



# Getting Started Guide

Exported from [JBoss Community Documentation Editor](#) at 2017-06-19 14:17:03 EDT  
Copyright 2017 JBoss Community contributors.



# Table of Contents

---

1	Getting Started with WildFly 10	4
1.1	Download	5
1.2	Requirements	6
1.3	Installation	6
1.4	WildFly - A Quick Tour	6
1.4.1	WildFly 10 Directory Structure	6
1.4.2	WildFly 10 Configurations	9
1.4.3	Starting WildFly 10	9
1.4.4	Starting WildFly 10 with an Alternate Configuration	9
1.4.5	Managing your WildFly 10	11
1.4.6	Modifying the Example DataSource	13
2	JavaEE 6 Tutorial	16
2.1	Standard JavaEE 6 Technologies	16
2.2	JBoss AS7 Extension Technologies	16
2.3	Standard JavaEE 6 Technologies	16
2.3.1	Java API for RESTful Web Services (JAX-RS)	17
2.3.2	Java Servlet Technology	38
2.3.3	Java Server Faces Technology (JSF)	38
2.3.4	Java Persistence API (JPA)	38
2.3.5	Java Transaction API (JTA)	38
2.3.6	Managed Beans	38
2.3.7	Contexts and Dependency Injection (CDI)	38
2.3.8	Bean Validation	38
2.3.9	Java Message Service API (JMS)	38
2.3.10	JavaEE Connector Architecture (JCA)	41
2.3.11	JavaMail API	41
2.3.12	Java Authorization Contract for Containers (JACC)	42
2.3.13	Java Authentication Service Provider Interface for Containers (JASPIC)	43
2.3.14	Enterprise JavaBeans Technology (EJB)	44
2.3.15	Java API for XML Web Services (JAX-WS)	44
2.4	JBoss AS7 Extension Technologies	53
2.4.1	Management Interface	53



- **Getting Started with WildFly 10**
  - [Download](#)
  - [Requirements](#)
  - [Installation](#)
  - [WildFly - A Quick Tour](#)
    - [WildFly 10 Directory Structure](#)
    - [WildFly 10 Configurations](#)
    - [Starting WildFly 10](#)
    - [Starting WildFly 10 with an Alternate Configuration](#)
    - [Managing your WildFly 10](#)
    - [Modifying the Example DataSource](#)



# 1 Getting Started with WildFly 10

WildFly 10 is the latest release in a series of JBoss open-source application server offerings. WildFly 10 is an exceptionally fast, lightweight and powerful implementation of the Java Enterprise Edition 7 Platform specifications. The state-of-the-art architecture built on the Modular Service Container enables services on-demand when your application requires them. The table below lists the Java Enterprise Edition 7 technologies and the technologies available in WildFly 10 server configuration profiles.

Java EE 7 Platform Technology	Java EE 7 Full Profile	Java EE 7 Web Profile	WildFly 10 Full Profile	WildFly 10 Web Profile
JSR-356: Java API for Web Socket	X	X	X	X
JSR-353: Java API for JSON Processing	X	X	X	X
JSR-340: Java Servlet 3.1	X	X	X	X
JSR-344: JavaServer Faces 2.2	X	X	X	X
JSR-341: Expression Language 3.0	X	X	X	X
JSR-245: JavaServer Pages 2.3	X	X	X	X
JSR-52: Standard Tag Library for JavaServer Pages (JSTL) 1.2	X	X	X	X
JSR-352: Batch Applications for the Java Platform 1.0	X	--	X	--
JSR-236: Concurrency Utilities for Java EE 1.0	X	X	X	X
JSR-346: Contexts and Dependency Injection for Java 1.1	X	X	X	X
JSR-330: Dependency Injection for Java 1.0	X	X	X	X
JSR-349: Bean Validation 1.1	X	X	X	X
JSR-345: Enterprise JavaBeans 3.2	X CMP 2.0 Optional	X (Lite)	X CMP 2.0 Not Available	X (Lite)
JSR-318: Interceptors 1.2	X	X	X	X
JSR-322: Java EE Connector Architecture 1.7	X	--	X	X
JSR-338: Java Persistence 2.1	X	X	X	X
JSR-250: Common Annotations for the Java Platform 1.2	X	X	X	X
JSR-343: Java Message Service API 2.0	X	--	X	--



JSR-907: Java Transaction API 1.2	X	X	X	X
JSR-919: JavaMail 1.5	X	--	X	X
JSR-339: Java API for RESTful Web Services 2.0	X	X	X	X
JSR-109: Implementing Enterprise Web Services 1.3	X	--	X	--
JSR-224: Java API for XML-Based Web Services 2.2	X	X	X	X
JSR-181: Web Services Metadata for the Java Platform	X	--	X	--
JSR-101: Java API for XML-Based RPC 1.1	Optional	--	--	--
JSR-67: Java APIs for XML Messaging 1.3	X	--	X	--
JSR-93: Java API for XML Registries	Optional	--	--	--
JSR-196: Java Authentication Service Provider Interface for Containers 1.1	X	--	X	--
JSR-115: Java Authorization Contract for Containers 1.5	X	--	X	--
JSR-88: Java EE Application Deployment 1.2	Optional	--	--	--
JSR-77: J2EE Management 1.1	X		X	
JSR-45: Debugging Support for Other Languages 1.0	X	X	X	X

### Missing HornetQ and JMS?

The WildFly Web Profile doesn't include JMS (provided by HornetQ) by default. If you want to use messaging, make sure you start the server using the "Full Profile" configuration.

This document provides a quick overview on how to download and get started using WildFly 10 for your application development. For in-depth content on administrative features, refer to the WildFly 10 Admin Guide.

## 1.1 Download

WildFly 10 distributions can be obtained from:

[wildfly.org/downloads](http://wildfly.org/downloads)

WildFly 10 provides a single distribution available in zip or tar file formats.

- **wildfly-10.0.0.Final.zip**
- **wildfly-10.0.0.Final.tar.gz**



## 1.2 Requirements

---

- Java SE 8 or later (we recommend that you use the latest update available)

## 1.3 Installation

---

Simply extract your chosen download to the directory of your choice. You can install WildFly 10 on any operating system that supports the zip or tar formats. Refer to the Release Notes for additional information related to the release.

## 1.4 WildFly - A Quick Tour

---

Now that you've downloaded WildFly 10, the next thing to discuss is the layout of the distribution and explore the server directory structure, key configuration files, log files, user deployments and so on. It's worth familiarizing yourself with the layout so that you'll be able to find your way around when it comes to deploying your own applications.

### 1.4.1 WildFly 10 Directory Structure

DIRECTORY	DESCRIPTION
appclient	Configuration files, deployment content, and writable areas used by the application client container run from this installation.
bin	Start up scripts, start up configuration files and various command line utilities like Vault, add-user and Java diagnostic report available for Unix and Windows environments
bin/client	Contains a client jar for use by non-maven based clients.
docs/schema	XML schema definition files
docs/examples/configs	Example configuration files representing specific use cases
domain	Configuration files, deployment content, and writable areas used by the domain mode processes run from this installation.
modules	WildFly 10 is based on a modular classloading architecture. The various modules used in the server are stored here.
standalone	Configuration files, deployment content, and writable areas used by the single standalone server run from this installation.
welcome-content	Default Welcome Page content



## Standalone Directory Structure

In "**standalone**" mode each WildFly 10 server instance is an independent process (similar to previous JBoss AS versions; e.g., 3, 4, 5, or 6). The configuration files, deployment content and writable areas used by the single standalone server run from a WildFly installation are found in the following subdirectories under the top level "standalone" directory:

DIRECTORY	DESCRIPTION
configuration	Configuration files for the standalone server that runs off of this installation. All configuration information for the running server is located here and is the single place for configuration modifications for the standalone server.
data	Persistent information written by the server to survive a restart of the server
deployments	End user deployment content can be placed in this directory for automatic detection and deployment of that content into the server's runtime. NOTE: The server's management API is recommended for installing deployment content. File system based deployment scanning capabilities remain for developer convenience.
lib/ext	Location for installed library jars referenced by applications using the Extension-List mechanism
log	standalone server log files
tmp	location for temporary files written by the server
tmp/auth	Special location used to exchange authentication tokens with local clients so they can confirm that they are local to the running AS process.



## Domain Directory Structure

A key feature of WildFly 10 is the managing multiple servers from a single control point. A collection of multiple servers are referred to as a "**domain**". Domains can span multiple physical (or virtual) machines with all WildFly instances on a given host under the control of a Host Controller process. The Host Controllers interact with the Domain Controller to control the lifecycle of the WildFly instances running on that host and to assist the Domain Controller in managing them. The configuration files, deployment content and writeable areas used by domain mode processes run from a WildFly installation are found in the following subdirectories under the top level "domain" directory:

DIRECTORY	DESCRIPTION
configuration	Configuration files for the domain and for the Host Controller and any servers running off of this installation. All configuration information for the servers managed within the domain is located here and is the single place for configuration information.
content	an internal working area for the Host Controller that controls this installation. This is where it internally stores deployment content. This directory is not meant to be manipulated by end users. Note that " <i>domain</i> " mode does not support deploying content based on scanning a file system.
lib/ext	Location for installed library jars referenced by applications using the Extension-List mechanism
log	Location where the Host Controller process writes its logs. The Process Controller, a small lightweight process that actually spawns the other Host Controller process and any Application Server processes also writes a log here.
servers	Writable area used by each Application Server instance that runs from this installation. Each Application Server instance will have its own subdirectory, created when the server is first started. In each server's subdirectory there will be the following subdirectories: data -- information written by the server that needs to survive a restart of the server log -- the server's log files tmp -- location for temporary files written by the server
tmp	location for temporary files written by the server
tmp/auth	Special location used to exchange authentication tokens with local clients so they can confirm that they are local to the running AS process.





## 1.4.2 WildFly 10 Configurations

### Standalone Server Configurations

- `standalone.xml` (*default*)
  - Java Enterprise Edition 7 web profile certified configuration with the required technologies plus those noted in the table above.
- `standalone-ha.xml`
  - Java Enterprise Edition 7 web profile certified configuration with high availability
- `standalone-full.xml`
  - Java Enterprise Edition 7 full profile certified configuration including all the required EE 7 technologies
- `standalone-full-ha.xml`
  - Java Enterprise Edition 7 full profile certified configuration with high availability

### Domain Server Configurations

- `domain.xml`
  - Java Enterprise Edition 7 full and web profiles available with or without high availability

Important to note is that the **domain** and **standalone** modes determine how the servers are managed not what capabilities they provide.

## 1.4.3 Starting WildFly 10

To start WildFly 10 using the default web profile configuration in "*standalone*" mode, change directory to `$JBOSS_HOME/bin`.

```
./standalone.sh
```

To start the default web profile configuration using domain management capabilities,

```
./domain.sh
```

## 1.4.4 Starting WildFly 10 with an Alternate Configuration

If you choose to start your server with one of the other provided configurations, they can be accessed by passing the `--server-config` argument with the server-config file to be used.



To use the full profile with clustering capabilities, use the following syntax from `$JBOSS_HOME/bin`:

```
./standalone.sh --server-config=standalone-full-ha.xml
```

Similarly to start an alternate configuration in *domain* mode:

```
./domain.sh --domain-config=my-domain-configuration.xml
```

Alternatively, you can create your own selecting the additional subsystems you want to add, remove, or modify.

## Test Your Installation

After executing one of the above commands, you should see output similar to what's shown below.

```
=====

JBoss Bootstrap Environment

JBOSS_HOME: /opt/wildfly-10.0.0.Final

JAVA: java

JAVA_OPTS:  -server -Xms64m -Xmx512m -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m
-Djava.net.preferIPv4Stack=true -Djboss.modules.system.pkgs=com.yourkit,org.jboss.byteman
-Djava.awt.headless=true

=====

11:46:11,161 INFO  [org.jboss.modules] (main) JBoss Modules version 1.5.1.Final
11:46:11,331 INFO  [org.jboss.msc] (main) JBoss MSC version 1.2.6.Final
11:46:11,391 INFO  [org.jboss.as] (MSC service thread 1-6) WFLYSRV0049: WildFly Full
10.0.0.Final (WildFly Core 2.0.10.Final) starting
<snip>
11:46:14,300 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Full
10.0.0.Final (WildFly Core 2.0.10.Final) started in 1909ms - Started 267 of 553 services (371
services are lazy, passive or on-demand)
```

As with previous WildFly releases, you can point your browser to <http://localhost:8080> (if using the default configured http port) which brings you to the Welcome Screen:



## Welcome to WildFly 10

**Your WildFly 10 is running.**

[Documentation](#) | [Quickstarts](#) | [Administration Console](#)

[WildFly Project](#) | [User Forum](#) | [Report an issue](#)

**JBoss Community**

To replace this page simply deploy your own war with / as its context path.  
To disable it, remove the "welcome-content" handler for location / in the undertow subsystem.

From here you can access links to the WildFly community documentation set, stay up-to-date on the latest project information, have a discussion in the user forum and access the enhanced web-based Administration Console. Or, if you uncover a defect while using WildFly, report an issue to inform us (attached patches will be reviewed). This landing page is recommended for convenient access to information about WildFly 10 but can easily be replaced with your own if desired.

### 1.4.5 Managing your WildFly 10

WildFly 10 offers two administrative mechanisms for managing your running instance:

- web-based Administration Console
- command-line interface



## Authentication

By default WildFly 10 is now distributed with security enabled for the management interfaces, this means that before you connect using the administration console or remotely using the CLI you will need to add a new user, this can be achieved simply by using the `add-user.sh` script in the bin folder.

After starting the script you will be guided through the process to add a new user: -

```
./add-user.sh
What type of user do you wish to add?
  a) Management User (mgmt-users.properties)
  b) Application User (application-users.properties)
(a):
```

In this case a new user is being added for the purpose of managing the servers so select option a.

You will then be prompted to enter the details of the new user being added: -

```
Enter the details of the new user to add.
Realm (ManagementRealm) :
Username :
Password :
Re-enter Password :
```

It is important to leave the name of the realm as 'ManagementRealm' as this needs to match the name used in the server's configuration, for the remaining fields enter the new username, password and password confirmation.

Provided there are no errors in the values entered you will then be asked to confirm that you want to add the user, the user will be written to the properties files used for authentication and a confirmation message will be displayed.

The modified time of the properties files are inspected at the time of authentication and the files reloaded if they have changed, for this reason you do not need to re-start the server after adding a new user.



## Administration Console

To access the web-based Administration Console, simply follow the link from the Welcome Screen. To directly access the Management Console, point your browser at:

<http://localhost:9990/console>

NOTE: port 9990 is the default port configured.

```
<management-interfaces>
  <native-interface security-realm="ManagementRealm">
    <socket-binding native="management-native"/>
  </native-interface>
  <http-interface security-realm="ManagementRealm">
    <socket-binding http="management-http"/>
  </http-interface>
</management-interfaces>
```

If you modify the *management-http* socket binding in your running configuration: adjust the above command accordingly. If such modifications are made, then the link from the Welcome Screen will also be inaccessible.

If you have not yet added at least one management user an error page will be displayed asking you to add a new user, after a user has been added you can click on the 'Try Again' link at the bottom of the error page to try connecting to the administration console again.

## Command-Line Interface

If you prefer to manage your server from the command line (or batching), the *jboss-cli.sh* script provides the same capabilities available via the web-based UI. This script is accessed from `$JBOSS_HOME/bin` directory; e.g.,

```
$JBOSS_HOME/bin/jboss-cli.sh --connect
Connected to standalone controller at localhost:9990
```

Notice if no host or port information provided, it will default to localhost:9990.

When running locally to the WildFly process the CLI will silently authenticate against the server by exchanging tokens on the file system, the purpose of this exchange is to verify that the client does have access to the local file system. If the CLI is connecting to a remote WildFly installation then you will be prompted to enter the username and password of a user already added to the realm.

Once connected you can add, modify, remove resources and deploy or undeploy applications. For a complete list of commands and command syntax, type **help** once connected.



## 1.4.6 Modifying the Example DataSource

As with previous JBoss application server releases, a default data source, *ExampleDS*, is configured using the embedded H2 database for developer convenience. There are two ways to define datasource configurations:

1. as a module
2. as a deployment

In the provided configurations, H2 is configured as a module. The module is located in the `$JBOSS_HOME/modules/com/h2database/h2` directory. The H2 datasource configuration is shown below.

```
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
      <driver>h2</driver>
      <pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>20</max-pool-size>
        <prefill>true</prefill>
      </pool>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
    <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS" pool-name="ExampleXADS">
      <driver>h2</driver>
      <xa-datasource-property name="URL">jdbc:h2:mem:test</xa-datasource-property>
      <xa-pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>20</max-pool-size>
        <prefill>true</prefill>
      </xa-pool>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </xa-datasource>
  </datasources>
  <drivers>
    <driver name="h2" module="com.h2database.h2">
      <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>
  </drivers>
</subsystem>
```

The datasource subsystem is provided by the [IronJacamar](#) project. For a detailed description of the available configuration properties, please consult the project documentation.



- IronJacamar homepage: <http://www.jboss.org/ironjacamar>
- Project Documentation: <http://www.jboss.org/ironjacamar/docs>
- Schema description:  
[http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingds\\_descriptor](http://docs.jboss.org/ironjacamar/userguide/1.0/en-US/html/deployment.html#deployingds_descriptor)

## Configure Logging in WildFly 10

WildFly 10 logging can be configured with the web console or the command line interface. You can get more detail on the [Logging Configuration](#) page.

**Turn on debugging for a specific category with CLI:**

```
/subsystem=logging/logger=org.jboss.as:add(level=DEBUG)
```

By default the `server.log` is configured to include all levels in it's log output. In the above example we changed the console to also display debug messages.



## 2 JavaEE 6 Tutorial

---

 **Coming Soon**

This guide is still under development, check back soon!

### 2.1 Standard JavaEE 6 Technologies

---

1. Enterprise JavaBeans Technology (EJB)
2. Java Servlet Technology
3. Java Server Faces Technology (JSF)
4. Java Persistence API (JPA)
5. Java Transaction API (JTA)
6. [Java API for RESTful Web Services \(JAX-RS\)](#)
7. Java API for XML Web Services (JAX-WS)
8. Managed Beans
9. Contexts and Dependency Injection (CDI)
10. Bean Validation
11. Java Message Service API (JMS)
12. JavaEE Connector Architecture (JCA)
13. JavaMail API
14. Java Authorization Contract for Containers (JACC)
15. Java Authentication Service Provider Interface for Containers (JASPIC)

### 2.2 JBoss AS7 Extension Technologies

---

1. OSGi Technology
2. Management Interface

### 2.3 Standard JavaEE 6 Technologies

---

 **Coming Soon**

This guide is still under development, check back soon!





1. Enterprise JavaBeans Technology (EJB)
2. Java Servlet Technology
3. Java Server Faces Technology (JSF)
4. Java Persistence API (JPA)
5. Java Transaction API (JTA)
6. [Java API for RESTful Web Services \(JAX-RS\)](#)
7. [Java API for XML Web Services \(JAX-WS\)](#)
8. Managed Beans
9. Contexts and Dependency Injection (CDI)
10. Bean Validation
11. Java Message Service API (JMS)
12. JavaEE Connector Architecture (JCA)
13. JavaMail API
14. Java Authorization Contract for Containers (JACC)
15. Java Authentication Service Provider Interface for Containers (JASPIC)

## 2.3.1 Java API for RESTful Web Services (JAX-RS)

### Content

- [Tutorial Overview](#)
- [What are RESTful Web Services?](#)
- [Creating a RESTful endpoint](#)
- [Package and build the endpoint](#)
- [Deploy the endpoint to OpenShift](#)
- [Building the mobile client](#)
- [Exploring the mobile client](#)



## Tutorial Overview

This chapter describes the Java API for RESTful web services (JAX-RS, defined in [JSR331](#)). [RESTEasy](#) is an portable implementation of this specification which can run in any Servlet container. Tight integration with JBoss Application Server is available for optimal user experience in that environment. While JAX-RS is only a server-side specification, RESTEasy has innovated to bring JAX-RS to the client through the RESTEasy JAX-RS Client Framework.

Detailed documentation on RESTEasy is available [here](#).

The source for this tutorial is in github repository [git://github.com/tdiesler/javaee-tutorial.git](https://github.com/tdiesler/javaee-tutorial.git)

[OpenShift](#), is a portfolio of portable cloud services for deploying and managing applications in the cloud. This tutorial shows how to deploy a RESTful web service on the free OpenShift Express JavaEE cartridge that runs [JBossAS 7](#).

An application running on [Android](#) shows how to leverage JBoss technology on mobile devices. Specifically, we show how use the RESTEasy client API from an Android device to integrate with a RESTful service running on a JBossAS 7 instance in the cloud.

The following topics are addressed

- What are RESTful web services
- Creating a RESTful server endpoint
- Deploying a RESTful endpoint to a JBossAS instance in the cloud
- RESTEasy client running on an Android mobile device



## What are RESTful Web Services?

### Coming Soon

This section is still under development.

RESTful web services are designed to expose APIs on the web. REST stands for **R**epresentational **S**tate **T**ransfer. It aims to provide better performance, scalability, and flexibility than traditional web services, by allowing clients to access data and resources using predictable URLs. Many well-known public web services expose RESTful APIs.

The Java 6 Enterprise Edition specification for RESTful services is JAX-RS. It is covered by JSR-311 (<http://jcp.org/jsr/detail/311.jsp>). In the REST model, the server exposes APIs through specific URIs (typically URLs), and clients access those URIs to query or modify data. REST uses a stateless communication protocol. Typically, this is HTTP.

The following is a summary of RESTful design principles:

- A URL is tied to a resource using the `@Path` annotation. Clients access the resource using the URL.
- Create, Read, Update, and Delete (CRUD) operations are accessed via `PUT`, `GET`, `POST`, and `DELETE` requests in the HTTP protocol.
  - `PUT` creates a new resource.
  - `DELETE` deletes a resource.
  - `GET` retrieves the current state of a resource.
  - `POST` updates a resource's state.
- Resources are decoupled from their representation, so that clients can request the data in a variety of different formats.
- Stateful interactions require explicit state transfer, in the form of URL rewriting, cookies, and hidden form fields. State can also be embedded in response messages.

## Creating a RESTful endpoint

A RESTful endpoint is deployed as JavaEE web archive (WAR). For this tutorial we use a simple library application to manage some books. There are two classes in this application:

- Library
- Book

The Book is a plain old Java object (POJO) with two attributes. This is a simple Java representation of a RESTful entity.



```
public class Book {  
  
    private String isbn;  
    private String title;  
  
    ...  
}
```

The Library is the RESTful Root Resource. Here we use a set of standard JAX-RS annotations to define

- The root path to the library resource
- The wire representation of the data (MIME type)
- The Http methods and corresponding paths

```
@Path("/library")  
@Consumes({ "application/json" })  
@Produces({ "application/json" })  
public class Library {  
  
    @GET  
    @Path("/books")  
    public Collection<Book> getBooks() {  
        ...  
    }  
  
    @GET  
    @Path("/book/{isbn}")  
    public Book getBook(@PathParam("isbn") String id) {  
        ...  
    }  
  
    @PUT  
    @Path("/book/{isbn}")  
    public Book addBook(@PathParam("isbn") String id, @QueryParam("title") String title) {  
        ...  
    }  
  
    @POST  
    @Path("/book/{isbn}")  
    public Book updateBook(@PathParam("isbn") String id, String title) {  
        ...  
    }  
  
    @DELETE  
    @Path("/book/{isbn}")  
    public Book removeBook(@PathParam("isbn") String id) {  
        ...  
    }  
}
```

The Library root resource uses these JAX-RS annotations:



Annotation	Description
@Path	Identifies the URI path that a resource class or class method will serve requests for
@Consumes	Defines the media types that the methods of a resource class can accept
@Produces	Defines the media type(s) that the methods of a resource class can produce
@GET	Indicates that the annotated method responds to HTTP GET requests
@PUT	Indicates that the annotated method responds to HTTP PUT requests
@POST	Indicates that the annotated method responds to HTTP POST requests
@DELETE	Indicates that the annotated method responds to HTTP DELETE requests

For a full description of the available JAX-RS annotations, see the [JAX-RS API](#) documentation.

## Package and build the endpoint

To package the endpoint we create a simple web archive and include a web.xml with the following content

### Review

[AS7-1674](#) Remove or explain why web.xml is needed for RESTful endpoints

```
<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">
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>*/</url-pattern>
  </servlet-mapping>
</web-app>
```

The root context is defined in jboss-web.xml

```
<jboss-web>
  <context-root>jaxrs-sample</context-root>
</jboss-web>
```

The code for the JAX-RS part of this tutorial is available on <https://github.com/tdiesler/javaee-tutorial/tree/master/jaxrs>. In this step we clone the repository and build the endpoint using [maven](#). There are a number of JAX-RS client tests that run against a local JBossAS 7 instance. Before we build the project, we set the JBOSS\_HOME environment variable accordingly.



[Arquillian](#), the test framework we use throughout this tutorial, can manage server startup/shutdown. It is however also possible to startup the server instance manually before you run the tests. The latter allows you to look at the console and see what log output the deployment phase and JAX-RS endpoint invocations produce.

```
$ git clone git://github.com/tdiesler/javaee-tutorial.git
Cloning into javaee-tutorial...

$ cd javaee-tutorial/jaxrs
$ export JBOSS_HOME=~/.workspace/jboss-as-7.0.1.Final
$ mvn install
...
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] JavaEE Tutorial - JAX-RS ..... SUCCESS [1.694s]
[INFO] JavaEE Tutorial - JAX-RS Server ..... SUCCESS [2.392s]
[INFO] JavaEE Tutorial - JAX-RS Client ..... SUCCESS [7.304s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12.142s
```

## Deploy the endpoint to OpenShift

First we need to create a free [OpenShift Express](#) account and select the JavaEE cartridge that runs JBossAS 7. Once we have received the confirmation email from OpenShift we can continue to create our subdomain and deploy the RESTful endpoint. A series of videos on the OpenShift Express page shows you how to do this. There is also an excellent [quick start document](#) that you have access to after login.

For this tutorial we assume you have done the above and that we can continue by creating the OpenShift application. This step sets up your JBossAS 7 instance in the cloud. Additionally a [Git](#) repository is configured that gives access to your deployed application.

```
$ rhc-create-app -a tutorial -t jbossas-7.0
Password:

Attempting to create remote application space: tutorial
Successfully created application: tutorial
Now your new domain name is being propagated worldwide (this might take a minute)...

Success! Your application is now published here:

    http://tutorial-tdiesler.rhcloud.com/

The remote repository is located here:

    ssh://79dcb9db5e134cccb9d1ba33e6089667@tutorial-tdiesler.rhcloud.com/~/.git/tutorial.git/
```

Next, we can clone the remote Git repository to our local workspace



```
$ git clone
ssh://79dcb9db5e134cccb9d1ba33e6089667@tutorial-tdiesler.rhcloud.com/~/.git/tutorial.git
Cloning into tutorial...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 24 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (24/24), 21.84 KiB, done.

ls -l tutorial
deployments
pom.xml
README
src
```

Because we want to deploy an already existing web application, which we'll build in the next step, we can safely remove the source artefacts from the repository.

```
$ rm -rf tutorial/src tutorial/pom.xml
```

Now we copy the JAX-RS endpoint webapp that we build above to the 'deployments' folder and commit the changes.

```
$ cp javaee-tutorial/jaxrs/server/target/javaee-tutorial-jaxrs-server-1.0.0-SNAPSHOT.war
tutorial/deployments
$ cd tutorial; git commit -a -m "Initial jaxrs endpoint deployment"
[master be5b5a3] Initial jaxrs endpoint deployment
 7 files changed, 0 insertions(+), 672 deletions(-)
 create mode 100644 deployments/javaee-tutorial-jaxrs-server-1.0.0-SNAPSHOT.war
 delete mode 100644 pom.xml
 delete mode 100644 src/main/java/.gitkeep
 delete mode 100644 src/main/resources/.gitkeep
 delete mode 100644 src/main/webapp/WEB-INF/web.xml
 delete mode 100644 src/main/webapp/health.jsp
 delete mode 100644 src/main/webapp/images/jbosscorp_logo.png
 delete mode 100644 src/main/webapp/index.html
 delete mode 100644 src/main/webapp/snoop.jsp

$ git push origin
Counting objects: 6, done.
...
remote: Starting application...Done
```

You can now use curl or your browser to see the JAX-RS endpoint in action. The following URL lists the books that are currently registered in the library.

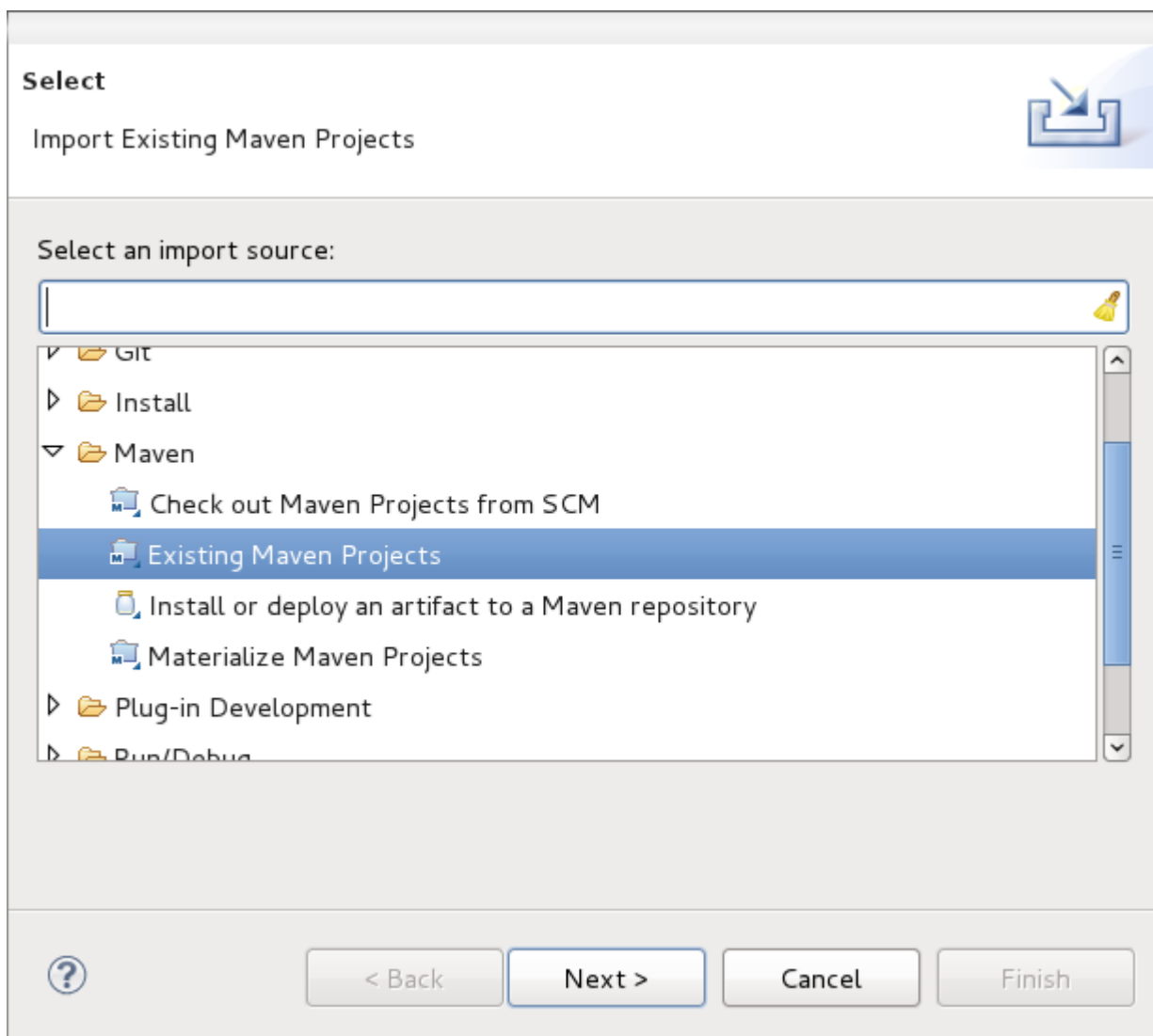


```
$ curl http://tutorial-todiesler.rhcloud.com/jaxrs-sample/library/books
[
  {"title":"The Judgment","isbn":"001"},
  {"title":"The Stoker","isbn":"002"},
  {"title":"Jackals and Arabs","isbn":"003"},
  {"title":"The Refusal","isbn":"004"}
]
```

## Building the mobile client

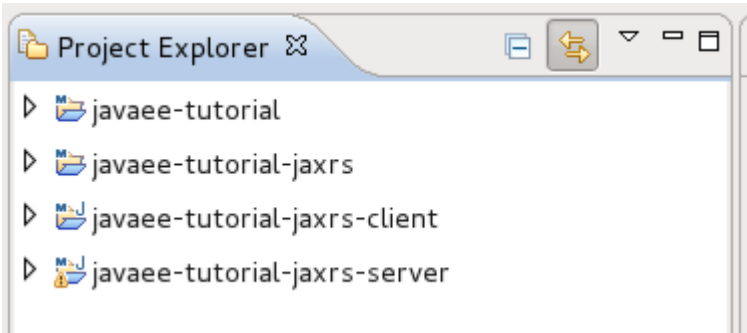
The source associated with this tutorial contains a fully working mobile client application for the Android framework. If not done so already please follow steps described in [Installing the SDK](#). In addition to the Android SDK, I recommend installing the [m2eclipse](#) and the [EGit](#) plugin to [Eclipse](#).

First, go to File|Import... and choose "Existing Maven Projects" to import the tutorial sources

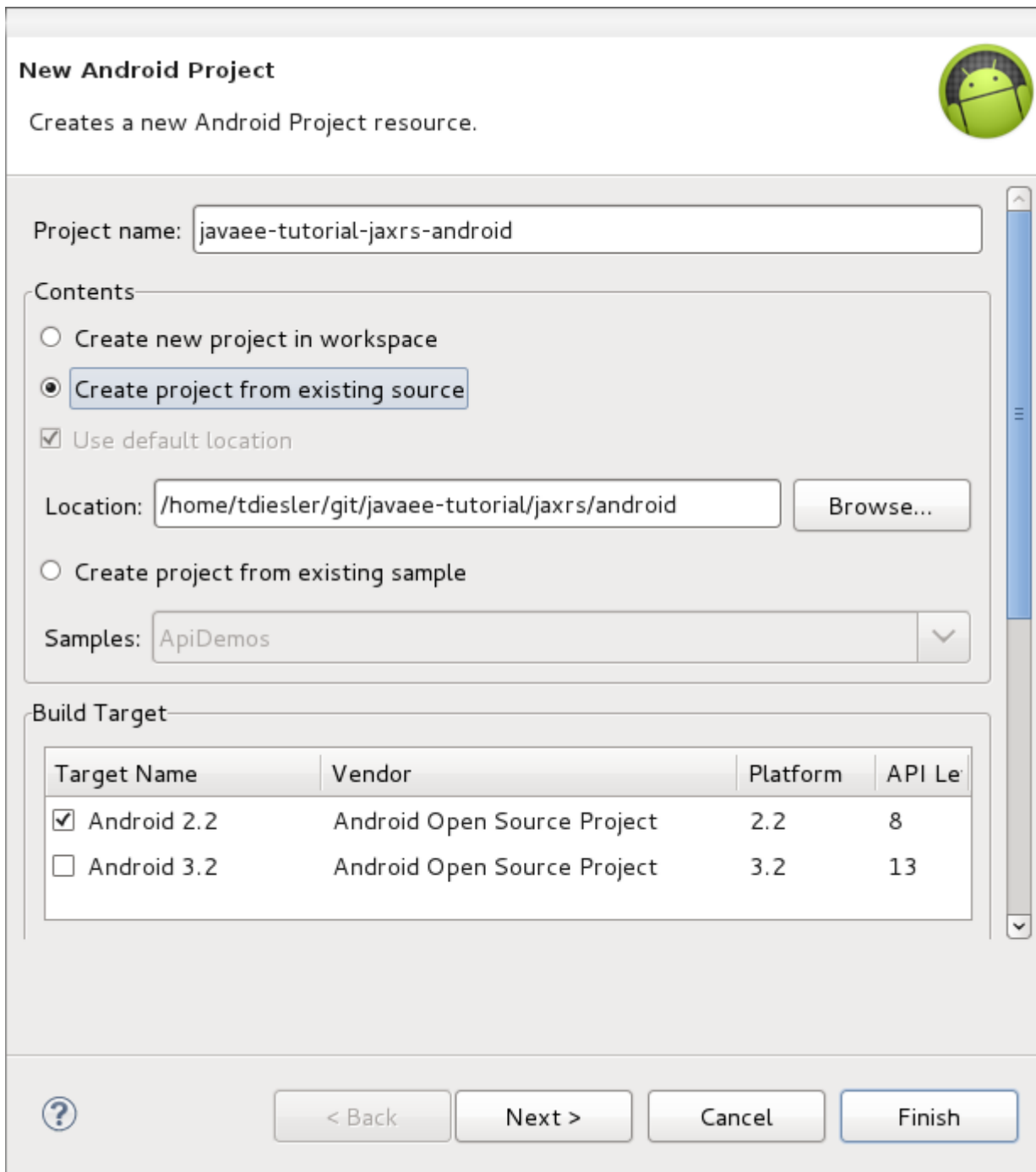


Your project view should look like this



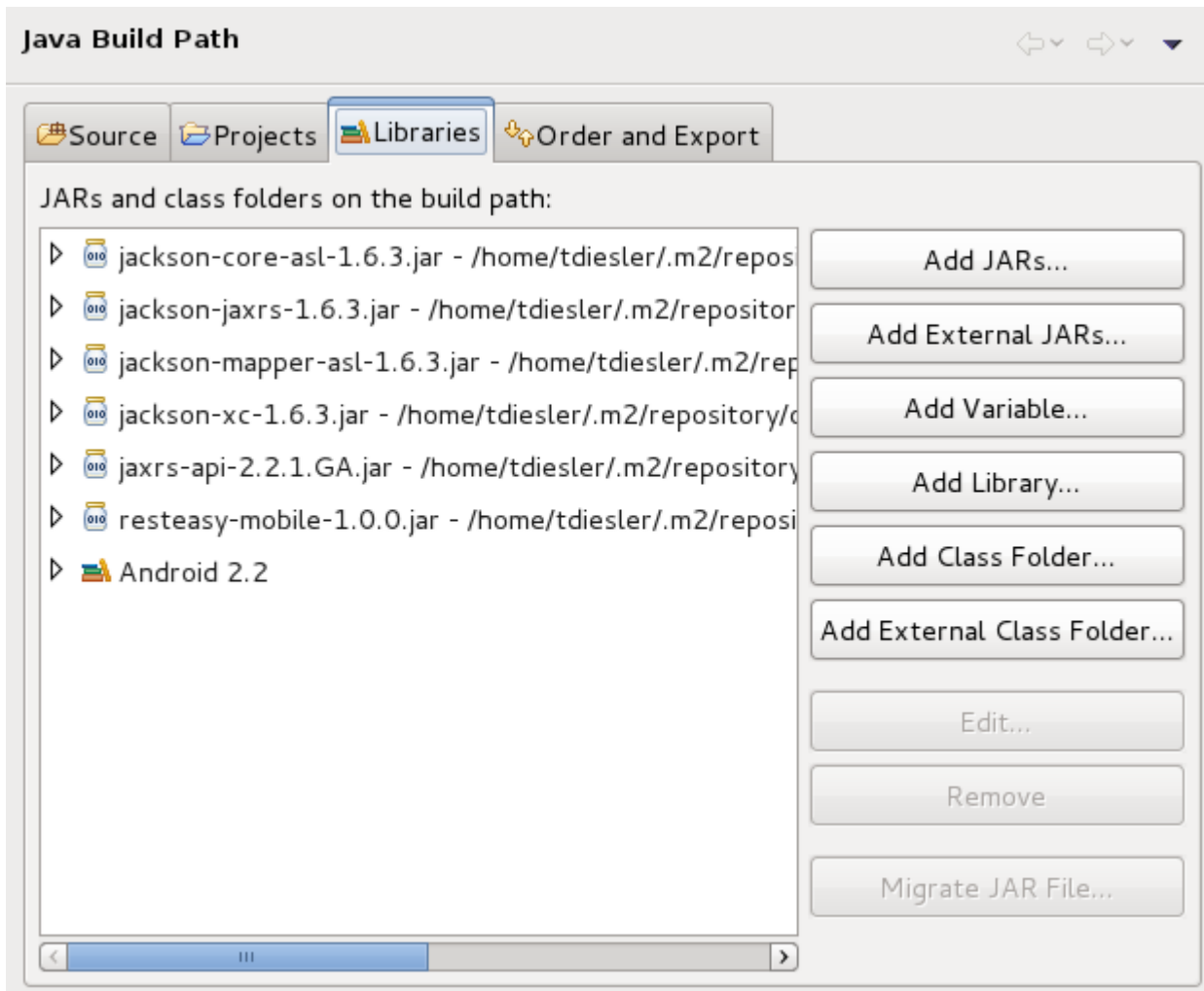


Then go to File|New|Android Project and fill out the first wizard page like this

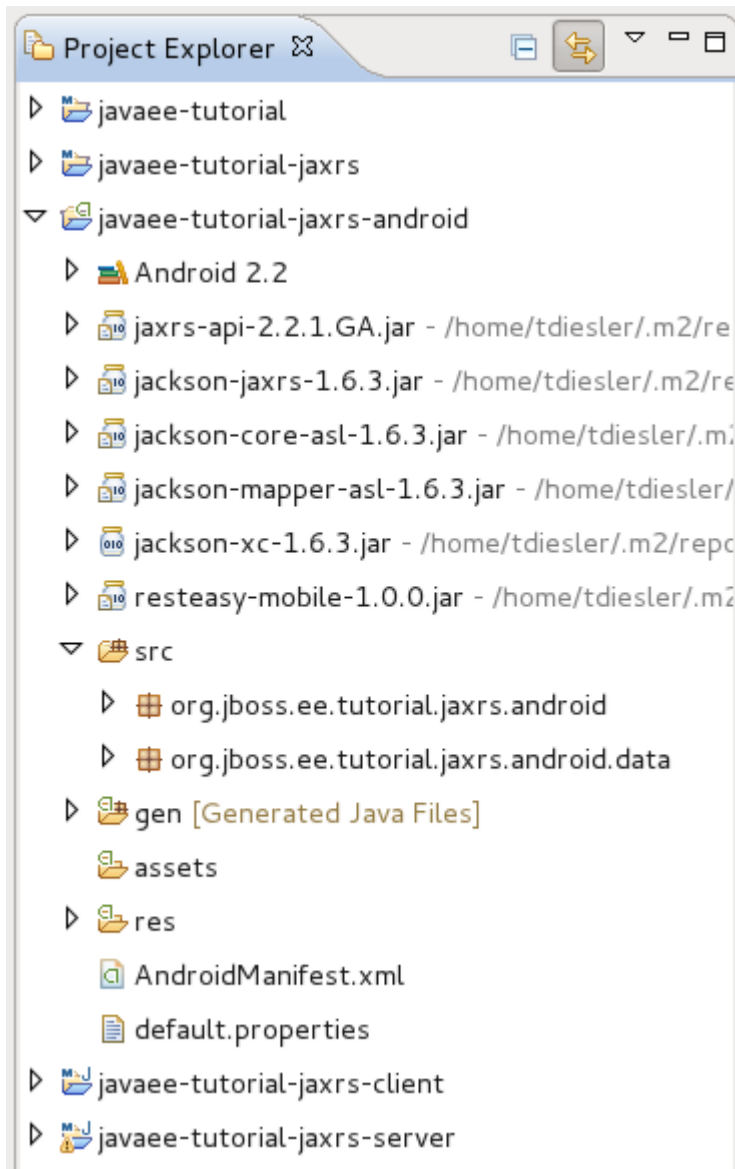




Click Finish. Next, go to Project|Properties|Build Path|Libraries and add these external libraries to your android project.



Your final project view should look like this



To run the application in the emulator, we need an Android Virtual Device (AVD). Go to Window|Android SDK and AVD Manager and create a new AVD like this



Name:

Target:

CPU/ABI:

SD Card:

Size:  MiB

File:

Snapshot:  Enabled

Skin:

Built-in:

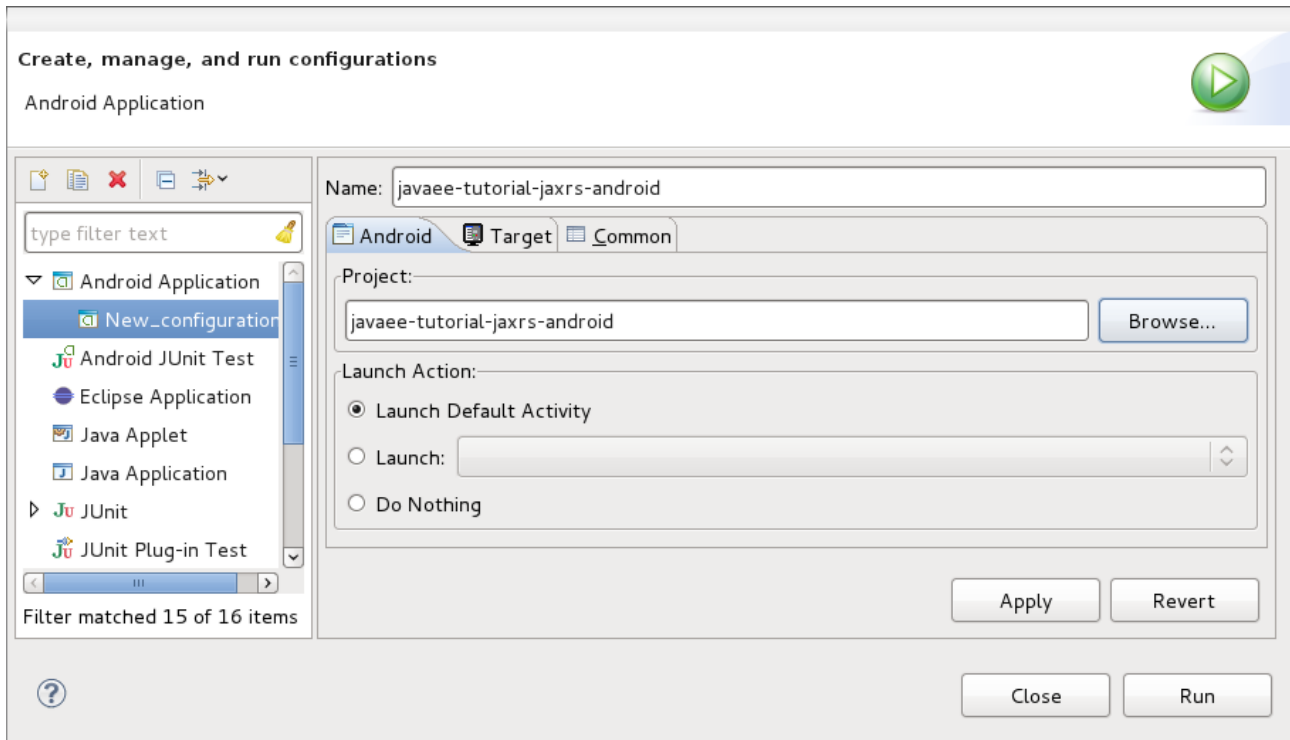
Resolution:  x

Hardware:

Property	Value	
Abstracted LCD density	160	<input type="button" value="New..."/>
Max VM application heap size	24	<input type="button" value="Delete"/>
<input type="text"/>		

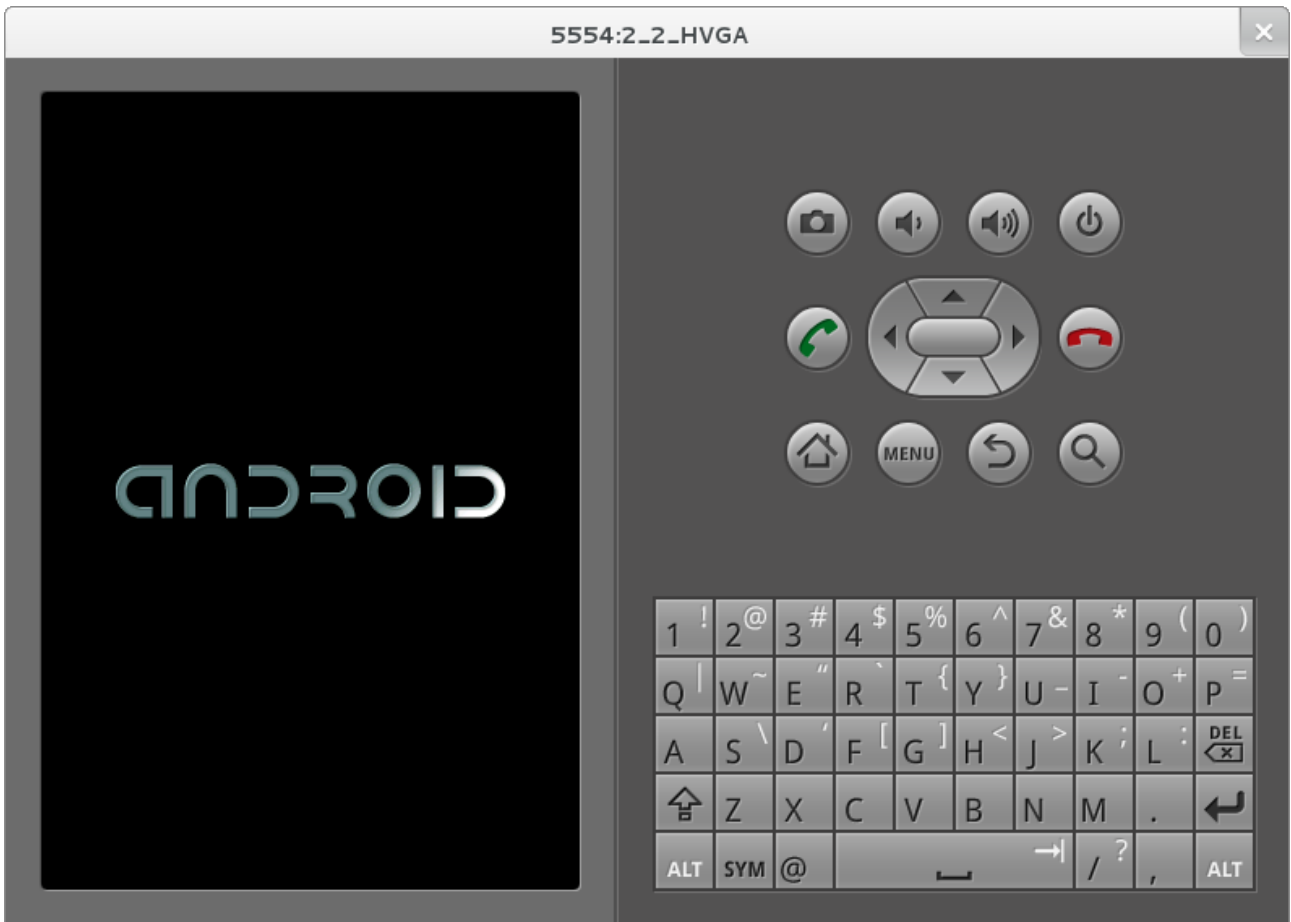
Override the existing AVD with the same name

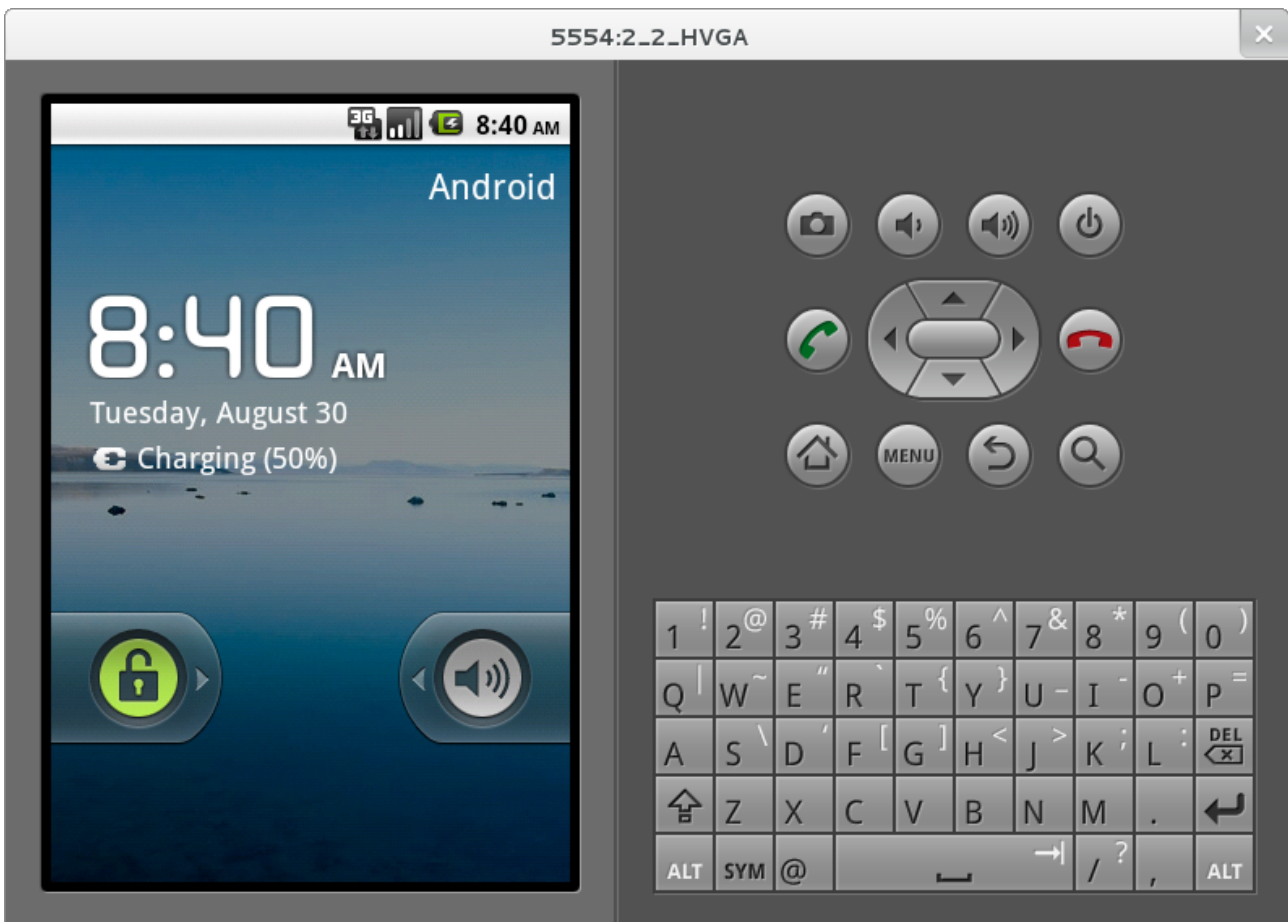
Now go to Run|Configuration to create a new run configuration for the client app.



Now you should be able to launch the application in the debugger. Right click on the `javaee-tutorial-jaxrs-android` project and select `Debug As|Android Application`. This should launch the emulator, which now goes through a series of boot screens until it eventually displays the Android home screen. This will take a minute or two if you do this for the first time.







When you unlock the home screen by dragging the little green lock to the right. You should see the the running JAX-RS client application.





Finally, you need to configure the host that the client app connects to. This would be the same as you used above to curl the library list. In the emulator click Menu|Host Settings and enter the host address of your OpenShift application.



When going back to the application using the little back arrow next to Menu, you should see a list of books.



You can now add, edit and delete books and switch between your browser and the emulator to verify that the client app is not cheating and that the books are in fact in the cloud on your JBossAS 7 instance.

In Eclipse you can go to the Debug perspective and click on the little Android robot in the lower right corner. This will display the LogCat view, which should display log output from that Android system as well as from this client app

```
08-30 09:05:46.180: INFO/JaxrsSample(269): removeBook: Book [isbn=1234, title=1234]
08-30 09:05:46.210: INFO/JaxrsSample(269): requestURI:
http://tutorial-tdiesler.rhcloud.com:80/jaxrs-sample/library
08-30 09:05:46.860: INFO/global(269): Default buffer size used in BufferedInputStream
constructor. It would be better to be explicit if an 8k buffer is required.
08-30 09:05:46.920: INFO/JaxrsSample(269): getBooks: [Book [isbn=001, title=The Judgment], Book
[isbn=002, title=The Stoker], Book [isbn=003, title=Jackals and Arabs], Book [isbn=004,
title=The Refusal]]
```

## Exploring the mobile client

There is a lot to writing high quality mobile applications. The goal of this little application is to get you started with JBossAS 7 / Android integration. There is also a portable approach to writing mobile applications. A popular one would be through [PhoneGap](#). With PhoneGap you write your application in HTML+CSS+JavaScript. It then runs in the browser of your mobile device. Naturally, **not the full set** of mobile platform APIs would be available through this approach.



## The JAX-RS client application uses an annotated library client interface

```
@Consumes({ "application/json" })
@Produces({ "application/json" })
public interface LibraryClient {

    @GET
    @Path("/books")
    public List<Book> getBooks();

    @GET
    @Path("/book/{isbn}")
    public Book getBook(@PathParam("isbn") String id);

    @PUT
    @Path("/book/{isbn}")
    public Book addBook(@PathParam("isbn") String id, @QueryParam("title") String title);

    @POST
    @Path("/book/{isbn}")
    public Book updateBook(@PathParam("isbn") String id, String title);

    @DELETE
    @Path("/book/{isbn}")
    public Book removeBook(@PathParam("isbn") String id);
}
```

There are two implementations of this interface available.

- LibraryHttpClient
- LibraryResteasyClient

The first uses APIs that are available in the Android SDK natively. The code is much more involved, but there would be no need to add external libraries (i.e. resteasy, jackson, etc). The effect is that the total size of the application is considerably smaller in size (i.e. 40k)



```
@Override
public List<Book> getBooks() {
    List<Book> result = new ArrayList<Book>();
    String content = get("books");
    Log.d(LOG_TAG, "Result content:" + content);
    if (content != null) {
        try {
            JSNTokener tokener = new JSNTokener(content);
            JSONArray array = (JSONArray) tokener.nextValue();
            for (int i = 0; i < array.length(); i++) {
                JSONObject obj = array.getJSONObject(i);
                String title = obj.getString("title");
                String isbn = obj.getString("isbn");
                result.add(new Book(isbn, title));
            }
        } catch (JSONException ex) {
            ex.printStackTrace();
        }
    }
    Log.i(LOG_TAG, "getBooks: " + result);
    return result;
}

private String get(String path) {
    try {
        HttpGet request = new HttpGet(getRequestURI(path));
        HttpResponse res = httpClient.execute(request);
        String content = EntityUtils.toString(res.getEntity());
        return content;
    } catch (Exception ex) {
        ex.printStackTrace();
        return null;
    }
}
```

The second implementation uses the fabulous RESTEasy client proxy to interact with the JAX-RS endpoint. The details of Http connectivity and JSON data binding is transparently handled by RESTEasy. The total size of the application is considerably bigger in size (i.e. 400k)

```
@Override
public List<Book> getBooks() {
    List<Book> result = new ArrayList<Book>();
    try {
        result = getLibraryClient().getBooks();
    } catch (RuntimeException ex) {
        ex.printStackTrace();
    }
    Log.i(LOG_TAG, "getBooks: " + result);
    return result;
}
```



Stay tuned for an update on a much more optimized version of the RESTEasy mobile client. Feasible is also a RESTEasy JavaScript library that would enable the portable PhoneGap approach.

## 2.3.2 Java Servlet Technology

### Coming Soon

This guide is still under development, check back soon!

## Content

- [Asynchronous Support](#)

## Asynchronous Support

## 2.3.3 Java Server Faces Technology (JSF)

## 2.3.4 Java Persistence API (JPA)

## 2.3.5 Java Transaction API (JTA)

### Coming Soon

This guide is still under development, check back soon!

## 2.3.6 Managed Beans

## 2.3.7 Contexts and Dependency Injection (CDI)

## 2.3.8 Bean Validation



## 2.3.9 Java Message Service API (JMS)

### Coming Soon

This guide is still under development, check back soon!

- [Configure JBossAS for Messaging](#)
- [Adding the message destinations](#)

## Configure JBossAS for Messaging

Currently, the default configuration does not include the JMS subsystem. To enable JMS in the standalone server you need to add these configuration items to standalone.xml or simply use standalone-full.xml.

```
<extension module="org.jboss.as.messaging"/>

<subsystem xmlns="urn:jboss:domain:messaging:1.0">
  <!-- Default journal file size is 10Mb, reduced here to 100k for faster first boot -->
  <journal-file-size>102400</journal-file-size>
  <journal-min-files>2</journal-min-files>
  <journal-type>NIO</journal-type>
  <!-- disable messaging persistence -->
  <persistence-enabled>false</persistence-enabled>

  <connectors>
    <netty-connector name="netty" socket-binding="messaging" />
    <netty-connector name="netty-throughput" socket-binding="messaging-throughput">
      <param key="batch-delay" value="50"/>
    </netty-connector>
    <in-vm-connector name="in-vm" server-id="0" />
  </connectors>

  <acceptors>
    <netty-acceptor name="netty" socket-binding="messaging" />
    <netty-acceptor name="netty-throughput" socket-binding="messaging-throughput">
      <param key="batch-delay" value="50"/>
      <param key="direct-deliver" value="false"/>
    </netty-acceptor>
    <acceptor name="stomp-acceptor">

  <factory-class>org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-class>
    <param key="protocol" value="stomp" />
    <param key="port" value="61613" />
  </acceptor>
    <in-vm-acceptor name="in-vm" server-id="0" />
  </acceptors>

  <security-settings>
    <security-setting match="#">
      <permission type="createNonDurableQueue" roles="guest"/>
    </security-setting>
  </security-settings>
</subsystem>
```



```
<permission type="deleteNonDurableQueue" roles="guest"/>
  <permission type="consume" roles="guest"/>
  <permission type="send" roles="guest"/>
</security-setting>
</security-settings>

<address-settings>
  <!--default for catch all-->
  <address-setting match="#">
    <dead-letter-address>jms.queue.DLQ</dead-letter-address>
    <expiry-address>jms.queue.ExpiryQueue</expiry-address>
    <redelivery-delay>0</redelivery-delay>
    <max-size-bytes>10485760</max-size-bytes>
    <message-counter-history-day-limit>10</message-counter-history-day-limit>
    <address-full-policy>BLOCK</address-full-policy>
  </address-setting>
</address-settings>

<!--JMS Stuff-->
<jms-connection-factories>
  <connection-factory name="InVmConnectionFactory">
    <connectors>
      <connector-ref connector-name="in-vm"/>
    </connectors>
    <entries>
      <entry name="java:/ConnectionFactory"/>
    </entries>
  </connection-factory>
  <connection-factory name="RemoteConnectionFactory">
    <connectors>
      <connector-ref connector-name="netty"/>
    </connectors>
    <entries>
      <entry name="RemoteConnectionFactory"/>
    </entries>
  </connection-factory>
  <pooled-connection-factory name="hornetq-ra">
    <transaction mode="xa"/>
    <connectors>
      <connector-ref connector-name="in-vm"/>
    </connectors>
    <entries>
      <entry name="java:/JmsXA"/>
      <!-- Global JNDI entry used to provide a default JMS Connection factory to EE
application -->
      <entry name="java:jboss/DefaultJMSConnectionFactory"/>
    </entries>
  </pooled-connection-factory>
</jms-connection-factories>

<jms-destinations>
  <jms-queue name="testQueue">
    <entry name="queue/test"/>
  </jms-queue>
  <jms-topic name="testTopic">
    <entry name="topic/test"/>
  </jms-topic>
</jms-destinations>
```





```
</subsystem>

<socket-binding name="messaging" port="5445" />
<socket-binding name="messaging-throughput" port="5455" />
```

Alternatively run the server using the preview configuration

```
$ bin/standalone.sh --server-config=standalone-preview.xml
```

## Adding the message destinations

For this tutorial we use two message destinations

- BidQueue - The queue that receives the client bids
- AuctionTopic - The topic that publishes the start of a new auction

You can either add the message destinations by using the Command Line Interface (CLI)

```
$ bin/jboss-admin.sh --connect
Connected to standalone controller at localhost:9999
[standalone@localhost:9999 /] jms-queue add --queue-address=BidQueue --entries=queue/bid
[standalone@localhost:9999 /] jms-topic add --topic-address=AuctionTopic --entries=topic/auction
[standalone@localhost:9999 /] exit
Closed connection to localhost:9999
```

or by adding them to the subsystem configuration as shown above.

## 2.3.10 JavaEE Connector Architecture (JCA)

## 2.3.11 JavaMail API



## 2.3.12 Java Authorization Contract for Containers (JACC)

In order to register your own JACC Module, you'll need to create a server module containing the required classes, and then set three system properties for WildFly to take it. Such a module would depend on the "javax.api" and "javaee.api" modules.

An example module.xml for such a module could be:

```
<module xmlns="urn:jboss:module:1.1" name="com.example.customjacc">
  <resources>
    <resource-root path="customjacc.jar" />
  </resources>
  <dependencies>
    <module name=" javax.api " />
    <module name=" javaee.api " />
  </dependencies>
</module>
```

The specified JAR needs to contain at least two classes, as mandated by the JACC spec:

- A `PolicyProvider` implementation: in our example, it'll be `com.example.customjacc.CustomPolicy`.
- A `PolicyConfigurationFactory` implementation: `com.example.customjacc.CustomPolicyConfigurationFactory` in our case.

The spec requires two system properties to be set for the server to register the JACC Module.

For a server running in standalone mode, put the following commands in the JBoss CLI:

```
[standalone@localhost:9990 /]
/system-property=javax.security.jacc.policy.provider:add(value=com.example.customjacc.CustomPolicy
/]
/system-property=javax.security.jacc.PolicyConfigurationFactory.provider:add(value=com.example.cus
```

Another property is needed to make WildFly know where to load the classes from:

```
[standalone@localhost:9990 /]
/system-property=org.jboss.as.security.jacc-module:add(value=com.example.customjacc)
```



## 2.3.13 Java Authentication Service Provider Interface for Containers (JASPIC)

JASPI is not available by default for deployments, and a specific Security Domain must be created to use it. For a simplified developer experience, a default JASPI Domain is already bundled, called `jaspitest`.

To make use of it, a Web Application only needs to specify the desired security domain in the `jboss-web.xml` deployment descriptor. This file should be located under the `WEB-INF` directory. An example `jboss-web.xml` enabling the default JASPI domain:

```
<?xml version="1.0"?>
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd"
  version="10.0">
  <security-domain>jaspitest</security-domain>
</jboss-web>
```

For EAR deployments, a `jboss-app.xml` like the following should be used instead, placed under the root `META-INF` directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-app xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="7.0">
  <security-domain>jaspitest</security-domain>
</jboss-app>
```



## 2.3.14 Enterprise JavaBeans Technology (EJB)

In this section we'll look at the two main types of beans (Session Beans and Message Driven Beans), the methods to access and the packaging possibilities of beans.

### Coming Soon

This guide is still under development, check back soon!

- [Session Beans](#)
  - [Stateful Session Beans](#)
  - [Stateless Session Beans](#)
  - [Singleton Session Beans](#)
- [Message Driven Beans](#)
- [How can Enterprise JavaBeans can be accessed?](#)
  - [Remote call invocation](#)
  - [Local call invocation](#)
  - [Web Services](#)
- [Packaging](#)

### **Session Beans**

#### **Stateful Session Beans**

#### **Stateless Session Beans**

#### **Singleton Session Beans**

### **Message Driven Beans**

### **How can Enterprise JavaBeans can be accessed?**

#### **Remote call invocation**

#### **Local call invocation**

#### **Web Services**

### **Packaging**




## 2.3.15 Java API for XML Web Services (JAX-WS)

JBossWS uses the JBoss Application Server as its target container. The following examples focus on web service deployments that leverage EJB3 service implementations and the JAX-WS programming models. For further information on POJO service implementations and advanced topics you need consult the [user guide](#).

### Developing web service implementations

JAX-WS does leverage annotations in order to express web service meta data on Java components and to describe the mapping between Java data types and XML. When developing web service implementations you need to decide whether you are going to start with an abstract contract (WSDL) or a Java component.

If you are in charge to provide the service implementation, then you are probably going to start with the implementation and derive the abstract contract from it. You are probably not even getting in touch with the WSDL unless you hand it to 3rd party clients. For this reason we are going to look at a service implementation that leverages [JSR-181 annotations](#).

 Even though detailed knowledge of web service meta data is not required, it will definitely help if you make yourself familiar with it. For further information see

- [Web service meta data \(JSR-181\)](#)
- [Java API for XML binding \(JAXB\)](#)
- [Java API for XML-Based Web Services](#)

### The service implementation class

When starting from Java you must provide the service implementation. A valid endpoint implementation class must meet the following requirements:

- It *must* carry a `javax.jws.WebService` annotation (see JSR 181)
- All method parameters and return types *must* be compatible with the JAXB 2.0

Let's look at a sample EJB3 component that is going to be exposed as a web service.

Don't be confused with the EJB3 annotation `@Stateless`. We concentrate on the `@WebService` annotation for now.



## Implementing the service

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless (1)
@WebService( (2)
    name="ProfileMgmt",
    targetNamespace = "http://org.jboss.ws/samples/retail/profile",
    serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE) (3)
public class ProfileMgmtBean {

    @WebMethod (4)
    public DiscountResponse getCustomerDiscount(DiscountRequest request) {
        return new DiscountResponse(request.getCustomer(), 10.00);
    }
}
```

1. We are using a stateless session bean implementation
2. Exposed a web service with an explicit namespace
3. It's a doc/lit bare endpoint
4. And offers an 'getCustomerDiscount' operation



## What about the payload?

The method parameters and return values are going to represent our XML payload and thus require being compatible with JAXB2. Actually you wouldn't need any JAXB annotations for this particular example, because JAXB relies on meaningful defaults. For the sake of documentation we put the more important ones here.

Take a look at the request parameter:

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(                                     (1)
    name = "discountRequest",
    namespace="http://org.jboss.ws/samples/retail/profile",
    propOrder = { "customer" }
)
public class DiscountRequest {

    protected Customer customer;

    public DiscountRequest() {
    }

    public DiscountRequest(Customer customer) {
        this.customer = customer;
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer value) {
        this.customer = value;
    }

}
```

1. In this case we use `@XmlType` to specify an XML complex type name and override the namespace.



If you have more complex mapping problems you need to consult the [JAXB documentation](#).



## Deploying service implementations

Service deployment basically depends on the implementation type. As you may already know web services can be implemented as EJB3 components or plain old Java objects. This quick start leverages EJB3 components, that's why we are going to look at this case in the next sections.

### EJB3 services

Simply wrap up the service implementation class, the endpoint interface and any custom data types in a JAR and drop them in the *deployment* directory. No additional deployment descriptors required. Any meta data required for the deployment of the actual web service is taken from the annotations provided on the implementation class and the service endpoint interface. JBossWS will intercept that EJB3 deployment (the bean will also be there) and create an HTTP endpoint at deploy-time.

### The JAR package structure

```
jar -tf jaxws-samples-retail.jar
```

```
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```



If the deployment was successful you should be able to see your endpoint in the application server management console.

## Consuming web services

When creating web service clients you would usually start from the WSDL. JBossWS ships with a set of tools to generate the required JAX-WS artefacts to build client implementations. In the following section we will look at the most basic usage patterns. For a more detailed introduction to web service client please consult the user guide.





## Creating the client artifacts

### Using wsconsume

The *wsconsume* tool is used to consume the abstract contract (WSDL) and produce annotated Java classes (and optionally sources) that define it. We are going to start with the WSDL from our retail example (ProfileMgmtService.wsdl). For a detailed tool reference you need to consult the user guide.

```
wsconsume is a command line tool that generates
portable JAX-WS artifacts from a WSDL file.

usage: org.jboss.ws.tools.jaxws.command.wsconsume [options] <wsdl-url>

options:
  -h, --help                Show this help message
  -b, --binding=<file>      One or more JAX-WS or JAXB binding files
  -k, --keep                Keep/Generate Java source
  -c --catalog=<file>       Oasis XML Catalog file for entity resolution
  -p --package=<name>       The target package for generated source
  -w --wsdlLocation=<loc>  Value to use for @WebService.wsdlLocation
  -o, --output=<directory> The directory to put generated artifacts
  -s, --source=<directory> The directory to put Java source
  -q, --quiet               Be somewhat more quiet
  -t, --show-traces        Show full exception stack traces
```

Let's try it on our sample:

```
~/wsconsume.sh -k -p org.jboss.test.ws.jaxws.samples.retail.profile ProfileMgmtService.wsdl
(1)

org.jboss.test.ws.jaxws.samples.retail.profile.Customer.java
org.jboss.test.ws.jaxws.samples.retail.profile.DiscountRequest.java
org.jboss.test.ws.jaxws.samples.retail.profile.DiscountResponse.java
org.jboss.test.ws.jaxws.samples.retail.profile.ObjectFactory.java
org.jboss.test.ws.jaxws.samples.retail.profile.ProfileMgmt.java
org.jboss.test.ws.jaxws.samples.retail.profile.ProfileMgmtService.java
org.jboss.test.ws.jaxws.samples.retail.profile.package-info.java
```

1. As you can see we did use the `-p` switch to specify the package name of the generated sources.



## The generated artifacts explained

File	Purpose
ProfileMgmt.java	Service Endpoint Interface
Customer.java	Custom data type
Discount*.java	Custom data type
ObjectFactory.java	JAXB XML Registry
package-info.java	Holder for JAXB package annotations
ProfileMgmtService.java	Service factory

Basically *wscconsume* generates all custom data types (JAXB annotated classes), the service endpoint interface and a service factory class. We will look at how these artifacts can be used the build web service client implementations in the next section.

## Constructing a service stub

Web service clients make use of a service stubs that hide the details of a remote web service invocation. To a client application a WS invocation just looks like an invocation of any other business component. In this case the service endpoint interface acts as the business interface. JAX-WS does use a service factory class to construct this as particular service stub:

```
import javax.xml.ws.Service;
[... ]
Service service = Service.create(                               (1)
    new URL("http://example.org/service?wsdl"),
    new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);  (2)

// do something with the service stub here...                 (3)
```

1. Create a service factory using the WSDL location and the service name
2. Use the tool created service endpoint interface to build the service stub
3. Use the stub like any other business interface

## Appendix

### Sample wsdl contract

```
<definitions
  name='ProfileMgmtService'
  targetNamespace='http://org.jboss.ws/samples/retail/profile'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  xmlns:ns1='http://org.jboss.ws/samples/retail'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns:tns='http://org.jboss.ws/samples/retail/profile'
```



```
xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

<types>

  <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
            version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
    <xs:complexType name='customer'>
      <xs:sequence>
        <xs:element minOccurs='0' name='creditCardDetails' type='xs:string' />
        <xs:element minOccurs='0' name='firstName' type='xs:string' />
        <xs:element minOccurs='0' name='lastName' type='xs:string' />
      </xs:sequence>
    </xs:complexType>
  </xs:schema>

  <xs:schema
    targetNamespace='http://org.jboss.ws/samples/retail/profile'
    version='1.0'
    xmlns:ns1='http://org.jboss.ws/samples/retail'
    xmlns:tns='http://org.jboss.ws/samples/retail/profile'
    xmlns:xs='http://www.w3.org/2001/XMLSchema'>

    <xs:import namespace='http://org.jboss.ws/samples/retail' />
    <xs:element name='getCustomerDiscount'
      nillable='true' type='tns:discountRequest' />
    <xs:element name='getCustomerDiscountResponse'
      nillable='true' type='tns:discountResponse' />
    <xs:complexType name='discountRequest'>
      <xs:sequence>
        <xs:element minOccurs='0' name='customer' type='ns1:customer' />
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name='discountResponse'>
      <xs:sequence>
        <xs:element minOccurs='0' name='customer' type='ns1:customer' />
        <xs:element name='discount' type='xs:double' />
      </xs:sequence>
    </xs:complexType>
  </xs:schema>

</types>

<message name='ProfileMgmt_getCustomerDiscount'>
  <part element='tns:getCustomerDiscount' name='getCustomerDiscount' />
</message>
<message name='ProfileMgmt_getCustomerDiscountResponse'>
  <part element='tns:getCustomerDiscountResponse'
    name='getCustomerDiscountResponse' />
</message>
<portType name='ProfileMgmt'>
  <operation name='getCustomerDiscount'
    parameterOrder='getCustomerDiscount'>

    <input message='tns:ProfileMgmt_getCustomerDiscount' />
    <output message='tns:ProfileMgmt_getCustomerDiscountResponse' />
  </operation>
</portType>
```



```
<binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
  <soap:binding style='document'
    transport='http://schemas.xmlsoap.org/soap/http' />
  <operation name='getCustomerDiscount'>
    <soap:operation soapAction='' />
    <input>

      <soap:body use='literal' />
    </input>
    <output>
      <soap:body use='literal' />
    </output>
  </operation>
</binding>
<service name='ProfileMgmtService'>
  <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

    <soap:address
      location='http://<HOST>:<PORT>/jaxws-samples-retail/ProfileMgmtBean' />
    </port>
  </service>
</definitions>
```



## 2.4 JBoss AS7 Extension Technologies

---

### Coming Soon

This guide is still under development, check back soon!

1. OSGi Technology
2. Management Interface

### 2.4.1 Management Interface

#### Coming Soon

This guide is still under development, check back soon!

- [Management via the Java Management Extension \(JMX\)](#)
- [Management via RESTful services](#)
- [Batch Management / Command Line Interface \(CLI\)](#)

#### **Management via the Java Management Extension (JMX)**

#### **Management via RESTful services**

#### **Batch Management / Command Line Interface (CLI)**