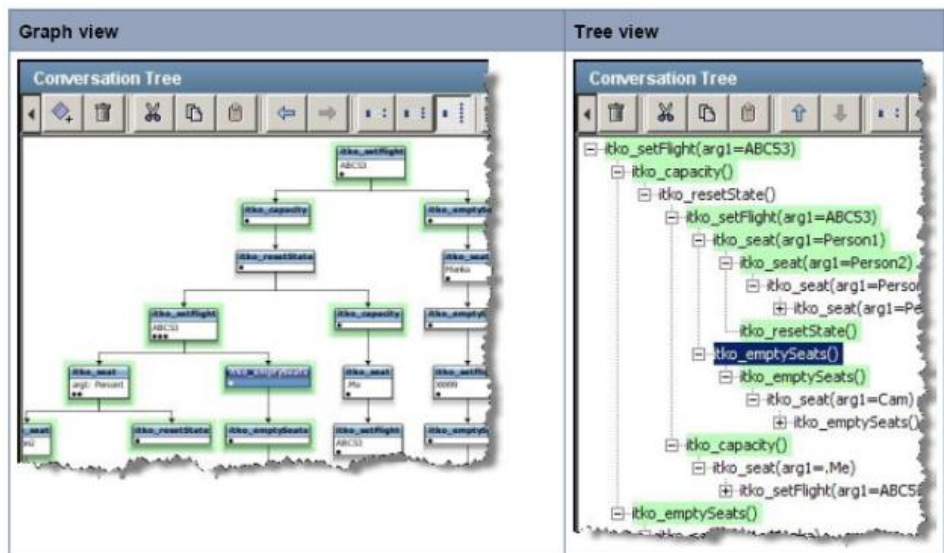
### Viewing Wide Navigation

Select a transaction and click the View Navigation button. The transactions that are searched in the selected transaction subtree and sibling subtrees are highlighted in blue.



### Viewing Loose Navigation

Select a transaction and click View Navigation. The transactions searched in the selected transaction subtree, sibling subtrees, and parent are highlighted in green. A full conversation restart is also possible.

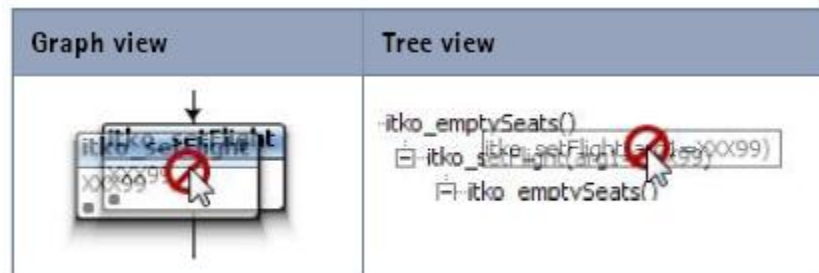


## Restructure Conversation

In some cases, you want to restructure a conversation by dragging and dropping a transaction and its subtree, if it has one. You restructure conversations in both the graph and tree views.

### Follow these steps:

1. Drag a transaction to the transaction that you want to be the new parent transaction.



As you drag the transaction, it displays the symbol for "not possible."

When it is possible to drop the transaction, the "not possible" symbol disappears.



2. Drop the transaction on the appropriate parent transaction.

The transaction and any transactions in its subtree move below the new parent.

## Service Images for JMS Transport Protocol

This topic describes specific characteristics of service images that are created using the [JMS transport protocol](#) (see page 139).

The following items apply to the request side:

- By default, the operation name for the transaction is set to the request channel name defined in the request step in the virtual service model.
- The metadata properties that are not grouped with a prefix contain the queue and connection information for the proxy request queue.
- The metadata properties that start with **liveRequest** contain the queue and connection properties for the live request queue.
- The **channel.name** metadata property must match the request channel name defined in the request and live invocation steps in the virtual service model. This property is the only required metadata property on the request side. You can change the virtual service model's request queues as needed, as long as the request channel name remains the same.

The following items apply to the response side:

- The metadata properties that are not grouped with a prefix contain the queue and connection information for the proxy response queue.
- The metadata properties that start with **msg** are used to reconstruct the JMS response message and its properties. The only required property is **msg.type**, which indicates the type of message to send.
- The **channel.name** metadata property must match a response channel name defined in the respond and live invocation steps in the virtual service model. The value specifies the response channel name to be used to send the response to the client. If there is more than one response, the responses can specify different channel names.
- The metadata properties that start with **liveResponse** contain the queue and connection properties for the live response queue.

## Chapter 10: Editing a VSM

---

A Virtual Service Image recording creates a Virtual Service Model (VSM) with six steps or eight steps, depending on the option chosen (More Flexible or More Efficient). Sometimes you would like to edit the VSM by editing generated steps or adding more steps.

You can skip this section unless you must edit a VSM or you must create a VSM without a recorder.

A VSM is a specialized test case and therefore editing or creating a VSM is similar to editing or creating a test case. Many types of steps can be added to a VSM. You can add a step that does not appear in this menu; however, some other step types can make a VSM undeployable.

Access the menu of step types from the VSM Editor by selecting Add a new step, Virtual Service Environment from the menu.

This section contains the following topics:

- [Virtual Service Router Step](#) (see page 307)
- [Virtual Service Tracker Step](#) (see page 308)
- [Virtual Conversational/Stateless Response Selector Step](#) (see page 309)
- [Virtual HTTP/S Listener Step](#) (see page 310)
- [Virtual HTTP/S Live Invocation Step](#) (see page 312)
- [Virtual HTTP/S Responder Step](#) (see page 314)
- [Virtual JDBC Listener Step](#) (see page 315)
- [Virtual JDBC Responder Step](#) (see page 316)
- [Socket Server Emulator Step](#) (see page 317)
- [Messaging Virtualization Marker Step](#) (see page 319)
- [Compare Strings for Response Lookup Step](#) (see page 320)
- [Compare Strings for Next Step Lookup Step](#) (see page 322)
- [Virtual Java Listener Step](#) (see page 324)
- [Virtual Java Live Invocation Step](#) (see page 325)
- [Virtual Java Responder Step](#) (see page 326)
- [Virtual TCP/IP Listener Step](#) (see page 327)
- [Virtual TCP/IP Live Invocation Step](#) (see page 329)
- [Virtual TCP/IP Responder Step](#) (see page 330)
- [Virtual CICS Listener Step](#) (see page 331)
- [Virtual CICS Responder Step](#) (see page 331)
- [CICS Transaction Gateway Listener Step](#) (see page 332)
- [CICS Transaction Gateway Live Invocation Step](#) (see page 334)
- [CICS Transaction Gateway Responder Step](#) (see page 335)
- [Virtual DRDA Listener Step](#) (see page 336)
- [Virtual DRDA Response Builder Step](#) (see page 336)
- [Virtual DRDA Live Invocation Step](#) (see page 337)
- [IMS Connect Listen Step](#) (see page 338)
- [IMS Connect Live Invocation Step](#) (see page 339)
- [Virtual IMS Connect Responder Step](#) (see page 340)
- [JMS VSE Steps](#) (see page 341)
- [JCo IDoc Listener Step](#) (see page 346)

[JCo IDoc Live Invocation Step](#) (see page 347)

[JCo IDoc Responder Step](#) (see page 347)

[JCo RFC Listener Step](#) (see page 348)

[JCo RFC Live Invocation Step](#) (see page 349)

[JCo RFC Responder Step](#) (see page 350)

## Virtual Service Router Step

This step routes a request from a virtual service listen step to the response selector step and the protocol-specific live invocation step, or both. The decision is made based on the current execution mode for the running model.

### Follow these steps:

1. Complete the following fields as described:

#### **Live invocation step**

Select the step for the live invocation from the list.

#### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

#### **When in dynamic mode, determine real mode using**

Select Subprocess or Script.

2. To test your parameters for the step, click Test.

## Virtual Service Tracker Step

Use this step to track responses in a running virtual service and (optionally) validate against a live system. This validation allows for easier service model debugging and model "healing".

Complete the following fields as described:

**Image Response**

Select the Image response file.

**Live Response**

Select the Live response file.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.



## Virtual Conversational/Stateless Response Selector Step

You could consider the Virtual Conversational/Stateless Response Selector step as the main step in any VSM. It reviews a specified service image and selects an appropriate virtual response for a specific request. Because there can be multiple responses for a request, the responses are always shown as a list. The step is typically created when you record and virtualize some form of service traffic.

Complete the following fields as described:

### **Service Image Location**

Select from the drop-down list of available service images to associate with this step. When you have chosen a service image here, view it or edit it by clicking Open and it opens in a new tab.

### **Request property name**

To define the property to review for the inbound request, set the property name. The property name is usually the response from the previous step.

### **Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual HTTP/S Listener Step

Use the Virtual HTTP/S Listener step to simulate an HTTP server, including SSL support. The step listens for incoming HTTP requests and converts them to a standard virtual request format.

The default name for the Virtual HTTP/S Listener step is **Virtual HTTPS Listener<portnumber>**. You can rename the step at any time.

Complete the following fields as described:

### Listen port

Enter the port on which DevTest listens for the HTTP/S traffic.

### Bind address

Enter the local IP address on which connections can come in. With no bind address specified, the listen step accepts connections on the specified port regardless of the NIC (or the IP address) on which it comes in.

### Bind only

To acquire the network resource and move to the next step, select this check box. A second Listen step that does not use the Bind only option is required. This option enables the model to listen on a port (the application queues requests until a listen step consumes them). The model performs setup tasks before dropping into the wait/process/respond loop. For example, Step 1 of the model acquires the listening port (using Bind only) and Step 2 triggers external software that sends requests.

### Use SSL to client

Select this check box to simulate a secure HTTP/S website. Then supply the SSL keystore information.

### SSL keystore file

Click Select... to browse to your SSL keystore file. The same keystore file must be available to the VSE server to which the VS model is deployed.

### Keystore password

Enter the keystore password, then click Verify.

### Base path

Identify the HTTP requested resource URIs that the listen step is to process. When the request comes in, the list of queue names is scanned for a name (base path) that starts the URI on the request. The queue name that matches is the one into which the request is placed. The listen step that is associated with the queue (by base path) processes the request.

### Format step response as XML

The VSE framework expects Respond steps to accept one of the following:

- A response object

- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual HTTP/S Live Invocation Step

Use the Virtual HTTP/S Live Invocation step to make a real HTTP call to real server in the context of a virtualized HTTP service. This step is typically created by recording and virtualizing some form of HTTP traffic. The step performs the real request, which is based on the current VSE request in use.

The default name for the Virtual HTTP/S Live Invocation step is **Virtual HTTPS LiveInvocation<portnumber >**. You can rename the step at any time.

Complete the following fields as described:

### Target server

Enter the name of the server to which the request is made.

### Target port

Enter the name of the port on which the request is made.

### Replacement URI

To replace the full URI in a GET/POST request, enter a new target path field. You can provide the URI as a DevTest property. This field can be blank, in which case the URI from the live request is used.

### Do not modify host header parameter received from client

If selected, this option instructs the live invocation to forward the host header received from the client application to the target server. If cleared, the live invocation regenerates the host header parameter to be **host: <target host>:<target port>**.

### Use SSL to server

Specifies whether to send an HTTPS (secured layer) request to the server.

#### Values:

- **Selected:** Sends an HTTPS (secured layer) request to the server. You can select Use SSL to Server and not select Use SSL to Client. In that case, a plain TCP connection is presented for recording, but those requests are sent to the server using SSL.

**Cleared:** The application does not send an HTTPS request to the server

### SSL keystore file

Click Select... to browse to your SSL keystore file. The same keystore file must be available to the VSE server to which the VS model is deployed.

The first certificate in the keystore is presented to the target server when client authentication is required. This certificate overrides the certificate that the **ssl.client.\*** properties in **local.properties** specify for the live invocation.

### Keystore password

Enter the keystore password, then click Verify.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**Bad Response Codes**

Defines a failure respons from the system.

**Format:** A comma-delimited list of three-character codes, with each character being either a numeric or the letter x (wildcard).

**Example:** 4xx,5xx

**VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**Note:** The Virtual HTTP/S Live Invocation step supports the **`lisa.http.timeout.socket`** and **`lisa.http.timeout.connection`** properties to control the client sockets used. The **`lisa.vse.http.live.invocation.max.idle.socket`** property controls how long an idle client socket waits before it is too old to use. This defaults to 2 minutes.

## Virtual HTTP/S Responder Step

Use this step with the Virtual HTTP/S Listener step to transmit responses to HTTP requests produced by the listener. The step uses a virtual response as the reply to the corresponding request using the HTTP/S protocol.

You can create this step by recording and virtualizing HTTP traffic.

Complete the following fields as described:

### **Responses list property name**

Specifies the name of the property in which to look for the response to send.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### **Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.

## Virtual JDBC Listener Step

Use the Virtual JDBC Listener step to control the simulation of JDBC database traffic. The step manages the communication with the simulation driver that is embedded in the database client.

The default name for the Virtual JDBC Listener step is **Virtual JDBC Listener<portnumber>**. You can rename the step at any time.

Complete the following fields as described:

### Endpoint Information

Set up the simulation host and range of ports (the default is 2999) as appropriate. JDBC VSE supports multiple endpoints during recording and playback. The endpoint information is a table that contains:

- Driver Host
- Base Port
- Max Port

When Base Port and Max Port differ, a unique endpoint is created for each port between Base Port and Max Port, inclusive.

### Format step response as XML

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

### If Environment Error

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### Connect/Disconnect

To connect to the JDBC simulator, click Connect. If connected, click Disconnect to end the connection. You can use this button to validate the connection information.

### Installed and Initialized JDBC Drivers

Lists the JDBC drivers that are installed and initialized in the database client.

### Current SQL Activity

Identifies the current SQL activity in the database client.

## Virtual JDBC Responder Step

This step takes the result of a JDBC data call as selected by a conversational response selection step. The step then sends the result to the simulation driver that is embedded in the database client.

Complete the following fields as described:

### **Responses list property name**

Specifies the name of the property in which to look for the response to send.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### **Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.



## Socket Server Emulator Step

Use this step to simulate any text-based server socket (typically HTTP). The Socket Server Emulator step supports listening, responding, and binding. The Socket Server Emulator step is low level. If you use this step, you must verify that the block of text that the respond step sends is fully HTTP-compliant.

**Note:** When you use the Socket Server Emulator step in response mode, the text to go out **must** result in a valid HTTP response message.

Complete the following fields as described:

### Process mode

From the list, select the process mode.

#### Values:

- Full Process
- Asynchronous setup: Acquire the network resource and move to the next step
- Listen Only
- Respond Only
- **Default:** Full Process

### Listen port

Enter the port on which DevTest listens for the HTTP/S traffic.

### Bind address

Enter the local IP address on which connections can come in. With no bind address specified, the listen step accepts connections on the specified port regardless of the NIC (or the IP address) on which it comes in.

### Close immediately

This option instructs the step to do the configured work and then immediately clean up its network resources. Select this check box to use design-time testing of this step.

### Use SSL

Select this check box to simulate a secure HTTPS website. Then supply the SSL keystore information.

### SSL keystore file

Click Select... to browse to your SSL keystore file. The same keystore file must be available to the VSE server to which the VS model is deployed.

### Keystore password

Enter the keystore password, then click Verify.

### Base path

Identify the HTTP requested resource URIs that the listen step is to process. When the request comes in, the list of queue names is scanned for a name (base path) that starts the URI on the request. The queue name that matches is the one into which the request is placed. The listen step that is associated with the queue (by base path) processes the request.

### If Environment Error

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### Record terminator

For a socket emulator that simulates a record-based service, enter the character to mark the end of a record. If you leave this field blank, either line-oriented records or the HTTP protocol are simulated.

### Ensure proper HTTP response format

When in a process mode that sends a response and the response is to be a valid HTTP response, this option verifies:

- The HTTP headers in the response text are correctly formatted
- (If necessary) The **Content-Length**: HTTP response header is present and correct.

This check box only verifies that the line separators are HTTP-compliant and that the **Content-Length** header is present and accurate. However, to work completely, the message must already be a well-formed HTTP message.

**Default:** Selected

### Listener status

Indicates whether the listener is running.

### Test

Click to test the listener setup.

### Clear Listener

Click to stop the test of the step.

### Response tab

The Response to Send includes the text for the response.

### Read Response From File

Click to browse the file system for a response.

### Request tab

Displays the Last/Original Request, which is used only at design time. This tab displays the last request that the step received.

The default name for the Socket Server Emulator step is **Socket Server Emulator** *<portnumber>*. You can rename the step at any time.

## Messaging Virtualization Marker Step

Use this step to designate that a message-based test case is designed for use in the Virtual Service Environment. If the test case listens or responds through JMS, add this step to the virtual service model to verify that it can be deployed to VSE.

**Note:** You do not need to use this step with the JMS transport protocol.

## Compare Strings for Response Lookup Step

This step reviews an incoming request to a virtual service. The step then determines the appropriate response in a stateless fashion, without referring to any service image. The stateful portions of VSE are not supported. You can match incoming requests using partial text match, regular expression, and others.

Complete the following fields as described:

**Text to match**

Enter the text against which the criteria are matched. This value is typically a property reference, such as LASTRESPONSE.

**Range to match**

Enter the start and end of the range.

**If no match found**

From the list, select the step to go to if no match is found.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**Store responses in a compressed form...**

Specify whether to compress the responses in the test case file.

**Default:** Selected

**Case Response Entries**

Add, move, and delete entries.

**Enabled**

Specifies whether to enable or ignore an entry. Clear this check box to ignore an entry.

**Default:** Selected when you add an entry.

**Name**

Enter a unique name for the case response entry.

**Delay Spec**

Enter the delay specification range. The default is **1000-10000**, which indicates to use a randomly selected delay time from 1000 milliseconds through 10000 milliseconds. The syntax is the same format as Think Time specifications.

**Criteria**

This area provides the string to compare against the Text to match field. To edit the criteria:

In the Case Response Entries area select the appropriate row.

From the Criteria list, select a setting.

**Compare Type**

Select an option from the list:

- Find in string
- Regular expression
- Starts with
- Ends with
- Exactly equals
- **Default:** Find in string

**Response**

This area provides the response of this step if the entry matches the Text to match field. To edit the response, in the Case Response Entries area select the appropriate row, then select a different setting from the Response list.

**Criteria**

You can update the criteria string for an entry.

**Response**

You can update the step response for an entry.

## Compare Strings for Next Step Lookup Step

Use this step to review an incoming request and determine the appropriate next step. You can match incoming requests using partial text match and regular expression, and others.

Each matching criterion specifies the name of the step to which to transfer if the match succeeds.

Complete the following fields as described:

**Text to match**

Enter the text against which the criteria are matched. This value is typically a property reference, such as LASTRESPONSE.

**Range to match**

Enter the start and end of the range.

**If no match found**

From the list, select the step to go to if no match is found.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**Next Step Entries**

Add, move, and delete entries.

**Enabled**

Specifies whether to enable or ignore an entry. Clear this check box to ignore an entry.

**Default:** Selected when you add an entry.

**Name**

Enter a unique name for the next step entry.

**Delay Spec**

Enter the delay specification range. The default is **1000-10000**, which indicates to use a randomly selected delay time from 1000 milliseconds through 10000 milliseconds. The syntax is the same format as Think Time specifications.

**Criteria**

This area provides the string to compare against the Text to match field. To edit the criteria:

In the Case Response Entries area select the appropriate row.

From the Criteria list, select a setting.

### Compare Type

Select an option from the list:

- Find in string
- Regular expression
- Starts with
- Ends with
- Exactly equals
- **Default:** Find in string

### Next Step

From the list, select the step to go to if the match is found.

### Criteria

You can update the criteria string for an entry.

## Virtual Java Listener Step

Use this step to handle virtualized JVM calls, such as calls to an EJB or other remote system. The step listens for the method calls that the DevTest Java Agent intercepts, and converts them to a standard VSE request.

### Available Online Agents

Lists all the online agents that are available to connect.

### Connected Agents

Lists all the online or offline agents that are connected for the virtual service model. The offline agents display in a gray italic font.

### Connecting Agents

You can select agents from the Available Online Agents list. Select the agent or agents and click the right arrow button. The agent moves to the connected agents list.

When you select an agent from either the Available Online Agents list or the connected agents list, the information bar below the list displays host and Main Class information for the agent.

To add an agent manually to the connected agents list, use the Add Agent field above the Connected Agents.

The agent name cannot be empty or already present in the connected agents list. If the agent name entered exists in the online agents list, it is moved from the online agents to the connected agents list.

If an existing agent is not in the Available Online Agents list, type it in the dialog and

click Add .

This mechanism is primarily provided for adding offline agents that were not previously in the connected list.


### Disconnecting Agents

To disconnect an agent, select the agent from the connected agents list and click the left arrow button.

### Selecting Classes and Protocols



To search for classes, select the Search for Classes arrow and enter a class name. These classes are entered as fully qualified names (including package), using regular expressions. To select classes, select the class name on the list and select the right arrow to move the class into the right pane. Some classes appear more than once; it is only necessary to select a class once for it to be virtualized.

To enter a class manually, select the Manually Enter a Class Name arrow. Enter the name of the class. To move the class in the right pane, select it and click the right arrow. To retrieve a list of classes that the DevTest agent suggested for virtualization, select the Agent Suggestions arrow and click Retrieve .

To add a protocol to the recording, select the Protocols arrow. From the list of available protocols, select any to record and click the right arrow to move them in the right pane.

To view or change the configuration information for a protocol, double-click any row with three dots (...) to the right of the protocol name. The Protocol Configuration window opens, and you can update the parameters.

## Virtual Java Live Invocation Step

Complete the following fields as described:

### Format step response as XML

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

### If Environment Error

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual Java Responder Step

Use this step with the Virtual Java Listener step to provide responses for virtualized JVM calls.

Complete the following fields as described:

### **Responses list property name**

Specifies the name of the property in which to look for the response to send.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### **Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.

## Virtual TCP/IP Listener Step

Use this step to simulate TCP/IP connections to a server application. The step listens for incoming TCP/IP traffic and converts it to a standard VSE request.

The default name for the Virtual TCP/IP Listener step is **Virtual TCP/IP Listener<portnumber>**. You can rename the step at any time.

Complete the following fields as described:

### Listen port

Enter the port on which DevTest listens for the TCP/IP traffic.

### Bind address

Enter the local IP address on which connections can come in. With no bind address specified, the listen step accepts connections on the specified port regardless of the NIC (or the IP address) on which it comes in.

### Use SSL to server

Specifies whether to send an SSL (secured layer) request to the server.

#### Values:

- **Selected:** Sends an SSL request to the server. You can select Use SSL to server and not select Use SSL to client. In that case, a plain TCP connection is presented for recording, but those requests are sent to the server using SSL.

**Cleared:** The application does not send an SSL request to the server.

### Use SSL to client

This option is only enabled if Use SSL to Server has been selected. Check whether we can play back an SSL request from the client using a custom client keystore. When you specify Use SSL to client, you are allowed to specify a custom keystore and a passphrase. If these values are entered, they are used rather than the hard-coded defaults.

### SSL keystore file

Click Select... to browse to your SSL keystore file. The same keystore file must be available to the VSE server to which the VS model is deployed.

### Keystore password

Enter the keystore password, then click Verify.

### Format step response as XML

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual TCP/IP Live Invocation Step

Use this step to make a real TCP/IP call to a real server in the context of a virtualized TCP/IP service. The step is typically created by recording and virtualizing some form of TCP/IP traffic. The step performs the real request, which is based on the current VSE request in use.

The default name for the Virtual TCP/IP Live Invocation step is **TCP Protocol Live Invocation<portnumber>**. You can rename the step at any time.

Complete the following fields as described:

### Target port

Enter the name of the port on which the request is made.

### Target server

Enter the name of the server to which the request is made.

### Use SSL to server

Specifies whether to send an SSL (secured layer) request to the server.

#### Values:

- **Selected:** Sends an SSL request to the server. You can select Use SSL to server and not select Use SSL to client. In that case, a plain TCP connection is presented for recording, but those requests are sent to the server using SSL.

**Cleared:** The application does not send an SSL request to the server.

### Use SSL to client

This option is only enabled when you select Use SSL to Server. Use SSL to client specifies whether the application can play back an HTTPS request from the client using a custom client keystore.

#### Values:

- **Selected:** You can specify a custom keystore and a passphrase. If you enter these values, the application uses them instead of the hard-coded defaults.
- **Cleared:** The application cannot play back an SSL request from the client using a custom client keystore.

### SSL keystore file

Click Select... to browse to your SSL keystore file. The same keystore file must be available to the VSE server to which the VS model is deployed.

### Keystore password

Enter the keystore password, then click Verify.

### Treat response as text

Specifies whether the application processes the response as text.

**Values:**

- **Selected:** The application process the response as text.

**Cleared:** The application does not process the response as text.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual TCP/IP Responder Step

Complete the following fields as described:

**Responses list property name**

Specifies the name of the property in which to look for the response to send.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.

## Virtual CICS Listener Step

The Virtual CICS Listener step simulates a CICS LINK server.

For information about the fields on this panel, see [Using the CICS Programs to Virtualize Panel](#) (see page 179).

## Virtual CICS Responder Step

The Virtual CICS Responder step is used with the Virtual CICS Listener step to transmit responses to requests produced by the listener. The step takes a virtual response and uses it as the reply to the corresponding request.

Complete the following fields as described:

**Responses list property name**

Specifies the name of the property in which to look for the response to send.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

To populate the Conversational Model Properties list, use the Add and Remove buttons.

## CICS Transaction Gateway Listener Step

Use the CICS Transaction Gateway Listener step to simulate a CTG server, including SSL support. The step listens for incoming CTG requests and converts them to a standard virtual request format.

Complete the following fields as described:

### **Listen on port**

Enter the port on which DevTest listens for the CTG traffic.

### **Bind address**

Enter the local IP address on which connections can come in. With no bind address specified, the listen step accepts connections on the specified port regardless of the NIC (or the IP address) it comes in on.

### **Bind only**

To acquire the network resource and move to the next step, select this check box. A second Listen step that does not use the Bind only option is required. This option enables the model to listen on a port (the application queues requests until a listen step consumes them). The model performs setup tasks before dropping into the wait/process/respond loop. For example, Step 1 of the model acquires the listening port (using Bind only) and Step 2 triggers external software that sends requests.

### **Expect SSL From Clients**

If you select this check box, the recorder expects clients to connect to it using SSL. The related keystore and password, if provided, are used to obtain security information (such as certificates).

### **SSL keystore file**

Specifies the name of the keystore file.

### **Keystore password**

Specifies the password associated with the specified keystore file.

### **CTG Protocol version**

Specifies the version of the CTG protocol to use.

### **Locale**

Defines the locale that represents the language and country code that are reported to the CTG client during the initial protocol handshake.

### **JVM text**

Describes the JVM on the mainframe.

### **Server class**

Defines a string that the application reports to the client. This field is not typically used.



**Client app ID**

Defines a string that the application reports to the client. This field is not typically used.

**Has security**

Specifies to the CTG client whether user authentication is required.

**Enable server ping**

When a CTG client connects to a server, that server starts sending "ping" messages to the client regularly to verify the connection. This is not strictly necessary where VSE is used in a testing environment, but selecting this option causes the VSE CTG server to emulate those "ping" messages.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## CICS Transaction Gateway Live Invocation Step

Use this step to make a real CTG call to a real server in the context of a virtualized CTG service. The step is typically created by recording and virtualizing some form of CTG traffic. It performs the real request based on the current VSE request in use.

Complete the following fields as described:

**Target server**

Enter the name or IP address of the target host where the CTG server runs.

**Target port**

Enter the number of the port on which the CTG server listens.

**Initiate SSL to the target server**

If selected, sends SSL (Secure Socket Layer) request to the server.

**SSL keystore file**

Specifies the name of the keystore file.

**Keystore password**

Specifies the password associated with the specified keystore file.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## CICS Transaction Gateway Responder Step

se this step with the CICS Transaction Gateway Listener step to transmit responses to CTG requests that the listener produces. This step uses a virtual response as the reply to the corresponding request using the CTG protocol.

You can create this step by recording and virtualizing CICS Transaction Gateway traffic.

Complete the following fields as described:

### **Responses list property name**

Specifies the name of the property in which to look for the response to send.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### **Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.

## Virtual DRDA Listener Step

Use the Virtual DRDA Listener step to virtualize DRDA traffic over TCP/IP. The step intercepts incoming DRDA data and converts it to a standard virtual request format.

Complete the following fields as described:

**Listen/Record on port**

**Target host**

**Target port**

**DB2 IP address**

**LISA IP address**

**Stored proc param delimiter**

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## Virtual DRDA Response Builder Step

This step decomposes DRDA requests into individual DRDA commands to mitigate the complexity of VSE receiving DRDA commands in different groupings. For example, DRDA sends the same set of commands in sets of one, two, three, or four for each request. Matching each command and building a composite response makes our models as flexible as possible for playback.

## Virtual DRDA Live Invocation Step

Use the Virtual DRDA Live Invocation step to make a real DRDA call to a real server in the context of a virtualized DRDA service. Recording and virtualizing DRDA traffic typically creates the step. The step performs the real request in regard to the current VSE request in play.

Complete the following fields as described:

**Listen/Record on port****Target host****Target port**

Enter the name of the port on which the request is made.

**DB2 IP address****LISA IP address****Stored proc param delimiter****Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

## IMS Connect Listen Step

Use the IMS Connect Listener step to respond to IMS Connect requests by the IMS Connect virtual service.

Enter the following fields as described:

**Listen/Record on port**

Defines the port on which the client communicates to DevTest.

**Target host**

Disabled for the Listen step.

**Target port**

Disabled for the Listen step.

**IMS Format file**

Disabled for the Listen step.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## IMS Connect Live Invocation Step

Complete the following fields:

**Listen/Record on port**

Not applicable for the Live Invocation step.

**Target host**

Specifies the name or IP address of the target host where the server runs.

**Target port**

Specifies the target port number listened to by the IMS server.

**IMS Format file**

To use the IMS Connect support that is included in DevTest by default, leave this field blank.

**Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

## Virtual IMS Connect Responder Step

Use this step with the IMS Connect Listener step to transmit IMS Connect responses.

You can create this step by recording and virtualizing IMS Connect traffic.

Enter the following fields as described:

**Responses list property name**

Specifies the name of the property in which to look for the response to send.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

**Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.



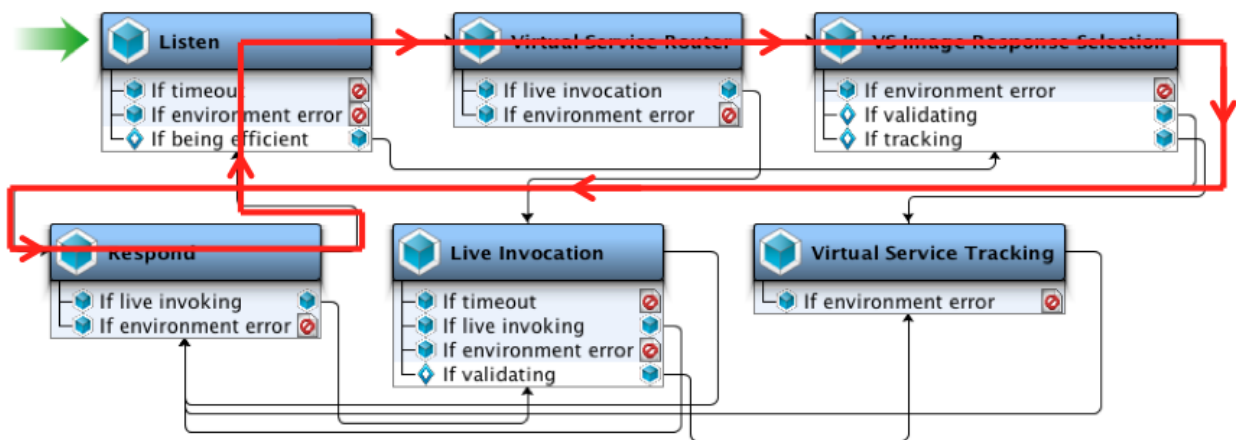
## JMS VSE Steps

The following steps are specific to virtual service models that are created using the [JMS transport protocol](#) (see page 139):

- JMS VSE Listen
- JMS VSE Respond
- JMS VSE Live Invocation

During normal operation, the execution flow of the virtual service model is like any other VSE service.

The following graphic uses an overlay to illustrate the flow.



The step with the name **Listen** is the JMS VSE Listen step. The step with the name **Respond** is the JMS VSE Respond step. The JMS VSE Live Invocation step is not used.

1. The **Listen** step receives a request message and converts it into a VSE request.
2. The **Virtual Service Router** step routes the flow to the response selection step.
3. The **VS Image Response Selection** step selects a matching transaction from the service image and produces a VSE response.
4. The **Respond** step sends one or more response messages in the VSE response.
5. Return to Step 1.

During live invocation, the execution flow of the virtual service model is more complicated.

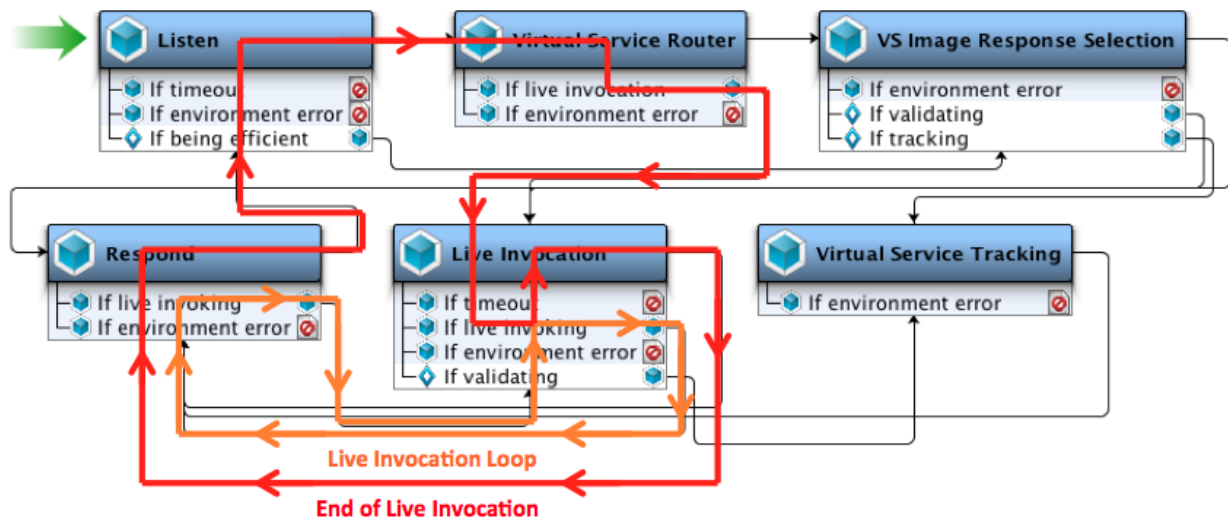
The following aspects of asynchronous messaging make live invocation difficult:

- A single request can have multiple responses.
- Responses can take any amount of time to be returned.

- In general, it is impossible to determine when all of the responses have been received.

As a result, the **Live Invocation** step and the **Respond** step are run in a loop.

The following graphic uses an overlay to illustrate the flow.



The step with the name **Listen** is the JMS VSE Listen step. The step with the name **Live Invocation** is the JMS VSE Live Invocation step. The step with the name **Respond** is the JMS VSE Respond step.

1. The **Listen** step receives a request message and converts it into a VSE request.
2. The **Virtual Service Router** step routes the flow to the live invocation step.
3. The **Live Invocation** step forwards the request to the live service.
4. The **Live Invocation** step starts listening on every live response queue.
5. The **Live Invocation** step receives a single response from any of the live response queues and converts it into a VSE response.
6. The **Respond** step returns one response message to the client.
7. Return to Step 5 and repeat until the **Live Invocation** step determines that the transaction is complete. Any of the following conditions can cause the **Live Invocation** step to make this determination:
  - The timeout is set and it passes without receiving another response from any of the live response queues. The default value is 30 seconds. The timeout is an advanced parameter that can be set in the VSE recorder or in the **Live Invocation** step.

- The maximum number of responses is set and that number has been reached. The default value is 1, which indicates that the loop can recur only once. The maximum number of responses is an advanced parameter that can be set in the VSE recorder or in the **Live Invocation** step.
  - The virtual service model is running close enough to capacity that it needs the current model execution thread to break out of its transaction and handle a new request. If the timeout and the maximum number of responses have not been set, this action is the only way for the model to break out of waiting for live responses and loop back to the **Listen** step.
8. The **Live Invocation** step constructs a final VSE response that contains all of the response messages that went through the loop. The step loops a final time to the **Respond** step.
  9. The **Respond** step does not send any messages on the final loop. Instead, the **Respond** step completes the typical VSE state cleanup tasks.
  10. Return to Step 1.

## JMS VSE Listen Step

The JMS VSE Listen step listens for incoming JMS requests and converts them to standard VSE requests.

The step editor has basic and advanced parameters. To display the advanced parameters, click PRO at the top of the editor.

The Request tab contains the list of receive operations.

The Channel Name field defines the request channel name. The value must match the operation name that is defined in the service image.

To disable and reenable individual request channels, use the Enabled check box. At least one request channel must be enabled.

The ReplyTo Mappings tab is applicable to the segregated messaging scenario. This type of scenario occurs under the following conditions:

- A set of queues is accessible to DevTest.
- Another set of queues is not accessible to DevTest.
- The application runs on the non-accessible queues.
- Messages are automatically forwarded word for word to the DevTest-accessible queues automatically.

Each mapping consists of the following items:

- The response-side channel name
- The client reply-to destination used by the client

## JMS VSE Respond Step

The JMS VSE Respond step sends one or more response messages in the VSE response.

The step editor has basic and advanced parameters. To display the advanced parameters, click PRO at the top of the editor.

The Channel Name field defines the response channel name. The value must match the **operation.name** metadata property that is defined in the service image.

To disable and reenable individual response channels, use the Enabled check box.

## JMS VSE Live Invocation Step

The JMS VSE Live Invocation step sends requests to the live service.

The step editor has basic and advanced parameters. To display the advanced parameters, click PRO at the top of the editor.

The Live Request Send tab contains the list of operations to use for sending a live request.

The Channel Name field defines the request channel name. The value must match the channel name in the JMS VSE Listen step.

To disable and reenable individual request channels, use the Enabled check box. At least one request channel must be enabled.

The Live Response Receive tab contains the list of operations to use for receiving live responses.

The Channel Name field defines the response channel name. The value must match the channel name in the JMS VSE Respond step.

To disable and reenable individual response channels, use the Enabled check box. At least one response channel must be enabled.

The Timeout parameter and the Maximum Responses parameter are among the criteria that the step can use to determine whether to leave the [live invocation loop](#) (see page 341).

The ReplyTo Mappings tab is applicable to the segregated messaging scenario. This type of scenario occurs under the following conditions:

- A set of queues is accessible to DevTest.
- Another set of queues is not accessible to DevTest.
- The application runs on the non-accessible queues.
- Messages are automatically forwarded word for word to the DevTest-accessible queues automatically.

Each mapping consists of the following items:

- The response-side channel name
- The service reply-to destination that must be sent to the service for its response to come through the live response queue on the response channel

## JCo IDoc Listener Step

Use this step to respond to JCo IDoc requests from the JCo IDoc virtual service. To create this step, record and virtualize JCo IDoc traffic.

Complete the following fields:

### **Client RFC Connection Properties**

Defines the Client RFC Connection properties file that contains connection properties that VSE uses to register itself under a program ID to an SAP gateway and receive IDocs. The properties should be the same as those specified in a .jcoServer file.

### **Client RFC Destination Name**

Specifies a unique name that identifies the RFC destination.

### **Client System Connection Properties**

Specifies the Client System Connection properties file that contains connection properties to return IDocs to the client SAP system. These properties should be the same as those specified in a .jcoDestination file that can be used to connect to the client SAP system.

### **Client System Name**

Specifies a unique name to identify the client SAP system.

### **Request Identifier XPath Expressions**

Specifies the XPath expressions that the protocol uses with the request IDoc XML to generate an identifier. The request identifier XPath expressions can be a single XPath expression. This identifier is used to correlate a request IDoc to a response IDoc. XPath expressions can also be a comma-separated list of XPath expressions, in which case the resulting values from the multiple expressions are concatenated (separated by dashes) and used as an identifier.

### **Response Identifier XPath Expressions**

Defines the XPath expressions that the protocol uses with the response IDoc XML to generate an identifier. The Response Identifier XPath Expressions can be a single XPath expression. This identifier is used to correlate a response IDoc to a request IDoc that was received earlier. XPath expressions can also be a comma-separated list of XPath expressions, in which case the resulting values from the multiple expressions are concatenated (separated by dashes) and used as an identifier.

### **Format step response as XML**

The VSE framework expects Respond steps to accept one of the following:

- A response object
- A list of response objects
- An XML document that represents either

**Note:** If this check box is cleared, the step produces a list of response objects. The step produces the list, even if it contains only one response.

**Default:** The step response is formatted as XML.

**If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

## JCo IDoc Live Invocation Step

Complete the following fields:

**Client RFC Destination Name**

Specifies a unique name that identifies the RFC destination.

**Server RFC Connection Properties**

Specifies a properties file that contains connection properties that VSE uses to register itself under a program ID to an SAP gateway and receive IDocs. The properties should be the same as those specified in a .jcoServer file to start a JCo server program that receives IDocs from the server SAP system.

**Server RFC Destination Name**

Specifies a unique name to identify the Server RFC destination.

**Server System Connection Properties**

The Server System Connection properties file contains connection properties to return IDocs to the client SAP system. These properties should be the same as those specified in a .jcoDestination file that can be used to connect to the SAP server system.

**Server System Name**

Specifies a unique name to identify the Server SAP system.

## JCo IDoc Responder Step

This step is used with the JCo IDoc Listener step to transmit JCo IDoc responses. To create this step, record and virtualize JCo IDoc traffic.

This step has no parameters.

## JCo RFC Listener Step

Use this step to respond to JCo RFC requests from the JCo RFC virtual service. To create this step, record and virtualize JCo RFC traffic.

Complete the following fields:

### Client System Name

Specifies a unique name to identify the client SAP system.

### Client System Connection Properties

The Client System Connection properties file contains connection properties with which to connect to the destination on the client system. This must be a .properties file in the Data directory of your project and contains properties that are typically found in a .jcoServer file. This file MUST NOT specify **jco.server.repository\_destination**. See the JavaDocs for **com.sap.conn.jco.ext.ServerDataProvider** in the **doc** folder of your installation directory for more information about the supported properties.

### Format step response as XML

If you select this check box, the step response (the incoming request) is serialized as XML and can be manipulated as text. This is deserialized at the response lookup step. Choosing this option slows the virtual service considerably.

### If Environment Error

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.



## JCo RFC Live Invocation Step

Complete the following fields:

### **Destination System Name**

Defines a unique name that identifies the SAP system on which the RFC executes. This is often the same as the Repository Name.

### **Destination System Connection Properties**

Specifies the Destination System Connection properties file that contains connection properties with which to connect to the system that has the repository. This must be a .properties file in the Data directory of your project and contains properties that are typically found in a .jcoDestination file. For more information about supported properties, see the JavaDocs for **com.sap.conn.jco.ext.DestinationDataProvider** in the **doc** folder of your installation directory. This can be the same file as the Repository Connection Properties.

### **Format step response as XML**

If you select this check box, the step response (the response from the live system) is serialized as XML and can be manipulated as text. This is deserialized at the respond step. Choosing this option slows the virtual service considerably.

### **If Environment Error**

Specifies the action to take or the step to go to if the test fails because of an environment error.

**Default:** Abort the test.

### **VSE Lookup Step**

For a live invocation step to support failover execution mode, it must know the step that is used to look up VSE responses so that it can redirect the VS model to the correct step when necessary. This field contains a list of steps in the VS model. Select the standard VSE Response Lookup step. That allows the live invocation step to move to the VSE response lookup step when necessary.

## JCo RFC Responder Step

This step is used with the JCo RFC Listener step to transmit JCo RFC responses. You can create this step by recording and virtualizing JCo RFC traffic.

Enter the following parameters:

**Responses list property name**

Specifies the name of the property in which to look for the response to send.

**Conversational Model Properties**

Enter a property, then click Add.

To delete a property, select it from the list and click Remove. The listed properties are associated with the current conversation session, which makes the values available to downstream conversational requests.

# Chapter 11: Desensitizing Data

---

In VSE, *desensitization* means attempting to recognize sensitive data and substituting random, but validly formatted, values for that data during recording. Use data desensitization when you do not want to use real customer data as your test data.

Activate data desensitization in VSE in the following ways:

- [Dynamic Desensitization](#) (see page 351)
- [Static Desensitization](#) (see page 352)
- [Data Desensitizer Data Protocol](#) (see page 232)

## Dynamic Desensitization

Dynamic desensitization occurs at the transport layer. Desensitization is invoked by enabling the Desensitize (transport layer) check box on the Basics tab of the Virtual Service Recorder.

This desensitization program uses volatile filters that ensure that sensitive information is never written to disk during the recording phase. The **desensitize.xml** file in the DevTest home directory configures data desensitizers to recognize known patterns such as credit card numbers. The file replaces the live data with realistic but unusable replacements. The file uses Regex pattern matching to recognize and find sensitive data. This file is parsed each time that the recorder is started.

You can use the built-in TestData string generation patterns as replacement data options. TestData provides 40,000 rows of test data, including replacement data for some common data types: names, addresses, telephone numbers, and credit card numbers.

You can customize these preset patterns to create your own. We recommend a regular expression toolkit such as RegexpBuddy. RegexpBuddy lets you paste in your recorded payload and interactively highlights Regexp matches as you fine-tune the Regexp.

Matches are processed in the order they exist in the file, so put your more specific matches first.

To avoid text escaping issues (especially with Regexp), you **must** enclose <regexp> and <replacement> child text in a CDATA element.

## Static Desensitization

Static data desensitization involves manually searching and replacing data in an existing service image. To access the Search and Replace menu, right-click a node in the service image and select Search and Replace.

Specify a specific string to replace with another, then indicate the scope of the change. Click Replace to run the search and replace function for the areas you selected.

## Data Desensitizer Data Protocol Handler

The Data Desensitizer data protocol handler allows the application of desensitization rules when another data protocol is required to "unopaque" a request or response body. For example, when an HTTP message body is gzipped.

For information about this feature, see [Data Desensitizer](#) (see page 232) in *Using CA Service Virtualization*.

# Chapter 12: Virtualizing a Service

---

*Virtualization* is the process by which VSE responds to the client in the absence of the server. Virtualization uses the VSM and the service image.

This section contains the following topics:

[Preparing for Virtualization](#) (see page 354)

[Deploy and Run a Virtual Service](#) (see page 355)

[Running Live Requests](#) (see page 357)

[Session Viewing and Model Healing](#) (see page 369)

[VSE Metrics](#) (see page 371)

## Preparing for Virtualization

### Starting the Virtual Service Environment

The Virtual Service Environment (VSE) must be started for the virtualization to run. The VSE must register with the DevTest registry.

To start the VSE, select Programs, DevTest, Virtual Service Environment from the Start menu.

A window opens to create a VSE named **lisa.VSEServer** and connects to the registry instance.

**Note:** You can minimize the Virtual Service Environment window, but do not close this window.

If you installed DevTest Windows system services, you can use the system service to start VSE.

You can also use the following command to start a named VSE from a command prompt:

**[LISA\_HOME]\bin\VirtualServiceEnvironment.exe -n VSEName -m RegistryName**

#### **VSEName**

Specifies the name of the VSE

#### **RegistryName**

Specifies the name of an existing registry

### Running Multiple Virtual Service Environments

To run multiple VSE servers on one computer, add the **-p port** command-line option while running **VirtualServiceEnvironment.exe**, where **port** specifies the port number. The port number differs for different VSE instances.

The **maxvirtualservices** property in your license limits the number of virtual services you can run.


### Using VSE Manager to Set up the VSE

VSE Manager lets you change your virtual service environments.

For more information about VSE Manager, see [VSE Manager Commands](#) (see page 388) in *Using CA Service Virtualization*.

## Deploy and Run a Virtual Service

### Follow these steps:

1. On the VSE Console, click Deploy a new virtual service to the environment  .  
The Deploy Virtual Service window opens.
2. Enter the name of a model archive (MAR) to upload, then click Deploy.
  - The MAR must contain a virtual service. When you click Deploy, the service loads into the VSE Console and the service is available to run.
  - Alternatively, go to the project panel in DevTest Workstation and right-click a virtual service model (VSM) to open the Deploy Virtual Service window.

The Deploy Virtual Service window opens.

3. Modify the fields as necessary:

#### Name

Displays the name of the virtual service that the VSM you selected references.

#### VS model

Displays the virtual service model that you selected.

#### Configuration

(Optional) Lets you select an alternate configuration file.

#### Group Tag

Specifies the name of the [virtual service group](#) (see page 402) for this virtual service. If deployed virtual services have group tags, they are available in the drop-down list. A group tag must start with an alphanumeric character and can contain alphanumeric characters and the following special characters:

- Period (.)
- Dash (-)
- Underscore (\_)
- Dollar sign (\$)

#### Concurrent capacity

Specifies a number that indicates the load capacity. *Capacity* is how many virtual users (instances) can simultaneously execute with the VSM. Capacity here indicates how many threads there are to service requests for this service model.

VSE allocates a number of threads equivalent to the total concurrent capacity. Each thread consumes some system resources, even when dormant. Therefore, for optimal overall system performance, set this setting as low as possible. Determine the correct settings empirically by adjusting them until you achieve the desired performance, or until increasing it further yields no performance improvement.

Out of the box protocols use a framework-level task execution service to minimize thread usage. For these protocols, a concurrent capacity of more than 2-3 per core is rarely useful, unless the VSM has been highly customized.

For extensions and any VSM that does not use an out of the box protocol, setting a long Think Time may consume a thread for the duration of the think time. In these cases, you may need to increase the concurrent capacity.

The following formula gives an approximate initial setting in these cases:

Concurrent Capacity = (Desired transactions per second / 1000)  
\* Average Think Time in ms \* (Think Scale / 100)

**Example:**

Assume that you are using a custom protocol that does not use the framework task execution service to handle think times. You want an overall throughput of 100 transactions per second. The average think time across the service image is 200 ms, and the virtual service is deployed with a 100 percent think scale.

$(100 \text{ Transactions per second} / 1000) * 200\text{ms} * (100 / 100) = 20$

In this case, each thread blocks for an average of approximately 200 ms before responding, and during that time is unable to handle new requests. We therefore need a capacity of 20 to accommodate 100 transactions per second. A thread would become available, on average, every 10 ms, which would be sufficient to achieve 100 transactions per second.

**Default: 1**

**Think time scale**

Specifies the think time percentage for the recorded think time.

**Note:** A step subtracts its own processing time from the think time to have consistent pacing of test executions.

**Default: 100**

**Examples:**

- To double the think time, use 200.
- To halve the think time, use 50.

**Start the service on deployment**

Specifies whether to deploy and start the service immediately.

**Values:**



- **Selected:** The service deploys and starts immediately.
- **Cleared:** The service deploys, then you start it manually from the VSE Console.

**If service ends, automatically restart it**

Specifies whether to keep the service running even after an emulation session reaches its end point.

**Values:**

- **Selected:** Continues running the service after the emulation service ends.
- **Cleared:** Stops the service when the emulation service ends.

**Default:** Selected

4. Click Deploy.

The VSE Console displays the virtual service status as loaded.

**Note:** A virtual service can be in the following states:

**Deployed**

No service with the name you entered is already deployed. The service is deployed.

**Redeployed**

A service with the name you entered is deployed with the same .vsm file as the service you entered. The service is redeployed.

**Overridden**

A service with the name you entered is deployed with a .vsm file that is different from the one associated with the service you entered. The application prompts you to override the deployed service.

## Running Live Requests

Run live requests against VSE after you deploy the virtual service. If possible, take the live service down and configure the gateway/proxy settings so the client communicates with VSE.

For more information, see:

- [Access the VSE Console](#) (see page 358)
- [Toolbar](#) (see page 359)
- [Services Tab](#) (see page 361)
- [Matching Tab](#) (see page 363)
- [Request Events Details Tab](#) (see page 364)

## Access the VSE Console

To manage and monitor deployed service images, use the VSE console. From this console, you can deploy, start, view, stop, redeploy, and remove virtual services.

**Follow these steps:**

1. From DevTest Workstation, click Server Console to open the DevTest Server Console.
2. Select the VSE service and double-click it to open the VSE console.
3. In the VSE Console, verify that the virtual service is deployed (status is Running).
4. In the VSE Console, verify that the service received the requests by viewing the transaction count (Txn count).









**Note:** If a deployed VS model shows no transactions, then the client is not configured properly. Reconfigure the client to reference the virtual model instead of the real system. If another service is using that port, stop that service or change the port setting to remove the conflict.



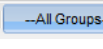
## VSE Console Toolbar

The VSE Console has a specific toolbar, which is comprised of the following buttons. You can also access these functions by pointing at a service and right-clicking.



The VSE Console toolbar contains the following commands:

		Deploys or redeploys a virtual service to the environment.
		Starts a virtual service that you selected.
Show		Displays the inspection view for the selected virtual service.
View		Displays session/tracking information for the selected virtual service.
Set		Specifies how the selected virtual service should behave.
Reset		Resets the transaction and error counts for the selected virtual service.
Stop		Stops the selected virtual service.
Remove		Removes the selected virtual service from the environment.

Configure		Configures tracking data cleanup.
Shut down		Shuts down the entire virtual service environment.
Groups		Selects whether to display services associated with all groups, with no groups, or with one group.

### Deploy a new virtual service to the environment

To deploy a virtual service from a model archive (MAR), select this option.


Enter the name of a model archive (MAR) to upload. The MAR must contain a virtual service. When you click Deploy/Redeploy, the service loads into the VSE Console and the service is available to run.

### Start the selected virtual service

Starts a virtual service that you have selected.

A confirmation window opens to indicate that the service has started. Click OK.

### Show the inspection view for the selected virtual service

The Show Inspection View  button opens a tab for an inspector panel that is tailored for virtual services. This panel has two tabs, [Matching](#) (see page 363) and [Request Event Details](#) (see page 364).

## VSE Console Services Tab

In the Services tab, you can sort on column values (ascending or descending) by clicking the down arrow to the right of the column name. Use this arrow also to select the columns to display on this window.

This tab contains the following fields:

**Name**

Identifies the currently deployed virtual service model.

**Resource / Type**

Identifies the port and the type or protocol of the service.

**Status**

Identifies the current state of the virtual service.

**Up-Time**

Indicates how much time has elapsed since the service was started.

**Txn Count**

Identifies the number of transactions that were recorded after the service started.

**Execution Mode**

Displays the [execution mode](#) (see page 364) of the virtual service.

**Group**

Displays the [virtual service group](#) (see page 402) of the virtual service.

**Errors:**

Displays a red dot to indicate that errors occurred while running the service.

The Virtual Service Details panel at the bottom of the window displays details about the service.

This panel contains the following fields"

**Model Name**

Identifies the name of the currently deployed virtual service model.

**Execution Mode**

Displays the execution mode for this virtual service. See [Specify How the Selected Model Should Behave](#) (see page 364) in *Using CA Service Virtualization*.

**Last Start**

Displays the date and time this service was last started.

**Transaction Count**

Displays the number of transactions that VSE recorded after the service started.

**Current txn/s**

The number of transactions currently executing.

**Capacity**

Defines how many virtual users (instances) can execute simultaneously with the virtual service model. Capacity indicates how many threads exist to service requests for this service model. You can update this field while the service is running.

**Group Tag**

Displays the name of the [virtual service group](#) (see page 402) for this virtual service. If deployed virtual services have group tags, those tags are available in the field when you enter a character. A group tag must start with an alphanumeric character and can contain alphanumerics and the following special characters:

- period (.)
- dash (-)
- underscore (\_)
- dollar sign (\$)

You must use the **Tab** or **Enter** key after you enter the group tag, then click Update to update the field.

To delete a group tag from a virtual service, click the **X** next to the group tag name, then click Update.

**Config Name**

Identifies the name of the configuration file that this service uses.

**Auto-Restart**

Specifies whether to use the auto-restart option for this service. To change this value while the service is running, click this field.

**Last End**

Displays the date and time this service stopped.

**Error Count**

Identifies the number of errors received.

**Peak txn/s**

Displays the largest number of transactions that have run concurrently.

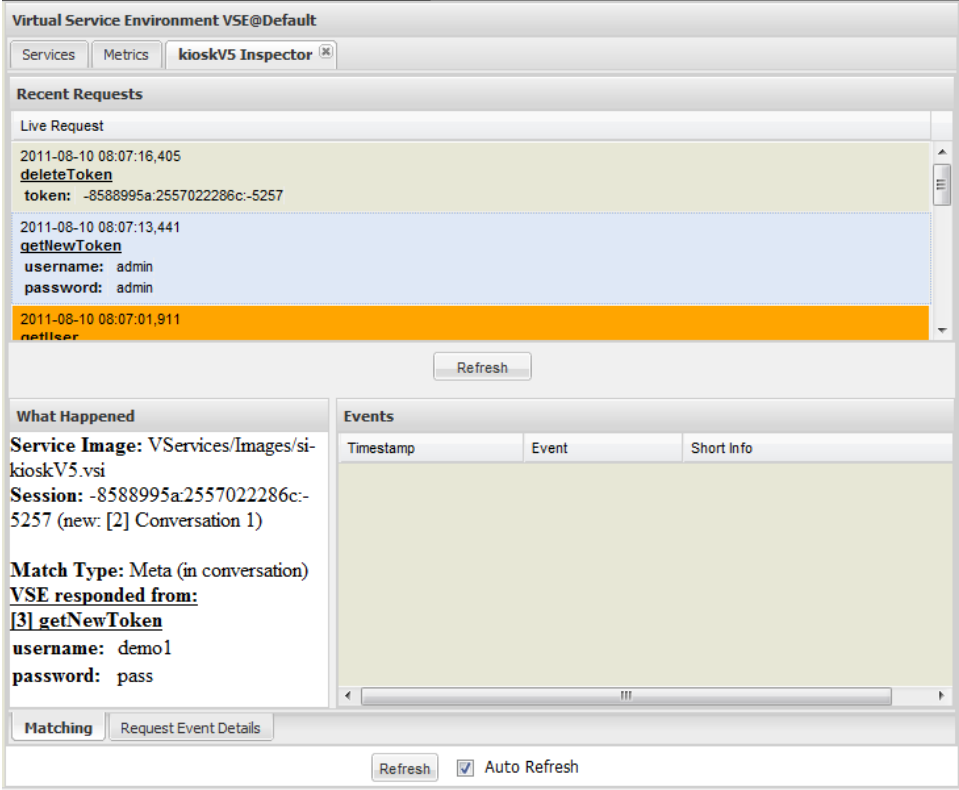
**Think Scale**

Displays the think time percentage regarding the recorded think time.

To download the backing archive for the virtual service, click the name of the virtual service in the VSE Console. The system prompts you to save the MAR file that is associated with this virtual service.

## VSE Console-Matching Tab

The Matching tab lists recent requests that the virtual services processed. When you select a request, the tab shows a description of how the request was (or was not) matched. The same information is recorded in the **vse\_matches.log** file. If any events were associated with the selected request, they display at the right side of the panel.



**Virtual Service Environment VSE@Default**

Services Metrics **kioskV5 Inspector** (X)

**Recent Requests**

Live Request

2011-08-10 08:07:16,405	<b>deleteToken</b>	token: -8588995a:2557022286c:-5257
2011-08-10 08:07:13,441	<b>getNewToken</b>	username: admin password: admin
2011-08-10 08:07:01,911	<b>getNewToken</b>	

Refresh

**What Happened**

**Service Image:** VServices/Images/si-kioskV5.vsi  
**Session:** -8588995a:2557022286c:-5257 (new: [2] Conversation 1)

**Match Type:** Meta (in conversation)  
**VSE responded from:**  
**[3] getNewToken**  
**username:** demo1  
**password:** pass

**Events**

Timestamp	Event	Short Info

Matching Request Event Details

Refresh ☒ Auto Refresh

## VSE Console Request Events Details Tab

The Request Event Details tab shows the list of inbound requests that caused the virtual service to error out. When you select a request, the list of VSM steps that executed is shown, with steps containing error events selected. To see the events that occurred during processing for that step (similar to the ITR), select a step.

Virtual Service Environment VSE@Default

Services Metrics **kioskV5 Inspector**

**Errored Requests**

Request

2011-08-12 13:41:36,985  
**deleteToken**  
token: -4655d680:656078c197c--6627

2011-08-12 13:41:35,225  
**getNewToken**  
username: itko

Refresh

**Steps Executed**

Name
HTTP/S Listen
<b>Prepare Request</b>
VS Image Response Selection
Prepare Response
HTTP/S Respond

**Events -- Prepare Request**

Timestamp	Event	Short Info
2011-08-12 13:41:36,978	Property set	lisa.vse.request
2011-08-12 13:41:36,978	Step response	Prepare Request
2011-08-12 13:41:36,978	Assertion fired	Prepare Request [if being efficient]

Matching **Request Event Details**

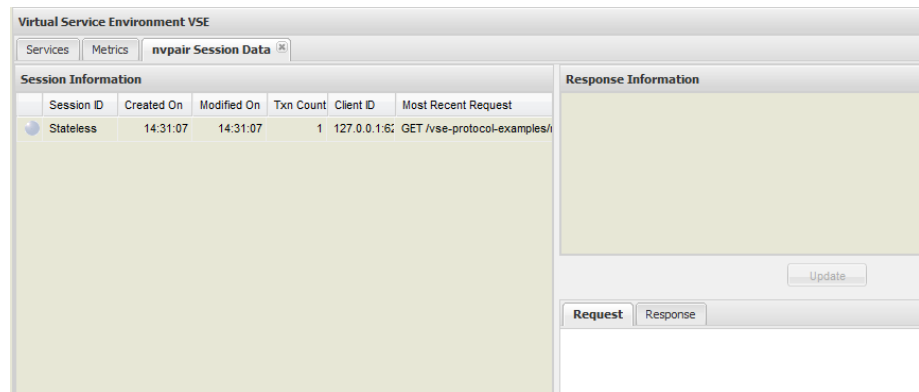
Refresh ☒ Auto Refresh

**Note:** When a virtual service exceeds 100 transactions for each second, the Property Set and Property Removed events are disabled to allow for greater overall performance.

### View Session and Tracking Information for the Selected Virtual Service

This option lets you see the session tracking information for the virtual service. For more information, see [Session Viewing and Model Healing](#) (see page 369).



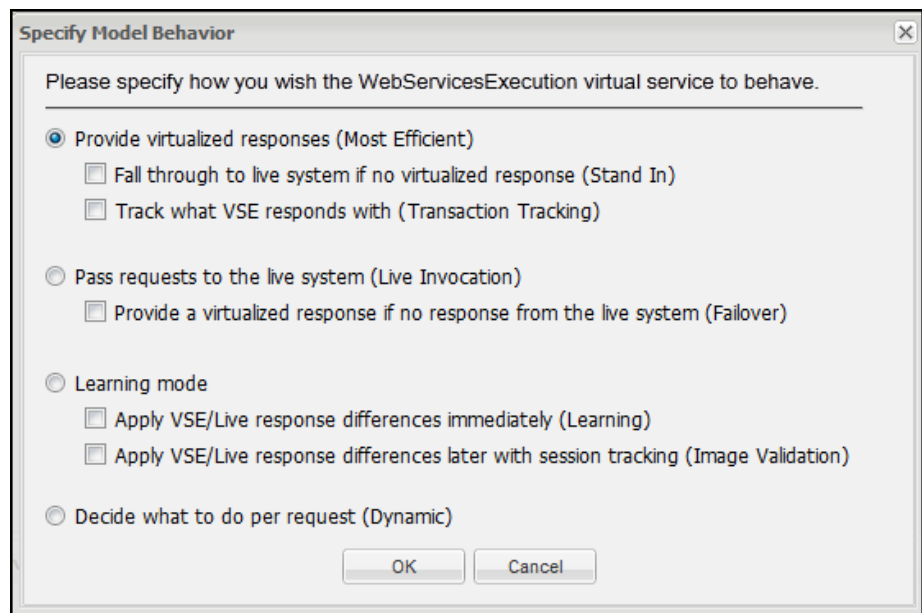


### Redeploy the Selected Virtual Service to the Environment

If you edit the service image or the VSM, save the changes and redeploy the modified service image in the VSE Console by clicking Deploy/Redeploy.

### Specify How the Selected Model Should Behave

This option lets you set an execution mode for the virtual service. The number of execution modes available for a virtual service depends on the type of test step. For example, if a model does not have a live invocation step, it does not support the Learning or Live Invocation options.



The available execution modes are:

#### Most Efficient

The fastest mode; it does not execute the routing or tracking steps. This mode also restricts generated event tracking.

### **Stand In**

Stand In mode first routes a request to the virtual service (the same as Most Efficient mode). However, if the virtual service does not have a response, the request is then automatically routed to the live system. You can only enable Stand In mode for a virtual service with a Live Invocation step. Stand In mode does not do any special tracking. It simply allows for a virtual service to fall back on the live service.

### **Transaction Tracking**

This mode fires more events than Most Efficient and remembers transaction flow through sessions. This transactional information is used to help determine why a specific response was chosen for a specific request. This mode does not perform as efficiently as Most Efficient. Transaction Tracking mode does not show live system responses; it only shows the response from the service image.

### **Live System**

This mode uses the Live Invocation step of the model to determine a response to the current request. Instead of using the response from the virtual service, it accesses the live service to get the response. The target system of the live invocation controls performance. This mode is also known as pass through.


### **Failover**

Failover mode first routes a request to the live system (the same as Live System mode). However, if the live system does not have a response, the request is then automatically routed to the virtual service. This mode is the opposite of Stand In mode. You can only enable Failover mode for a virtual service with a Live Invocation step. Failover mode uses the service image to determine a response if the Live Invocation step actually fails (as might happen if the live system were not available).

### **Learning**

Learning mode is like Image Validation mode but it automatically "heals" or corrects the virtual service to have the new or updated response from the live system. The next request that is passed into the virtual service automatically sees the new response that was "learned". Not only one system is being checked to learn, but both are, and the live system currently prevails.

When a virtual service is running in Learning mode and it has acquired new

knowledge, there is an icon  to the left of the virtual service name in the VSE Console. The icon remains visible until you redeploy the virtual service.

### **Image Validation**

This mode uses both the VSE and the live system to derive a response to the current request. The responses are logged for applying later to the service image using the View Session and Tracking panel. This mode allows a live comparison between the responses that VSE provides and a corresponding live system and, where differences exist, patches or heals the VSE service image to keep in sync with the live system. This mode is also known as "live healing mode." Image Validation is the least efficient of all the modes.

### Dynamic

This mode enables the model to determine for each request which of the other modes to use. Performance is, therefore, unpredictable. The only requirement is that the VS Routing step is present in the model after the protocol's listen step or steps.

Reset the Transaction and Error Counts for the Selected Virtual Service

Sets both the transaction count and error count to zero for the selected virtual service.

### Stop the selected virtual service

To stop the selected virtual service, click this button. The application displays a confirmation message before the service stops.

### Remove the selected virtual service from the environment

To remove the selected virtual service from the console display, click this button. The application displays a confirmation message before it removes the service.

### Configure tracking data cleanup

Opens the Configure Tracking Data Cleanup window.

Configure Tracking Data Cleanup

Scan for data to delete every: 1 hours

Delete data older than: 8 hours

Note that these numbers will reset to their default or properties file values when the virtual environment server is stopped and restarted.

OK Cancel

Enter data cleanup values here that remain in effect until the service is stopped and restarted. When the service restarts, the values reset according to their defaults or the values in a properties file.

### Scan for data to delete every

Defines the interval (in milliseconds, seconds, minutes, hours, days, or weeks) after which to scan for tracking data to delete.

**Delete data older than**

Defines the age of data (in milliseconds, seconds, minutes, hours, days, or weeks) to delete.

**Shut down the entire virtual service environment**

To shut down the complete VSE environment, click this button. If you reply affirmatively to the shutdown confirmation message, your VSE shuts down and the corresponding window closes.

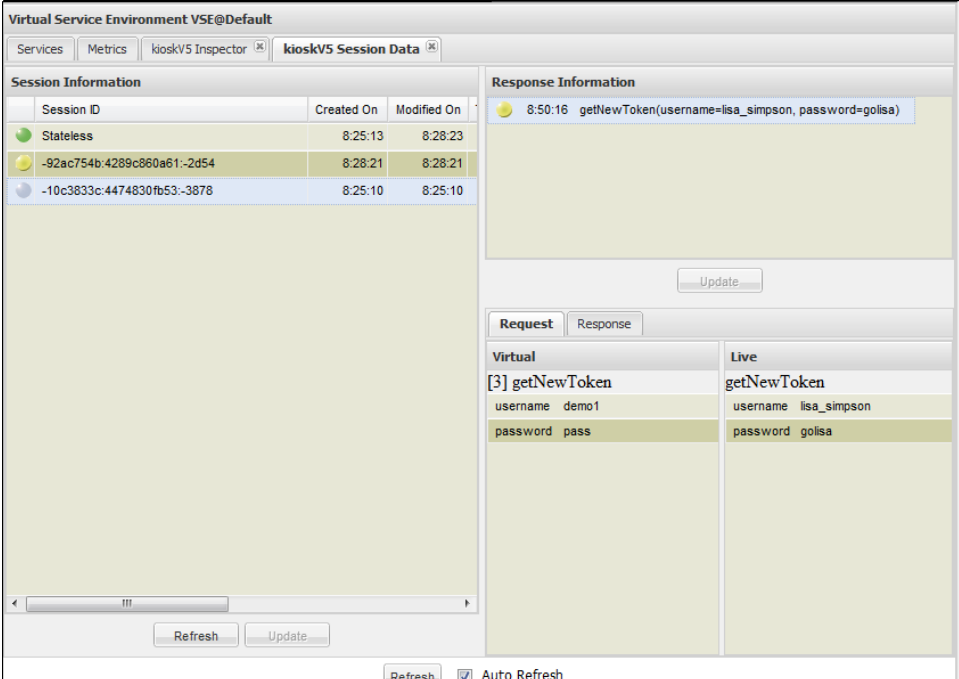
## Session Viewing and Model Healing

*Session viewing* lets a VSE user actually see the behavior of current (or recent) sessions on a VSE server. The user can determine why the response for a specific request was given. Session viewing also enables a live comparison between the responses that VSE provides and a corresponding live system. Where differences exist, the application can use *model healing* to patch or heal the VSE service image to keep it in sync with the live system.

Session viewing is only available for virtual services that run with an Execution Mode of Transaction Tracking or Image Validation. Model healing is only available for virtual services that run in Image Validation mode. For more information about execution modes, see [Specify How the Selected Model Should Behave](#) (see page 364).

Model healing differs from learning because learning changes a service image immediately when CA Service Virtualization and live response differences are detected. Healing logs those differences for later review and application to the service image with the View Session and Tracking information panel.

You can manage session viewing and model healing with a panel that is accessible from either the VSE dashboard or while editing a virtual service model.




The screenshot shows the 'Virtual Service Environment VSE@Default' console. The 'Services' tab is selected, and the 'kioskV5 Session Data' panel is active. The 'Session Information' table lists sessions with their IDs, creation times, and modification times. The 'Response Information' panel shows a log entry for a 'getNewToken' request. Below this, the 'Request' and 'Response' panels compare the 'Virtual' and 'Live' responses for the same request.

Session ID	Created On	Modified On
Stateless	8:25:13	8:28:23
-92ac754b4289c860a61-2d54	8:28:21	8:28:21
-10c3833c4474830fb53-3878	8:25:10	8:25:10

Request	Response
[3] getNewToken	getNewToken
username demo1	username lisa_simpson
password pass	password golisa

To view the session tracking for a selected virtual service, from the VSE Console Services tab, select a service and click Session/Tracking Information .

When you click the icon, the session panel opens to show the recorded session and transactions. If a deployed virtual service has no recorded transactions, the session panel opens with no session information.

The session panel is divided into two panes: Session Information and Response Information.

#### Session Information Pane

The Session Information pane lists all the sessions for the selected virtual service in tabular format. To reorder the table or to select or clear the columns that show, click the arrow at the right of each column heading.

##### **Session Status**

Displays and icon to indicate the current session status.

A gray ball indicates a session that was recorded in Transaction Tracking mode, or a session that has been healed using model healing.

A red ball indicates a session that was recorded in Image Validation mode and is a candidate for healing.

A green ball indicates a session that was recorded in Image Validation mode where the live and VSE responses match.

##### **Session ID**

Identifies the unique ID for each session.

##### **Created On**

Displays the timestamp of the first transaction in that session.

##### **Modified On**

Displays the timestamp of most recent transaction.

##### **Txn Count**

Identifies the number of transactions that were recorded after the service started.

##### **Client ID**

(Protocol-specific) For HTTP, displays the endpoint of the client that submitted the transaction.

##### **Most Recent Request**

Identifies the most recent request that came through on the specific session.

#### Response Information Pane

The Response Information pane shows the list of transactions for the selected session. When you point the mouse to the colored ball in the first column of this pane, a tooltip identifies the match for that transaction. If you click any specific transaction, the request and response tabs appear at the bottom of the pane. These tabs compare request/response between the VSE system and the live system.

In the Response Information pane, the following icons represent transactions:

- Green ball: Indicates a signature match on a meta transaction.
- Yellow ball: Indicates a signature match on a meta transaction; the image navigation is successful, but the response body differs between VSE and the live system.
- Red ball: Indicates a conversational transaction that diverged between the live system and VSE image.

The Update buttons are used to update the service image with live session/stateless transactions. This process is referred to as model healing. You use model healing to remove the disparity between the VSE image and the live system so that the VSM works correctly. When you click Update, the session marked with a red ball changes to a gray ball. This session is now tracked.

- The Response Information Update button updates the service image for the selected transaction.
- The Session Information Update button lets you select multiple sessions that display in the Service Information pane and update them all simultaneously.

## VSE Metrics

### View VSE Metrics

The Metrics tab on the DevTest Console allows you to view VSE metrics.

**Follow these steps:**

1. To display a list of active services, click Select VSE Service.
2. Select the service to display.
3. To see the available charts for the selected service, click Select Chart.
4. Select a chart and click Generate Chart.

To refresh the display, click Refresh. Use the Zoom slider and the Scroll slider to move the chart to make your data more visible.

**Note:** To view the metrics charts, the Adobe Flash plugin must be installed on the browser that is used to access the server console.

## Define Metrics Collection

To define the parameters of VSE metric collecting, update the following properties in the **lisa.properties** file:

- `lisa.vse.metrics.collect`
- `lisa.vse.metrics.txn.counts.level`
- `lisa.vse.metrics.sample.interval`
- `lisa.vse.metrics.delete.cycle`
- `lisa.vse.metrics.delete.age`

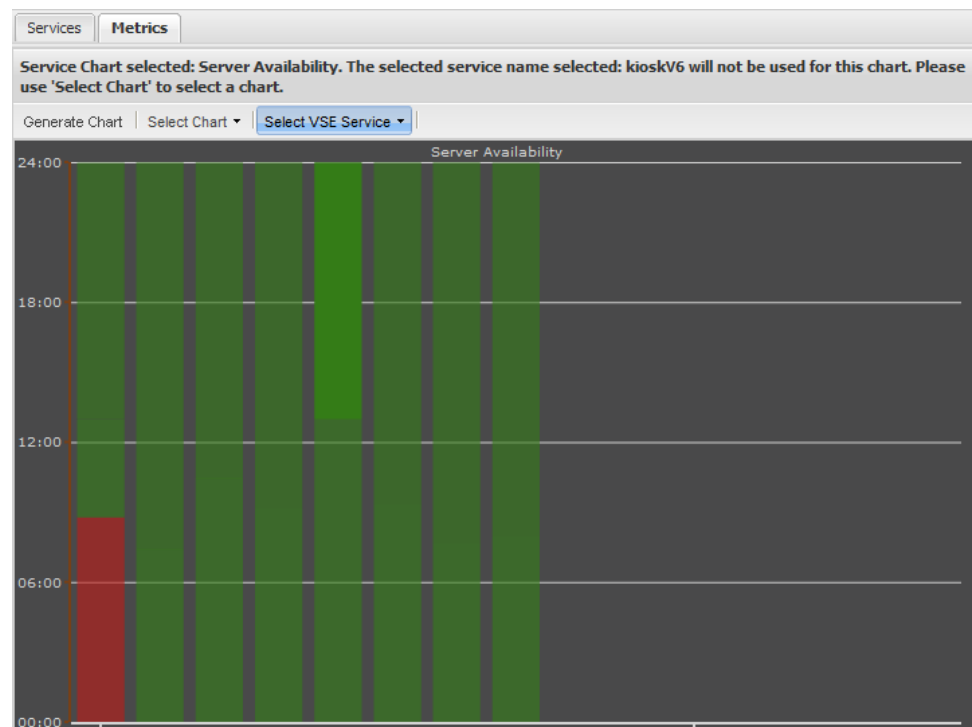
**Note:** For more information, see "Appendix A - LISA Property File" in *Using*.

## Server Chart Metrics

*Server chart metrics* apply to all activity on a specific virtual server, or VSE.

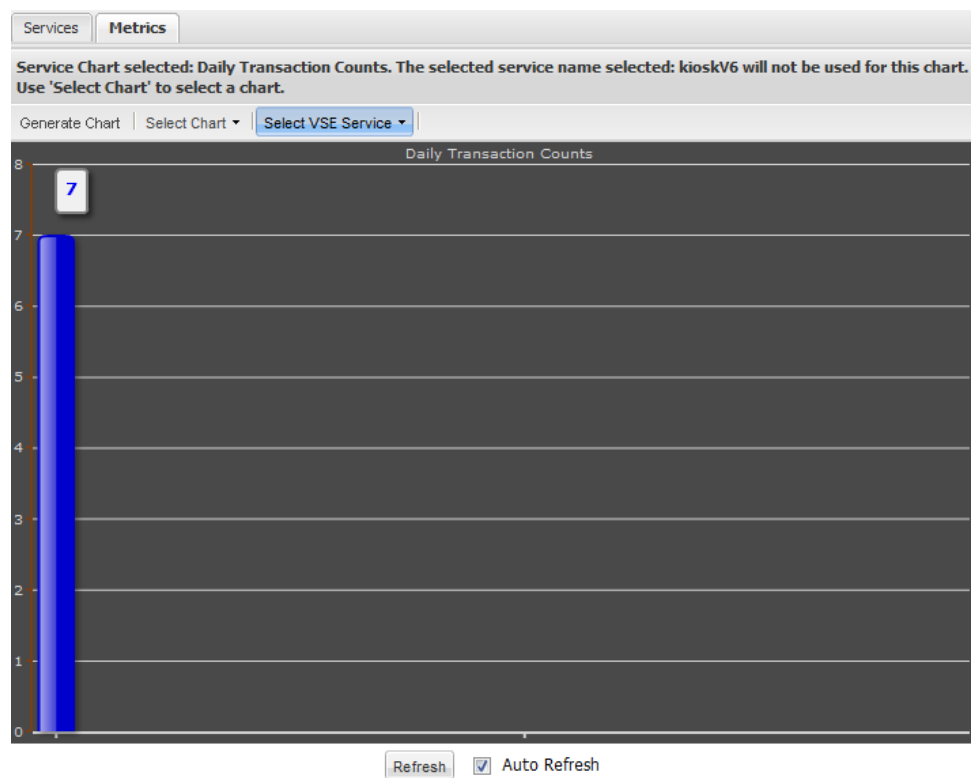
**Note:** If you select a server chart, the panel header indicates that the selected virtual service is not used for this chart. These charts apply to the entire server, not only the selected service.

## Server Availability

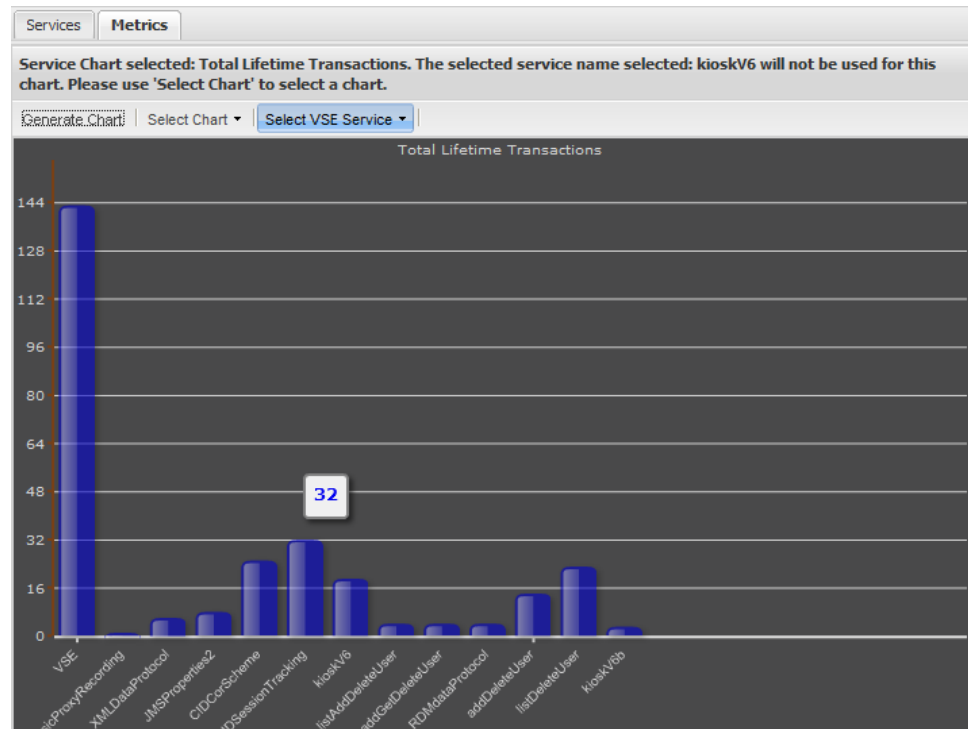




## Daily Transaction Counts



## Total Lifetime Transactions



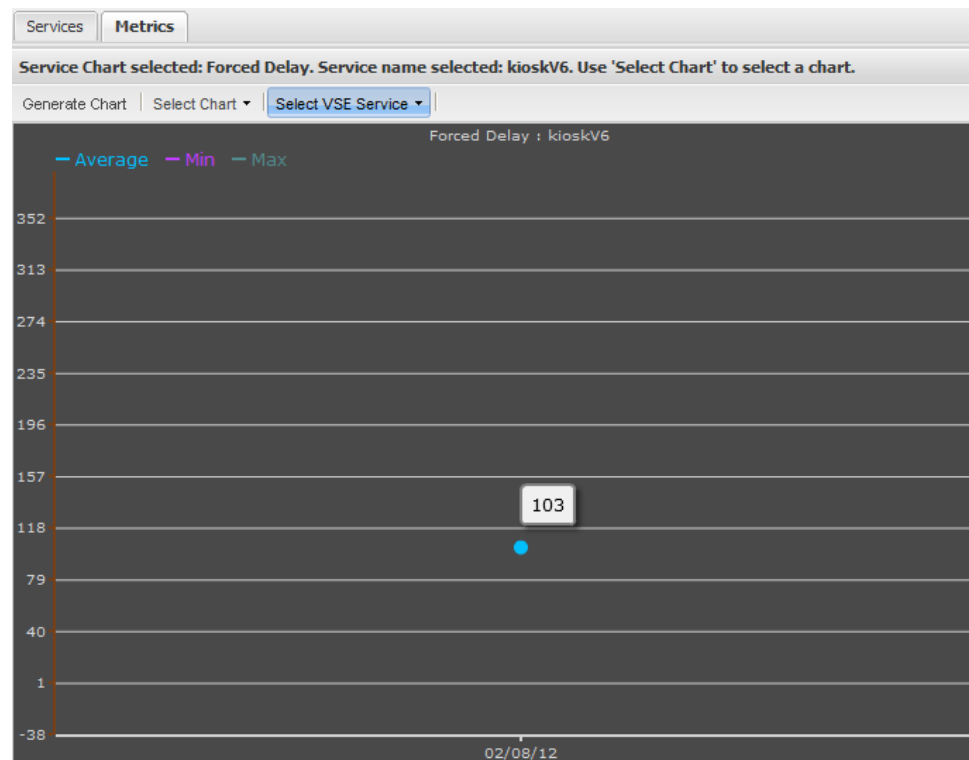
## Service Chart Metrics

*Service chart metrics* apply to each virtual service. On the right pane, there is a list of virtual services that were deployed in this environment. You can select one of them, then select a chart from the list.

## Response Time



## Forced Delay



### Forced Delay

Indicates the number of milliseconds before VSE sent a result.

A positive number means VSE deliberately waited to send a result because of the specified think time. For example, if think time was 10ms and the time to listen and respond was 8ms, VSE waits 2ms.

A negative number means VSE took longer to generate the result than the think time. For example, the think time was 10ms but it took 12 ms to listen and respond was 12 ms. The report shows -2.

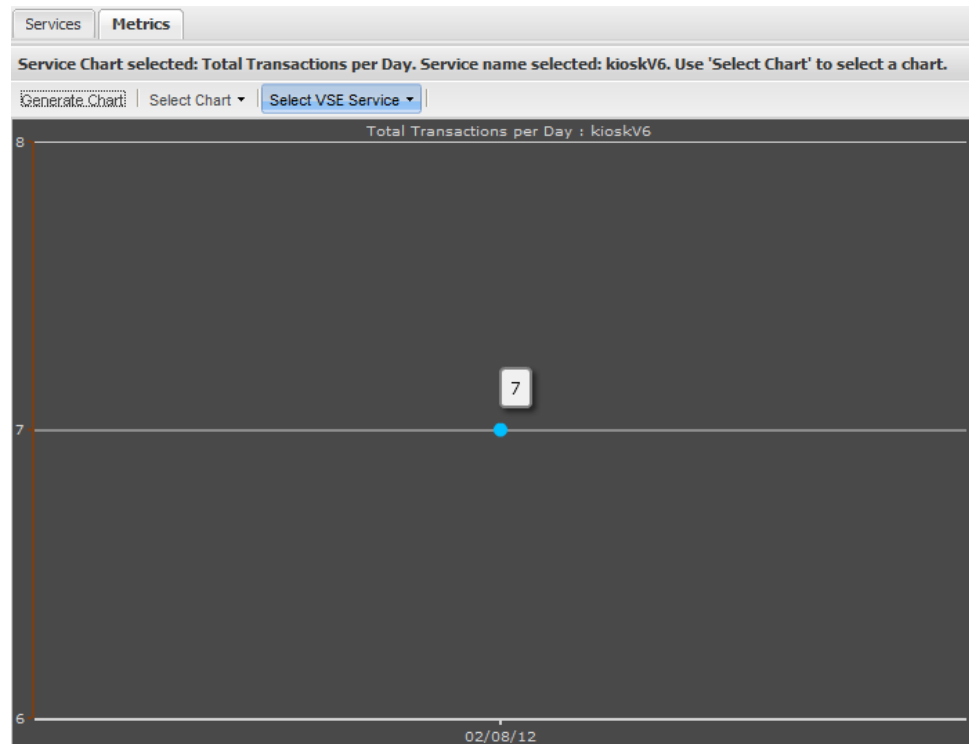
## Transactions Per Second



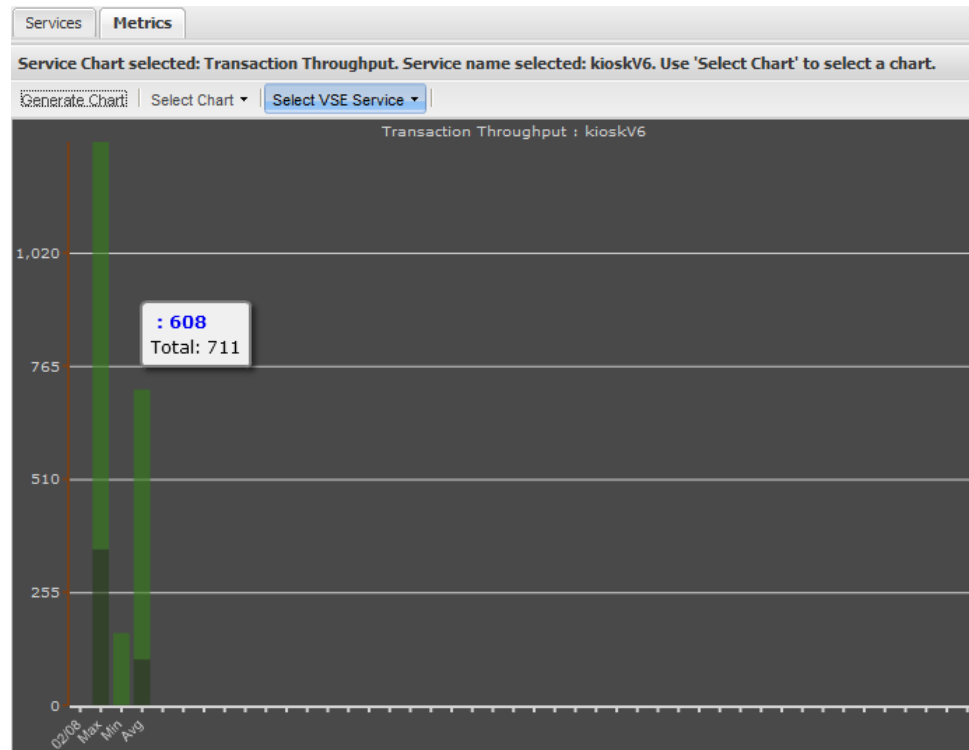
## Transaction Hits and Misses



## Total Transactions per Day



## Transaction Throughput







# Chapter 13: VSE Manager - Manage and Deploy Virtual Services

---

VSE Manager is an Eclipse plug-in that lets you deploy and manage DevTest virtual services.

VSE Manager is supported on Eclipse release 4.3 and later. Be sure to run Eclipse with a Java 7 JVM.

This section contains the following topics:

[Install VSE Manager](#) (see page 381)

[Use VSE Manager](#) (see page 382)

## Install VSE Manager

To install VSE Manager, follow the standard Eclipse procedure for adding new software.

**Follow these steps:**

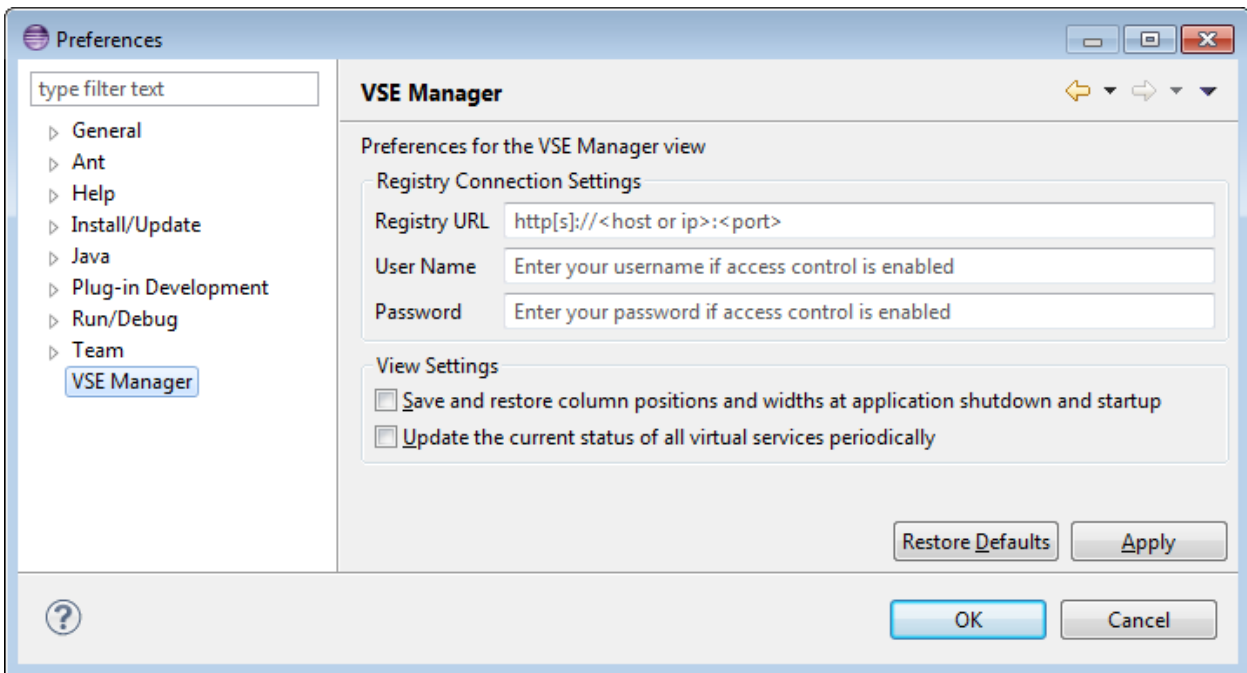
1. In Eclipse, select Help, Install New Software.  
The Install dialog opens.
2. Select Add to add the VSE Provisioning Platform (p2) repository URL:  
`http://www.itko.com/downloads/eclipse/8.0/updates/`
3. Select the VSE Manager UI feature, then click Next.  
The Install Details window opens.
4. Review the install details, then click Next.  
The Review Licenses window opens.
5. Review and accept the licenses, then click Finish.

## Use VSE Manager

### Configure VSE Manager

Before you use VSE Manager, you must configure the VSE Manager page in the Eclipse Preferences dialog.

The following graphic shows the Preferences dialog with the VSE Manager page selected.



### Notes:

- Set the Registry URL field to an **https** URL. This type of URL requires the enabling of HTTPS communication with the DevTest Console, as described in *Administering*.
- Be sure to enter values in the User Name and Password fields. Leaving these fields blank can result in the appearance of the Eclipse Password Required dialog, which is not the correct dialog for VSE Manager. If the Password Required dialog does appear, click Cancel instead of entering a user name and password.

### Follow these steps:

1. To configure the registry connection settings, enter the URL for the VSE Webserver.
2. If necessary, enter your user name and password.
3. (Optional) To save the order of columns and their associated widths when closing and reopening the view, select the Save and restore... check box.

4. (Optional) To have the view automatically refresh the content approximately every 10 seconds, select the Update the current status... check box.
5. To finish and close the dialog, click OK.

### Import a LISA Project

#### Follow these steps:

1. From the main menu, open the Eclipse Import wizard.
2. From the CA LISA category, select LISA Project.
3. Select a LISA project and enter its name in the Project Name field.
4. Click Finish.

The project imports.

### Use the VSE Manager View

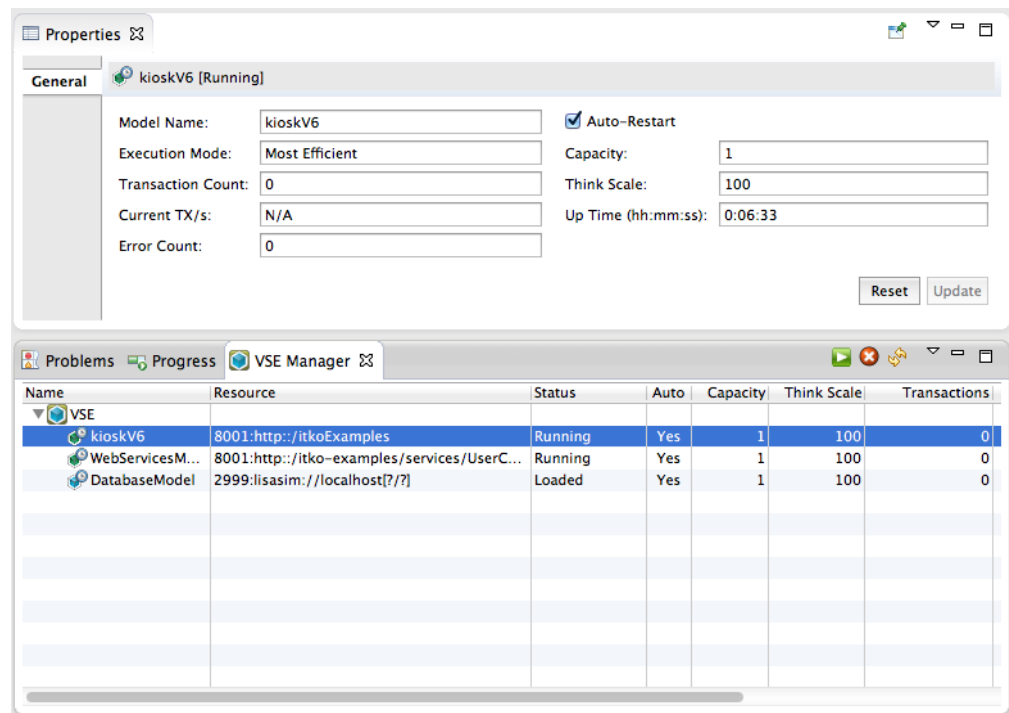
Name	Resource	Status	Auto	Capacity	Think Scale	Transactions
VSE						
kioskV6	8001:http://itkoExamples	Running	Yes	1	100	0
WebServicesM...	8001:http://itko-examples/services/UserC...	Running	Yes	1	100	0
DatabaseModel	2999:lisasim://localhost[?/?]	Loaded	Yes	1	100	0

The VSE Manager view supports the following actions:

- Drag-and-drop to rearrange the columns of the view.
- Select the columns to hide or view with the drop-down list.
- Start, stop, or undeploy the selected virtual service.
- Drag-and-drop from an Eclipse-based product and from the native file system to deploy new virtual services to VSE in the MAR file format.

### Properties View Integration

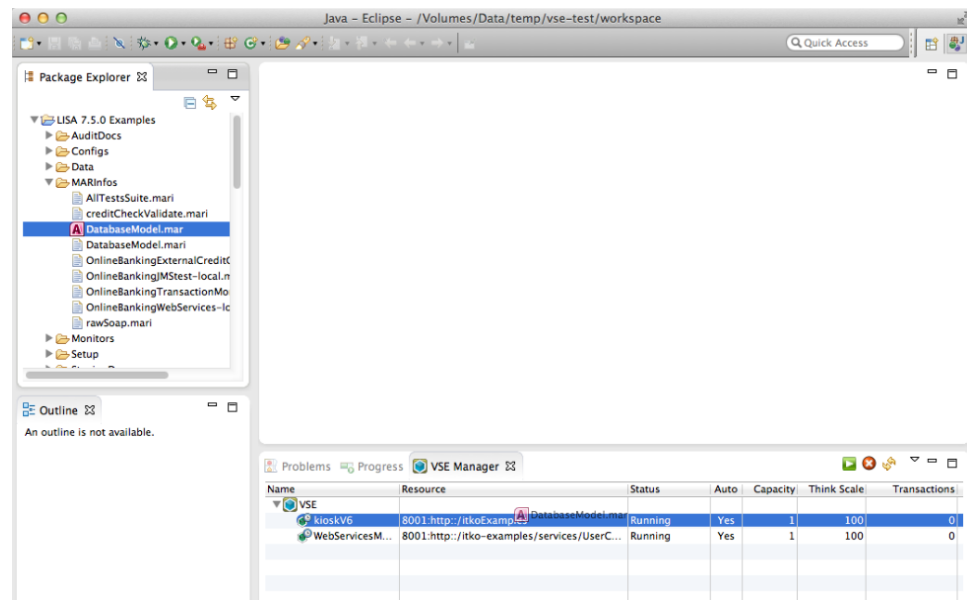
When a virtual service is selected in the VSE Manager view and the Properties view is open, you can view and update more information about the selected service.



With this view, you can:

- Control automatically starting the service when the VSE server is started.
- Change the capacity of a deployed virtual service.
- Change the think scale for the requests to the virtual service.

You can drag a MAR file to the VSE Manager to deploy it.







# Chapter 14: VSE Commands

---

This section contains the following topics:

[VSE Manager Command - Manage Virtual Service Environments](#) (see page 388)

[ServiceManager Command - Manage Services](#) (see page 389)

[ServiceImageManager Command - Manage Service Images](#) (see page 390)

[VirtualServiceEnvironment Commands](#) (see page 395)

## VSE Manager Command - Manage Virtual Service Environments

To manage virtual service environments, use the VSE Manager command-line tool. We recommend that you use the command-line tool for LISA 6.0 and later implementations. For more information, see [Service Manager Commands](#) (see page 389).

VSE Manager is included as a tool under the bin directory in <lsrv>.

**The VSE Manager command has the following format:**

```
VSEManager [--registry name | --vse name | --username=name  
|--password=name ]  
  
[--status [vs-name | ALL ] | --deploy archive-file | --redeploy  
archive-file | --update vs-name [capacity capacity] [thinkscale  
scale] [auto-restart [ON|OFF] [group tag group tag] | --remove vs-name  
| --start vs-name | --set-exec-mode vs-name --mode exec-mode | --stop  
vs-name | --help | --version
```

**--registry *name***

Sets the name of the DevTest registry through which to work.

**Note:** The --registry parameter must be defined before a command is run. Otherwise, the command will, by default, run on the local registry. If there is no local registry, the command will not run.

**--vse *name***

Sets the name of the virtual service environment with which to work.

**--username=*name***

Sets the ID with which to authenticate to ACL.

**--password=*name***

Sets the password that is associated with the specified --username value.

**--status [*vs-name* | ALL]**

Prints the status of a specific virtual service or of all virtual services in the current environment.

**--deploy *archive-file***

Deploys a new virtual service to the current environment. The specified archive file must have a virtual service model as its primary asset.

**--redeploy *archive-file***

Redeploys the specified virtual service to the current environment.

**--update *vs-name* [*capacity capacity*] [--thinkscale *scale* ] [--auto-restart [ON|OFF]  
[--group tag *group tag* ]**



Updates the capacity, think scale, group tag, auto-restart setting, or any combination of these parameters for the named virtual service.

**--remove *vs-name***

Removes the named virtual service from the current environment.

**--start *vs-name***

Starts the named virtual service in the current environment.

**--set-exec-mode *vs-name* --mode [DYNAMIC | VALIDATION | LIVE | TRACK | EFFICIENT | LEARNING | STAND\_IN | FAILOVER]**

Sets the execution mode for the named virtual service in the current environment.

**--stop *vs-name***

Stops the named virtual service in the current environment.

**--help**

Displays help text.

**--version**

Prints the VSE version number.

## ServiceManager Command - Manage Services

Use the ServiceManager command-line tool to monitor, reset, and stop DevTest services. The commands apply to any DevTest registry, simulator, or coordinator, and to the VSE server. The Service Manager is included under the bin/ directory in DevTest Server.

For more information, see ServiceManager in *Administering*.

## ServiceImageManager Command - Manage Service Images

To import transactions (raw or session) into a new or existing service image, use the VSE ServiceImageManager command-line tool. The tool can also combine two or more service images. The ServiceImageManager is located in the LISA\_HOME\bin directory.

**Control recording in one of the following ways: interactive, timed, or disconnected.**

**Interactive:**

Use the interactive style when you specify only the --record argument. When the recorder is ready, it waits for you to click Enter at the console before starting the recording process. The recorder then waits for Enter again to stop the process.

**Timed:**

Use the timed style when you specify the --record and --stop= (and, optionally, the --start=) arguments.

**Disconnected:**

Use the disconnected style when you specify the --record and --port= arguments. This sets up a listener that the --signal argument uses to pass control signals to the recorder. After a recorder is initiated with the disconnected style, the ServiceImageManager tool, in a separate process, can be used to control it by sending start and stop signals over the port. This control requires specifying only the --signal= and --port= arguments.

All three styles require the --vrs= and --si-file= arguments. Optionally, for the interactive and disconnected styles, specify --go to skip waiting for a start signal and begin recording immediately.

To import, specify the raw or session traffic transaction file to import after the **--import** argument. To control how the import happens, use the transport, request, and response data protocols and navigation tolerances. To use the **--vrs=** argument with the protocol or tolerance arguments, specify it before the others. The **--vrs=** argument resets parameters to their initial state. To specify the name of the file containing the service image in which to import, use the **--si-file** argument. Specify at least the following arguments:

- **--import=**
- **--si-file=**
- Either **--vrs=** or **--transport=**

You can also specify these arguments:

- **--request-data=**
- **--response-data=**

Either **--non-leaf=** or **--leaf=**

To combine two or more service images, specify the target service image after the **combine** argument. The file name that **--combine=** defines is the target VSI and other files that are listed are the sources. To control how like transactions combine and list the file names of the source images to combine into the target image, use **favor**. Specify at least the following argument:

**--combine=**

You can also specify the following argument:

**--favor=**

List the source files for the combine operation.

**Note:** Arguments with values that contain spaces (such as the names of protocols) must be enclosed in quotation marks. For example: "**--import=my file.xml**" and **-i"my file.xml"**

The ServiceImageManager command has the following format:

**--help -h**

Displays help text.

**--record -d**

Records the transactions into a service image file.

**--vrs=*recording-session-file* -v *recording-session-file***

Specifies the recording session file that contains all the configuration information for the recording or import operation.

**--go -G**

Specifies that for interactive or disconnected styles, the wait for a start signal is bypassed.

**--start=*time-spec* -S *time-spec***

Indicates that the recorder starts recording after the specified interval.

**--stop=*time-spec* -E *time-spec***

Indicates that the recorder will stop recording after the specified interval. This interval is relative to the time at which the recorder started recording, and is specified in milliseconds.

**--port=*port-number* -P *port-number***

Specifies the port to be used for recorder control. When used with **--record**, this command notes the port on which the recorder listens for control. When used with **--signal**, it notes the port to which the start/stop signal is sent.

**--signal=*start* | *stop* -g *start* | *stop***

Specifies the signal to send to a recorder in a different process. This command requires the **--port=** argument.

**--import=*raw/traffic-file* -i *raw/traffic-file***

Imports the specified raw or session traffic XML document into a service image file.

**--transport=*protocol* -t *protocol***

Specifies the transport protocol to use during an import operation.

**Values:**

- HTTP/S
- IBM MQ Series
- JMS
- Standard JMS (Deprecated)
- Java
- TCP
- JDBC (Driver based)
- CICS LINK <DPL>, DTP <MRO, LU6.1, LU6.2>
- CICS Transaction Gateway <ECI>
- DRDA
- IMS Connect
- SAP RFC via JCo
- JCo IDoc Protocol

Opaque Data Processing

**--request-data=*protocol* -r *protocol***

Specifies the request-side data protocol to use during an import operation.

**Values:**

- Web Services (SOAP)
- Web Services (SOAP Headers)
- Web Services Bridge
- WS-Security Request
- Request Data Manager
- Request Data Copier
- Auto Hash Transaction Discovery
- Generic XML Payload Parser
- Data Desensitizer

- Copybook Data Protocol
- Delimited Text Data Protocol
- Scriptable Data Protocol
- DRDA Data Protocol
- XML Data Protocol
- CICS Copybook Data Protocol
- CTG Copybook Data Protocol
- EDI X12 Data Protocol
- JSON
- SWIFT Data Protocol

REST Data Protocol

**`--response-data=protocol -R protocol`**

Specifies the response-side data protocol to use during an import operation.

**Values:**

- WS-Security Response
- Data Desensitizer
- Copybook Data Protocol
- Delimited Text Data Protocol
- Scriptable Data Protocol
- DRDA Data Protocol
- CICS Copybook Data Protocol
- CTG Copybook Data Protocol
- JSON

SWIFT Data Protocol

**`--non-leaf=CLOSE | WIDE | LOOSE -n CLOSE | WIDE | LOOSE -n tolerance`**

Specifies the default navigation tolerance to assume for any non-leaf conversation nodes created.

**Default:** WIDE

**`--leaf=CLOSE | WIDE | LOOSE -l CLOSE | WIDE | LOOSE -l tolerance`**

Specifies the default navigation tolerance to assume for any leaf conversation nodes created.

**Default:** LOOSE

**`--config=config-file -C config-file`**

In Recording mode, specifies a configuration file to use.

**--si-file=*vsi-file* -s *vsi-file***

Specifies the name of the service image file to which to import transactions. If this file does not exist, the application creates it. Otherwise, the imported transactions are merged with the existing service image.

**--vsm\_file=*vsm-file* -m *vsm\_file***

Specifies the name of the virtual service model file to create during a recording.

**--combine=*target-vsi-file* -c *target-vsi-file***

Combines one or more service image files into the named service image file. Unless the **favor** argument is specified, the source service images are favored.

**--favor=source | target -f source | target**

Specifies how to combine like transactions when combining service images.

**Values:**

■ **source:** Like transactions (and other data) update the target side from the source side.

**target:** Like transactions (and other data) leave the target side unchanged.

**--version**

Prints the VSE version number.

The ServiceImageManager command has two error codes:

- 1 - If an argument exception occurs (exit status bad param)
- 2 - If other general failure occurs

## VirtualServiceEnvironment Commands

The VirtualServiceEnvironment command is used to manage the VSE application. The executable is located in the [LISA\_HOME]\bin directory.

This command has the following format:

```
VirtualServiceEnvironment [-h|--help] [-n=name|-n=name]
[-m=registry-spec|--registry=registry-spec]
[-l=lab-name|--labName=lab-name] [-f|--force]
[-P=port-number|--port=port-number] [--version]
```

**--help -h**

Displays help text.

**--name=name -n name**

Defines the service name for the environment server.

**Default:** The system property **lisa.vseName**. The default value for the system property is **VSEServer**.

**--registry=registry-spec -m registry-spec**

The registry to which to connect.

**Example:**

**-m=tcp://localhost:2010/registry1**

**--labName=lab-name -l lab-name**

Specifies the name of the lab to use.

**Default:** Default

**--force -f**

Forces this server into the DevTest registry, replacing any object that is already registered by the service name of the environment.

**--port=port-number -P port-number**

Specifies the service port to which the VSE publishes. It is the same as specifying the port as part of the **--name** argument.

**--username=username -u username**

Specifies the DevTest security username.

**--password=password -p password**

Specifies the DevTest security password.

**--version**

Prints the VSE version number.





# Chapter 15: Java Agent VSE Properties

---

The configuration properties for the DevTest Java Agent include some properties that apply to VSE.

You can configure these properties from the Agents window of the DevTest Portal. The properties appear in the Settings tab.

#### **Startup mode**

Specifies the virtualization mode at startup>

##### **Values:**

- Passthrough
- Recording
- Playback

#### **Shallow recording**

Specifies to perform callbacks only on top level (in the stack frames) virtualized methods.

#### **Maximum graph size**

Defines the maximum size (in bytes) of a serialized-to-XML object graph.

#### **Reply timeout**

Specifies the maximum interval that is allowed for a VSE reply at playback. This value should exceed any think time in your tests.

#### **Omit by reference**

Specifies whether to record or replay arguments that reference and void methods modify.

##### **Values:**

- **Selected:** Do not record or replay arguments that reference and void methods modify. Select this check box if you are not virtualizing any void methods that change the arguments state. No void methods go to VSE and return immediately to enhance performance.
- **Cleared:** Record or replay arguments the reference methods and void methods modify.

#### **Cache responses**

Specifies whether to cache response objects key by their string representation to bypass unmarshalling.

##### **Values:**

- **Selected:** Cache response objects key by their string representation. When enabled, this property still causes a request to VSE. When the XML payload is received back in the agent, instead of deserializing it, we try to look for the object in a cache. The key is essentially an XML representation of the response. This setting is helpful to improve performance if responses are not in a mutable state. Selecting this property can increase memory usage.
- **Cleared:** Do not cache response objects key by their string representation.

**Enable JIT**

VSE jit means model/image logic is partially (or entirely) cached in the agent.

**JIT threshold**

Defines how many method invocations cause VSE jitting to occur.



# Glossary

---

## assertion

An *assertion* is an element that runs after a step and all its filters have run. An assertion verifies that the results from running the step match the expectations. An assertion is typically used to change the flow of a test case or virtual service model. Global assertions apply to each step in a test case or virtual service model. For more information, see Assertions in *Using CA Application Test*.

## asset

An *asset* is a set of configuration properties that are grouped into a logical unit. For more information, see Assets in *Using CA Application Test*.

## audit document

An *audit document* lets you set success criteria for a test, or for a set of tests in a suite. For more information, see Building Audit Documents in *Using CA Application Test*.

## companion

A *companion* is an element that runs before and after every test case execution. Companions can be understood as filters that apply to the entire test case instead of to single test steps. Companions are used to configure global (to the test case) behavior in the test case. For more information, see Companions in *Using CA Application Test*.

## configuration

A *configuration* is a named collection of properties that usually specify environment-specific values for the system under test. Removing hard-coded environment data enables you to run a test case or virtual service model in different environments simply by changing configurations. The default configuration in a project is named project.config. A project can have many configurations, but only one configuration is active at a time. For more information, see Configurations in *Using CA Application Test*.

## Continuous Service Validation (CVS) Dashboard

The *Continuous Validation Service (CVS) Dashboard* lets you schedule test cases and test suites to run regularly, over an extended time period. For more information, see Continuous Validation Service (CVS) in *Using CA Application Test*.

## conversation tree

A *conversation tree* is a set of linked nodes that represent conversation paths for the stateful transactions in a virtual service image. Each node is labeled with an operation name, such as withdrawMoney. An example of a conversation path for a banking system is getNewToken, getAccount, withdrawMoney, deleteToken. For more information, see *Using CA Service Virtualization*.

---

**coordinator**

A *coordinator* receives the test run information as documents, and coordinates the tests that are run on one or more simulator servers. For more information, see *Coordinator Server* in *Using CA Application Test*.

**data protocol**

A *data protocol* is also known as a data handler. In CA Service Virtualization, it is responsible for handling the parsing of requests. Some transport protocols allow (or require) a data protocol to which the job of creating requests is delegated. As a result, the protocol has to know the request payload. For more information, see [Using Data Protocols](#) (see page 209) in *Using CA Service Virtualization*.

**data set**

A *data set* is a collection of values that can be used to set properties in a test case or virtual service model at run time. Data sets provide a mechanism to introduce external test data into a test case or virtual service model. Data sets can be created internal to DevTest, or externally (for example, in a file or a database table). For more information, see *Data Sets* in *Using CA Application Test*.

**desensitize**

*Desensitizing* is used to convert sensitive data to user-defined substitutes. Credit card numbers and Social Security numbers are examples of sensitive data. For more information, see [Desensitizing Data](#) (see page 351) in *Using CA Service Virtualization*.

**event**

An *event* is a message about an action that has occurred. You can configure events at the test case or virtual service model level. For more information, see *Understanding Events* in *Using CA Application Test*.

**filter**

A *filter* is an element that runs before and after a step. A filter gives you the opportunity to process the data in the result, or store values in properties. Global filters apply to each step in a test case or virtual service model. For more information, see *Filters* in *Using CA Application Test*.

**group**

A *group*, or a *virtual service group*, is a collection of virtual services that have been tagged with the same group tag so they can be monitored together in the VSE Console.

**Interactive Test Run (ITR)**

The *Interactive Test Run (ITR)* utility lets you run a test case or virtual service model step by step. You can change the test case or virtual service model at run time and rerun to verify the results. For more information, see *Using the Interactive Test Run (ITR) Utility* in *Using CA Application Test*.

**lab**

A *lab* is a logical container for one or more lab members. For more information, see *Labs and Lab Members* in *Using CA Application Test*.

---

**magic date**

During a recording, a date parser scans requests and responses. A value matching a wide definition of date formats is translated to a *magic date*. Magic dates are used to verify that the virtual service model provides meaningful date values in responses. An example of a magic date is

`{{=doDateDeltaFromCurrent("yyyy-MM-dd","10");/*2012-08-14*/}}`. For more information, see [Magic Strings and Dates](#) (see page 47) in *Using CA Service Virtualization*.

**magic string**

A *magic string* is a string that is generated during the creation of a service image. A magic string is used to verify that the virtual service model provides meaningful string values in the responses. An example of a magic string is `{{=request_fname;/chris/}}`. For more information, see [Magic Strings and Dates](#) (see page 47) in *Using CA Service Virtualization*.

**match tolerance**

*Match tolerance* is a setting that controls how CA Service Virtualization compares an incoming request with the requests in a service image. The options are EXACT, SIGNATURE, and OPERATION. For more information, see [Match Tolerance](#) (see page 58) in *Using CA Service Virtualization*.

**metrics**

*Metrics* let you apply quantitative methods and measurements to the performance and functional aspects of your tests, and the system under test. For more information, see *Generating Metrics in Using CA Application Test*.

**Model Archive (MAR)**

A *Model Archive (MAR)* is the main deployment artifact in DevTest Solutions. MAR files contain a primary asset, all secondary files that are required to run the primary asset, an info file, and an audit file. For more information, see *Working with Model Archives (MARs)* in *Using CA Application Test*.

**Model Archive (MAR) Info**

A *Model Archive (MAR) Info* file is a file that contains information that is required to create a MAR. For more information, see *Working with Model Archives (MARs)* in *Using CA Application Test*.

**navigation tolerance**

*Navigation tolerance* is a setting that controls how CA Service Virtualization searches a conversation tree for the next transaction. The options are CLOSE, WIDE, and LOOSE. For more information, see [Navigation Tolerance](#) (see page 56) in *Using CA Service Virtualization*.

**network graph**

The network graph is an area of the Server Console that displays a graphical representation of the DevTest Cloud Manager and the associated labs. For more information, see *Start a Lab in Using CA Application Test*.

---

**node**

Internal to DevTest, a test step can also be referred to as a *node*, explaining why some events have node in the EventID.

**path**

A *path* contains information about a transaction that the Java Agent captured. For more information, see *Using CA Continuous Application Insight*.

**path graph**

A *path graph* contains a graphical representation of a path and its frames. For more information, see Path Graph in *Using CA Continuous Application Insight*.

**project**

A *project* is a collection of related DevTest files. The files can include test cases, suites, virtual service models, service images, configurations, audit documents, staging documents, data sets, monitors, and MAR info files. For more information, see Project Panel in *Using CA Application Test*.

**property**

A *property* is a key/value pair that can be used as a run-time variable. Properties can store many different types of data. Some common properties include LISA\_HOME, LISA\_PROJ\_ROOT, and LISA\_PROJ\_NAME. A configuration is a named collection of properties. For more information, see Properties in *Using CA Application Test*.

**quick test**

The *quick test* feature lets you run a test case with minimal setup. For more information, see Stage a Quick Test in *Using CA Application Test*.

**registry**

The *registry* provides a central location for the registration of all DevTest Server and DevTest Workstation components. For more information, see Registry in *Using CA Application Test*.

**service image (SI)**

A *service image* is a normalized version of transactions that have been recorded in CA Service Virtualization. Each transaction can be stateful (conversational) or stateless. One way to create a service image is by using the Virtual Service Image Recorder. Service images are stored in a project. A service image is also referred to as a *virtual service image* (VSI). For more information, see [Service Images](#) (see page 42) in *Using CA Service Virtualization*.

**simulator**

A *simulator* runs the tests under the supervision of the coordinator server. For more information, see Simulator Server in *Using CA Application Test*.

**staging document**

A *staging document* contains information about how to run a test case. For more information, see Building Staging Documents in *Using CA Application Test*.



---

**subprocess**

A *subprocess* is a test case that another test case calls. For more information, see Building Subprocesses in *Using CA Application Test*.

**test case**

A *test case* is a specification of how to test a business component in the system under test. Each test case contains one or more test steps. For more information, see Building Test Cases in *Using CA Application Test*.

**test step**

A *test step* is an element in the test case workflow that represents a single test action to be performed. Examples of test steps include Web Services, JavaBeans, JDBC, and JMS Messaging. A test step can have DevTest elements, such as filters, assertions, and data sets, attached to it. For more information, see Building Test Steps in *Using CA Application Test*.

**test suite**

A *test suite* is a group of test cases, other test suites, or both that are scheduled to execute one after other. A suite document specifies the contents of the suite, the reports to generate, and the metrics to collect. For more information, see Building Test Suites in *Using CA Application Test*.

**think time**

*Think time* is how long a test case waits before executing a test step. For more information, see Add a Test Step (example) and Staging Document Editor - Base Tab in *Using CA Application Test*.

**transaction frame**

A *transaction frame* encapsulates data about a method call that the DevTest Java Agent or a CAI Agent Light intercepted. For more information, see Business Transactions and Transaction Frames in *Using CA Continuous Application Insight*.

**Virtual Service Environment (VSE)**

The *Virtual Service Environment (VSE)* is a DevTest Server application that you use to deploy and run virtual service models. VSE is also known as CA Service Virtualization. For more information, see *Using CA Service Virtualization*.

**virtual service model (VSM)**

A *virtual service model* receives service requests and responds to them in the absence of the actual service provider. For more information, see [Virtual Service Model \(VSM\)](#) (see page 41) in *Using CA Service Virtualization*.