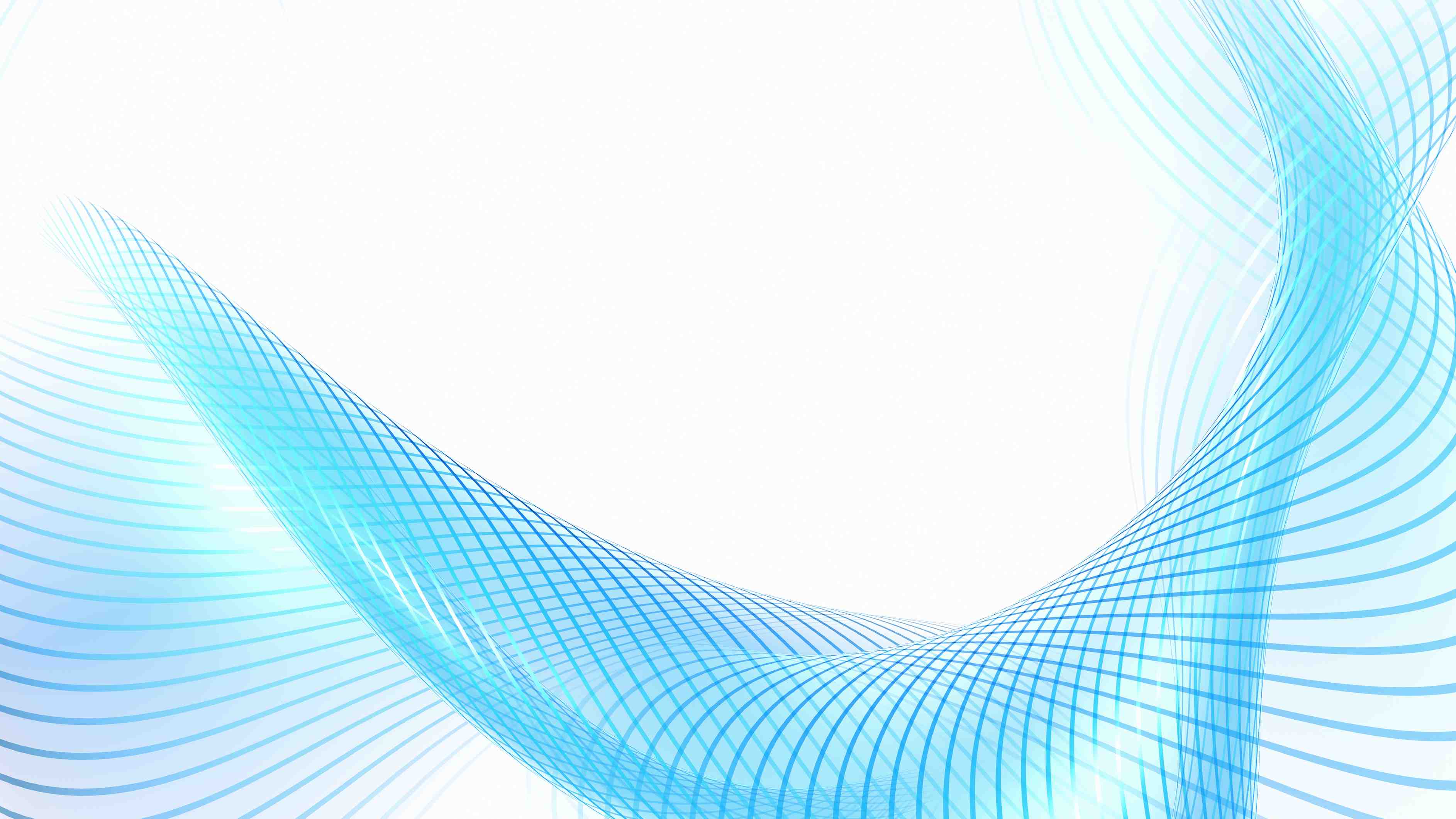
Fabric3 Reference Guide

Version 1.1



© Metaform Systems 2009

Table of Contents

1. Getting Started 5

Choosing a Runtime 5

The Tutorials 6

Prerequisites 6

Building and Deploying WebCalc 6

Building and Deploying BigBank 7

Deploying BigBank to a Distributed Domain 8

Getting Help 9

2. JPA and Hibernate 10

JPA and Hibernate Features 10

Using JPA 10

Injection 10

Transactions 11

Conversations and Extended Persistence Contexts 12

Accessing the Hibernate API 13

Enabling JPA 13

Installing the JPA Profile 13

Configuring Datasources in the Standalone Runtime 14

Enabling JPA in the ITest and Webapp Runtimes 14

3. Transactions and Transactional Resources 15

Declaring Transactional Behavior 15

Transactional Annotations 15

Composite Configuration 16

Runtime Configuration 16

DataSource Configuration 16

JMS Connection Factory Configuration 17

4. Policy 18

Writing Policy Extensions 18

Defining and Applying Policies 18

Including Policies in a Contribution 20

Packaging Policies Separately 20

Creating Custom Intent and PolicySet Annotations 20

The Interceptor Builder SPI 20

5. The Web Services Binding 21

6. The JMS Binding 22

Using the JMS Binding 22

Wire Formats 24

Configuring Connection Factories 24

Connection Factories and Destinations using JNDI 25

XA Transactions 26

Configuring ActiveMQ 26

Binding.SCA 27

7. The Net Binding 28

Using HTTP Communications 28

Wire Formats 29

Binding.SCA 30

Using TCP Communications 30

Wire Formats 31

Binding.SCA 31

8. REST and JAX-RS 32

9. The FTP Binding 33

Implementing an FTP Service 33

Handling Different Data Types 34

Configuring and Provisioning FTP Services 35

Provisioning Services 35

Connecting to External FTP Servers 36

Installing the FTP Profile 38

Installing the FTP Profile 38

Setting up the FTP Server 38

Configuring an Integration Test Environment 39

Supported Commands 39

10. Web Components 41

Implementing Web Components 41

Accessing Services 42

Accessing Stateless Services 42

Accessing Conversational Services 43

The SCA Tag Library 43

Fabric3 Server Features 44

Reference injection 44

Installing the Web Profile 44

11. Timer Components 45

Implementing a Timer Component 45

Configuring Trigger Events 46

Fixed rate 46

Repeat interval 46

Cron expression 46

Fire Once 46

Configuring Policy 47

Clustering 47

12. Running the Fabric3 Server 48

Installation 48

Extensions and Profiles 48

Packaging an Application 49

JAR contributions 49

The Fabric3 Contribution Plugin 49

Contribution Imports and Exports 50

XML Resource Sharing 50

Java Package Sharing 50

Fabric3 and OSGi Classloading 51

OSGi Bundles 52

WAR Archives 52

Deploying an Application 53

Runtime Administration 53

Command Line Administration 53

Server Shut Down 53

Runtime Configuration 53

Logging 53

Base Server Configuration 53

Extension Configuration 54

13. Distributed Domains 55

Key Concepts 55

The Domain 55

Zones 55

The Controller, Participants, and Zone Manager 55

Deployment Plans 56

Clustering 56

Scaling Down 56

Setting up a Distributed Domain 56

Installing the Controller 56

Installing Participants 57

Installing the Administration tool 57

14. The Webapp Runtime 58

Installing the WebApp Runtime 58

Configuring the Web Application 59

Required web.xml Settings 59

Default Deployment Settings 60

Using Features and Extensions 61

Classloading 62

15. Testing 63

Unit Testing 63

Integration Testing 63

Simple Example 64

Service Interface 64

Service Implementation 65

Service Composite 65

Service Test 65

Integration Composite 66

Maven Project Descriptor 67

Project Dependencies 68

Plugins 68

Running the Test 69

Using Extensions 69

Using Features 72

Working with Easymock 72

Adding Mock Functionality to Hello World 73

# 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, JBoss, Websphere 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 Webapp Runtime**: The Fabric3 runtime can be embedded in a Web Application Archive (WAR) and deployed to any JEE application server or Servlet container. In application servers that provide support for clustering, Fabric3 will take advantage of those features to provide high-availability.
* **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 Tutorials

The tutorials are intended to demonstrate the capabilities of Fabric3 using the standalone server. The tutorials contain individual projects designed to showcase specific features:

* **WebCalc**. This tutorial demonstrates how to create a basic SCA application with a web UI. WebCalc is a simple calculator assembled from add, subtract, delete, and multiple services with a servlet-based UI.
* **BigBank Loan Application**. This tutorial showcases advanced features of the Fabric3 runtime. It is intended to provide a complete, real-world application that demonstrates SCA and Fabric3 best-practices.

### Prerequisites

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

* JDK 5.0 or later.
* Maven 2.0.8 or later to build the tutorial projects. Maven can be downloaded from <http://maven.apache.org/download.html>.

### Building and Deploying WebCalc

To build and deploy WebCalc, do the following:

* **Build WebCalc**

In the directory where you extracted the tutorials distribution, go to the webcalc 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

A WAR containing the application will be created in the /target output directory.

* **Build the Fabric3 server distribution.**

Download the Fabric3 standalone runtime by excuting the Maven build script from the tutorials /server directory:

mvn -o clean install

This will create three server images for use with the tutorials in the /target directory: a controller for managing a distributed domain and two runtimes configured to run in different cluster zones (zone1 and zone2).

* **Start the server.**

WebCalc is designed to deploy to a single VM server. To launch a Fabric3 server in single-VM mode, execute the following from the /bin directory of the controller server by executing:

java -jar server.jar

* **Deploy the application**.

After the server has booted, deploy the WebCalc WAR by copying it to the Fabric3 runtime /deploy directory.

The runtime will write a message to the console after the war has been deployed. The calculator UI can be accessed at <http://localhost:8181/calculator/entry.html>.

## 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 four archives: bigbank-api-1.0.jar, bigbank-services-1.0.jar, bigbank-loan-1.0.jar, and fabric3-tutorial-bigbank-webclient-1.0-SNAPSHOT.war.

* **Start the server.**

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

java -jar server.jar

* **Deploy the archives**.

Copy the archives to the server /deploy directory.

The runtime will write a message to the console after the war has been deployed. The BigBank UI can be accessed at <http://localhost:8181/lending/applicationForm.html>. After a loan application is submitted, you can view its status at <http://localhost:8181/lending/status.html>.

## Deploying BigBank to a Distributed Domain

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/bin directory:

java -jar server.jar controller

* From the zone1/bin directory:

java -jar server.jar participant

* From the zone2/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 the deployment archives to the controller/deploy directory. The controller will provision the loan service and UI archives to zone1 and the services archive to zone2.

If run on the same machine, the Fabric3 zone1 runtime will select an available HTTP port to expose the web interface. The HTTP port may vary depending on the start order of the three runtimes. When the runtime boots, it will report the HTTP to the console. The BigBank UI can be accessed at t[http://localhost:<zone1 port>/lending/applicationForm.html](http://localhost:8181/lending/applicationForm.html). After a loan application is submitted, you can view its status at [http://localhost:<zone2 port>/lending/status.html](http://localhost:8181/lending/status.html).

## 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.

# JPA and Hibernate

Fabric3 provides first-class SCA/JPA integration by allowsing JPA persistence units and contexts to be injected into components. These injected instances are managed by the Fabric3 runtime, thereby alleviating the need in most cases for applications to directly manage EntityManager instances 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 JPA, whether they are deployed to a JEE, JSE or other environment.

## JPA and Hibernate Features

JPA 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.
* Support for extended and transaction scope entity managers
* Support for using JPA in JSE and JEE environments
* Integration with SCA transaction policies
* Support for zero code DAOs
* Transaction optimization across persistence units

## Using 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 {

@PersistenceUnit(unitName="employee")

  protected EntityManagerFactory emf;

public void save(Employee employee) {

EntityManager em = emf.createEntityManager();

em.persist(employee);

}

}

and

public class EmployeeDAOImpl implement EmployeeDAO {

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

public void save(Employee employee) {

   m.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">

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<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>

### Transactions

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

import org.fabric3.api.annotation.transaction.ManagedTransaction

@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;

// ...

}

### Accessing the Hibernate API

Some applications require access to Hibernate APIs (e.g. the Criteria API). To access the underlying Hibernate Session instance (Fabric3 uses Hibernate as its JPA provider), inject an EntityManager instance using the @PersistenceContext annotation and use EntityMamager.getDelegate():

public class EmployeeDAOImpl implement EmployeeDAO {

@PersistenceContext(name="employeeEmf" unitName="employee")

protected EntityManager em;

public List<Employee> searchWithCriteria(String name) {

Session session = (Session) employeeEM.getDelegate();

Criteria criteria = session.createCriteria(Employee.class);

criteria.add(Restrictions.eq("name", name));

return (List<Employee>) criteria.list();

}

}

## Enabling JPA

JPA is supported in the Standalone Server, iTest plugin, and Webapp runtime by installing a set of optional extensions. These extensions contain the Fabric3 JPA infrastucture based on Hibernate as well as a JTA transaction manager and XA-compliant datasource.

### 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 /host and /extensions directories. After you restart the server, JPA support will be activated.

### Configuring Datasources in the Standalone Runtime

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.

### Enabling JPA in the ITest and Webapp Runtimes

To use JPA in the ITest and Webapp environments, you must enable the transaction and JPA feature sets. The following Maven POM demonstrates how this is done:

TBD

# Transactions and Transactional Resources

Fabric3 provides consistent transaction management across all supported runtime environments. In the Maven and Server runtimes, Fabric3 uses an embedded JTA transaction manager and transparently handles 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.fabric3.api.annotation.transaction.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>

## Runtime Configuration

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. To use transactions without installing JPA or JMS, include the following extensions in the runtime configuration:

* fabric3-tx
* fabric3-tx-jotm
* fabric3-tx-xapool
* The JTA API jar packaged as an OSGi (we recommend the SpringSource repository distribution)

Generally, you will use transactions in conjunction with JPA or JMS so this manual configuration step is avoided.

### 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.

#### Fabric3 Server

To configure the Fabric3 server, include the appropriate JDBC drivers in the extensions/datasource directory, creating it if needed. Next, you will need to update the serverConfig.xml file for the profile you are running, adding a <datasource> entry. Note, currently only one datasource configuration is supported. The following demonstrates how to configure a connection pool using the MySQL drivers:

<datasource>

<driver>com.mysql.jdbc.Driver</driver>

<url>jdbc:mysql://localhost/bigbank</url>

<user>bigbank</user>

<password>bigbank</password>

<keys>LoanApplicationDS</keys>

</datasource>

The <keys> element is a space separated list of datasource aliases. These aliases may be used to configure subsystems that require datasources, such as JPA/Hibernate (i.e. the persistence.xml).

#### iTest Plugin

TBD

#### Web app

TBD

### 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.

# 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

# The Web Services Binding

# The JMS Binding

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 enlistement

## 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).

**TBD using custom wire formats.**

## 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>

*Note that the JMS binding currently only supports local transacted messaging on references. The Fabric3 1.2 is scheduled to add support for global transacted messaging on references.*

## 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>

#### Disabling the Broker

TBD

## 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>

#### Diabling Binding.SCA

TBD

# 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>

# REST and JAX-RS

# 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, it's 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>

## Configuring 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"/>

### Cron expression

Timer components can be configured to fire according to a cron expression. The following demonstrates using a cron-based trigger:

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

### 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 @requires attribute:

   <component name="TimerComponent">

        <f3-impl:implementation.timer class="org.foo.timer.TimedComponent" requires="sca:managedTransaction" repeatInterval="10"/>

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

    </component>

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

## Clustering

Note clustered timer components are not currently supported.

# Running 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 operation of the server.

## Installation

The Standalone Server requires JRE 5.0 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|2008.09.14|10:22:05] Jetty extension started

\[INFO\|main|2008.09.14|10:22:05] HTTP listener started on port 8181

\[INFO\|main|2008.09.14|10:22:05] Fabric3 ready [Mode:VM, JMX domain:standalone, JMX port:1099, Monitor port: 8083]

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 /deploy 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 /repository directory. Some extension profiles may also contain libraries that are shared between application code and the runtime (e.g. JPA annotations). These libraries are noted in the extension profile and must be copied to the Standalone Server /host directory.

## Packaging an Application

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 15 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 1.1, 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.

## 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.

## 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 as well as make configuration changes (such as thread pools). 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

## Runtime Configuration

### Logging

The server is configured by default to use JDK logging. Log levels and output can be adjusted by editing the monitor.propertes file located in the /config directory.

### Base Server Configuration

The base server configuration can be changed by editing the runtime.properties file located in the /config directory. This section describes the supported configuration properties.

#### Changing the domain name

The domain name may be changed by setting the domain property to a valid URI, for example, domain=fabric3://mydomain.

#### JMX configuration

The JMX domain name and port may be changed by setting the fabric3.jmx and fabric3.jmx.port properties respectively. The default JMX port is 1099.

#### Shutdown listener

The server is configured to listen on a port for shutdown events issued by the shutdown command. The default port is 8083. The port may be changed by setting the fabric3.monitor.port property.

The shutdown listener can also be configured to require a special key. By default, no key is required. A key may be required by setting the fabric3.monitor.key property.

### 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.

#### 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>

# 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 GlassFish Shoal (<https://shoal.dev.java.net/>) 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, zone manager and participant runtimes 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 - 1098
* 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

# The Webapp Runtime

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 Features and Extensions

Additional capabilities (such as support for the Web Services binding and JPA) can be installed in the WebApp runtime by configuring features or specific extensions using the WebApp plugin. Features are groupings of related extensions. For example, the JPA feature includes the core JPA, Hibernate, transaction manager, and XA datasource extensions. Generally, applications will configure features as opposed to individual extensions. The following shows how to configure a feature 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>

<features>

<!-- JPA Hibernate Extensions -->

<dependency>

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

<artifactId>fabric3-hibernate-jotm-feature</artifactId>

<version>0.7/version>

<type>xml</type>

</dependency>

</features>

</configuration>

</plugin>

Multiple features can be installed by including additional dependency elements.

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>0.7</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>0.7</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.

# 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.

## Using Features

If you use a number of extensions in your system, the plugin configuration can get verbose. To cut down this verbosity, Fabric3 provides another mechanism called features, which group extensions and required shared dependencies together. You can use the Fabric3 provided features or build your own features using the feature plugin. The feature plugin is explained in the plugin chapter (TODO link required).

The snippet below shows the example from last section redefined using the Fabric3 Hibernate JOTM feature, which supports JPA using Hibernate, JTA transactions using JOTM and declarative transaction demarcation.

<plugin>

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

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

<configuration>

<features>

<!-- JPA Hibernate Extensions -->

<dependency>

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

<artifactId>fabric3-hibernate-jotm-feature</artifactId>

<type>xml</type>

<version>RELEASE</version>

</dependency>

</features>

</configuration>

</plugin>

## 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>