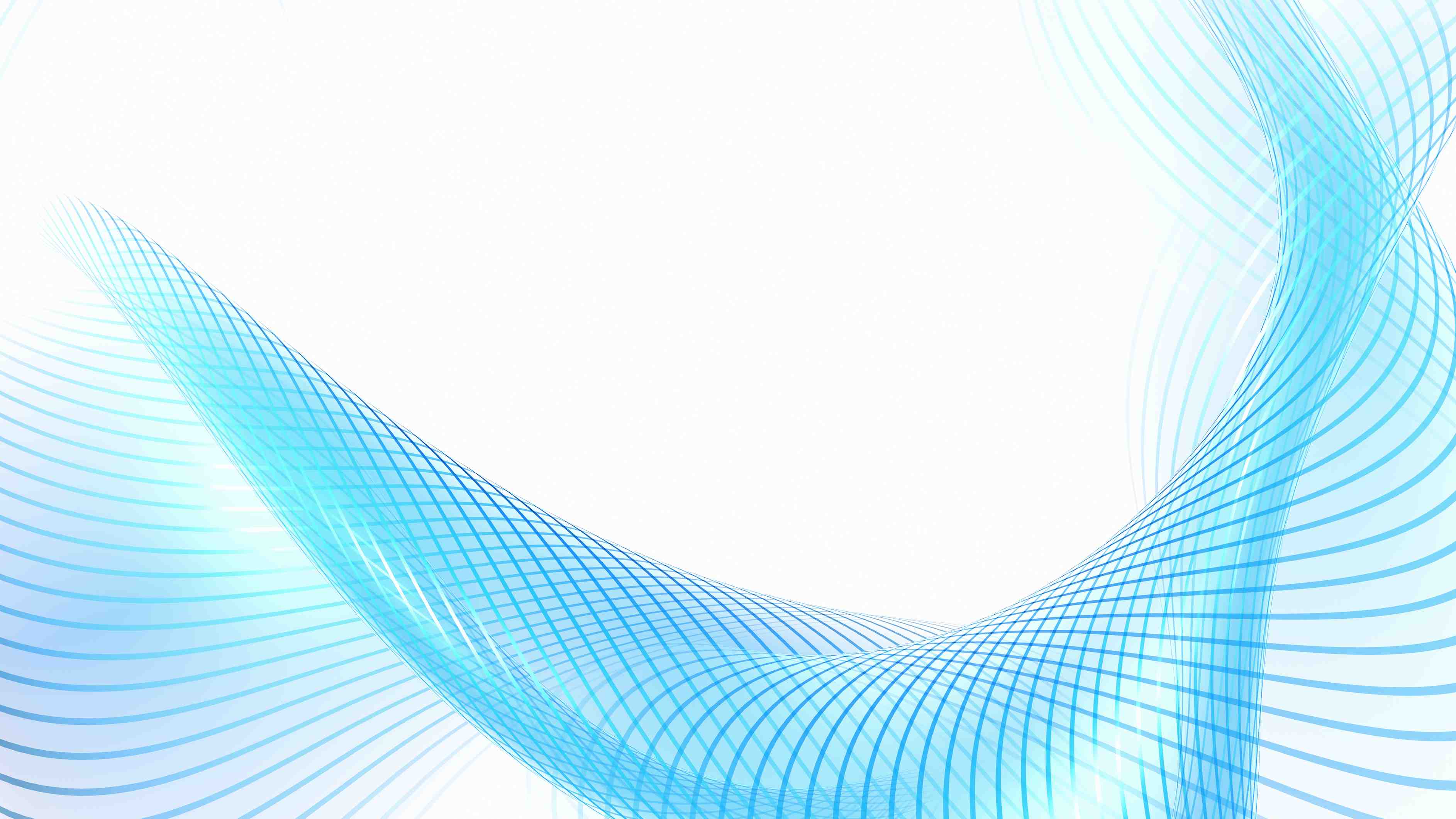


Reference Guide

Version 1.6



© Metaform Systems 2009-2010. All Rights Reserved.

Table of Contents

1. Getting Started 7

Choosing a Runtime 7

Runtime Layout 8

The Samples 9

Prerequisites 10

Building and Deploying The Starter Applications 10

Building and Deploying BigBank 11

Deploying BigBank to a Distributed Domain 12

Getting Help 13

2. Data Access: Hibernate, JPA and DataSources 14

Data Access Features 14

Using Hibernate and JPA 14

Injection 14

Transactions 16

Conversations and Extended Persistence Contexts 16

Low Level Data Access 17

Managing Persistence Contexts and Sessions 17

Enabling Hibernate and JPA 18

Installing the JPA Profile 18

JPA in the Maven Runtime 18

3. Transactions and Transactional Resources 19

Declaring Transactional Behavior 19

Transactional Annotations 19

Composite Configuration 20

DataSource Configuration 20

Runtime DataSource Configuration 20

Application DataSource Configuration 21

Managing DataSources 22

Transaction Manager Configuration 22

JMS Connection Factory Configuration 23

Enabling Transactions 23

4. Spring 24

Enabling Spring 24

Spring Extensions 24

Installing the Spring profile 24

Using Spring 24

Spring application context as composite implementation 24

JPA and Hibernate in Spring 26

JTA in Spring 26

5. Policy 28

Writing Policy Extensions 28

Defining and Applying Policies 28

Including Policies in a Contribution 30

Packaging Policies Separately 30

Creating Custom Intent and PolicySet Annotations 30

The Interceptor Builder SPI 30

6. Application Security 31

The Basic Security Provider 31

The Tomcat Realms Security Provider 32

The Spring Security Provider 32

Installing the Spring Security Profile 32

Enabling LDAP and JDBC Providers 32

Custom Security Providers 33

Using Authentication 33

Simulating Authentication in Integration Tests 33

Using Authorization 34

Using Authorization With Spring Beans 34

7. The Web Services Binding 36

Interoperability with .NET 36

Asynchronous Web Services 36

Standard Web Services Policies 37

Configuring Policy on Services and References 37

Configuring Connections 39

Debugging Options 40

8. The JMS Binding 41

Using the JMS Binding 41

Wire Formats 43

Configuring Connection Factories 43

Connection Factories and Destinations using JNDI 44

XA Transactions 45

Message Autoscaling 45

Configuring ActiveMQ 46

Binding.SCA 46

9. The Net Binding 48

Using HTTP Communications 48

Wire Formats 49

Binding.SCA 50

Using TCP Communications 50

Wire Formats 51

Binding.SCA 51

10. JAX-RS and RESTFul Services 52

Exposing Spring Beans as RESTful Resources 53

Enabling JAX-RS 54

11. The FTP Binding 55

Implementing an FTP Service 55

Handling Different Data Types 56

Configuring and Provisioning FTP Services 57

Provisioning Services 57

Connecting to External FTP Servers 58

Installing the FTP Profile 60

Installing the FTP Profile 60

Setting up the FTP Server 60

Configuring an Integration Test Environment 61

Supported Commands 61

12. Web Components 63

Implementing Web Components 63

Accessing Services 64

Accessing Stateless Services 64

Accessing Conversational Services 65

The SCA Tag Library 65

Fabric3 Server Features 66

Reference injection 66

Installing the Web Profile 66

13. Timer Components 67

Implementing a Timer Component 67

Trigger Events 68

Fixed rate 68

Repeat interval 68

Interval Class 68

Fire Once 69

Configuring Policy 69

Timer Pools 69

Timer Scopes 70

Clustered Singletons and Domain Scoped Timers 70

Using the Scheduled Executor Service 71

14. PubSub Eventing 72

A Basic Application 72

Enabling Eventing 74

15. Application Packaging 75

JAR contributions 75

The Fabric3 Contribution Plugin 75

Contribution Imports and Exports 76

XML Resource Sharing 76

Java Package Sharing 76

Fabric3 and OSGi Classloading 77

OSGi Bundles 78

WAR Archives 78

16. Running and Managing the Fabric3 Server 79

Installation 79

Extensions and Profiles 79

Deploying an Application 80

Runtime Administration 80

Command Line Administration 80

Server Shut Down 80

Runtime Configuration 80

Runtime Layout 80

Instantiating Multiple Runtime Instances From a Single Image 81

Runtime Cloning 82

Customizing the Runtime Image 82

Base Configuration 82

Securing the Runtime 83

HTTPS Configuration 83

Secure Artifact Provisioning 84

Secure Clustered Communications 85

Enabling JMX Authentication and Authorization 86

Monitoring and Logging 87

Monitor Hierarchy 87

Configuration 88

Work Management 88

Stopping Transports 89

Extension Configuration 89

Pass-By-Value 90

17. Distributed Domains 91

Key Concepts 91

The Domain 91

Zones 91

The Controller, Participants, and Zone Manager 91

Deployment Plans 92

Clustering 92

Scaling Down 92

Setting up a Distributed Domain 92

Installing the Controller 92

Installing Participants 92

Installing the Administration tool 93

18. Writing Manageable Applications 94

Publishing Components as MBeans 94

JMX Security 95

Monitoring 96

Segregating Monitor Events 96

19. The Tomcat Runtime 98

Configuration 98

Setting the Default HTTP Connector 98

Setting the Runtime Name 98

Enabling and Accessing JMX 99

20. Testing 100

Unit Testing 100

Integration Testing 100

Simple Example 101

Service Interface 101

Service Implementation 102

Service Composite 102

Service Test 102

Integration Composite 103

Maven Project Descriptor 104

Project Dependencies 105

Plugins 105

Running the Test 106

Using Extensions 107

Working with Easymock 109

Adding Mock Functionality to Hello World 109

Configuring Security Performing Security Tests 114

21. The Webapp Runtime 116

Installing the WebApp Runtime 116

Configuring the Web Application 117

Required web.xml Settings 117

Default Deployment Settings 118

Using Extensions and Profiles 119

Classloading 120

22. Extending Fabric3 121

# Getting Started

Fabric3 is a platform for developing, assembling, and managing distributed applications. Fabric3 provides the following features:

* A programming model based on Service Component Architecture (SCA) that is specifically designed for building loosely-coupled systems.
* The ability to use a variety of remote communication protocols in a unified, consistent manner without tying application logic to specific transport APIs.
* A cross-application policy framework for implementing and enforcing policies such as security and SLAs throughout an organization.
* A management framework for provisioning, controlling, and monitoring production deployments.
* Portability across a variety of middleware environments including Tomcat, and WebLogic.

In this chapter, we cover the basics of setting up and deploying an application using Fabric3. Subsequent chapters will cover developing loosely-coupled services, runtime configuration and topics such as integration with Java Persistence Architecture (JPA).

This manual assumes a basic understanding of SCA concepts. Before proceeding, if you have not done so, we recommend familiarizing yourself with SCA. The specifications themselves (downloadable from <http://www.oasis-opencsa.org/>) are generally not the most accessible source of introductory information. We recommend instead the paper by David Chappell, "Introducing SCA" (<http://www.davidchappell.com/articles/Introducing_SCA.pdf>), which presents a balanced and accurate overview of SCA. Alternatively, Jim Marino and Michael Rowley, *Understanding SCA* (Addison-Wesley) provides an in-depth treatment including Fabric3.

## Choosing a Runtime

Fabric3 is designed to run in a variety of environments and provide application portability between them. This is important for flexibility when deploying into different production environments but also essential for effective iterative development. Fabric3 supports running applications out-of-container, in an integration test environment, or in a managed environment such as a JEE application server with minor configuration changes. This provides an efficient and automated mechanism for conducting local testing, integration testing, and production deployment. The following are brief descriptions of the available Fabric3 runtimes:

* **The Standalone Server:** Fabric3 includes a server that can be run in a single VM or in a distributed environment with multiple runtime clusters.
* **The Fabric3 Tomcat Runtime**:This runtime is hosted in a Tomcat server. The Fabric3 Tomcat Runtime is integrated with Tomcat facilities including JMX and logging. This runtime provides additional services including support for service clustering, JTA transactions, and datasource pooling.
* **The Fabric3 WebLogic Runtime**: This runtime is hosted in a WebLogic server. The Fabric3 WebLogic Runtime is integrated with WebLogic facilities including clustering, transactions, datasources, JMX and logging.
* **The iTest Runtime**: The iTest runtime allows applications to be tested as part of a Maven project build. Fabric3 provides facilities for writing automated tests and mock services that can used to verify a complete application or individual subsystems in an automated fashion.
* **The Ant Runtime**: The Ant runtime allows applications to be tested as part of a Ant project build. Similar to the Maven runtime, this distribution provides facilities for writing automated tests and mock services that can used to verify a complete application or individual subsystems in an automated fashion.

### Runtime Layout

The runtime image is organized as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| /bin | | | | Startup modules |
| /lib | | | | Modules required to start the runtime host |
| /boot | | | | Modules required for the runtime bootstrap and primordial system services |
| /host | | | | Libraries shared between the runtime and application (e.g. web services annotations) |
| /extensions | | | | Extension modules that are able to be loaded by all runtime instances |
| /runtimes | | | | Specific runtime instance configuration is hosted by default under this directory |
|  | <runtime-name> | | | top level directory for a runtime configuration |
|  |  | /config | | Contains systemConfig.xml for configuring the runtime and extensions |
|  |  | /deploy | | File system deploy directory for the controller and single-VM runtimes |
|  |  | /repository | |  |
|  |  |  | /runtime | Extensions only loaded for the runtime image |
|  |  |  | /user | User contributions (only populated on the controller and single-VM runtimes) |
|  |  | /data | | Persistent data directory for a runtime instance (e.g. transaction log) |
|  |  | /tmp | | Temporary data and artifact cache for a runtime instance |

For basic startup, the runtime name is required, which maps to a configuration under runtimes:

|  |  |
| --- | --- |
| java -jar server.jar controller | launches a runtime using the runtimes/controller image |
| java -jar server.jar vm | launches a runtime using the runtimes/vm image |
| java -jar server.jar participant | launches a runtime using the runtimes/participant image |
| java -jar server.jar foo | launches a custom runtime using the runtimes/foo image |

## The Samples

The samples are intended to demonstrate the capabilities of Fabric3 using the standalone or Tomcat distributions. The samples contain individual applications designed to showcase specific features and are organized as follows:

* **Starter:** Contains several variations of a calculator application that show how to create services, wire them, and expose them as a web services endpoint, and REST resource.
* **Features:** Contains applications that demonstrate how to use specific Fabric3 features, including JPA/Hibernate integration and Pub/Sub eventing.
* **Apps**. Contains complete applications that demonstrate SCA and Fabric3 best-practices. There is currently one application, BigBank.

For projects that intend to use Spring, Fabric3 also ships with a set of dedicated Spring samples. These samples are ports of the SCA Java samples described in this chapter, with SCA Java components replaced by Spring beans. Basic layout, configuration and deployment remain the same across both sets of samples.

### Prerequisites

The samples may be downloaded from <http://www.fabric3.org/downloads> and require the following software:

* JDK 6.0 or later.
* Maven 2.2.1 or later to build the tutorial projects. Maven can be downloaded from <http://maven.apache.org/download.html>.

***Note that it is not necessary to download the Fabric3 runtime distribution in addition to the samples as the sample build process will automatically download a distribution and configure a Fabric3 runtime cluster.***

### Building and Deploying The Starter Applications

The starter calculator applications are the recommended way to get familiar with Fabric3. To build and deploy the calculator applications, do the following:

* **Build the starter modules**

In the directory where you extracted the samples distribution, go to the */starter* project folder and execute:

mvn clean install



Note internet access is required the first time the project is built so Maven can download the required project dependencies. Remote access can be turned off for subsequent builds by executing:

mvn -o clean install

JARs containing the application artifacts will be created in the /target output directories for each application.

* **Build the Fabric3 server distribution.**

Download the Fabric3 standalone runtime by executing the Maven build script from the samples */servers/vm* directory:

mvn -o clean install

This will create a server image for use with the samples in the /target/image directory.  Note the server is configured to run as a single instance. The */servers/cluster* directory contains modules which will configure a multi-cluster Fabric3 installation. The starter applications can be deployed to either topology.

* **Start the server.**

To launch a Fabric3 server in single-VM mode, execute the following from the /bin directory */servers/vm/target/image* by executing:

java -jar server.jar

Alternatively, the Fabric3 server can be started from another working directory by executing:

java -jar <path to bin directory>server.jar

* **Deploy the application**.

After the server has booted, deploy a calculator application archive by copying it to the Fabric3 runtime /deploy directory.

The runtime will write a message to the console after the war has been deployed. Depending on the application deployed, one of the following will be available.

* The web calculator UI can be accessed at:

[http://localhost:8181/ calculator/entry.html](http://localhost:8181/%20calculator/entry.html).

* The REST calculator resource can be accessed using a GET with the formula to calculate included in the URI as in:

<http://localhost:8181/calculator/1+1>

* The web services calculator can be accessed at:

<http://localhost:8181/calculator>

and its WSDL from:

<http://localhost:8181/calculator?wsdl>

The samples also contain a separate client for the web services calculator. The client module is located at */samples/wscalc-client.* The WSCalcClient class can be executed using the Java command line or via an IDE.

## Building and Deploying BigBank

To build and deploy BigBank, do the following:

* **Build the BigBank source**.

To build BigBank, go to the bigbank project folder and execute:

mvn clean install

The build will produce two archives: bigbank-loan-1.6.jar (the server module) and bigbank-client-1.6.jar (the web service client)

* **Start the server.**

Boot the Fabric3 runtime from the server/vm/image/bin directory by executing:

java -jar server.jar

* **Deploy the archives**.

Copy bigbank-loan-1.6.jar to the runtimes/vm/deploy directory.

* **Execute the client**

In your IDE or from the command line, execute *org.fabric3.samples.bigbank.client.ws.LoanServiceClient* located in the bigbank-client-16.jar.

## Deploying BigBank to a Distributed Domain

The samples distribution also contains an automated build process for producing a set of clustered servers. To create the clustered servers, execute the following from the servers/cluster directory:

mvn clean install

The build will create three server images located in the target directory of each module under /servers/cluster: controller, zone1, and zone2.

Follow the steps in the previous section to build the BigBank application. When the build completes, launch the controller and zone runtimes:

* From the controller/target/image/bin directory:

java -jar server.jar controller

* From the zone1/target/image/bin directory:

java -jar server.jar participant

* From the zone2/target/image/bin directory:

java -jar server.jar participant

After booting, the runtimes will discover each other and form a distributed domain consisting of two cluster zones. Note the runtimes may be on the same machine or different machines.

Copy bigbank-loan-1.6.jar to the controller/target/image/runtimes/controller/deploy directory. The controller will provision the loan service to zone1 and the backend services to zone2.

To run additional zone participants, copy one of the images and follow the instructions above for launching the servers. If more than one server is run on the same machine, you will need to modify the HTTP and HTTPS ports in config/participant/systemConfig.xml.

## Getting Help

At some point you may require help with Fabric3. The best place to obtain pointers, advice or assistance troubleshooting a problem is the user mailing list, which can be accessed at <http://xircles.codehaus.org/projects/fabric3/lists>. There are several mailing lists for Fabric3. Please post questions to the user list. The developer list is intended for topics related to ongoing Fabric3 development and is not a general forum for questions. However, if you are interested in Fabric3 development, we encourage you to participate in discussions.

Should you encounter a bug, we encourage you to file a report in the online JIRA system at <http://jira.codehaus.org/browse/FABRICTHREE>. If possible, please include a detailed description and failing testcase (or other appropriate means) to reproduce the problem. Assisting Fabric3 developers in reproducing the problem generally leads to faster resolution.

# Data Access: Hibernate, JPA and DataSources

Fabric3 provides first-class SCA/Hibernate integration by allowing EntityManager and Hibernate Session instances to be injected into components. These injected instances are managed by the Fabric3 runtime, thereby alleviating the need in most cases for applications to manage them through such mechanisms as setting ThreadLocal variables. In addition, Fabric3 provides consistent transactional behavior (i.e. no code or configuration changes required) for applications using Hibernate, whether they are deployed to a JEE, JSE or other environment.

## Data Access Features

Data access features supported by Fabric3 include:

* Injection of entity manager factories using @PersistenceUnit annotation
* Injection of entity managers using @PersistenceContext annotation
* Injection of Hibernate Session objects using the @PersistenceContext. Hibernate Session injection provides components access to Hibernate APIs.
* Injection of XA and non-XA DataSource instances
* JDBC connection pooling
* Support for extended and transaction scope entity managers
* Integration with SCA transaction policies
* Transaction optimization across persistence units

## Using Hibernate and JPA

### Injection

EntityManager and EntityManagerFactory instances are injected into component implementations using the @PersistenceContext and @PersistenceUnit from the javax.persistence package respectively. These annotations are made available to all contributions at runtime. The following examples demonstrate injecting an EntityManager and EntityManagerFactory:

public class EmployeeDAOImpl implement EmployeeDAO {

@PersistenceContext(name="employeeEmf" unitName="employee") protected EntityManager em;

public void save(Employee employee) {

   m.persist(employee);

}

}

public class EmployeeDAOImpl implement EmployeeDAO {

@PersistenceUnit(unitName="employee")

  protected EntityManagerFactory emf;

public void save(Employee employee) {

EntityManager em = emf.createEntityManager();

em.persist(employee);

}

}

In the above examples, the persistence unit "employee" and persistence context "employeeEmf" are defined in the JPA persistence.xml file. This file can be located in any directory in the contribution containing the components, although it is customary to place it in /META-INF for JAR-based contributions.

The following is an example persistence.xml file:

<persistence xmlns="http://java.sun.com/xml/ns/persistence"…>

<persistence-unit name="employee" transaction-type="JTA">

<jta-data-source>EmployeeDS</jta-data-source>

<class>org.fabric3.jpa.model.Employee</class>

<properties>

<property name="hibernate.dialect"

value="org.hibernate.dialect.HSQLDialect"/>

<property name="hibernate.hbm2ddl.auto"

value="create-drop"/>

</properties>

</persistence-unit>

</persistence>

Similarly, Hibernate Session instances can be injected by changing the injected type to *org.hibernate.Session:*

public class EmployeeDAOImpl implement EmployeeDAO {

@PersistenceContext(name="employeeEmf" unitName="employee") protected Session session;

public void save(Employee employee) {

   session.save(employee);

}

}

### Transactions

Fabric3 integrates JPA with SCA transaction policies. By using SCA transaction policies, operations performed against an EntityManager or Session can be done in the context of a transaction. Fabric3 handles associating EntityManager (and Session) instances with transaction contexts transparently to applications. For example, to enable global transactions, use the SCA @ManagedTransaction as shown in the following example:

@ManagedTransaction

public class EmployeeDAOImpl implement EmployeeDAO {

@PersistenceContext(name="employeeEmf" unitName="employee") protected EntityManager em;

public void save(Employee employee) {

   m.persist(employee);

}

}

In the above example, a new transaction will be started if one does not exist. All operations performed against the injected EntityManager instance will take place in the newly-created or existing transaction context. If the transaction completes successfully, the EntityManager will be flushed and changes written to the database. Otherwise, a rollback will be issued, discarding any changes.

Alternatively, transaction policy can be configured using the @requires attribute on a component entry in a composite:

<component name="EmployeeDAO" requires="sca:managedTransaction">

<implementation.java class="com.foo.EmployeeDAOImpl"/>

</component>

### Conversations and Extended Persistence Contexts

The default lifecycle for injected EntityManager instances is tied to the current transaction context. This means that the persistence context will be flushed when the transaction completes. In certain scenarios, it is useful to use a JPA extended persistence context where entities are maintained in memory in a disconnected state accross transaction boundaries. Fabric3 provides support for container-managed extended persistence contexts by associating them with an SCA conversation. EntityManagers instances configured to use an extended persistence context will be managed by Fabric3 for the duration of a conversation. If multiple transactions are committed during the course of a conversation, an EntityManager instance will be flushed and changes written to the database. Between transactions, while a conversation is still active, the persistence context associated with the EntityManager will be maintained in a disconnected state. This provides performance benefits as entities can be cached in memory as well as changes made to them without having to retain resources such as JDBC connections. When a conversation ends, the extended persistence context is closed.

The following is an exampe of injecting an EntityManager with an extended persistence context on a composite-scoped component:

@Scope("CONVERSATION")

public class ConversationEmployeeDAOImpl implements ConversationEmployeeDAO {

private EntityManager employeeEM;

@PersistenceContext(name = "employeeEmf", unitName = "employee", type = PersistenceContextType.EXTENDED)

protected EntityManager em;

// ...

}

## Low Level Data Access

Applications may require direct access to a DataSource in order to use JDBC. DataSource instances can be injected using the *org.fabric3.api.annotation.Resource* annotation where the mappedName value corresponds to a configured DataSource name (for specifics of DataSource configuration, see Chapter 5, “Transactions and Transactional Resources”):

public class EmployeeDAOImpl implement EmployeeDAO {

@Resource(mappedName=”EmployeeDataSource”)

protected DataSource dataSource;

// …..

}

## Managing Persistence Contexts and Sessions

Hibernate Session statistics are available via JMX. A JMX client such as JConsole can connect to a runtime using the following URL form:

*service:jmx:rmi:///jndi/rmi://<Runtime IP Address>:<JMX Port>/server*

Statistics MBeans are available under /fabric3/runtime/resource/Hibernate.

## Enabling Hibernate and JPA

JPA and Hibernate support is provided via a a set of optional extensions which must be installed in the runtime. These extensions contain the Fabric3 Hibernate extensions as well as a JTA transaction manager (Atomikos) and an XA-compliant DataSource pooling implementation. As a convenience, the required extensions are packaged together in a Fabric3 profile.

### Installing the JPA Profile

To install JPA support in the Standalone Server, download the JPA profile from the Fabric3 web site and copy the appropriate jars into the /extensions directory. After you restart the server, JPA support will be activated.

See “Transactions and Transactional Resources” for more information. Note you will need to configure persistence.xml files to use the datasources setup in the runtime.

### JPA in the Maven Runtime

To use JPA in the Maven runtime, you must enable the JPA profile as part of the Fabric3 plugin configuration. See the Maven runtime chapter for details on activating profiles in the Fabric3 plugin configuration.

# Transactions and Transactional Resources

Fabric3 provides consistent transaction management across all supported runtime environments. In the Maven and Server runtimes, Fabric3 uses Atomikos TransactionEssentials to manage JTA transactions and transparently handle XA resource enlistment for JDBC DataSources and JMS Sessions. The Fabric3 Webapp runtime can be configured to use the host application server’s JTA transaction manager.

## Declaring Transactional Behavior

Transactional behavior can be declared using annotations or in a component configuration contained in a composite file.

### Transactional Annotations

The following transactional annotations are provided, which correspond to SCA transaction policies:

* org.oasisopen.sca.annotation.ManagedTransaction

Note currently only global (XA) transactions are supported.

* org.fabric3.api.annotation.transaction.PropagatesTransaction
* org.fabric3.api.annotation.transaction.SuspendsTransaction

Transaction annotations may be applied to an implementation class, method or service reference The following component implementation demonstrates how the various annotations are used:

@ManagedTransaction

public class ManagedTransactionService implements

TransactionalService {

@Reference

@SuspendsTransaction

protected TransactionalService suspendedTransactionService;

@Reference

@PropagatesTransaction

protected TransactionalService propagatesTransactionService;

public void call() throws Exception {

suspendedTransactionService.call();

propagatesTransactionService.call();

}

}

In the above example, when invoked, the component will either start a new transaction or join an existing one. When it invokes the two services, the current transaction will be suspended and propagated respectivel.

### Composite Configuration

Alternatively, transaction semantics can be declared in a composite using the @requires attribute:

<component name="TransactionalService">

<implementation.java class=".."

requires="sca:managedTransaction"/>

<reference name=" suspendedTransactionService"...>

requires="sca:suspendsTransaction"/>

<reference name="propagatesTransactionService"...>

requires="sca:propagatesTransaction"/>

</component>

Generally, transaction annotations are the recommended approach since transactional semantics are generally a fixed implementation requirement as opposed to a configuration option.

## DataSource Configuration

Fabric3 transparently manages JDBC DataSource enlistment and provides XA connection pooling. The following sections describe how to configure datasources and JDBC drivers for each runtime environment.

DataSources can either be *runtime* or *application* scoped. Runtime scoped DataSources are defined as part of the runtime configuration and remain for the duration of the runtime instance. Application scoped DataSources are defined as part of a composite and are activated (and de-activated) when the composite is deployed.

### Runtime DataSource Configuration

To configure runtime data sources, include the appropriate JDBC drivers in the extensions/datasource directory, creating it if needed. Next, you will need to update the systemConfig.xml file for the profile you are running, adding a <datasource> entry. The following demonstrates how to configure a connection pool using the non-XA MySQL driver:

<datasources>

<datasource name=”LoanApplicationDS”

driver=”com.mysql.jdbc.Driver”

url=”jdbc:mysql://localhost/bigbank”

username=”bigbank”

password=”bigbank”>

<!—key value pairs for JDBC properties in the form

<key>value</key>

-->

</datasource>

</datasources>

The following sub-elements can be set under the <datasource> element:

* type - the data source type: either “non\_xa” (default) or “xa”
* minPoolSize – the minimum number of connections to keep active in the connection pool
* maxPoolSize – the maximum number of connections to create in the connection pool
* poolSize – used to set the minimum and maximum pool size to a single value
* connectionTimeout –the maximum amount of time in seconds the pool will block waiting for a connection to become available in the pool when it is empty
* loginTimeout – the maximum time in seconds that the data source will wait while attempting to connect to a database.
* maintenanceInterval – the time in seconds between maintenance periods
* maxIdle – the time in seconds before a pooled connection can be reclaimed
* reap – the time in seconds the connection pool will allow a connection to be borrowed before claiming it back

If a data source is an XA datasource, driver-specific properties can be configured by using sub-elements of <datasource> in the format:

*<property\_name>property\_value</property\_name>*

### Application DataSource Configuration

Application data sources can be configured in a composite using the Fabric3 <datasources> element, as shown in the following:

<?xml version="1.0" encoding="UTF-8"?>

<composite

xmlns="http://docs.oasis- open.org/ns/opencsa/sca/200912"

**xmlns:f3-other="urn:fabric3.org:other"**..>

**<f3-other:datasources>**

**<datasource name="MessageDS"**

**driver="org.h2.jdbcx.JdbcDataSource" type="xa">**

**<URL>jdbc:h2:mem:MessageDS;DB\_CLOSE\_DELAY=-1</URL>**

**<minPoolSize>5</minPoolSize>**

**<maxPoolSize>10</maxPoolSize>**

**</datasource>**

**</f3-other:datasources>**

</composite>

The data source will be activated when the composite is deployed and removed when the composite is undeployed.

The same sub-elements as used in runtime data source configuration can be used.

## Managing DataSources

DataSource statistics are available via JMX. A JMX client such as JConsole can connect to a runtime using the following URL form:

*service:jmx:rmi:///jndi/rmi://<Runtime IP Address>:<JMX Port>/server*

Statistics MBeans are available under /fabric3/runtime/resource/datasources.

## Transaction Manager Configuration

Transaction manager properties can be set using the <transaction.manager> element in the systemConfig.xml file:

<transaction.manager>

<key>value</key>

</transaction.manager>

Key/value pairs correspond to Atomikos transaction manager settings which are documented here: http://www.atomikos.com/Documentation.

The transaction manager logging level can be set using the @logging attribute on <transaction.manager>.

### JMS Connection Factory Configuration

Fabric3 transparently manages JMS Session enlistment with the current transaction if global transacted messaging is configured. No further configuration is necessary other than ensuring the JMS ConnectionFactory has XA support.

## Enabling Transactions

The Fabric3 runtimes must be configured with the required transaction extensions. If a runtime has the JPA or JMS profiles installed, the transaction extensions will be included.

# Spring

Fabric3 provides integration with Spring XML-based applications. Fabric3 is able to expose a Spring bean as an endpoint via SCA.

## Enabling Spring

### Spring Extensions

There are two extensions to enable Spring support in Fabric3: the Spring implementation type extension and an extension that supplies the Spring library classes.

By separating these two extensions, it is possible to use custom versions of the Spring library extension. Under the covers, the Spring implementation extension will use the Fabric3 extension point mechanism to resolve the Spring library extension.

#### Spring implementation type extension

This extension handles creating and wiring Spring components in a domain. The domain can be single-VM or distributed.

#### Spring Library classes extension

This extension is a jar containing Spring classes and other dependencies. To create a custom library extension, a jar containing other jars in META-INF/lib/ and an SCA manifest needs to be produced.

### Installing the Spring profile

To install Spring support into the Fabric3 runtime, download the Spring profile from the Fabric3 web site and copy the appropriate jars into the correct directory. To find out in which directory these jars should be copied, read the [runtime layout description](#_Runtime_Layout). After restarting the runtime, Spring support will be activated.

## Using Spring

### Spring application context as composite implementation

To expose a Spring Bean as a SCA service in a composite component, the Spring implementation should be used:

<component name="MyComponent">

<implementation.spring location="META-INF/appContext.xml"/>

...

</component>

The location attribute of the implementation.spring element, should refer to the applicationcontext xml file where the Spring bean is declared.

In the Spring application context, the SCA service should be declared using the SCA namespace:

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

...

http://docs.oasis-open.org/ns/opencsa/sca/200912 http://docs.oasis-open.org/opencsa/sca-assembly/sca-1.1.xsd">

<sca:service name="*MySCAService*" target="*mySpringBean*"/>

<bean id="*mySpringBean*" class="... .*MyClass*"/>

</beans>

The above code snippet shows an explicit declaration of a Spring SCA service. The target attribute of the sca:service element refers to a Spring Bean.

To use an SCA reference in your Spring bean, use the sca:reference element:

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

...

http://docs.oasis-open.org/ns/opencsa/sca/200912 http://docs.oasis-open.org/opencsa/sca-assembly/sca-1.1.xsd">

<sca:service name="MySCAService" target="mySpringBean"/>

<bean id="mySpringBean" class="... .MyClass">

<property name="myRefBean" ref="*myRefService*"/>

</bean>

<sca:reference name="*myRefService*" type="foo.bar.MyServiceI"/>

</beans>

This reference is also declared in a composite file, using the spring implementation element for declaring the component.

## JPA and Hibernate in Spring

Fabric3 provides support for using JPA and Hibernate in Spring. Fabric3 provides a Spring Bean org.fabric3.implementation.spring.api.Fabric3EntityManagerFactoryBean, that extends the standard Spring LocalContainerEntityManagerFactoryBean. This gives the ability to have pooled datasources and it makes it unnecessary to specify a persistence provider in persistence.xml. With this approach, vendor-specific configurations can be avoided in application artifacts .

<bean id="EntityManagerFactory" class="org.fabric3.implementation.spring.api.Fabric3EntityManagerFactoryBean">  
        <property name="persistenceUnitName" value="*unitName*"/>  
</bean>

<bean id="PersistenceAnnotationBeanPostProcessor" class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>  
  
<tx:annotation-driven/>

Alternatively, the standard Spring LocalContainerEntityManagerFactoryBean can be used instead of the Fabric3EntityManager. In that case, the naming of the datasource needed, should be placed in the persistence.xml.

<bean id="EntityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">  
         <property name="persistenceUnitName" value="unitName"/>  
         <property name="dataSource" ref="dataSourceName"/>  
</bean>

## JTA in Spring

Fabric3 supports Spring JDK Proxy-based AOP and transactions. (Use the Spring @Transactional annotation to use transactional behavior).

Fabric3 JTA transaction manager and pooled datasources are automatically aliased in each Spring application context. This provides full transactional support (including XA and pooled datasources) to Spring persistence (JPA, Hibernate, JDBC) and makes it also possible to automatically propagate transactions from other SCA components and bindings to Spring Beans.

# Policy

Fabric3 support for SCA Policy uses a powerful interceptor-based infrastructure that can be used in a range of scenarios:

* Policies can enforce a number of behaviors such as transactions, security, or SLA alerts.
* Policies can be explicitly configured on components and remote communications
* Policies can be enforced on a domain level and “attached” to components and remote communications at deployment
* Policies can be dynamically enforced on a domain level and attached to existing components and remote communications

## Writing Policy Extensions

Fabric3 support standard policies such as transactions and also provides a way to write custom policies. These custom policies may be bundled directly with user code (e.g. if it is only applicable to a service or set of services) or as a separate contribution.

### Defining and Applying Policies

Policies are defined in an SCA definitions file. A definitions file can contain *intents*, *policy sets, binding types,* and *implementation types*. The specifics of SCA policy are beyond the scope of this reference. Briefly, though, intents are abstract requirements that can be declared by a component or on a reference or service. Intents are matched to policy sets, which provide concrete configuration for a behavior. For example, a the “message authentication” intent may be mapped to a policy set that specifies WS-Security. Intents are therefore a way to specify a requirement without tying a component to a specific underlying technology. In an environment that does not use WS-Security, the “message authentication” intent would be mapped to a different security technology.

Policy set configuration in Fabric3 varies by binding. For example, defining a policy set for use with the Web Services binding is done using Axis2’s XML configuration dialect or WS-Policy if Metro is used.

Fabric3 also provides a general mechanism for defining policy sets that can be used across bindings and wires. This involves writing an interceptor that will be called to process a message invocation. The following XML definitions file demonstrates how to define an interceptor policy:

<?xml version="1.0" encoding="ASCII"?>

<definitions

xmlns="<http://docs.oasis-open.org/ns/opencsa/sca/200903>”

xmlns:f3-policy="urn:fabric3.org:policy">

<policySet name="testImplementationPolicy"

f3-policy:phase="INTERCEPTION"

attachTo="//component">

<f3-policy:interceptor

class="org.fabric3.interceptor.TestInterceptor"/>

</policySet>

</definitions>

The “attachTo” attrbitute instructs Fabric3 to apply the interceptor to all components in the domain (note you would probably not want to do this in a real-world application). This is termed “SCA external policy attachment”. Again, the details of these SCA mechanisms are beyond the scope of this reference. Briefly, though, the value of the attachTo attribute is an XPath expression that is applied to the domain infoset. This is an extremely powerful capability. Policies can be dynamically attached to any component, binding, or wire in the domain by specifying an XPath expression. This attachment can happen at deployment or be applied to already deployed components, bindings, and wires.

Note that in addition to external attachment, Fabric3 also supports a “pull” policy model where policies (or intents) are specified in the component configuration or via annotations.

The interceptor class for the previous policy set is shown below:

package org.fabric3.interceptor;

import org.fabric3.spi.invocation.Message;

import org.fabric3.spi.wire.Interceptor;

public class TestInterceptor implements Interceptor {

private Interceptor next;

public Message invoke(Message message) {

// perform some processing.

return next.invoke(message);

}

public void setNext(Interceptor interceptor) {

next = interceptor;

}

public Interceptor getNext() {

return next;

}

}

Note the interceptor class implements org.fabric3.spi.wire.Interceptor.

### Including Policies in a Contribution

Policies (e.g. the definitions.xml file and supporting classes such as an interceptor) can be packaged and deployed as part of a contribution. This is useful if the policy only applies to a particular set of services. The only required step to do this is to ensure the contribution manifest (sca-contribution.xml) contains an import.java entry for the org.fabric3.spi package:

<contribution xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200903" xmlns:f3-core="urn:fabric3.org:core" f3-core:extension="true">

<import.java package="org.fabric3.spi.\*"/>

</contribution>

### Packaging Policies Separately

Policies can also be packaged and deployed as individual contributions. Again, the only specific requirement is that the contribution manifest import the org.fabric3.spi package. One important difference, however, with this approach is that the interceptor class will be loaded in a different classloader than the source and targets of a wire. In most cases, this will not matter as the component classloaders will not be needed. In cases where the source and target classloaders must be accessed, the policy contribution must use the lower level Fabric3 interceptor builder SPI.

### Creating Custom Intent and PolicySet Annotations

TBD

### The Interceptor Builder SPI

TBD

# Application Security

Fabric3 includes an extensible security framework that supports authentication and authorization. Authentication is typically specified as a policy intent on a binding to perform client, server, or mutual authentication. When a client is authenticated, a security subject is associated with request messages sent by the client. This security subject can then be used to authorize access to service operations based on roles.

The security provider varies by runtime. The Standalone, Maven, and Ant runtimes are configured by default to use a basic security provider. The Tomcat runtime is configured with a provider that delegates to Tomcat security realms. There is also a Fabric3 extension that uses Spring Security, which can be installed in any of the Fabric3 runtimes.

## The Basic Security Provider

The Standalone runtime includes a basic security provider that allows users and roles to be statically defined in a configuration file, security.xml, located in the runtime /config directory. An example file is shown below:

<users>

<user>

<username>foo</username>

<password>bar</password>

<roles>

<role>role1</role>

<role>role2</role>

</roles>

</user>

</users>

In the Maven runtime, the same security information is configured using a systemConfig entry:

<systemConfig>

<![CDATA[

<config>

<users>

<user>

<username>foo</username>

<password>bar</password>

</user>

</users>

</config>

]]>

</systemConfig>

## The Tomcat Realms Security Provider

Fabric3 integrates with Tomcat security realms in place of the basic Fabric3 provider. When using authentication and authorization policies in applications, Fabric3 will transparently delegate to Tomcat security.

## The Spring Security Provider

Fabric3 includes integration with Spring Security that can be used in place of the basic security provider. The Spring Security extension supports LDAP and JDBC-based access control. Since it is dynamic in nature (users and roles can be added at runtime), it is much more powerful than the basic and Tomcat providers.

### Installing the Spring Security Profile

The Spring Security provider is included in the Spring profile. When installing the profile in the standalone runtime, it is important to remove the fabric3-security-impl extension that is configured by default.

### Enabling LDAP and JDBC Providers

The Spring Security provider is configured as part of the system configuration using the <spring.security> element. Configuration follows the standard Spring Security configuration format. The following illustrates how JDBC security is configured using a Fabric3 JDBC connection pool:

<datasources>

<datasource name="SecurityDS" …/>

</datasources>

<security>

<spring.security>

<authentication-manager>

<authentication-provider>

<jdbc-user-service data-source-ref="SecurityDS"/>

</authentication-provider>

</authentication-manager>

</spring.security>

</security>

The next example demonstrates configuring LDAP-based security:

<security>

<spring.security>

<authentication-manager>

<ldap-server

url="ldap://localhost:1389/dc=example,dc=com"

manager-dn="cn=Directory Manager"

manager-password="password"/>

<ldap-authentication-provider

user-search-base="ou=people"

user-search-filter="uid={0}"

group-search-filter="member={0}"

group-search-base="ou=groups"/>

</authentication-manager>

</spring.security>

</security>

## Custom Security Providers

The basic provider can be replaced by a more capability (and dynamic) alternative by substituting the fabric3-security-impl.jar in the extensions repository. For details on implementing an alternative provider, see the Javadoc for the org.fabric3.spi.security package in fabric3-spi.

## Using Authentication

Authentication is typically enabled on a binding configuration. Please refer to the binding chapters for specific examples.

### Simulating Authentication in Integration Tests

In integration test environments, it is often required to simulate authentication credentials. For example, a test client may need to supply credentials to authenticate with the secure service it tests. Fabric3 JUnit test components can be configured with authentication credentials, and those credentials propagated over a remote transport such as Web Services. The following shows how to simulate username/password credentials:

<component name="SecurityTest">

<f3-impl:junit

class="org.fabric3.tests.spring.SecurityTest">

<configuration>

<username>scott</username>

<password>wombat</password>

</configuration>

</f3-impl:junit>

<reference name="service" target="SCASecureService"/>

</component>

## Using Authorization

The Fabric3 API includes the org.fabric3.api.annotation.security.RolesAllowed annotation, which is used to specify roles required to execute a portion of code. The RolesAllowed annotation can be placed on a method or class (in which case it will be applied to all methods contained in the class) to restrict access to security subjects with certain roles as follows:

public class SecureRolesServiceImpl implements SecureService {

**@RolesAllowed({"role1", "role2"})**

public void call() {

// …

}

}

Note that the current security subject can be injected using the SCA @Context annotation on a field or setter method that takes the SCA *RequestContext* type. Alternatively, additional Fabric3 APIs can be accessed by using the *org.fabric3.api.Fabric3RequestContext* type in place of the SCA *RequestContext* type.

### Using Authorization With Spring Beans

When installed, Fabric3 transparently associates the Spring Security provider with application contexts that are configured as SCA components. To enable authorization for Spring beans contained in those application contexts, use the Spring Security elements as in any typical Spring application. The following is an example that enables authorization on all Spring beans. Note that namespace declarations have been omitted from the <beans> element:

<beans xmlns=<http://www.springframework.org/schema/beans> …>

<security:global-method-security

pre-post-annotations="enabled"/>

<sca:service name="SecureTestService"

target="SpringSecureService"/>

<bean name="SpringSecureService"

class="org.fabric3.sample.SomeSpringBean"/>

</beans>

For more information, refer to the Spring Security reference.

# The Web Services Binding

The Web Services binding is based on Metro, the JAX-WS RI. When developing web services, the service contract can be a plan Java interface or decorated with JAX-WS annotations. If a plain Java interface is used, Fabric3 will map it to WSDL using JAX-WS defaults. If default behavior is not desired, standard JAX-WS annotations may be used to customize the service WSDL.

***Note that when creating a web service that uses JAX-WS annotations, it is NOT necessary to import the JAX-WS, JAXB, and javax.soap packages into the contribution as the runtime makes these available to applications automatically.***

### Interoperability with .NET

Fabric3 provides out-of-the-box interoperability with .NET and Windows Communication Foundation (WCF). To assist with interoperability, Fabric3 will automatically generate a WSDL for deployed services. This WSDL will include schemas for message types, service contract information (portType), and all necessary endpoint information (service and bindings). In addition, the WSDL will also contain attached policy if the service is deployed with intents or policy sets. For example, if a service is configured with a security intent, Fabric3 will merge the corresponding policy (based on WS-SecurityPolicy) into the generated WSDL. This merging process is performed according to the rules laid out by the WS-PolicyAttachement and WS-SecurityPolicy specifications. Fortunately, WSDL generation is transparent to end-users so in most cases you will not need to be concerned with the details of how it is performed.

Generated service WSDL can be accessed by appending “?wsdl” to the service URL, as in:

http://www.foo.com/someService?wsdl

The generated WSDL can be used to create client proxies. In most cases, Microsoft web services tooling, including Visual Studio and SvcUtil, will be able to consume a Fabric3 generated WSDL and create a proxy that can be used directly without manual configuration changes. This out-of-the-box interoperability extends to Fabric3 provided security intents and policy sets (detailed below), which have been designed specifically to work with .NET clients.

### Asynchronous Web Services

Fabric3 supports asynchronous web service invocations. To enable a one-way (asynchronous) operation, annotate a method with the SCA @OneWay annotation. Note that the JAX-WS @Oneway annotation is also supported. It is important to note that asynchronous web services are fully interoperable with .NET: Fabric3 services may invoke .NET asynchronous services via a reference and WCF clients may invoke Fabric3 asynchronous services via a .NET “service reference”.

### Standard Web Services Policies

Fabric3 includes support for the following intents when configuring a service with the web services binding. Note, these intents are designed for interoperability with .NET (and other web services-based) clients:

|  |  |
| --- | --- |
| **Intent Name** | **Description** |
| sca:confidentiality.transport | Provides transport (HTTPS) level confidentiality. |
| sca:clientAuthentication.message | Provides message-based authentication with a username/password token over HTTPS. |
| f3-policy:clientAuthentication | Provides message-based authentication over HTTPS. The default qualified intent uses a X.509 token for authentication. |
| f3-policy:clientAuthentication.X509 | The default qualified intent for f3-policy:clientAuthentication. |

The “sca:” prefix denotes the OASIS namespace, <http://docs.oasis-open.org/ns/opencsa/sca/200903> and indicates the intent is part of the standard OASIS SCA intents. The “f3-policy” intent denotes the Fabric3 policy namespace, “urn:fabric3.org:policy”.

At deployment, the intents will be mapped to Fabric3 defined policy sets, which will then be enabled on the deployed service. In addition, the policy sets will be attached to the generated service WSDL. The policy sets are defined using WS-Policy/WS-SecurityPolicy and are based on the Web Service Interoperability Group policy definitions (http://mssoapinterop.org/ilab/).

### Configuring Policy on Services and References

Policy is configured on services or references (or on the containing component) using the SCA “requires” attribute to specify one or more intents (or policy sets). At deployment, Fabric3 will match configured intents to policy sets.

One important consideration when configuring policy is to account for interoperable attachment on services. Fabric3 will attach policy to a service WSDL based on where it is configured in a composite file or Java annotation. Specifically, if the “requires” attribute is specified on a service operation, it will be attached to the WSDL operation subject. However, if the WSDL is attached to the binding, service, or component definition, it will be attached to the WSDL binding subject. .NET (specifically Visual Studio and SvcUtil) follow WS-SecurityPolicy closely and will not recognize security policy attached to the WSDL operation subject. Consequently, for interoperability, it is import security intents and policy sets be configured on the binding, service or component.

#### Username/password Authentication

The following is an example of enabling username/password authentication on a service:

<component name="SecureService">

<implementation.java class="…"/>

<service name="SecureService">

**<binding.ws uri="/authenticationService" requires="sca:clientAuthentication.message"/>**

</service>

</component>

The above example will secure the web services endpoint with username/password authentication. During an invocation, Fabric3 will compare the username and password credentials against users configured through the security framework (see the security chapter for details on configuring users). It is also possible to perform authorization where only users with specific roles gain access to a service. To configure authorization, the @RolesAllowed intent attribute can be used on the SecureService implementation:

public class SecureServiceImpl implements SecureService {

**@RolesAllowed({"role1", "role2"})**

public void call() {

//….

}

}

References are configured to pass username/password information using the <configuration> element:

<binding.ws uri="https://www.foo.com/authenticationService" requires="sca:clientAuthentication.message">

<configuration>

<username>foo</username>

<password>bar</password>

</configuration>

</binding.ws>

#### X.509 Certificate-based Authentication

Using X.509-based authentication is similar to configuring username/password authentication:

<component name="SecureService">

<implementation.java class="…"/>

<service name="SecureService">

**<binding.ws uri="/authenticationService" requires="f3-policy:clientAuthentication.X509"/>**

</service>

</component>

The above assumes a certificate store has been configured for Fabric3. For details on how to do this, see Chapter 5 on security. During an invocation, Fabric3 will match the certificate credential with one in the configured trust store.

References are configured to pass a certificate using the <configuration> element and the certificate alias:

<binding.ws uri="https://www.foo.com/authenticationService" requires="sca:clientAuthentication.message">

<configuration>

<alias>certAlias</alias>

</configuration>

</binding.ws>

If no alias is specified, Fabric3 will attempt to use the default certificate, assuming the key store has only one certificate.

### Configuring Connections

Outgoing reference connections can be configured for timeouts and chunking using the <configuration> element and the following sub-elements:

* connectTimeout – the time in milliseconds to wait when attempting to open a connection.
* requestTimeout - the time in milliseconds to wait for an HTTP response when making an invocation.
* clientStreamingChunkSize the chunking size to use in bytes

The following shows how to configure connection open and request timeouts:

<binding.ws uri="https://www.foo.com/authenticationService" requires="sca:clientAuthentication.message">

<configuration>

<connectTimeout>10000</connectTimeout>

<requestTimeout>10000</requestTimeout>

</configuration>

</binding.ws>

### Debugging Options

To dump HTTP traffic to the console, set the following variable when starting the Fabric3 JVM:

-Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true

To change the web service binding log level, configure the <web.services> element in systemConfig.xml:

<web.services>

<metro logging=”..”/>

</web.services>

# The JMS Binding

XXXXXXXXXXX Put note about thread pool size: <thread.pool size="100"/>

Fabric3 supports both synchronous and asynchronous remote communications using a third-party JMS provider. The JMS profile provides integration with ActiveMQ in the form of an embedded broker extension, although the profile will work with any JMS-compliant messaging middleware.

The following features are supported:

* Non-blocking operations
* Request-reply operations
* Callbacks
* Local and global (XA) transacted one-way messaging with transparent resource enlistment and recovery

## Using the JMS Binding

A JMS provider can be used as the transport for one-way and request-response operations. A minimal one-way configuration is shown below:

<component name="OneWayClient">

<implementation.java class="…"/>

<reference name="service">

<binding.jms>

<destination name="serviceQueue"/>

</binding.jms>

</reference>

</component>

<component name="OneWayService">

<implementation.java class="…"/>

<service>

<binding.jms>

<destination name="serviceQueue"/>

</binding.jms>

</reference>

</component>

The above configuration uses the “serviceQueue” queue to enqueue messages. The JMS queue may be configured externally using the JMS provider and bound to JNDI or setup using Fabric3 server configuration (more on this later).

Configuring request-response operations is also straightforward and involves specifying a separate queue, which will be used to send responses (for specifics on how messages are correlated, see the SCA JMS Binding Specification):

<component name="RequestResponseClient">

<implementation.java class="…"/>

<reference name="service">

<binding.jms>

<destination name="serviceQueue"/>

<response>

<destination name="responseQueue"/>

</response>

</binding.jms>

</reference>

</component>

<component name="RequestResponseService">

<implementation.java class="…"/>

<service>

<binding.jms>

<destination name="serviceQueue"/>

<response>

<destination name="responseQueue"/>

</response>

</binding.jms>

</service>

</component>

While JMS is an asynchronous model, it is important to note that the client component will block on request-response operations until a response is received. In some cases, this is the desired behavior. In other situations, such as long-running interactions, looser coupling is required where the client can continue processing without waiting for a response to be returned. Callbacks can be used to provide responses at some later point in time. Configuring callbacks involves specifying a callback queue:

<component name="CallbackClient">

<implementation.java class="…"/>

<reference name="service">

<binding.jms>

<destination name="serviceQueue"/>

</binding.jms>

<callback>

<binding.jms>

<destination name="callbackQueue"/>

</binding.jms>

</callback>

</reference>

</component>

<component name="CallbackService">

<implementation.java class="…"/>

<service>

<binding.jms>

<destination name="serviceQueue"/>

</binding.jms>

<callback>

<binding.jms>

<destination name="callbackQueue"/>

</binding.jms>

</callback>

</service>

</component>

When the CallbackClient invokes the CallbackService, the call will return immediately. At some later point in time, a reponse will be delivered asynchronously using the “callbackQueue” queue.

In the previous examples, queues where assumed to be externally configured using Fabric3 server settings or the JMS provider. The JMS binding can also be configured to create queues dynamically by using the @create attribute on the destination element and setting it to “ifnotexist” or “always”. Similarly, the JMS connection factory can be configured directly on the binding using the connectionFactory element. See the SCA JMS Binding Specification for more detail on how to do this.

### Wire Formats

The JMS binding supports multiple wire formats including object serialization, JMS message types, and JAXB serialization. If a parameter type is annotated with the JAXB @XmlRootElement annotation, parameters will be sent as XML using a JMS text message. Otherwise, the JMS binding will introspect the parameter types and select the most appropriate message type (e.g. object, bytes, etc).

## Configuring Connection Factories

Often, it is useful to be able to configure a set of JMS connection factories that can be used for multiple services, particularly for connection pooling. This can be done by creating a <jms> entry in the runtime systemConfig.xml. It is important to note that this is currently only supported with the ActiveMQ provider. The following shows how to setup reusable connection factories

<config>

<jms>

<connection.factories>

<connection.factory name="xaFactory"

broker.url="vm://broker" type="xa"/>

</connection.factories>

</jms>

</config>

The type attribute indicates the connection factory type to create: XA, local, or pooled. The connection factories can then be used in binding configurations as illustrated below:

<component name="CallbackClient">

<implementation.java class="…"/>

<reference name="service">

<binding.jms>

<destination name="serviceQueue"/>

</binding.jms>

</reference>

</component>

Default XA and non-XA connection factories can also be setup in systemConfig.xml. If no connection factory is specified, either the XA or non-XA factory will be used depending on whether global or local transacted messaging is required for a particular operation. Below is an example of how to set up default ActiveMQ connection factories:

<config>

<jms>

<connection.factories>

<connection.factory name="xaDefault"

broker.url="vm://broker" type="xa"/>

<connection.factory name="default"

broker.url="vm://broker"/>

</connection.factories>

</jms>

</config>

## Connection Factories and Destinations using JNDI

The JMS binding can be used with external third-party providers by binding connection factories and destinations into the Fabric3 runtime’s JNDI context. The following demonstrates how to access a queue bound to the name “serviceQueue” in JNDI:

<component name="CallbackService">

<implementation.java class="…"/>

<service>

<binding.jms>

<destination name="serviceQueue" create=”never”/>

</binding.jms>

</service>

</component>

## XA Transactions

Fabric3 supports transparent enlistment of JMS sessions in XA transactions as well as local transactions (the default). To enable transactions use the @requires attribute with transactedOneWay.global or transactedOneWay.local on a binding configuration:

<composite …

xmlns:sca=<http://docs.oasis-open.org/ns/opencsa/sca/200903>>

<component name="CallbackService">

<implementation.java class="…"/>

<service>

<binding.jms requires=”sca:transactedOneWay.global”>

<destination name="serviceQueue"/>

</binding.jms>

</service>

</component>

</composite>

## Message Autoscaling

Fabric3 supports autoscaling where the number of message listeners for a service endpoint are dynamically resized based on workload. The following are the autoscaling attributes which can be configured per endpoint as attributes on binding.jms:

* idle.limit
* transaction.timout
* receive.timeout
* max.messages
* recovery.interval
* max.receivers
* min.receivers

## Configuring ActiveMQ

The JMS profile by default uses an embedded ActiveMQ broker per runtime that offers basic configuration. For simple use cases, this will likely be sufficient. For more advanced use cases, seprate broker process may be required.

ActiveMQ network and transport connectors can be setup in systemConfig.xml as follows:

<config>

<jms>

<active.mq>

<networkConnectors>

<networkConnector uri=”multicast://default”/>

</networkConnectors>

<transportConnectors>

<transportConnector name=”openwire”

uri=”tcp://localhost:61616”/>

</transportConnectors >

</active.mq>

</jms>

</config>

## Binding.SCA

When used with ActiveMQ, the JMS binding extension is configured by default to be a provider of binding.sca. This means components can be wired without configuring transports or physical endpoint information – basically as if they were components locally wired. The JMS binding extension will manage queue setup and connections transparently.

Connection factories can be configured for use with binding.sca in systemConfig.xml in the following way:

<config>

<jms>

<connection.factories>

<connection.factory name="xaFactory"

broker.url="vm://broker" type="xa"/>

<connection.factory name="nonXaFactory"

broker.url="vm://broker" type="local"/>

</connection.factories>

<binding.sca xa.factory=”xaFactory”

factory= “nonXaFactory”/>

</jms>

</config>

# The Net Binding

The Net Binding enables HTTP and TCP-based remote communications. The binding is based on the Netty client/server socket framework (<http://www.jboss.org/netty/>). It provides a small-footprint, highly-performant asynchronous remote communications infrastructure. Binding features include:

* Support for the SCA HTTP binding
* Low-level TCP binary communications
* Multiple wire formats including XML, JSON, Hessian and Java Serialization.
* Non-blocking invocations
* Callbacks
* A binding.sca provider

## Using HTTP Communications

The Net binding supports HTTP communications as defined by the SCA HTTP Binding Specification. The following is an example of a request-response operations can be configured as follows:

<component name="RequestResponseClient">

<implementation.java class="…"/>

<reference name="service">

<binding.http

uri=<http://machine1/requestResponseService>/>

</reference>

</component>

<component name="RequestResponseService">

<implementation.java class="…"/>

<service>

<binding.http uri=[/requestResponseService](http://machine1/requestResponseService)/>

</service>

</component>

One-way operations are configured the same way – the binding extension will no from the service operation signature (e.g. the presence of the @OneWay annotation) that the client should not block:

<component name="OneWayClient">

<implementation.java class="…"/>

<reference name="service">

<binding.http

uri=<http://machine2/oneWayService>”/>

</reference>

</component>

<component name="SomeService">

<implementation.java class="…"/>

<service>

<binding.http uri=[/oneWayService](http://machine1/requestResponseService)”/>

</service>

</component>

Oftentimes, a client will not want to block on a response, particularly for long-running interactions. Callbacks can be used to provide responses at some later point in time. Configuring HTTP callbacks involves specifying a callback address:

<component name="OneWayClient">

<implementation.java class="…"/>

<reference name="service">

<binding.http uri=<http://machine2/oneWayService>”/>

<callback>

<binding.http uri=[/callbackService](http://machine1/callbackService)”/>

</callback>

</reference>

</component>

<component name="SomeService">

<implementation.java class="…"/>

<service>

<binding.http uri=[/oneWayService](http://machine1/requestResponseService)”/>

<callback>

<binding.http uri=<http://machine1/callbackService>”/>

</callback>

</reference>

</service>

</component>

When the CallbackClient invokes the CallbackService, the call will return immediately. At some later point in time, a reponse will be delivered asynchronously to <http://machine1/callbackService>.

### Wire Formats

Wire formats are specified using the wireFormat element as in:

<component name="RequestResponseClient">

<implementation.java class="…"/>

<reference name="service">

<binding.http

uri=<http://machine1/requestResponseService>/>

<wireFormat.json/>

</reference>

</component>

The default wire format is JAXB (i.e. parameters will be serialized using JAXB), which is for conformance to the SCA HTTP Binding Specification. The following alternatives are supported:

* JSON – wireFormat.json
* Hessian – wireFormat.hessian
* JDK serialization- wireFormat.jdk

### Binding.SCA

The Net Binding extension includes an HTTP binding.sca provider that is enabled by default. To disable it, set the @httpBindingProvider attribute on the net.binding element to false in systemConfig.xml:

<config>

<binding.net httpBindingProvider=”false”/>

</config>

## Using TCP Communications

The Net Binding also supports asynchronous, binary TCP communications. While the protocol used is currently not interoperable (i.e. it is proprietary to Fabric3 requiring both the client and service provider to be hosted in a Fabric3 runtime), TCP communications are useful when performance is an overriding concern. Request-response and one-way operations as well as callbacks are supported. Configuring TCP communications is the same as HTTP, except binding.tcp is used:

<component name="OneWayClient">

<implementation.java class="…"/>

<reference name="service">

<binding.tcp uri=<http://machine2/oneWayService>”/>

<callback>

<binding.tcp uri=[/callbackService](http://machine1/callbackService)”/>

</callback>

</reference>

</component>

<component name="SomeService">

<implementation.java class="…"/>

<service>

<binding.tcp uri=[/oneWayService](http://machine1/requestResponseService)”/>

<callback>

<binding.tcp uri=<http://machine1/callbackService>”/>

</callback>

</reference>

</service>

</component>

### Wire Formats

The TCP communications support the same wire formats as HTTP, except that data is sent as binary.

### Binding.SCA

The Net Binding extension includes a TCP binding.sca provider that is enabled by default. To disable it, set the @httpBindingProvider attribute on the net.binding element to false in systemConfig.xml:

<config>

<binding.net tcpBindingProvider=”false”/>

</config>

# JAX-RS and RESTFul Services

The RS Binding provides support for exposing SCA services as RESTful resources using the JAX-RS annotations and APIs. The binding supports both JAXB and JSON (via the Jackson JSON parser) representations.

Exposing an SCA service as a RESTFul endpoint is straightforward – simply use JAX-RS annotations and configure the service with <binding.rs> in a composite. As a convenience, JAX-RS annotations can be used directly on a component implementation class without the need for a service interface:

@Path("/")

@Consumes({MediaType.APPLICATION\_XML})

@Produces({MediaType.APPLICATION\_XML})

@Scope("COMPOSITE")

public class MessageService {

@PUT

@Path("/message")

public Response create(Message message) {

// …

}

@GET

@Path("message/{id}")

public Message retrieve(@PathParam("id") Long id) {

// …

}

@DELETE

@Path("message/{id}")

public Response delete(@PathParam("id") Long id) {

// …

}

}

The above component is configured in a composite as follows:

<composite

xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"

xmlns:f3-binding="urn:org.fabric3:binding" …>

<component name="MessageService">

<implementation.java

class="org.fabric3.tests.rs.MessageService"/>

<service name="MessageService">

<f3-binding:binding.rs uri="/messages"/>

</service>

</component>

</composite>

The binding configuration will expose the MessageService resource at <ip address>/messages.

To consume and produce both XML and JSON, the JAX-RS *Consumes and Produces* annotations are used:

@Path("/")

@Consumes({MediaType.APPLICATION\_XML,**MediaType.APPLICATION\_JSON**})

@Produces({MediaType.APPLICATION\_XML,**MediaType.APPLICATION\_JSON**})

@Scope("COMPOSITE")

public class MessageService {

//….

}

## Exposing Spring Beans as RESTful Resources

Spring beans can also be exposed as RESTful resources using JAX-RS annotations on a bean implementation and the <binding.rs> element. In this case, the MessageService implementation remains exactly the same as in the above example. The Spring application context is configured as follows:

<beans

xmlns:sca=<http://docs.oasis-open.org/ns/opencsa/sca/200912>…>

<sca:service name="MessageService" target="MessageService"/>

<bean id="MessageService"

class="org.fabric3.samples.hibernate.MessageService"/>

</beans>

The MessageService is configured in the composite as follows:

<composite …>

<component name="MessageService">

<implementation.spring location="…"/>

**<service name="MessageService">**

**<f3-binding:binding.rs uri="/messages"/>**

**</service>**

</component>

</composite>

***For further examples, the Samples and Spring Samples contain several applications that use the RS binding.***

## Enabling JAX-RS

JAX-RS support is enabled by installing the rs profile in any of the Fabric3 runtimes.

# The FTP Binding

Many enterprise architectures rely on FTP to integrate loosely-coupled systems. Fabric3 provides the ability to receive inbound PUTs and to perform outbound PUTs to external FTP servers via a streaming infrastructure. This enables the transfer of large data sets without having to write files to disk. Inbound PUTs are sent to services exposed over the Fabric3 FTP binding while outbound PUTs are sent using a reference configured with the FTP binding. As will be detailed below, in addition to allowing streaming data transfers, this FTP binding simplifies application code by removing the need to interact directly with an FTP protocol API.

## Implementing an FTP Service

To implement a servce that receives incoming FTP PUTs, define a Java interface with a single method that takes a String file name and InputStream parameters as shown in the following example:

public interface FtpDataTransferService {

void transferData(String fileName, InputStream data)

throws Exception;

}

Next, implement the service using a component as demonstrated here:

public class FtpDataTransferServiceImpl implements FtpDataTransferService {

public void transferData(String fileName, InputStream data)

throws Exception {

// process the stream

}

}

One common use case for receiving FTP files is to place a Fabric3 runtime in a DMZ where it receives inbound puts and forwards the stream via another protocol (e.g. web services) to another server in a trusted zone. This configuration avoids directly exposing servers in a domain to inbound traffic while minimizing performance overhead as the data is streamed from the origin server through the DMZ runtime to the trusted zone. Further, these operation can be configured to be transactional.

Implementing such a fowarding service is straightforward. The previous example can be extended to forward the InputStream to another service via a reference bound to another protocol:

public class FtpDataTransferServiceImpl implements FtpDataTransferService {

@Reference

protected InternalService internalService;

public void transferData(String fileName, InputStream data)

throws Exception {

// forward over a different binding

internalService.write(data);

}

}

### Handling Different Data Types

The FTP binding supports both ASCII and binary data. To determine the type of an incoming stream, Fabric3 provides the F3RequestContext API. The F3RequestContext is a specialized form of the SCA RequestContext API and is accessed through injection using the @Context annotation. The following is an example of how to use the API to determine the type of incoming data:

public class FtpDataTransferServiceImpl implements

FtpDataTransferService {

@Context

protected F3RequestContext context;

public void transferData(String fileName, InputStream data)

throws Exception {

String type =

context.getHeader(String.class, "f3.contentType");

if ("BINARY".equals(type)) {

handleBinary(data);

} else {

handleText(fileName, data);

}

}

}

The header options for f3.contentType are TEXT or BINARY. In addition to the SCA API jar (fabric3-sca), the Fabric3 API jar (fabric3-api) is required to be on the classpath to use the F3RequestContext API. This jar is automatically made available to all deployed contributions.

## Configuring and Provisioning FTP Services

### Provisioning Services

A service is bound over FTP by specifying a relative URI at which it will be made available to clients. The relative URI maps to the directory FTP clients PUT data to. For example, the following maps the FtpDataTransferService to the "transfer" directory:

<component name="FtpService">

<implementation.java

class="...FtpDataTransferServiceImpl"/>

<service name="FtpDataTransferService">

<f3-binding:binding.ftp uri="transfer"/>

</service>

</component>

After logging in at the FTP base address for the domain (configuring the Fabric3 streaming FTP server is described below), clients can send data to the service by changing the working directory to transfer and issuing a PUT command. The data will then be streamed to the component.

A service can also be mapped to the root FTP directory using the '/' token:

<component name="FtpService">

<implementation.java class="...FtpDataTransferServiceImpl"/>

<service name="FtpDataTransferService">

<f3-binding:binding.ftp uri="/"/>

</service>

</component>

If a service is mapped to the root FTP directory, clients do not need to change working directories to send data after connecting to the domain FTP server.h6. Configuring Security

FTP-bound services can be configured to require authentication-based security using a policy definition. Policies are covered in detail in [Chapter 16 - Policies and Intents](http://www.metaform-systems.com/display/F3/Chapter+07+-+Policies+and+Intents). The following policy definition shows how to secure a service using an authentication policy:

<?xml version="1.0" encoding="ASCII"?>

<definitions

xmlns=" http://docs.oasis-open.org/ns/opencsa/sca/200903"

xmlns:sca=" http://docs.oasis-open.org/ns/opencsa/sca/200903"

targetNamespace="urn:foo.com:policy"

xmlns:f3-policy="urn:fabric3.org:policy">

<policySet name="authenticationPolicy"

provides="sca:authentication.message"

appliesTo="@name='ftpDataTransferService'">

<f3-policy:security user="user" password="password"/>

</policySet>

</definitions>

The above policy requires FTP clients to login prior to sending data to the service.

## Connecting to External FTP Servers

The FTP binding can also be used to connect to remote FTP servers. The following demonstrates how to bind a reference to a remote FTP server using authentication and PASSIVE data transfer:

<component name="DataTransferClient">

<f3-impl:junit class="…DataTransferSClient"/>

<reference name="ftpDataTransferService">

<f3-binding:binding.ftp uri="ftp.baz.com:2000"

requires="sca:authentication.message" mode="PASSIVE"/>

</reference>

</component>

The above authentication intent is matched to the policy definition below:

<?xml version="1.0" encoding="ASCII"?>

<definitions

xmlns=" http://docs.oasis-open.org/ns/opencsa/sca/200903"

xmlns:sca=" http://docs.oasis-open.org/ns/opencsa/sca/200903"

targetNamespace="urn:fabric3.org:policy"

xmlns:f3-policy="urn:fabric3.org:policy">

<policySet name="authenticationPolicy"

provides="sca:authentication.message"

appliesTo="../@name='DataTransferClient'">

<f3-policy:security user="meeraj" password="password"/>

</policySet>

</definitions>

#### Setting the Data Type

The F3RequestContext API can be used to set the outgoing data type transfer to ASCII or BINARY via request headers. The header name and values are specified in org.fabric3.api.ftp.FtpConstants (contained in the fabric3-api jar). For example:

public class FtpClientImpl implements FtpClient {

@Context

protected F3RequestContext context;

@Refeence

protected FtpService service;

public void transferData(InputStream data) {

context.setHeader(ftpConstants.HEADER\_CONTENT\_TYPE, FtpConstants.BINARY\_TYPE);

 // to set ASCII use: context.setHeader(ftpConstants.HEADER\_CONTENT\_TYPE, FtpConstants.ASCII\_TYPE);

// send data

service.send(data);

}

}

Note the header must be set on each invocation.

#### Sending proprietary FTP commands before a PUT (STOR) operation

Some FTP servers require custom command sequences prior to a PUT operation. Sequences of commands can be configured on the reference binding using the <commands> element. These sequences will be executed prior to the PUT operation:

<component name="DataTransferClient">

<f3-impl:junit class="org.fabric3.tests.binding.ftp.DataTransferSClient"/>

<reference name="ftpDataTransferService">

<f3-binding:binding.ftp uri="ftp.baz.com:2000" requires="sca:authentication.message" mode="PASSIVE">

<commands>

<command>custom command1</command>

<command>custom command2</command>

</commands>

</f3-binding:binding.ftp>

</reference>

</component>

## Installing the FTP Profile

### Installing the FTP Profile

The FTP profile can be downloaded from the Fabric3 web site. To install it in the Standalone Server, unzip its contents into the respective /host and /extensions directory. When the server is rebooted, the FTP extensions will be activated.

### Setting up the FTP Server

The FTP profile installs an embedded FTP server. This server is used to listen for incoming requests and dispatch them to bound services.

#### FTP Configuration Settings

The FTP server has a number of configuration options:

* **commandPort** - The port to accept FTP commands on. The FTP server is set to listen on port 2000 by default.
* **minPassivePort** and **maxPassivePort**- The lowest and highest port number for passive connections. The default passive port range for PUT operations is set to 6000-7000.
* **listenAddress** - The machine address the server should bind to. Used for multi-homed machines. The default is set to the value returned by InetAddress.getLocalHost().
* **idleTimeout** - The timeout in milliseconds to use for socket connections. The default is one minute.
* **users -** Contains users and password information.

FTP defaults can be changed by editing the runtime system configuration. In the standalone runtime, the system configuration file (systemConfig.xml) is located in the profile subdirectories under /config. In the Maven and Webapp runtimes, it is part of the plugin configuration. The following provides an example of how to modify the default settings:

<ftp.server>

<commandPort>2000</commandPort>

<minPassivePort>6000</minPassivePort>

<maxPassivePort>7000</maxPassivePort>

<users>

<user>password</user>

</users>

</ftp.server>

In addition to the above configuration settings, user accounts (name and password) can be specified as show below:

<component name="FtpServerComponent">

<implementation.composite name="sample:FtpServerExtension"/>

   <property name="config">

    <config xmlns="">

      <commandPort>2000</commandPort>

      <minPassivePort>6000</minPassivePort>

      <maxPassivePort>7000</maxPassivePort>

      <users><user>password</user></users>

      </config>

   </property>

</component>

## Configuring an Integration Test Environment

For an example of using the FTP binding in an integration test environment, see the Fabric3 test-binding-ftp modules located in the source repository under tests/trunk.

## Supported Commands

The following commands are supported by the FTP server:

* USER
* PASS
* PASV
* PASS
* STOR
* QUIT
* SYST
* CWD
* LIST
* TYPE

#### Custom commands

The FTP server may be extended to add custom commands. Custom commands must implement the org.fabric3.ftp.server.protocol.RequestHandler inteface and be configured as a system component in an extension contribution. For examples, see the fabric3-ftp-server module in the source repository.

# Web Components

Fabric3 supports wiring services to Servlets and JSPs in a web application. It does this by treating web applications as components. In this chapter, we cover how to implement a web component and wire it to other services in a domain.

## Implementing Web Components

Web components are essentially web applications (WARs) with additional SCA artifacts that allow Servlets and JSPs to be wired to services in a domain.

The first step in implementing a web component is to create a *component type* file. A component type file defines the references (and properties) for a web component. Component type files are not required for Java-based components as references can be introspected from implementation classes. However, there is no equivalent metadata in a web applicaton so this information must be provided explicitly. The following is an example of a web component type file defining one reference:

<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">

    <reference name="testService">

        <interface.java interface="org.foo.TestService"/>

    </reference>

</componentType>

The component type file must be named web.componentType and placed in the WEB-INF directory of the WAR.

In addition to the component type file, the WAR must contain a web.compoiste file in its WEB-INF directory. This composite configures the web application as a component. An example is provided below:

<composite ...>

    <component name="WebComponent">

        <implementation.web/>

        <reference name="testService" target="TestService"/>

    </component>

</composite>

In the example, the testService reference is wired to the TestService (which could be a service deployed previously in the domain).

It is also possible to configure additional components in the WAR. The following is the same web composite expanded to configure the TestService as a Java-based component:

<composite ...>

    <component name="WebComponent">

        <implementation.web/>

        <reference name="service" target="TestService"/>

    </component>

    <component name="TestService">

        <implementation.java class="org.fabric3.runtime.webapp.smoketest.ContextTest"/>

    </component>

</composite>

## Accessing Services

Services can be accessed from a web application by looking them up in the Servlet context, via injection on Servlets, or through use of the SCA taglib.  Injection on Servlets is only supported in the Standalone Runtime (as opposed to the Web App Runtime, so it will be discussed in separately in the next section).

Due to threading constrains, stateless services are accessed differently than conversational services. We discuss those differences in the next two sections.

### Accessing Stateless Services

Stateless services (and services backed by composite-scoped components) can be looked up by their reference name from the Servlet context. The following Servlet looks up the TestService used in the previous examples:

public class TestServlet extends HttpServlet {

private ServletContext servletContext;

public void init(ServletConfig config)

throws ServletException {

super.init(config);

servletContext = config.getServletContext();

}

protected void doGet(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

TestService service =

(TestService) servletContext.getAttribute(testName);

service.invoke(..);

}

}

### Accessing Conversational Services

Conversational services can be looked up using the Servlet's session context. If the previous TestService were conversational, it would be accessed in the following manner:

public class TestServlet extends HttpServlet {

protected void doGet(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

TestService service = (TestService)

request.getSession().getAttribute("testService");

service.invoke(..);

}

}

 When using conversational services it is important to keep in mind the multi-threaded nature of web applications. Specifically, when accessing a conversational service from a Servlet or JSP, you must take into account the possibility of dispatching multiple simultaneous requests to the service. As a general rule, you should handle access to conversational services in the same way HTTP session-related data is managed.

### The SCA Tag Library

Fabric3 includes support for the OSOA-defined SCA TagLib. This taglib is separately downloaded from the Fabric3 web site and contains support for adding <reference> and <property> tags in JSPs.

## Fabric3 Server Features

There are several web component features only available when using the Fabric3 Server. These are described here.

### Reference injection

When deployed to the Fabric3 Standalone Server, references may be directly injected on Servlets using the SCA @Reference annotation. This avoids having to look up services in the Servlet or session contexts. The servlet in the previous examples could be re-written to take advantage of reference injection:

public class TestServlet extends HttpServlet {

 @Reference

protected TestService testService;

protected void doGet(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

service.invoke(..);

}

}

## Installing the Web Profile

In order to deploy web components to the Standalone Server, it is necessary to install the Web Profile. The profile can be downloaded from the Fabric3 web site. To install the profile, unzip the contents of the distribution and copy the JARs in the /extensions and /host directories to the same directories of the Standalone Server. After the server has booted, the Web Profile extensions will be activated. If you are using the SCA taglib, you will also need to include it in the web component WAR.

To deploy a web component, copy the WAR to the server /deploy directory or install it as a contribution via the command line admin tool.

# Timer Components

Fabric3 provides a custom component type for implementing timers. Timers are useful for application functionality that is triggered at a certain time or interval. For example, a timer may be used to periodically delete records from a database or fire notifications.

## Implementing a Timer Component

Implementing timer components is similar to implementing Java-based components. References, properties, and resources may be injected and used when the timer is fired. However, since timer components are not invoked by clients, they do not implement a service interface. Rather, they implement java.lang.Runnable. When the timer is fired, its run() method will be invoked. The following is an example of a timer implementation that uses a JPA EntityManager to retrieve a list of records and invoke a notification service:

public class TimedComponent implements Runnable {

   @PersistenceContext(name="notifyEM")

   protected EntityManager entityManager;

   @Reference

   protected NotificationService service;

    public void run() {

       // use the entityManager to retrieve records ...

       List<Record> records = (List<Record>) entityManager.createQuery("").getResultList();

        // iterate through the records and fire notifications

       for (Records record : records) {

service.notify;

     }

}

}

Timer components are configured using the <implementation.timer> element, which is in the urn:fabric3.org:implementation namespace:

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"

           xmlns:f3-impl="urn:fabric3.org:implementation"

           xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"

           name="TimerComposite">

<component name="TimerComponent">

   <f3-impl:implementation.timer

class="org.foo.timer.TimedComponent"

repeatInterval="10"/>

      <reference name="service" target="NotificationService"/>

   </component>

</composite>

## Trigger Events

Timer components can be configured to trigger in the following ways.

### Fixed rate

Fixed rate fires a timer at the specified rate in milliseconds. A best effort will be made to maintain the rate over time. For example, the following timer will fire on average every 10 seconds:

<f3-impl:implementation.timer class="org.foo.timer.TimedComponent" fixedRate="10000"/>

### Repeat interval

Repeat interval fires a timer according to the specified interval in milliseconds. For example, the following configures a timer to be triggered in intervals of 10 seconds:

<f3-impl:implementation.timer class="org.foo.timer.TimedComponent" repeatInterval="10000"/>

### Interval Class

A class can be specified which returns when to fire a timer. This can be used to implement a custom trigger algorithm. The trigger class must implement a public method, nextInterval(), which returns a *long* value indicating the delay in milliseconds before the timer is to be fired next. The trigger class is specified using the intervalClass attribute:

<f3-impl:implementation.timer

class="org.foo.timer.TimedComponent"

intervalClass="org.fabric3.samples.TestInterval"/>

### Fire Once

Fire once triggers the timer once at the specified time in milliseconds. This method of configuring a timer is likely to have limited use. The following shows how to configure a timer to fire once:

<f3-impl:implementation.timer class="org.foo.timer.TimedComponent" fireOnce="...."/>

## Configuring Policy

Timer components may be configured with policy. One common policy used with timer components is transactions. If a managed transaction is configured on a timer component, it will be triggered in the context of a transaction. This is useful if the timer must perform transacted work, such as persisting to a database or enqueing messages. Policy is configured in the same was as any other component via the @ManagedTransaction annotation:

@ManagedTransaction

public class TransactionalTimedComponent implements Runnable {

public void run() {

//….

}

}

When the timer run method finishes executing, the transaction will be commited or rolled back.

## Timer Pools

Timers are scheduled against a timer pool. This allows fine-grained workload management by varying the number of available threads to execute a group of timers. If no timer pool is specified, the default pool is used.

Timer pools are configured similar to datasources in a composite. The poolName attribute is used to associate a timer component with a pool:

<composite>

**<f3-other:timer.pool name="TestPool" size="5"/>**

<component name="TimerComponent">

   <f3-impl:implementation.timer

class="org.foo.timer.TimedComponent"

**poolName=”TestPool”**

repeatInterval="10000"/>

   </component>

</composite>

The above example creates a timer pool with 5 threads.

The number of threads in the default pool can be configured in systemConfig.xml using the <timers> element:

<timers default.pool.Size=”5”/>

If not specified, the default pool size is 2.

## Timer Scopes

Timer components can specified as stateless, composite, or domain scoped using the SCA @Scope annotation on the timer component class. A new timer component instance will be created when a trigger is fired for a stateless scoped implementation. In contrast, the same instance will be used when a trigger is fired for a composite scoped implementation. Domain scoped timers are described in the next section.

### Clustered Singletons and Domain Scoped Timers

Some applications require highly available singleton timers. If a timer component is domain scoped, one and only one instance will exist in a cluster (termed a “zone”) at a given time. In other words, a domain scoped timer component is guaranteed to have one instance per cluster (zone) and if the hosting runtime fails, the timer instance will be migrated to another runtime in the cluster (zone). This is different than a composite scoped timer component in that one timer instance per runtime will be created for the latter. Consequently, when deployed to a cluster, a composite scoped timer instance will be active on each runtime.

Implementing domain scoped components is straightforward. The only metadata needed is the SCA @Scope annotation; otherwise, the component configuration is the same. Below is an example of a domain scoped timer component:

@Scope(“DOMAIN”)

public class TimedComponent implements Runnable {

public void run() {

       //…

}

}

Fabric3 will transparently manage deployment and fail-over migration for domain scoped timers. The activation and failover algorithm relies on the underlying Fabric3 clustering service. When a domain scoped component is deployed to a cluster, its singleton instance will be activated on the cluster leader (each cluster as a dynamically elected leader which is determined by the clustering service; see Chapter 17, “Distributed Domains” for details). If the cluster leader fails, the timer will be migrated to the newly-elected leader and activated.

## Using the Scheduled Executor Service

Applications can access the low-level runtime scheduler service to schedule tasks dynamically. The runtime scheduler service implements *java.util.concurrent.ScheduledExecutorService*. A reference to the runtime scheduler is obtained by using Fabric3 resource injection, as shown in the following code fragment:

import org.fabric3.api.annotations.Resource;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.ScheduledExecutorService;

public class SomeComponent implements SomeService{

@Resource

protected ScheduledExecutorService executor;

public void invoke() {

Runnable runnable = //…

executor.schedule(runnable, 10000, TimeUnit.MILLISECONDS);

}

}

***Note dynamically scheduled Runnable instances cannot make service invocations. as the component invocation context is not propagated to the scheduler thread. If a Runable needs to make a service invocation, consider using a timer component instead.***

Dynamically scheduled tasks are executed using the default timer pool. If scheduling against a different timer pool is required, an application will need to use the Fabric3 SPI timer service interface *org.fabric3.timer.spi.TimerService* instead of *java.util.concurrent.ScheduledExecutorService.*

# PubSub Eventing

Wires are used to connect clients directly to a service. Many applications, however, require support for broadcast-style interactions where clients send messages to a destination that consumers can listen on. Often referred to as “PubSub”, Fabric3 supports composing these highly decoupled interactions through SCA eventing. This chapter provides an overview of how to use Fabric3’s eventing facilities.

***Note that as an evolving part of the SCA specifications, eventing is not complete. Therefore, while the Fabric3 implementation is robust, the specifications are subject to change. Depending on how the specifications evolve, it is possible for API and configuration changes to be introduced. While the Fabric3 implementation attempts to steer clear of implementing parts of eventing that may be subject to significant change, users should be aware that eventing is still a work in progress.***

## A Basic Application

Eventing-style interactions involve a component that acts as a source of events, which are dispatched to a channel. Consumer components can be configured to listen on a channel for events. Similar to reference injection, a source component is injected with a *producer* proxy using the Fabric3 *@Producer* annotation. This proxy is responsible for dispatching messages to a *channel*. A component subscribes to a channel using a *consumer* method. The following is an example of a source component with a an injected producer:

import org.fabric3.api.annotation.Producer;

public class BuyComponent implements BuyService {

**@Producer**

private BuyChannel buyChannel;

public void process() {

// …

BuyEvent event = //…

**buyChannel.publish(event);**

}

}

The above example uses the default producer name “buyChannel”. Alternatively, a name could be specified on the @*Producer* annotation. The next excerpt subscribes to receive *BuyEvents:*

import org.fabric3.api.annotation.Consumer;

public class BuyListener {

**@Consumer(“buyChannel”)**

public void onEvent(BuyEvent event) {

// …

}

}

Producers, consumers, and channels are configured in a composite:

<composite …>

<component name="BuyComponent">

<implementation.java …/>

<producer name="buyChannel" target="BuyChannel"/>

</component>

<component name="BuyListener">

<implementation.java …/>

<consumer name="buyChannel" source="BuyChannel"/>

</component>

<channel name="BuyChannel"/>

</composite>

In this example, the producer, consumer, and channel are defined in a single composite. In many applications, these may be defined in different composites. For example, a composite may only contain channel definitions, while others contain definitions for producers and consumers.

As with references, producers and channels may be configured with bindings (currently the only supported binding is JMS). This allows events to be sent across clusters. Bindings are configured in the same way as references.

If no bindings are specified and a producer is defined in a composite that is deployed to a different cluster (zone) than a channel, Fabric3 will automatically create a remote binding between the two. For example, if the *BuyComponent* in the previous example were deployed to a different cluster than the *BuyChannel*, Fabric3 will generate a remote binding to propagate messages from the producer component to the channel.

In addition to SCA Java components, Fabric3 supports injecting producers on Spring beans and timer components. Timer components are particularly useful for modeling events that are periodically generated. Fabric3 also supports configuring Spring beans as consumers.

***For further examples, both the samples and Spring samples contain an eventing application.***

## Enabling Eventing

Eventing is a core runtime capability. Consequently, optional extensions do not need to be installed.

# Application Packaging

In SCA, applications are packaged in one or more contributions. Contributions can be a variety of formats. Fabric3 supports the following formats and can be extended to support others:

* JAR archives
* OSGi bundles
* WAR archives
* ZIP archives
* XML documents

### JAR contributions

Most SCA applications will be packaged as one or more JARs. In addition to including application classes and artifacts, a JAR-based contribution may contain an sca-contribution.xml manifest file in the META-INF directory. This manifest file contains contribution metadata, including a list of deployable composites. Deployable composites are those composites that are contained in the contribution which may be deployed to a domain. A contribution may contain other composites but if they are not marked as deployable, they may not be directly included in the domain (i.e. they may only be used by a deployable composite). An example sca-contribution.xml file is shown below:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"

xmlns:tutorial="urn:org.fabric3:tutorials">

<deployable composite="tutorial:LoanAppComposite"/>

</contribution>

Applications often require third-party libraries. Fabric3 supports two ways of packaging and deploying these libraries: by embedding them in the JAR; and importing them from another contribution. Similar to WARs, Fabric3 allows contribution JARs to bundle third-party libraries by placing their JARs in the META-INF/lib directory of the contribution. Any JAR placed in the META-INF/lib directory will be made available on the contribution classpath.

### The Fabric3 Contribution Plugin

For Maven users, Fabric3 includes a *contribution plugin*\* \*that automates the process of embedding third-party libraries in a contribution. This plugin allows projects to specify a set of Maven modules which will be included in the META-INF/lib directory. The plugin automatically calculates and includes transitive dependencies as well. For more information on using the plugin, see Chapter 21 Testing.

### Contribution Imports and Exports

Embedding artifacts and libraries in the META-INF/lib directory of a contribution is simple but lacks the flexibility required by many applications. For example, several applications may need to share the same WSDL document or library. Fabric3 supports two forms of sharing:

* XML resource sharing
* Java package sharing

In both cases, a resource (or set of resources) are ***exported*** by one contribution and ***imported*** by another. Imports and exports are specified in the contribution manifest file.

### XML Resource Sharing

XML resources are shared by exporting and importing their qualified name (qname). For example, assume a set of portTypes in a WSDL document need to be shared among several contributions. The contribution manifest file contining the WSDL document will export the document's qname using the <export> element:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<export name="urn:somenamespace:1.0"/>

</contribution>

Contributions that require access to the portTypes defined in the **u*rn:somenamespace:1.0*** namespace may import it using the <import> element in their manifest:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<import name="urn:somenamespace:1.0"/>

</contribution>

When the qname is imported, Fabric3 will ensure the portTypes may be referenced by artifacts such as composite files contained in the importing contribution.

### Java Package Sharing

Java resources (i.e. classes) are shared by exporting and importing their packages. Java package sharing in Fabric3 is based on OSGi, so if you are familiar with that technology, you already understand Fabric3's approach and capabilities. Classes contained in Java packages are made available to other contributions using the <export.java> element in the contribution manifest:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<export.java package="com.foo.bar"/>

</contribution>

Exported packages may then be imported using the <import.java> element in the manifest of another contribution:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<import.java package="com.foo.bar"/>

</contribution>

The previous examples make classes in the *com.foo.bar* package available to the importing contribution.

### Fabric3 and OSGi Classloading

Fabric3 loads contributions in separate classloaders using OSGi. This provides contribution isolation (Java classes and artifacts are not visibile to other contributions unless they are exported, thereby reducing the potential for conflicts) and allows versioning. Further, each contribution is associated a classloader space. When a package is imported, a "wire" is created between the importing and exporting contribution. This wire is used by the importing contribution's classloader to load classes belonging to the package using the exporting conribution's classloader. A classloader space therefore consists of the contribution classloader and the classloaders it is wired to via a set of import/export pairs.

Since package imports and exports can specify versions, it is possible to control contribution isolation is a very precise manner. For example, by specifying a version, is is possible for two contributions to use different versions of the same package. For example, versions 1.0 and 2.0 of package *com.bar.foo* can be provided by using the @version attribute of the <export.java> element:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<export.java package="com.foo.bar" version="2.0"/>

</contribution>

An importing contribution can control which version it receives by specifying the @version attribute of the <import.java> element:

<?xml version="1.0" encoding="ASCII"?>

<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0">

<export.java package="com.foo.bar" version="2.0"/>

</contribution>

Often, it is useful to specify a version range instead of an exact version. This can be done using the @min, @minInclusive, @max, and @maxInclusive attributes of the <import.java> element:

 <import.java package="org.foo.bar" min="1.0.0" minInclusive="false" max="2.0.0" maxInclusive="true"/>

By default, @minInclusive and @maxInclusive are true.

It is also possible to specify a '\*' wildcard when exporting and importing packages. For example, the following will export packages *com.foo.bar* and *com.foo.baz*:

 <export.java package="org.foo.\*"/>

### OSGi Bundles

Fabric3 also supports packaging contributions as OSGi bundles. In this case, OSGi bundle manifests may be used to export and import packages from other contributions.

Note in Fabric3 1.5, only Export-Package and Import-Package OSGi manifest headers are supported in a limited fashion. Specifically, only versions and version ranges are supported. The "uses", "required", and attribute directives are not supported.

### WAR Archives

Fabric3 supports packaging contributions as WAR files. This is useful for deploying web applications that are wired to services in a domain.

# Running and Managing the Fabric3 Server

The Fabric3 Server is a modular runtime that can be deployed in single-VM mode or as part of a distributed domain potentially spread across multiple physical machines. This chapter covers basic server operation and management.

## Installation

The Standalone Server requires JRE 6.0 or later. To install the server, download the distribution from [http://www.fabric3.org](http://www.fabric3.org/), unzip its contents in a directory, and execute the following command from the /bin directory:

$ java -jar server.jar

[INFO|main|2010.09.14|10:22:05] HTTP extension installed

…

[INFO|main|2010.09.14|10:22:05] Fabric3 ready [Mode:VM, JMX port:1199]

The "Fabric3 ready" message indicates the server has booted and is ready to receive requests.

## Extensions and Profiles

The Standalone Server can be extended to add support for different remote communications protocols (bindings), programming languages, and enterprise services such as security and transactions. These features are added through extensions and profiles. The latter are groups of related extensions such as the web profile for deploying web applications, or the JPA profile, which includes persistence capabilities based on Hibernate and XA datasources. Since the Standalone Server includes only basic functionality, it is likely you will need to install a set of extensions or profiles.

Profiles and individual extensions can be downloaded from the Fabric3 web site. Individual extension JARs are deployed to the Standalone Server by copying them to the /extensions directory and starting the runtime (note installing extensions in a running server is not yet supported). Extension profiles are installed by extracting their contents to the / extensions directory.

## Deploying an Application

Contributions can be deployed to the Standalone Server in one of two ways. The command line administration tool as described in the next section can be used to deploy the contribution to a local or remote server instance. Alternatively, contributions can be deployed by copying them to the server's /deploy directory. The server periodically scans the directory for new contributions and will deploy them when found. If an application is comprised of multiple contributions that depend on each other, the server will calculate the proper order the contributions must be deployed based on the import metadata contained in their manifests, including transitive dependencies. For example, if A depends on B which dependences on C, the server will deploy C, then B, followed by A.

***Note directory deployment is intended primarily for development use; the admin utility should be used for production deployments as it is less error prone.***

## Runtime Administration

### Command Line Administration

Fabric3 includes a separately downloaded command line administration tool. This tool can be used to deploy contributions to a local or remote runtime and view system status. The command line tool is described later in this chapter.

### Server Shut Down

The Standalone Server can be shut down using the following command located in the /bin directory:

java -jar shutdown.jar

Alternatively, as detailed later in this chapter, a runtime instance can be shutdown using a JMX console.

## Runtime Configuration

### Runtime Layout

The runtime image is organized as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| /bin | | | | Startup modules |
| /lib | | | | Modules required to start the runtime host |
| /boot | | | | Modules required for the runtime bootstrap and primordial system services |
| /host | | | | Libraries shared between the runtime and application (e.g. web services annotations) |
| /extensions | | | | Extension modules that are able to be loaded by all runtime instances |
| /runtimes | | | | Specific runtime instance configuration is hosted by default under this directory |
|  | <runtime-name> | | | top level directory for a runtime configuration |
|  |  | /config | | Contains systemConfig.xml for configuring the runtime and extensions |
|  |  | /deploy | | File system deploy directory for the controller and single-VM runtimes |
|  |  | /repository | |  |
|  |  |  | /runtime | Extensions only loaded for the runtime image |
|  |  |  | /user | User contributions (only populated on the controller and single-VM runtimes) |
|  |  | /data | | Persistent data directory for a runtime instance (e.g. transaction log) |
|  |  | /tmp | | Temporary data and artifact cache for a runtime instance |

For basic startup, the runtime name is required, which maps to a configuration under runtimes:

|  |  |
| --- | --- |
| java -jar server.jar controller | launches a runtime using the runtimes/controller image |
| java -jar server.jar vm | launches a runtime using the runtimes/vm image |
| java -jar server.jar participant | launches a runtime using the runtimes/participant image |
| java -jar server.jar foo | launches a custom runtime using the runtimes/foo image |

### Instantiating Multiple Runtime Instances From a Single Image

It is often desirable to run multiple runtime instances from a single disk image. This is supported in Fabric3 by creating runtime configurations under the /runtimes directory. The top-level configuration directory under /runtimes is the unique runtime name. This name is used to launch the runtime instance. For example, the launch a runtime whose configuration is located at /runtimes/foo, execute:

java –jar server.jar foo

Similarly, a second runtime instance could be launched from the same image by creating another runtime configuration, foo2, under /runtimes/foo and executing:

java –jar server.jar foo2

### Runtime Cloning

For high-density clustered topologies such as cloud environments, Fabric3 provides the ability to clone runtimes. This allows new instances to be spawned from a configuration template using a single command without the need for manual setup. The following command clones a runtime image and launches the cloned instance:

java -jar server.jar clone:template runtime2

### Customizing the Runtime Image

### Base Configuration

#### The Domain Name

If more than one domain is run on the same physical network, it is necessary to create unique domain names. The domain name is configured using the domain attribute of the <runtime> element:

<config>

<runtime domain=”mydomain”/>

</config>

The domain name must be a valid URI host and not contain characters such as “:” or “/”. The default domain name is “domain”/

#### Runtime Mode

The runtime mode is configured using the mode attribute of the <runtime> element:

<config>

<runtime mode=”controller”/>

</config>

Valid values are: *vm*; *participant*; and *controller*. For more information on the runtime mode setting, see Chapter 18, “Distributed Domains.”

#### JMX Port

The JMX port is configured using the jmx.port attribute of the <runtime> element. The port can be a single numeric value or a range, as shown below:

<config>

<runtime jmx.port=”1100-1200”/>

</config>

The default JMX port is 1199.

#### HTTP Port

The server HTTP extension can be configured to listen to a different port other than the default (8181). This is done by editing the @port attribute of the <http> element in systemConfig.xml as in:

<web.server>

    <http port="8181"/>

</web.server>

In addition to a numeric value, a port range can be specified in the form “min-max”.

## Securing the Runtime

### HTTPS Configuration

To enable HTTPS, add the following to the systemConfig.xml file:

<config>

<web.server>

<http port="8181"/>

<https enabled="true" port="8901"/>

</web.server>

<security>

<keystore>fabric3-keystore.jks</keystore>

<truststore>fabric3-truststore.jks</truststore >

<keystore.password>password</keystore.password>

<truststore.password>password</truststore.password>

<cert.password>password</cert.password>

</security>

</config>

Keystore and truststore entries are optional. If not supplied, Fabric3 will look for a fabric3-keystore.jks store in the server /config directory. If a keystore is defined but a truststore is not, Fabric3 will default the truststore to the keystore value. Note the keystore value is an absolute file path.

### Secure Artifact Provisioning

In a distributed domain, Fabric3 uses HTTP or HTTPS to provision artifacts to runtime instances during deployment. By default HTTP is used. To enable HTTPS it is necessary to configure secure provisioning on the controller and participant runtimes (the roles of the controller and participant runtimes are explained in detail in Chapter 18, “Distributed Domains”).

The following systemConfig.xml shows how to configure a controller instance to use HTTPS-based provisioning:

<config>

<!-- … -->

<federation>

<provision secure="true" address="localhost"/>

</federation>

<security>

<keystore.password>password</keystore.password>

<cert.password>password</cert.password>

</security>

<users>

<user>

<username>foo</username>

<password>bar</password>

<roles>

<role>provision.client</role>

</roles>

</user>

</users>

</config>

There are several items to note from the above example:

* Secure provisioning is enabled through the stanza:

<provision secure="true" address="localhost"/>

* The address attribute on the <provision> element is used to set the provision server address. This must match the address encoded in the SSL certificate. Otherwise, if the addess attribute is not specified, the IP address is used by default.
* The keystore is configured using the <security> element.
* A user must be configured with the provision.client role. Participant runtimes will need to authenticate as that user.

Since participant runtimes may provision artifacts to peers, they are configured in a similar way as the controller. The main difference is the username and password attributes on the <provision> element. These are used to authenticate with the controller and must match the values set in the controller configuration:

<config>

<!-- … -->

<federation>

<provision secure="true"

address="localhost"

**username="foo"**

**password="bar"**/>

</federation>

<security>

<keystore.password>password</keystore.password>

<cert.password>password</cert.password>

</security>

<users>

<user>

<username>foo</username>

<password>bar</password>

<roles>

<role>provision.client</role>

</roles>

</user>

</users>

</config>

### Secure Clustered Communications

In the Standalone and Tomcat runtimes, Fabric3 uses JGroups as the cluster service provider. To enable secure cluster communications, it is necessary to configure JGroups appropriately. JGroups XML configuration is specified using the <federation>/<config> element in systemConfig.xml:

<config>

<federation>

<config>

<!--- JGroups XML configuration -->

</config>

</federation>

</config>

### Enabling JMX Authentication and Authorization

JMX authentication is enabled by setting the jmx.security attribute on the <runtime> element and adding users in systemConfig.xml:

<config>

<runtime jmx.security=’authentication’ />

   <users>

       <user>

           <username>foo</username>

           <password>foo</password>

       </user>

   </users>

</config>

If enabled, JMX authentication will require JMX clients to supply a user name and password to gain access to runtime and application MBeans.

JMX security can also be configured for authorization. Authorization is enabled by setting the jmx.security attribute, specifying security roles allowed to access MBeans, and a set of users:

<config>

<runtime jmx.security='authorization'

jmx.access.roles='ROLE\_FABRIC3\_ADMIN,

ROLE\_FABRIC3\_OBSERVER'/>

   <users>

    <user>

           <username>foo</username>

           <password>foo</password>

           <roles>

               <role>ROLE\_FABRIC3\_OBSERVER</role>

           </roles>

       </user>

       <user>

           <username>bar</username>

           <password>bar</password>

           <roles>

               <role>ROLE\_FABRIC3\_ADMIN</role>

           </roles>

       </user>

   </users>

</config>

For details on enabling MBean authorization, see Chapter 19, “Writing Manageable Applications”.

## Monitoring and Logging

Fabric3 uses SCA eventing for monitoring runtime and application events. Runtimes are configured with two channels by default, the *RuntimeMonitorChannel* where runtime events are sent and the *ApplicationMonitorChannel* where applications can send events. Each channel by default is configured with a consumer that listens to events and logs them using LogBack (<http://logback.qos.ch/>). This eventing architecture provides flexibility by allowing custom listeners to be added to the default channels for reacting to particular events. In addition, new channels can be added as destinations for custom events.

Enabling and using monitoring in applications is covered in Chapter , “.” This section covers administrative and operational aspects of the Fabric3 monitor framework.

### Monitor Hierarchy

Applications and runtime extension components can emit events by configuring a specialized event producer called a *monitor proxy*. A monitor proxy is responsible for dispatching an event from a component source to a channel. For extension components, this channel will typically be the *RuntimeMonitorChannel.* Similarly, unless configured differently, application components will emit monitor events to the *ApplicationMonitorChannel.*

Fabric3 provides the ability to dynamically adjust monitoring levels using the *MonitorService* JMX MBean. Valid levels are:

* SEVERE – Errors encountered by the runtime or an application
* WARN – Denote an event that may potentially lead to an error or incorrect operation.
* INFO – Informational messages
* DEBUG – Used for debugging events

Monitoring levels can be adjusted for a component or deployable composite. Since components are hierarchical, setting a parent component will recursively propagate to child components. For example, given a component hierarchy of *domain://foo/bar/baz,* setting the monitor level for *domain://foo/bar* will also set the monitor level for *domain://foo/bar/baz.* To set the monitor level of all application components, use the following URI: *<domain name>://* Similarly, to sett the monitor level for all runtime components, use the URI: *fabric3://*.

### Configuration

Channel logging can be configured with custom LogBack appenders. Custom appenders are added under the <application.monitor> and <runtime.monitor> elements in systemConfig.xml as shown below:

<config>

<application.monitor>

<configuration>

<appender name="CUSTOM"

class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>

[%level %thread %d{YY:MM:DD HH:mm:ss.SSS}] [%logger] %msg%n%ex

</pattern>

</encoder>

</appender>

</configuration>

</application.monitor>

<runtime.monitor>

<configuration>

<appender name="CUSTOM"

class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>

[%level %thread %d{YY:MM:DD HH:mm:ss.SSS}] [%logger] %msg%n%ex

</pattern>

</encoder>

</appender>

</configuration>

</runtime.monitor>

</config>

For a description of available appenders and configuration options, see the LogBack documentation at: <http://logback.qos.ch/documentation.html>.

## Work Management

Fabric3 processes all runtime tasks and requests via a single thread pool. This provides centralized work management and statistics for all runtime operations. The thread pool is a specialized implementation of *java.util.concurrent. AbstractExecutorService.* The following attributes can be configured on the <thread.pool> element in systemConfig.xml:

* coreSize – the number of threads to keep in the pool, even if they are idle.
* size - the maximum number of threads to allow in the pool.
* queueSize – the size of the queue used to receive work requests. If an attempt is made to schedule a task when the queue has reached is maximum size, the task will be rejected. The default is 100,000.
* keepAliveTime - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.
* allowCoreThreadTimeOut - If false (default) core threads stay alive even when idle. If true, core threads use keepAliveTime to time out waiting for work.
* checkStalledThreads – true if the runtime should check for stalled threads. When a stalled thread is encountered, the runtime will emit a warning event with a stack trace to assist in identifying the current executing code.
* stallThreshold – the threshold in milliseconds after which a thread processing a request is considered stalled.
* stallCheckPeriod – the time in milliseconds between checking for stalled threads.

The runtime thread pool can be managed using the *runtime/kernel/RuntimeThreadPoolExecutor* MBean.

**Note it is important to understand the relationship between coreSize, size, and queueSize. As explained in detail in the Javadoc for *java.util.concurrent.******ThreadPoolExecutor,* a thread will not be added to the thread pool until the core size has been reached *and* the work queue has reached the queueSize capacity. This may lead to tasks not being performed as the number of core threads has been reached but the maximum queue size has not. In this case, the runtime will not add additional threads to process work. The correct approach is to set the coreSize to the maximum pool size and allowCoreThreadTimeOut to true. This will allow the thread pool to grow even if the maximum queue size has not been reached and non-utilized threads to be discarded.**

## Stopping Transports

The runtime/component/TransportService MBean can be used to suspend and resume requests at a transport level such as HTTP or HTTPS.

## Extension Configuration

Extensons are configured by adding entries to the systemConfig.xml located in the /config directory. Specific extension configuration options are detailed in the respective extension chapters.

## Pass-By-Value

By default, Fabric3 is configured to use pass-by-reference semantics when invoking remotable services collocated with their clients. This avoids the overhead of parameter copying. In some cases, this may not be desirable, for example, if a service directly modifies parameter data that should not be visible to clients. To avoid this, the runtime can be configured to follow SCA by-value semantics by setting the @enableByValue attribute to true on the SCA setting in serviceConfig.xml:

<sca enableByValue=”true”/>

# Distributed Domains

Fabric3 provides support for distributed domains that span multiple processes.

## Key Concepts

This section explains key concepts necessary to understand when designing, deploying and maintaining distributed applications with Fabric3.

### The Domain

In SCA, a domain is a realm of control and administration. Specifically, a domain is a set of Fabric3 runtimes that act in that host applications and are managed together. A domain may be large, comprising multiple runtime instances on separate physical machines, or small, consisting of a single runtime. In a domain, policy and contributions may be shared.

### Zones

It is often convenient to partition large domains into a set of smaller regions, or zones. For example, an organization may decide to deploy one application to a set of runtimes and another application to a different set of runtimes. If the organization assigned the applications to different domains, they would not be able to share policies or contributions.

Zones partition a domain into smaller managed units.  A zone is a set of one or more runtimes where applications (contibutions) are deployed. A domain always has at least one zone and may contain more.

While a domain may be heterogeneous (i.e. it may be composed of many different runtime types), a zone is homogenous. All runtimes in a zone are the same. Furthermore, all runtimes in a zone host replicas of the same applications.

### The Controller, Participants, and Zone Manager

A distributed domain is composed of runtimes that perform several different roles: the controller, participant, and zone manager.

The controller manages the domain, including contributions in a repository and deployment to zones. A participant runtime hosts application components for the zone it is a member of. Each zone also has a dynamically elected zone manager which is responsible for coordinating with the domain controller for operations such as deployment. A Fabric3 domain is segmented such that the controller only communicates directly to zone managers, which in turn are responsible for managing individual participants.

Fabric3 uses JGroups (http://www.jgroups.org) as its underlying communications technology, although this may be substituted by alternative technologies.

### Deployment Plans

Deployment plans are XML files that specify zones components in a contribution should be deployed to. They may also contain deployment-specific configuraton. Deployment plans can be installed in the domain as part of a contribution or separately. In addition, contributions may be deployed multiple times using different deployment plans.

### Clustering

Clustering is provided through zones. When components in a contribution are deployed to a zone, they are replicated to all runtimes configured as part of the zone.

### Scaling Down

Using this architecture, it is possible to construct domain topologies that are easy to manage and scale. A domain may span multiple runtime types (e.g. different application servers). However, it is also important to note that this architecture scales down. A domain can be confined to one runtime instance, in which case the controller and participant runtime are the same.

## Setting up a Distributed Domain

Setting up a distributed domain requires the Fabric3 standalone distribution. Note that each runtime must be booted from a separate file system directory.

### Installing the Controller

To install the controller, download and unzip the standalone distribution from the Fabric3 site and execute the following command from the /bin directory:

java -jar server.jar controller

The controller will boot and join the domain. The default HTTP port for the Controller is 8180.

### Installing Participants

Download the standalone distribution (or make a copy of the previous download), unzip it, and execute the following command from the /bin directory:

java -jar server.jar participant

When the particpant boots, it will join the same domain as the controller. Since there is only one participant, it will act as a zone manager for the default zone, zone1. Both runtimes should display a console message indicating the participant has joined the domain as a zone manager.

The default port settings for the participant are:

* HTTP - 8181
* JMX - 1198
* Stop daemon – 8084

If more than one participant is run on a physical machine, Fabric3 will select an available port for these services.

### Installing the Administration tool

The controller is administered through a separate admin tool that can connect to a local or remote running instance via JMX. Download and unzip the adminstration tool distribution. The tool can be run from the commandline or using an interactive shell. To start the interactive shell, execute the following from the /bin directory:

java -jar f3.jar

The shell will display the f3> prompt. Type the following command to display a list of installed contributions and their status:

f3> st

Alternatively, operations can can be executed from the command line. For example, the previous command can be issued directly using:

java -jar f3.jar st

# Writing Manageable Applications

One of the main benefits of Fabric3 is it enables developers to write more manageable applications. This chapter covers how to expose components as MBeans as well as accessing runtime event monitoring from application code.

## Publishing Components as MBeans

Composite scoped components (and system extension components) can be exposed as JMX MBeans using the *org.fabric3.api.annotations.management.Management* the *org.fabric3.api.annotations.management.ManagementOperation* annotations. The following is an example that publishes an MBean with an attribute and two management operations:

@Scope("COMPOSITE")

@Management(description = "A calculator component")

public class HelloServiceImpl implements HelloService{

@ManagementOperation(description = "The greeting text")

public void setGreeting(String greeting){

//…

}

@ManagementOperation(description = "The greeting text")

public String getGreeting(){

//…

}

@ManagementOperation(description = "Start auditing")

public void startAudit() {

// …

}

@ManagementOperation(description = "Stop auditing")

public void stopAudit() {

// ..

}

}

It is also possible to set the JMX group on the @Management attribute, which is used to display MBeans in a common hierarchy in a JMX client:

@Scope("COMPOSITE")

@Management(group = “Hello Components”,

description = "A calculator component")

public class HelloServiceImpl implements HelloService{

//…

}

## JMX Security

JMX authorization can be enabled at an MBean or operation level using the *readRoles* and *writeRoles* parameters for @Management and the *rolesAllowed* parameter for @ManagementOperation. By default on system components, readRoles is set to ROLE\_FABRIC3\_ADMIN and ROLE\_FABRIC3\_OBSERVER while writeRoles is set to ROLE\_FABRIC3\_ADMIN. The following example enables custom authorization:

@Scope("COMPOSITE")

@Management(readRoles={“ROLE\_OPERATIONS”, “ROLE\_ADMIN”},

writeRoles={“ROLE\_ADMIN”})

public class HelloServiceImpl implements HelloService{

@ManagementOperation(rolesAllowed={“ROLE\_SUPER\_ADMIN”})

public void setSecretGreeting(String greeting){

//…

}

@ManagementOperation(rolesAllowed={“ROLE\_SUPER\_ADMIN”})

public String getSecretGreeting(){

//…

}

@ManagementOperation

public void startAudit() {

// …

}

@ManagementOperation

public void stopAudit() {

// ..

}

}

## Monitoring

In place of traditional logging, Fabric3 provides facilities for application code to emit events to channels. This provides added flexibility by allowing component consumers to subscribe to receive events in addition to logging operations.

Fabric3 uses LogBack (<http://logback.qos.ch/>) to log events sent to the default application channel (for details on the default application channel, see “Monitoring and Logging” in Chapter 17, “Running and Managing the Fabric3 Server”).

The org.fabric3.annotations.monitor.Monitor annotation is used to inject a monitor proxy, which is responsible for dispatching application events to a channel. The monitor proxy is defined using an interface supplied by the application:

public class CalculatorImpl implements Calculator{

private CalculatorMonitor monitor;

public CalculatorImpl(@Monitor monitor){

this.monitor = monitor;

}

public double add(double n1, double n2) {

monitor.add(n1,n2);

return n1+n2;

}

//…

}

Fabric3 will search for a resource bundle named f3.properties in the classpath by converting the monitor interface package to a directory hierarchy. The properties file can be used to provide internationalized text output in a log message. An example f3.properties file is shown below:

org.fabric3.samples.CalculatorMonitor#add=Added: {0} and {1}

org.fabric3.samples.CalculatorMonitor#add=An error occured:\n {0}

## Segregating Monitor Events

It is also possible to defile custom channels to send events. This allows application events to be segregated. Below is an example of a configuring a channel. Note that the channel is set as synchronous, which means events will be dispatched to listeners synchronously. This ensures log events are recorded in order:

<composite

xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"

xmlns:f3-core="urn:fabric3.org:core"…>

<channel name="CustomMonitorChannel"

requires="f3-core:synchronous"/>

</composite>

To attach a logger to the channel, use implementation.monitor:

<composite …>

<component name="TestChannelMonitor">

<implementation.monitor/>

<consumer name="monitor" source="CustomMonitorChannel"/>

</component>

</composite>

An event can be dispatched to a custom channel by specifying the channel name on the @Monitor annotation:

public class CalculatorImpl implements Calculator{

private CalculatorMonitor monitor;

public CalculatorImpl(@Monitor(“CustomMonitorChannel” monitor){

this.monitor = monitor;

}

public double add(double n1, double n2) {

monitor.add(n1,n2);

return n1+n2;

}

//…

}

# The Tomcat Runtime

Fabric3 includes a distribution that runs within a Tomcat server instance. Once installed, runtime configuration is identical to the Standalone Server. Please refer to the chapter on the Standalone Server for details.

***One important thing to note is that SCA archives must be deployed to the /fabric3/deploy directory, and not the Tomcat /webapps directory. Standard web application archives, i.e. those that do not contain SCA code or features, may be deployed to the /webapps directory.***

## Configuration

### Setting the Default HTTP Connector

By default, Fabric3 uses the Tomcat connector configured for the following ports:

* VM: 8080
* Controller: 8080
* Participant: 8181

If the Tomcat host where Fabric3 is installed is configured to use a different HTTP connector, it is necessary to update the Fabric3 systemConfig.xml. The following demonstrates a participant configuration which uses the HTTP connector associated with port 8888:

<config>

<runtime mode="participant"/>

<tomcat httpPort="8888"/>

</config>

### Setting the Runtime Name

To set the runtime name to start a specific Fabric3 configuration, modify the CATALINA\_OPTS settings in the Catalona startup scripts, adding:

CATALINA\_OPTS="$CATALINA\_OPTS -Dorg.fabric3.name=<name>"

For example, to start the standard controller configuration, do the following:

CATALINA\_OPTS="$CATALINA\_OPTS -Dorg.fabric3.name=controller"

### Enabling and Accessing JMX

Fabric3 integrates directly with the Tomcat JMX provider. To setup JMX on Tomcat, add the following *CATALINA\_OPTS* values to the Catalina startup script (e.g. catalina.sh or catalina.bat):

CATALINA\_OPTS="-Dcom.sun.management.jmxremote"

CATALINA\_OPTS="$CATALINA\_OPTS

-Dcom.sun.management.jmxremote.port=1199"

CATALINA\_OPTS="$CATALINA\_OPTS

-Dcom.sun.management.jmxremote.ssl=false"

CATALINA\_OPTS="$CATALINA\_OPTS

-Dcom.sun.management.jmxremote.authenticate=false"

This will make Fabric3 MBeans avalable at the URL:

service:jmx:rmi:///jndi/rmi://localhost:1199/jmxrmi

To enable the Fabric3 command line tool, add the following domain entry to the settings file :

<domain name="tomcat"

url="service:jmx:rmi:///jndi/rmi://localhost:1199/jmxrmi"/>

# Testing

Automated tests are a very useful way of improving code quality. They provide a context for development, documentation, and an on-going check of code correctness. These benefits give you the confidence to build on and refactor code knowing that it will continue to work as designed. This chapter covers the application testing facilities provided by Fabric3. Some familiarity with Maven, JUnit and the concepts of mock objects are assumed.

## Unit Testing

Java-based components in SCA are in most cases simple POJOs with optional annotations. This means that this guide has little to add to the already well-covered topics of test-driven development and unit testing. It is important to note, however, that even with Fabric3's strong support for integration testing (examined below), it remains easier to find and fix a whole class of bugs by running lightweight tests as opposed to a (slower) test-harness.

## Integration Testing

Fabric 3 integration tests ensure that component implementations and composites provide the expected functionality and interact with other services and runtime resources in the expected way. They are referred to as 'itests'.

While Fabric 3 does provide a lightweight standalone runtime environment, the recommended way to drive automated integration tests is through the specialized Maven plug-in. This handles the creation of an embedded Fabric3 runtime, deployment, and test execution. Fast runtime bootstrap and test execution makes iterative development easier. Since the Maven plug-in provides the same execution environment as the standalone server, component behavior verification can be done with little overhead. Use of the Maven plug-in also allows integration tests to be initiated in exactly the same way on development machines and the build server.

The Maven plug-in is provided by the 'fabric3-itest' mojo which has a single goal 'test'. This goal is bound to the 'integration-test' phase of the Maven life cycle which means that integration tests are run after application packaging and ordinary unit tests. A basic configuration looks like this:

<build>

<defaultGoal>verify</defaultGoal>

<plugins>

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-itest-plugin</artifactId>

<configuration>

<runtimeVersion>RELEASE</runtimeVersion>

</configuration>

<executions>

<execution>

<goals>

<goal>test</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

### Simple Example

The example illustrates the use of Fabric3 ITest plugin for testing a simple SCA service. The service will be injected into a JUnit test case using the SCA Java programming model, and the test case will invoke the injected service in the ITest host environment. The example uses:

* Service interface
* Service implementation
* Test class
* Service composite
* Test composite
* Maven project descriptor

The snippet below shows the structure of the project,

/pom.xml

/src/main/java/HelloWorld.java

/src/main/java/HelloWorldImpl.java

/src/main/resources/helloWorld.composite

/src/test/java/HelloWorldITest.java

/src/test/resources/itest.composite

### Service Interface

The snippet below shows the Java code for the service interface. The service interface has one operation that is available for the service consumers to invoke.

public interface HelloWorld {

String sayHello(String name);

}

### Service Implementation

The service component is implemented as a Java class and implements the interface shown below,

public class HelloWorldImpl implements HelloWorld {

public String sayHello(String name) {

return "Hello, " + name;

}

}

### Service Composite

The service composite uses the Service Component Description Language (SCDL) to describe the service. The composite shown below has one component implemented in Java. The implementation class for the Java component is HelloWorldImpl and the service offered by the component is promoted from the composite, so that the service can be used when the composite itself is used in a higher level composite as a composite component.

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="HelloWorldComposite" targetNamespace="urn:helloWorld">

<service name="helloWorldService" promote="HelloWorldComponent"/>

<component name="HelloWorldComponent">

<implementation.java class="HelloWorldImpl"/>

</component>

</composite>

### Service Test

The test for the service is written as a standard JUnit test. However, Fabric3 Maven iTest runtime allows unit tests to be written in line with the SCA programming model, and enable the tests to be run in an embedded SCA container, as an SCA component. This means the service being tested can be injected into the test case using SCA semantics.

import org.osoa.sca.annotations.Reference;

import junit.framework.TestCase;

public class HelloWorldITest extends TestCase {

@Reference protected HelloWorld helloWorld;

public void testSayHello() {

assertEquals("Hello, Fred", helloWorld.sayHello("Fred"));

}

}

### Integration Composite

The job of a composite file is to wire together components to create more complex components and services which may then be deployed to a domain. The itest.composite file is no exception to this. It allows you to wire together your production contribution and your test classes such that they may be deployed together at which time the test classes can verify that the behaviour of the production contribution is as expected. For this wiring and deployment to take place, your JUnit test classes must themselves become components and this facility is provided by the special Fabric3 component implementation type junit.

Components of this special type, that is to say JUnit test classes, are represented within the itest.composite in the following way:

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"

xmlns:f3-impl="urn:fabric3.org:implementation"

name="HelloWorldTestComposite">

<component name="HelloWorldTest">

<f3-impl:junit class="HelloWorldITest"/>

<reference name="helloWorld" target="HelloWorldComposite"/>

</component>

<component name="HelloWorldComposite">

<implementation.composite name="helloWorldComposite" scdlResource="helloWorld.composite" />

</component>

</composite>

The ITest composite contains two components:

* The test case implemented using the Fabric3 JUnit component. This component has the reference to the original component that is being tested.
* The component being tested, which is a composite component.

Instead of autowiring the reference to the service, the reference on the JUnit component explictly targets the user component being tested. Please note that implementation.composite belongs to the SCA namespace whereas the junit component is a Fabric3 specific feature and belongs to the Fabric3 implementation namespace. Also, itest.composite is a default name, which the Maven ITest host automatically picks, however, you may use an alternative name and specify that in the plugin configuration.

### Maven Project Descriptor

Next, let’s look at the Maven project descriptor (pom.xml) that specifies the Fabric3 ITest plugin, that allows the test to be run in an SCA environment.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>helloworld</groupId>

<artifactId>helloworld</artifactId>

<packaging>jar</packaging>

<version>0.1</version>

<name>Hello World Test</name>

<dependencies>

<dependency>

<groupId>org.codehaus.fabric3.spec</groupId>

<artifactId>sca-api-r1.0</artifactId>

<version>0.2.1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

</dependency>

</dependencies>

<build>

<defaultGoal>verify</defaultGoal>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.5</source>

<target>1.5</target>

<compilerArgument>-g</compilerArgument>

</configuration>

</plugin>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<excludes>

<exclude>\*\*/\*ITest.java</exclude>

</excludes>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-itest-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>test</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

### Project Dependencies

The project has the following dependencies:

* SCA API: This is used for adding the SCA annotations etc
* JUnit: This is for the purpose of writing the test case.

### Plugins

The project uses the following plugins:

* Java Compiler Plugin: This is to specify the source and target versions as 1.5
* Fabric3 ITest Plugin: This plugin provides the embedded SCA container in the Maven build environment and allow the JUnit test case to be run as an SCA component. The snippet above uses the default configuration, we will look at the configuration details later. The plugin is bound to the integration test lifecycle phase and integrates with Surefire.
* Maven Surefire Plugin: This is to exclude any tests with the \*ITest pattern to be excluded from the normal surefire run. Otherwise, you will get a NullPointerException, as with normal Surefire runs the reference wouldn't have been injected in the test case.

### Running the Test

You can run the test by typing in the mvn integration-test. This will generate the following output:

[INFO] [fabric3-itest:test {execution: default}]

[INFO] Starting Embedded Fabric3 Runtime ...

[INFO] Deploying test composite from c:\projects\tutorials\hellowworld\target\test-classes\itest.composite

[INFO] Executing tests...

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running HelloWorldTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] Stopping Fabric3 Runtime ...

[INFO] ----------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] ----------------------------------------------------------

[INFO] Total time: 12 seconds

[INFO] Finished at: Fri Dec 19 22:32:25 GMT 2008

[INFO] Final Memory: 18M/33M

[INFO] ----------------------------------------------------------

## Using Extensions

Fabric3 ITest hosts, like standalone and web app hosts, provide all the SCA functionailty out of teh box. All the host environments are architected in a modular manner, that allow new features to be added to as extensions to the base runtime. These extensions are released as separate artifacts and users are free to write their own extensions. Extensions provided by Fabric3 include:

* Binding Extensions
* Web Services
* JMS
* Network (HTTP and TCP)
* FTP
* Implementation Extensions
* JAX-RS
* Timer Components
* Groovy
* JPA
* JTA Transaction and Datasource pooling

Extensions are enabled in the ITest environment by specifying them as plugin configuration using standard Maven dependency conventions. The snippet below shows how the ITest host can be extended to support JPA using Hibernate, declarative transaction demarcation using JTA and an out-of-container JTA transaction manager using JOTM.

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-itest-plugin</artifactId>

<configuration>

<extensions>

<!-- JPA Hibernate Extensions -->

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-jpa-hibernate</artifactId>

<version>RELEASE</version>

</dependency>

<!-- JPA Core Components -->

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-jpa-spi</artifactId>

<version>RELEASE</version>

</dependency>

<!-- Transaction Manager Using JOTM -->

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-tx-jotm</artifactId>

<version>RELEASE</version>

</dependency>

<!-- Declarative Transaction Policies -->

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-tx</artifactId>

<version>RELEASE</version>

</dependency>

<!-- JSR 250 Resource Support and Datasource Management -->

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-resource</artifactId>

<version>RELEASE</version>

</dependency>

</extensions>

<!-- API classes need to be shared between the application and extension classloaders -->

<shared>

<dependency>

<groupId>javax.persistence</groupId>

<artifactId>persistence-api</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>geronimo-spec</groupId>

<artifactId>geronimo-spec-jta</artifactId>

<version>1.0.1B-rc4</version>

</dependency>

<dependency>

<groupId>geronimo-spec</groupId>

<artifactId>geronimo-spec-j2ee-connector</artifactId>

<version>1.5-rc4</version>

</dependency>

</shared>

</configuration>

</plugin>

Both extensions and shared libraries are defined as Maven dependencies within the plug-in configuration. Fabric3 uses classloader isolation between user contributions and individual extensions. The shared element is used to declare any libraries that are expected to be shared between user and extension code. In the above example, the JPA API used by both the user code as well as the Hibernate extension.

## Working with Easymock

When you write service based applications, services seldom function in isolation. Services may depend on other services for implementing a cohesive piece of functionality. And, of course, dependency from the implementation of one service to another would be through well-defined service contracts. In the previous chapter we saw these dependencies in the SCA world are expressed using references.

When you unit test a service implementation, you may or may not have the implementation of other services it depend on. For example, the implementation of the dependency may come from the same composite, in which case, when you unit test the composite all the dependencies would have been catered for. In other scenarios, implementations of the dependencies may come from extrenal composites, in which case, the references would have been promoted. In such scenarios, you may want to mock those references and verify the behaviour of the composites in terms of the references being called expected number of times.

Fabric3 provides service mocking using the mock implementation using Easymock, which we will cover in this section.

### Adding Mock Functionality to Hello World

In this section we will have a look at how service mocking can be used in the hello world example. Let us say each time the sayHello method is called, the component implementation will log the call using a monitor service, whose service contract is shown below:

public interface HelloWorldMonitor {

void onSayHello(String name);

}

The code for the service implementation class has now changed to incorporate the call to the above service:

import org.osoa.sca.annotations.Reference;

public class HelloWorldImpl implements HelloWorld {

@Reference protected HelloWorldMonitor monitor;

public String sayHello(String name) {

monitor.onSayHello(name);

return "Hello, " + name;

}

}

Let us say the monitor service comes from a different composite, so our service composite promotes the reference as shown below, so that it can be provided in the context in which the composite will be used as a component.

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="HelloWorldComposite" targetNamespace="urn:helloWorld">

<service name="helloWorldService" promote="HelloWorldComponent"/>

<reference name="monitor" promote="HelloWorldComponent/monitor"/>

<component name="HelloWorldComponent">

<implementation.java class="HelloWorldImpl"/>

</component>

</composite>

Now, when we test the above composite the monitor reference will have to be provided. However, in our integration test, rather than using a real implementation of the monitor service, we use a mock implementation using Fabric3 mock support.

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"

xmlns:f3-impl="urn:fabric3.org:implementation"

name="HelloWorldTestComposite"

autowire="true">

<component name="HelloWorldTest">

<f3-impl:junit class="HelloWorldITest"/>

<reference name="helloWorld" target="HelloWorldComposite"/>

<reference name="monitor" target="MockComponent/HelloWorldMonitor"/>

</component>

<component name="HelloWorldComposite">

<implementation.composite name="helloWorldComposite" scdlResource="helloWorld.composite" />

<reference name="monitor" target="MockComponent/HelloWorldMonitor"/>

</component>

<component name="MockComponent">

<f3-impl:implementation.mock>

HelloWorldMonitor

</f3-impl:implementation.mock>

</component>

</composite>

The implementation type, implementation.mock like junit belongs to the Fabric3 namespace and is an implementation provided by Fabric3 to support mocking service references in integration tests. The implementation takes a list of token separated fully-qualified names of interfaces that need to be mocked.

Note: When autowire is switched on you dont need to explictly specify all the references and target them. This would make the composite less verbose.

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"

xmlns:f3-impl="urn:fabric3.org:implementation"

name="HelloWorldTestComposite"

autowire="true">

<component name="HelloWorldTest">

<f3-impl:junit class="HelloWorldITest"/>

</component>

<component name="HelloWorldComposite">

<implementation.composite name="helloWorldComposite" scdlResource="helloWorld.composite" />

</component>

<component name="MockComponent">

<f3-impl:implementation.mock>

HelloWorldMonitor

</f3-impl:implementation.mock>

</component>

</composite>

Now in the test code, we can use the Easymock API to verify the right number of calls are made to the monitor service by the HelloWorldImpl component being tested.

import org.osoa.sca.annotations.Reference;

import junit.framework.TestCase;

import org.easymock.EasyMock;

import org.easymock.IMocksControl;

public class HelloWorldITest extends TestCase {

@Reference protected HelloWorld helloWorld;

@Reference protected IMocksControl control;

@Reference protected HelloWorldMonitor monitor;

public void testSayHello() {

control.reset();

monitor.onSayHello("Fred");

control.replay();

assertEquals("Hello, Fred", helloWorld.sayHello("Fred"));

control.verify();

}

}

Before you can run the test, you need to modify the POM to add dependency on Easymock and also enable the Fabric3 Easymock extension,

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>helloworld</groupId>

<artifactId>helloworld</artifactId>

<packaging>jar</packaging>

<version>0.1</version>

<name>Hello World Test</name>

<dependencies>

<dependency>

<groupId>org.codehaus.fabric3.spec</groupId>

<artifactId>sca-api-r1.0</artifactId>

<version>0.2.1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

</dependency>

<dependency>

<groupId>org.easymock</groupId>

<artifactId>easymock</artifactId>

<version>2.2</version>

</dependency>

</dependencies>

<build>

<defaultGoal>verify</defaultGoal>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.5</source>

<target>1.5</target>

<compilerArgument>-g</compilerArgument>

</configuration>

</plugin>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<excludes>

<exclude>\*\*/\*ITest.java</exclude>

</excludes>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-itest-plugin</artifactId>

<configuration>

<extensions>

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-mock</artifactId>

<version>RELEASE</version>

</dependency>

</extensions>

<shared>

<dependency>

<groupId>org.easymock</groupId>

<artifactId>easymock</artifactId>

<version>2.2</version>

</dependency>

</shared>

</configuration>

<executions>

<execution>

<goals>

<goal>test</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

### Configuring Security Performing Security Tests

JUnit components may be configured with a context for invocations. For example, possible to configure authentication information, which will allow a subject for each test invocation to be set automatically based on that information:

<component name="SecureServiceTest">

<f3-impl:junit class="org.fabric3.policy.security.SecureServiceTest">

<configuration>

<username>foo</username>

<password>bar</password>

</configuration>

</f3-impl:junit>

<reference name="secureRoleService" target="SecureRolesService"/>

</component>

<component name="SecureRolesService">

<implementation.java class="org.fabric3.policy.security.SecureRolesServiceImpl"/>

</component>

The Maven POM is configured with the following:

<systemConfig>

<![CDATA[

<config>

<users>

<user>

<username>foo</username>

<password>bar</password>

<roles>

<role>role1</role>

<role>role2</role>

</roles>

</user>

</users>

</config>

]]>

</systemConfig>

This provides an easy way to test components that require authentication and/or authorization.

# The Webapp Runtime

***Note: The Webapp Runtime is officially deprecated in the Fabric3 1.6. Users are advised to deploy on one of the other Fabric3 supported runtimes.***

Running Fabric3 in a Servlet container.Fabric3 can be run in a Servlet container as part of a WAR. This allows for portable deployment across a variety of servers, including Tomcat, Jetty, and JEE application servers. When deployed using the Fabric3 Web Application (WebApp) runtime, applications can leverage the underlying capabilities of the host server such as clustering and transaction management.

When deployed in this mode, the SCA domain is contained within the WAR the Fabric3 runtime is embedded in. In a single-VM environment, the SCA domain will be hosted in one process. In a clustered enviroment, the SCA domain will be replicated across multiple instances. Multiple WARs with embedded Fabric3 runtimes deployed to the same server or cluster will contain separate SCA domains.

## Installing the WebApp Runtime

The WebApp runtime is embedded along with application code in a WAR. Specifically, the Fabric3 runtime jars are placed in the WEB-INF/lib directory and the runtime is boostrapped via a ServletContextListener.

To facilitate embedding, Fabric3 provides a Maven WebApp plugin that downloads and assembles the necessary runtime JARs. The following is an example Maven POM that configures the plugin:

<groupId>com.foo.ui</groupId>

<artifactId>webapp/artifactId>

<version>1.0</version>

<packaging>war</packaging>

< !-- -- >

<build>

<plugins>

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-webapp-plugin</artifactId>

<version>0.7</version>

<executions>

<execution>

<id>fabric3-war</id>

<goals>

<goal>fabric3-war</goal>

</goals>

</execution>

</executions>

<configuration>

<!-- specify the runtime version -->

<runTimeVersion>0.7</runTimeVersion>

</configuration>

</plugin>

</plugins>

<build>

When the Maven module is built, a WAR will be produced containing the application code and Fabric3 runtime JARs.

## Configuring the Web Application

### Required web.xml Settings

The WebApp runtime requires several settings in the web.xml file. These are used to boostrap the Fabric3 runtime and map web application events to runtime events.

The following shows a web.xml with the necessary listener and Servlet mappings:

<web-app version="2.4"

xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Fabric3 Webapp Example</display-name>

<listener>

<listener-class>org.fabric3.runtime.webapp.Fabric3ContextListener</listener-class>

</listener>

<listener>

<listener-class>org.fabric3.runtime.webapp.Fabric3RequestListener</listener-class>

</listener>

<listener>

<listener-class>org.fabric3.runtime.webapp.Fabric3SessionListener</listener-class>

</listener>

<servlet>

<servlet-name>Fabric3Servlet</servlet-name>

<servlet-class>org.fabric3.runtime.webapp.Fabric3Servlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Fabric3Servlet</servlet-name>

<url-pattern>/services</url-pattern>

</servlet-mapping>

</web-app>

Note that the servlet-mapping for the Fabric3Servlet is used to forward requests to services bound to HTTP-based protocols. For example, a service named SomeService bound using Web Services will map to a URL corresponding to <base server address>/<servlet context>/services/SomeService.

### Default Deployment Settings

The Fabric3 WebApp runtime uses default deployment settings. On startup, if no composite is specified (see below), the runtime looks for a WEB\_INF/web.composite and attempts to deploy it. The name of the composite must be set to "WebAppComposite" as in:

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="WebappComposite">

#### Explicit Deployment Settings

The composite deployed by the WebApp runtime can be configured using the following web.xml settings (currently, it must still be named WEB-INF/web.composite):

<context-param>

<param-name>fabric3.composite</param-name>

<param-value>WebappSmoketestComposite</param-value>

</context-param>

<context-param>

<param-name>fabric3.compositeNamespace</param-name>

<param-value>urn:fabric3.org:sample</param-value>

</context-param>

#### Work Thread Configuration

The number of worker threads can be configured by setting the fabric3.work.scheduler.numWorkers Servlet context parameter. Also, the runtime can be configured to pause worker threads on startup by setting the Servlet contect parameter fabric3.work.scheduler.pauseOnStart to true.

## Using Extensions and Profiles

Additional capabilities (such as support for the Web Services binding and JPA) can be installed in the WebApp runtime by configuring profiles or specific extensions using the WebApp plugin. Profiles are groupings of related extensions. For example, the JPA profile includes the core JPA, Hibernate, transaction manager, and XA datasource extensions. Generally, applications will configure profiles as opposed to individual extensions. The following shows how to configure a profile using Maven depenency syntax as part of the <configuration> element of the WebApp plugin:

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-webapp-plugin</artifactId>

<version>0.7</version>

<executions>

<execution>

<id>fabric3-war</id>

<goals>

<goal>fabric3-war</goal>

</goals>

</execution>

</executions>

<configuration>

<profiles>

<profile>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>profile-jpa</artifactId>

<version>${fabric3.version}</version>

</profile>

</profiles>

</configuration>

</plugin>

Multiple profiles can be installed by including additional profile elements.

***Note when using a binding profile such as web services, it is generally necessary to configure the fabric3-jetty extension for use as an HTTP transport. See below on how to configure a specific extension.***

It is also possible to install individual extensions using the extensions element in the WebApp plugin configuration. The following demonstrates how this is done:

<plugin>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-webapp-plugin</artifactId>

<version>${fabric3.version}</version>

<executions>

<execution>

<id>fabric3-war</id>

<goals>

<goal>fabric3-war</goal>

</goals>

</execution>

</executions>

<configuration>

<extensions>

<dependency>

<groupId>org.codehaus.fabric3</groupId>

<artifactId>fabric3-groovy</artifactId>

<version>${fabric3.version}</</version>

</dependency>

</extensions>

</configuration>

</plugin>

## Classloading

The web app runtime does not support classloader isolation. this is due to a limitation imposed by third-party libraries that require visibility of application classes. Consequently, all application and runtime classes will be loaded by the WAR classloader.

# Extending Fabric3

**Custom intents**

- TBD usage of @Intent and @IntentMetaData (for an example, see @RolesAllowed).

New admin tool run command to run a script:

java -jar f3.jar run test.text

Federation configuration

<federation logging="debug" timeout="20000">

<transport>

<value>

<entry><key>http</key><value>localhost:8181</value></entry>

<entry><key>binding.net.http</key><value>localhost:8282</value></entry>

</value>

</transport>

<runtimeName>runtime1</runtimeName>

</federation>

Enbabling transactional deployment:

on distributed setup, always enabled.

on single-VM, add the following to systemConfig.xml:

<deployment transactional="true"/>

itests no longer support specifying scdl location. Default test composite is targetNamespace="org.codehaus.fabric3", TestComposite

Admin tool setup for WebLogic:

1. Create a domain configuration in settings.xml:

<?xml version="1.0" encoding="UTF-8"?>

<settings>

<domains>

<domain name="wls"

url="service:jmx:iiop://127.0.0.1:7001/jndi/weblogic.management.mbeanservers.runtime"

protocolPackages="weblogic.management.remote"

username="jmarino"

password="jmarino1024"/>

</domains>

</settings>

1. Copy wlclient.jar and wljmxclient.jar from the WebLogic distribution to the /lib directory

to bind Federation (JGroups) and contribution upload MBean to a specific address: add top-level element to systemConfig:

host.bind.address

Note on some multi-homed machines, JGroups may throw an error when IPv4 and IPv6 stacks are enabled. Tow work around this, use the system property when F3 is started:

-Djava.net.preferIPv4Stack=true