Set up the Atlassian Plugin SDK and Build a Project

This tutorial takes you step-by-step through creating your first Atlassian add-on, a plugin, using the Atlassian Plugin SDK. The purpose of the tutorial is to guide you through setting up your machine environment and installation of the SDK. It also teaches you about the tools, concepts, and basic processes used to create plugins. When you complete the tutorial, you will have added a custom menu to the JIRA application.

This tutorial consists of the following pages:

- Set up the SDK Prerequisites on a Windows System (or on Linux/Mac)
- Install the Atlassian SDK on a Windows System (or on Linux/Mac)
- Explore the Installed SDK and the atlas Commands
- Create a HelloWorld Plugin Project
- Set Up the Eclipse IDE for Windows (or on Linux/Mac)
- Put the Final Polish on the Project in Eclipse

The tutorial requires little or no programming knowledge. You should have a good understanding of the shell (command line environment) for your operating system. If you consider yourself an absolute beginner programmer, you can still complete this tutorial.

How to work through the tutorial

You should work through each page sequentially as each new page builds on the material from the previous page. At the end of each page is a **Next Steps** heading that tells you where to go next. Beginners should allow two hours to work through the entire tutorial.

If you are experienced or just skimming pages, much of the tutorial will be familiar to you and it should not take too long. If you get lost, you can use the navigation bar (to your left) to locate the next page. If you feel confident skipping pages or just going to pages you need, feel free to do that too.

Even though this tutorial customizes JIRA, the tools, concepts, and basic processes are the same regardless of which Atlassian product you want to customize or extend.

Set up the SDK Prerequisites on a Windows System

Before you install the Atlassian software developer kit (SDK), you must make sure your Microsoft Windows system has the prerequisite software, that your environment is configured, and that the systems you need to use the SDK exist. This page contains the following sections:

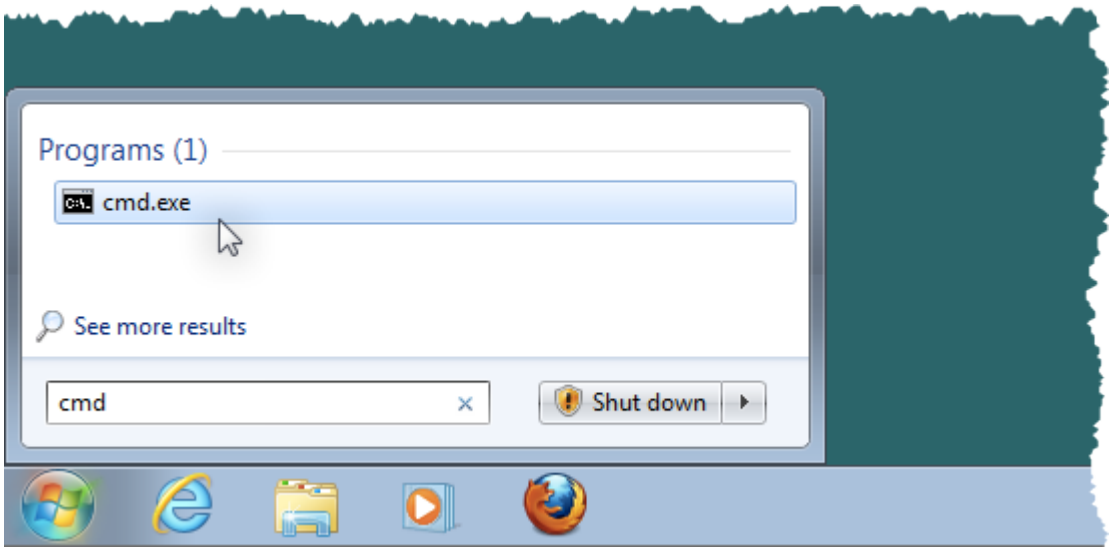
Linux or Mac User?

See this page.

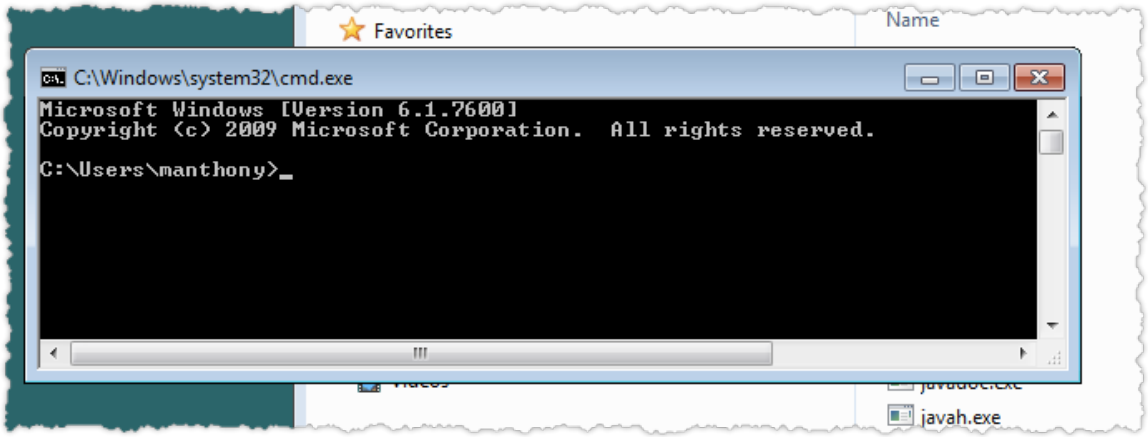
Step 1: Verify the Java Developer Kit (JDK) is Installed

The Atlassian SDK relies on 1.6.X of the Oracle Java SE Development Kit 6 (JDK). Before installing the SDK, verify that you have installed the JDK 1.6.X. To do this:

1. Locate the **cmd** program in the Windows **Start** menu.



2. Click on the **cmd** menu item.
   The system displays a Windows command window:



3. Verify that the `JAVA_HOME`variable is set by entering the following at the prompt:

```
C:\Users\manthony>echo %JAVA_HOME%
C:\Program Files\Java\jdk1.6.0_32
```

4. If the environment variable is set, verify your `Path` includes the JDK `bin` by entering `javac -version` at the command line:

```
C:\Users\manthony>javac -version
javac jdk1.6.0_32
```

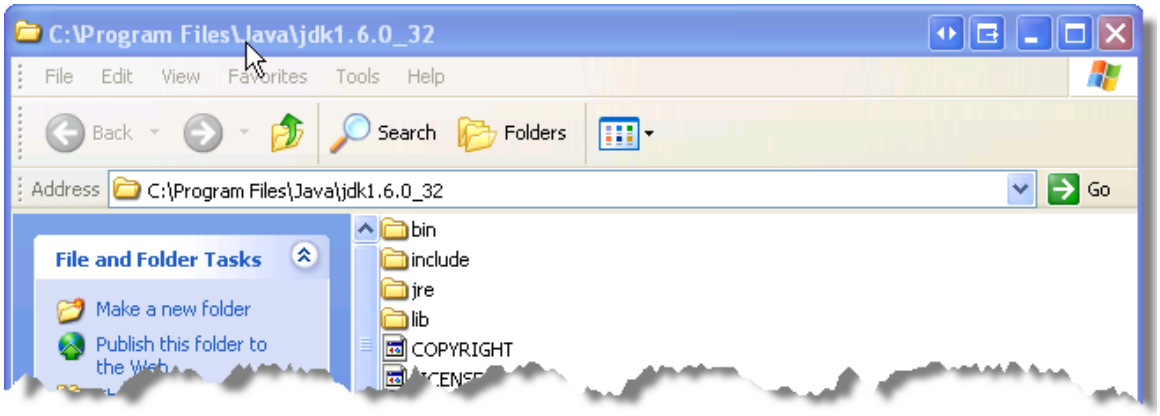
5. If you have the JDK installed and your Path configured, skip the rest of this procedure and go to Step 2.
6. If you don't have the JDK 1.6.X installed, follow the Oracle website instructions for installing it.

7. Restart the Windows command window after installing the JDK.
   The Windows command window does not automatically detect environment variables set by installing new software. For that reason, you should always close and restart the command window after installing new software in your Windows environment or after you add environment variables manually.

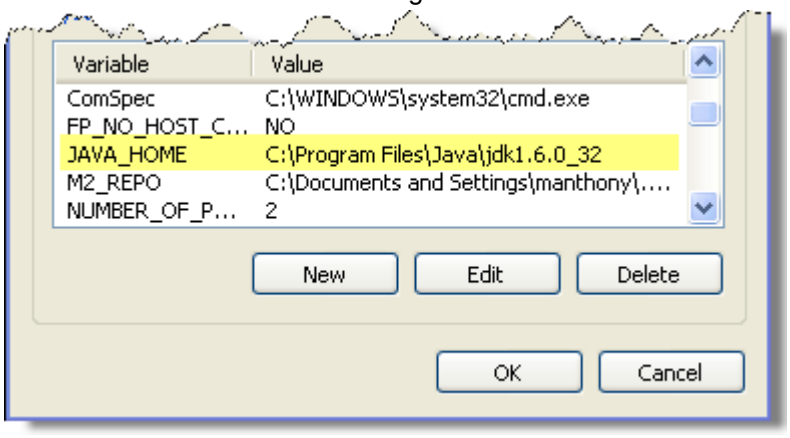
Step 2: Set the JAVA_HOME system variable

You only need to do this procedure if the `JAVA_HOME` variable is not set. The `JAVA_HOME` environment variable specifies the location of the JDK on your system. If you accepted the defaults when you installed the JDK, this directory should be in the `C:\Program Files\Java\jdk`*version* folder on your system.

1. Browse to the `C:\Program Files\Java\jdk`*version* folder on your system.
2. Copy the path to the folder.



3. Right-click the **My Computer** icon on your desktop.
   The right-click menu displays.
4. Choose **Properties**.
5. Click the **Advanced system settings** option.
   The **System Properties** dialog displays.
6. Click the **Advanced** tab.
7. Click the **Environment Variables** button.
   The **Environment Variables** dialog appears.
8. Locate the **System variables** section
9. Click the **New** button.
   The **New System Variable** dialog appears.
10. Enter JAVA_HOME in the **Variable name** field.
11. Paste the folder path you copied in Step. 2 above into the **Variable value** field.
12. Press **OK** to close the dialog.
    The **Environment Variables** dialog shows the new variable.



13. Press **OK** to close this dialog and then again to close the **System Properties** dialog.

14. Open a command window and enter the following command:

```
C:\Documents and Settings\manthony>echo %JAVA_HOME%
C:\Program Files\Java\jdk1.6.0_32
```

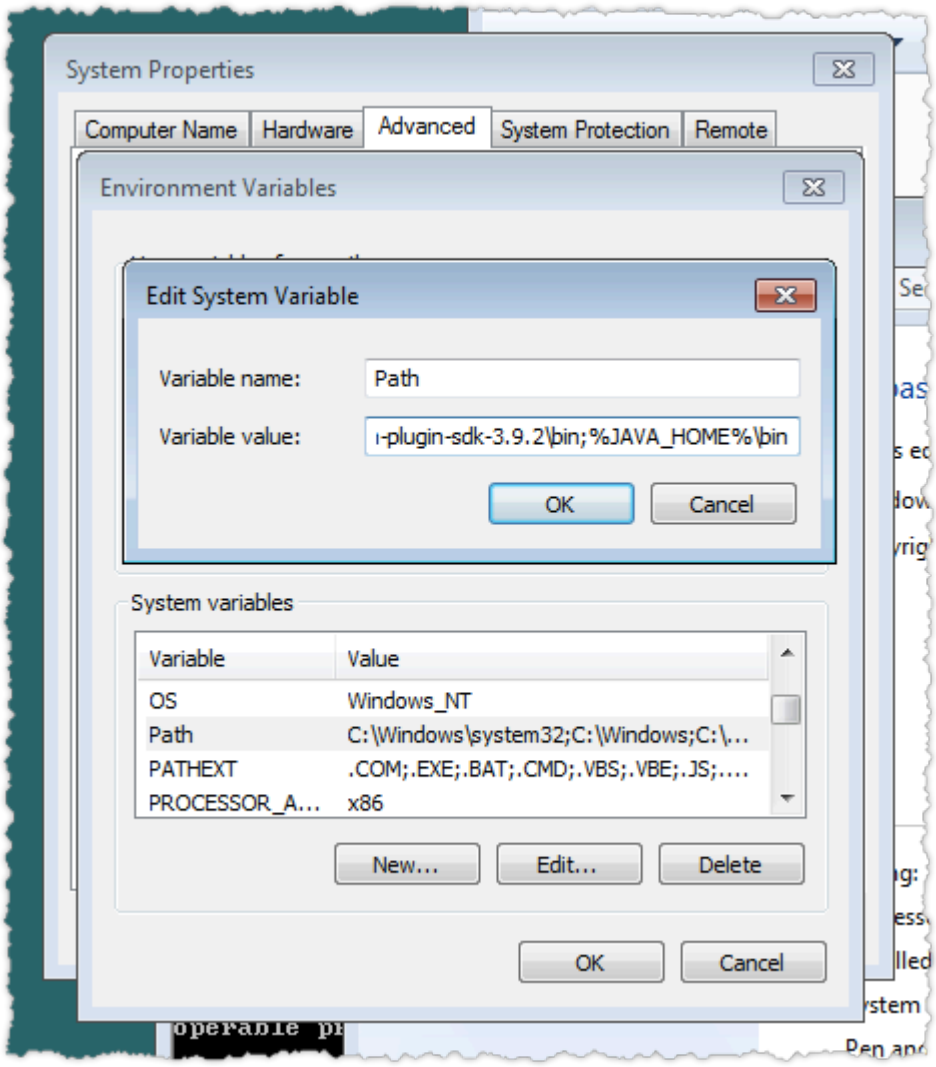
Step 3: Add JAVA_HOME\bin to your PATH

You should add the JDK's bin directory to your `PATH` environment variable as well. This ensures your environment is configured and can locate the `javac` command.

1. Right-click the **My Computer** icon on your desktop.
   The right-click menu displays.
2. Choose **Properties**.
3. Click the **Advanced system settings** option.
   The **System Properties** dialog displays.
4. Choose the **Advanced** tab.
5. Click the **Environment Variables** button.
   The **Environment Variables** dialog appears.
6. Locate the **System variables** section
7. Locate and select the `Path` environment variable.
8. Click `Edit`.
   The **Edit System Variable** dialog appears.
9. Move your cursor to the very end of the the Variable value field's entry.
10. Add the following to the end of the line:

```
;%JAVA_HOME%\bin
```

    When you are done the entry will look similar to the following:



11. Press **OK** until you have closed all the dialogs.
12. Open a command window and enter the following command:

```
javac -version
```

Step 4: Check your ports

The tutorial makes a modification to the JIRA application. The JIRA application runs in an Apache Tomcat web server. The server is installed and configured automatically for you when you work through this tutorial. However, the SDK uses default port settings for each application, including JIRA.

The table below shows the applications currently supported by the Atlassian Plugin SDK, the default port, and the product key for each host application.

Atlassian Application	Default Port	Product Key	Caveats
Bamboo	6990	bamboo	
Confluence	1990	confluence	
Crowd	4990	crowd	
Crucible	3990	fecru	

FishEye	3990	fecru	
JIRA	2990	jira	Plugins developed for versions of JIRA before 4.0 are supported, but using the SDK with versions of JIRA earlier than 4.0 is not. For developing plugins for JIRA 3.13 and earlier, take a look at the JIRA Documentation Archives.
RefApp	5990	refapp	
Stash	7990	stash	

For the purposes of this tutorial, make sure port 2990 is available on your system. Unless you've been adjusting ports on your system, it is most likely this port is available. You can use the `netstat` command to find out if any processes are currently using this port:

```
C:\Users\manthony>netstat -an |find /i "2990"

C:\Users\manthony>
```

If `netstat` is not installed or you are unsure, you can ask your system administrator.

Next steps

At this point, have set up all the prerequisites you need to install the Atlassian SDK on a Windows system. In the next page, you set up the Atlassian SDK and run a quick test.

Set up the SDK Prerequisites for Linux or Mac

Before you install the Atlassian software developer kit (SDK), you must make sure your system has the prerequisite software, that your environment is configured, and that the systems you need to use the SDK exist. This page contains the following sections:

Windows User?

See this page.

Step 1: Verify the Java Developer Kit (JDK) is Installed

The Atlassian SDK relies on 6 (1.6.X) of the Oracle JDK. Before installing the SDK, verify that you have installed the JDK 6 (1.6.X). To do this:

1. Open a terminal window.
2. Verify that the JDK 6 is installed by entering the following at the prompt:

```
myhost:~ manthony$ javac -version
javac 1.6.0_31
```

   You should see output simliar the what is shown above. The version should be 1.6 or higher.
3. If you have installed JDK 1.6 and your `PATH` is configured, skip the rest of this procedure.
4. If you don't have JDK 1.6.X installed, follow the Oracle website instructions for installing it.

> ⚠ If you are a Mac OSX user, you cannot obtain the JDK directly from Oracle. See this post on Stack Overflow.

Step 2: Configure your shell environment (optional)

This is optional. If, after installing the JDK according to Oracle's instructions, your system can locate the javac command with the following:

```
javac -version
```

Skip this section and go to Remove ATLAS_HOME\bin from PATH Settings (Optional).

If You Installed the JDK but Your Environment Can't Find the javac Command

The `JAVA_HOME` environment variable specifies the location of the JDK on your system. On Mac OS X, if you accepted the defaults when you installed the JDK, this is `/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home`. On Linux, it may be `/usr/local/jdk`, or a similar location. You should add the JDK's `bin` directory to your `PATH` environment variable as well. This ensures your environment is configured and can locate the `javac` command.

To configure these two values, do the following:

1. Edit the `.bashrc` file in your home directory using your favorite editor.

```
myhost:~ manthony$ vi ~/.bash_profile
```

2. Add the following three lines at the end of the file.

```
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Conten
ts/Home
export JAVA_HOME
export PATH=$PATH:$JAVA_HOME/bin
```

3. Save and close the file.
4. Enter the following at the command line to pick up your changes:

```
myhost:~ manthony$ source ~/.bash_profile
```

5. Verify that the `JAVA_HOME` variable is set by entering the following at the prompt:

```
myhost:~ manthony$ env | grep JAVA_HOME
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Conten
ts/Home
```

6. Verify your `PATH` includes the JDK `bin` by entering `javac -version` at the command line:

```
myhost:~ manthony$ javac -version
javac 1.6.0_31
```

Step 3: Remove ATLAS_HOME\bin from PATH Settings (optional)

Only do this step if you have installed a 3.X version of the Atlassian Plugin SDK or older. For example, you need to do this step if you installed version 3.11 of the Atlassian Plugin SDK.

1. Check if the Atlassian Plugin SDK `bin` directory is on your path:

```
echo $PATH | grep atlassian
```

2. If the command returns you to the prompt without finding a match, you are all done. Skip the rest of this procedure and go to Step 4. Check your ports.
3. Edit your shell profile files.
   For example, if your shell is Bash your `.bash_profile` executes whenever you log into your system and the `.bashrc` is read and executed when you start a subshell.
4. Remove the Atlassian Plugin SDK `bin` directory from your `PATH` declaration.
5. Close and save your shell file.
6. Start a new terminal and verify the SDK bin is no longer in your path.

Step 4: Check your ports

The tutorial makes a modification to the JIRA application. The JIRA application runs in an Apache Tomcat web server. The server is installed and configured automatically for you when you work through this tutorial. However, the SDK uses default port settings for each application, including JIRA.

The table below shows the applications currently supported by the Atlassian Plugin SDK, the default port, and the product key for each host application.

Atlassian Application	Default Port	Product Key	Caveats
Bamboo	6990	bamboo	
Confluence	1990	confluence	
Crowd	4990	crowd	
Crucible	3990	fecru	
FishEye	3990	fecru	
JIRA	2990	jira	Plugins developed for versions of JIRA before 4.0 are supported, but using the SDK with versions of JIRA earlier than 4.0 is not. For developing plugins for JIRA 3.13 and earlier, take a look at the JIRA Documentation Archives.
RefApp	5990	refapp	
Stash	7990	stash	

For the purposes of this tutorial, make sure port 2990 is available on your system. Unless you've been adjusting ports on your system, it is most likely this port is available. You can use the `netstat` command to find out if any processes are currently using this port:

```
netstat –lp --inet
```

If `netstat` is not installed or you are unsure, you can ask your system administrator.

Next steps

At this point, have set up all the prerequisites you need to install the Atlassian SDK on a Linux or Mac system. In the next page, you set up the Atlassian SDK and run a quick test.

Install the Atlassian SDK on a Windows System

On this page, you install the SDK on your Windows System. You also configure your operating system to recognize the SDK commands in your environment. Before working through the procedures on this page, make sure you have already Set up the SDK Prerequisites on a Windows System. This page contains the following sections:

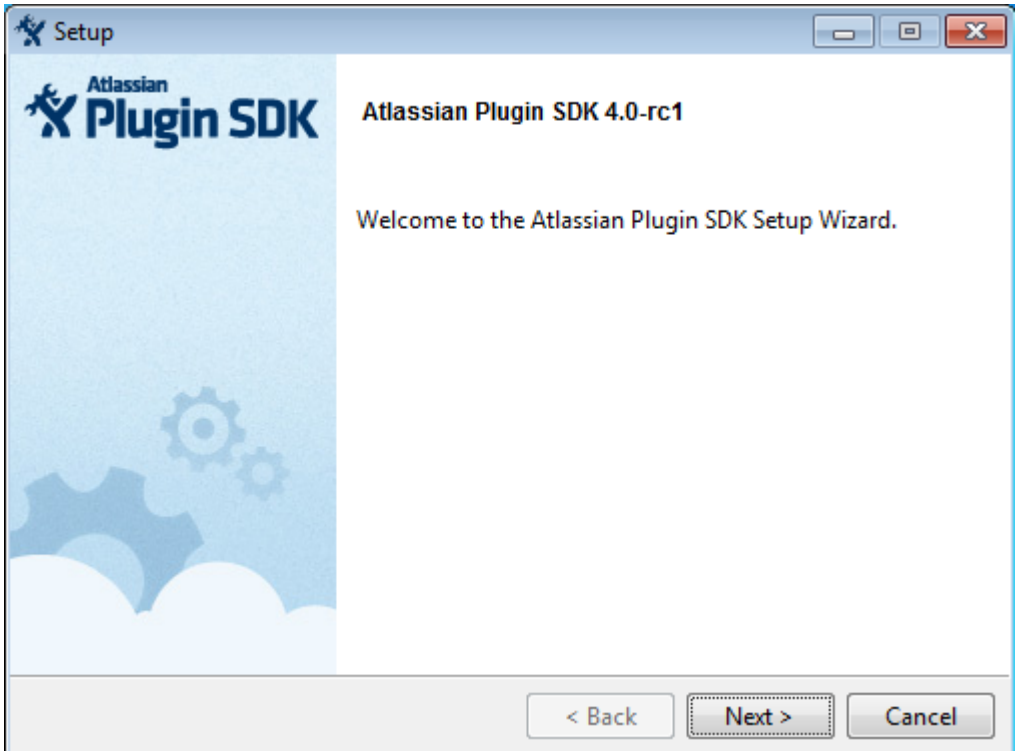
Linux or Mac User?

See this page.

Step 1: Download and Install the SDK

Atlassian maintains the latest and older versions of the SDK on a public Maven repository. To download and install the latest version of SDK, do the following:

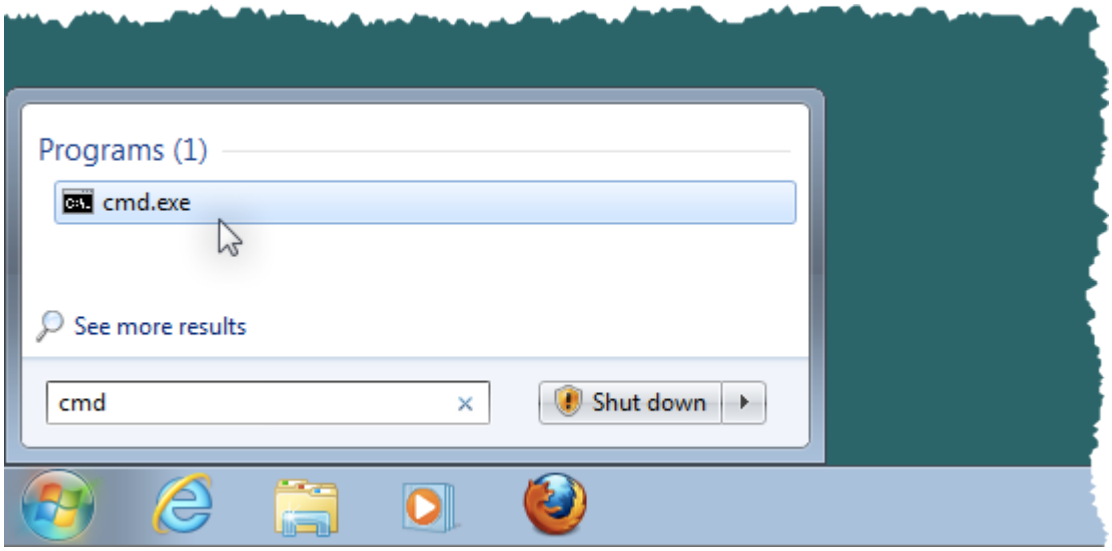
1. Download the SDK.
   Your browser downloads the EXE to a folder on your local system.
2. Locate the downloaded SDK installer.
3. Double-click the installer EXE file to launch it.



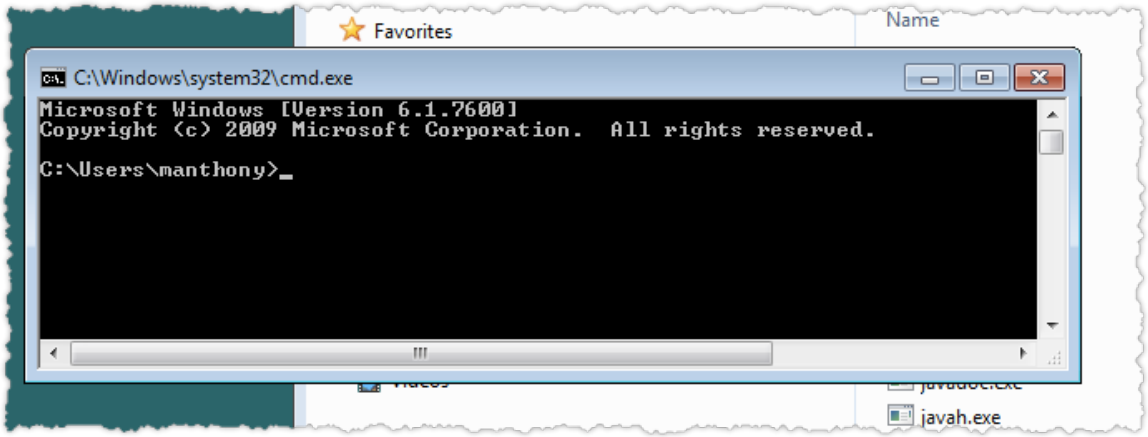
   The version you see on your installer may be different.
4. Follow the prompts to install the SDK.
5. After the installation completes, the installer prompts you to restart your computer.

Step 2: Verify that the SDK is Configured Correctly

1. Locate the **cmd** program in the Windows **Start** menu.



2. Click on the **cmd** menu item.
   The system displays a Windows command window:



3. Open a command window and enter the `atlas-version` command.
   The system responds with information similar to the following:

```
C:\Users\manthony>atlas-version

ATLAS Version:    4.0
ATLAS Home:       C:\Users\manthony\atlassian-plugin-sdk-4.0
ATLAS Scripts:    C:\Users\manthony\atlassian-plugin-sdk-4.0\bin
ATLAS Maven Home:
C:\Users\manthony\atlassian-plugin-sdk-4.0\apache-maven
--------
Executing:
"C:\Users\manthony\atlassian-plugin-sdk-4.0\apache-maven\bin\mvn.
bat" --version
Apache Maven 2.1.0 (r755702; 2009-03-18 12:10:27-0700)
Java version: 1.6.0_32
Java home: C:\Program Files\Java\jdk1.6.0_32\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7" version: "6.1" arch: "amd64" Family: "windows"
```

   If the Java version is not correct or not found, make sure you have Set up the SDK Prerequisites on a Windows System correctly.
4. Verify you are running the version of Maven that comes with the SDK.
   To do this, review the output of the `atlas-version` command and make sure the `mvn` executable is

called from the SDK.

The Next Step

You've installed the SDK and made sure your environment recognizes the commands. In the next step, you go on to explore the SDK by trying some of the commands.

Have a comment or question on this page? Ask on Atlassian Answers.

Last searched Atlassian Answers at 7:17 PM.

Install the Atlassian SDK on a Linux or Mac System

On this page, you install the SDK on your Linux or Mac system. You also configure your operating system to recognize the SDK commands in your environment. Before working through the procedures on this page, make sure you have already Set up the SDK Prerequisites for Linux or Mac. This page contains the following sections:

Windows User?

See this page.

Step 1: Download and Install the SDK

Atlassian maintains the latest and older versions of the SDK on a public Maven repository. Atlassian offers native installers for most operating systems as well as a TGZ (GZipped tar file).

Mac OSX Installer

Click here to view the instructions...

1. Download the PKG file.
2. Double click the PKG file to launch the installer.
3. Follow the prompts to complete the installation.

Homebrew

Click here to expand...

If you use Homebrew to manage packages on OSX, add the Atlassian "Tap" to your Brew:

```
brew tap atlassian/tap
```

Then install the SDK via the atlassian/tap:

```
brew install atlassian/tap/atlassian-plugin-sdk
```

Debian /Ubuntu Linux

Click here to view the instructions...

On a Debian based Linux like Ubuntu, you can install the Atlassian Plugin SDK using `apt-get` or `aptitude`, but first you have to set up the Atlassian repositories. You only need to do this once and it requires `sudo` privileges. To set up the Atlassian repositories,

1. Open a terminal.
2. Enter the following at the prompt.

```
sudo sh -c 'echo "deb https://sdkrepo.atlassian.com/debian/
stable contrib" >>/etc/apt/sources.list'
```

3. After the prompt returns, add the public key:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv-keys B07804338C015B73
```

Once you have set up the Atlassian repositories, enter the following to install the SDK:

```
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install atlassian-plugin-sdk
```

First line makes sure that you have `apt-transport-https` package installed to make `apt` be able to

access https repositories.

You can also use the same `apt` tools to upgrade the SDK to a new version when it's available.

Centos/Fedora Yum Installation

Click here to view the instructions...

1. Download the repo files to your `/etc/yum.repos.d/` folder:

```
cd /etc/yum.repos.d/
sudo wget http://sdkrepo.atlassian.com/atlassian-sdk-stable.repo
```

   You only need to do this step once. Subsequent Yum installs of the SDK you can skip this step.
2. Install using the following three commands:

```
sudo yum clean all
sudo yum updateinfo metadata
sudo yum install atlassian-plugin-sdk
```

.tgz File Installation

Click here to view the instructions...

To install the latest version of SDK, do the following:

1. Download a TGZ (GZipped tar file) of the SDK.
   Your browser downloads the jar to a folder on your local system.
2. Locate the downloaded SDK file.
3. Extract the file to your local directory.

```
sudo tar -xvzf atlassian-plugin-sdk-4.0.tar.gz -C /opt
```

4. Rename the extracted folder to `atlassian-plugin-sdk`.

```
sudo mv /opt/atlassian-plugin-sdk-4.0 /opt/atlassian-plugin-sdk
```

   If you are comfortable working with symbolic links, you can set up a symbolic link instead of renaming the directory.

Step 2: Verify that you have set up the SDK correctly

1. Open a terminal window.
2. Enter `atlas-version` command at the prompt.
   The system responds with information similar to the following:

```
ATLAS Version:    4.0
ATLAS Home:       /usr/share/atlassian-plugin-sdk-4.0
ATLAS Scripts:    /usr/share/atlassian-plugin-sdk-4.0/bin
ATLAS Maven Home: /usr/share/atlassian-plugin-sdk-4.0/apache-maven
--------
Executing: /usr/share/atlassian-plugin-sdk-4.0/apache-maven/bin/mvn
--version
Apache Maven 2.1.0 (r755702; 2009-03-18 12:10:27-0700)
Java version: 1.6.0_33
Java home:
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x" version: "10.7.4" arch: "x86_64" Family: "mac"
```

If the Java version is not set properly, make sure you have followed the instructions to Set up the SDK Prerequisites for Linux or Mac.

3. Verify you are running the version of Maven that comes with the SDK.
   To do this, review the output of the `atlas-version` command and make sure the `mvn` executable is called from the SDK.

The Next Step

You've installed the SDK and made sure your environment recognizes the commands. In the next step, you go on to explore the SDK by trying some of the commands.

Explore the Installed SDK and the atlas Commands

You should understand the basic contents and functions of the SDK before you develop your first add-on. On this page, you get a summary of the SDK contents, a very brief intro to Apache Maven, and you run an Atlassian application through the SDK. The following topics are covered:

Step 1: Browse the SDK contents and get introduced to Maven

Maven is a popular tool and you may already be using it. You don't need to be a Maven expert to use this tutorial. You do need to understand some Maven basics to develop with the SDK. Examine the contents of the SDK installation:

1. Navigate to the directory where the Atlassian SDK (the ATLAS_HOME) was installed.
   Not sure where the SDK was installed? Use the `atlas-version` command. If you are on a Mac or other Linux system you may need to preface your command with `sudo`.
2. List the contents of the `ATLAS_HOME` directory.
   The SDK contains the following directories:

Directory	Description
`apache-maven`	Contains the Apache Maven 2 installation. Maven is a popular build management tool. Atlassian built a suite tools around Maven called the Atlassian Maven Plugin Suite (AMPS). These tools automate many add-on development tasks.
`bin`	Contains command-line wrappers for the most common AMPS calls. All of the commands start with the `atlas-` prefix. In this tutorial, you do all your development with the wrapper commands. This is the typical case. Only very rarely, and then only for very advanced techniques, should you need to work with AMPS directly.
`repository`	Contains code repositories that the Atlassian SDK is dependent on. This directory is prepopulated during the SDK installation.

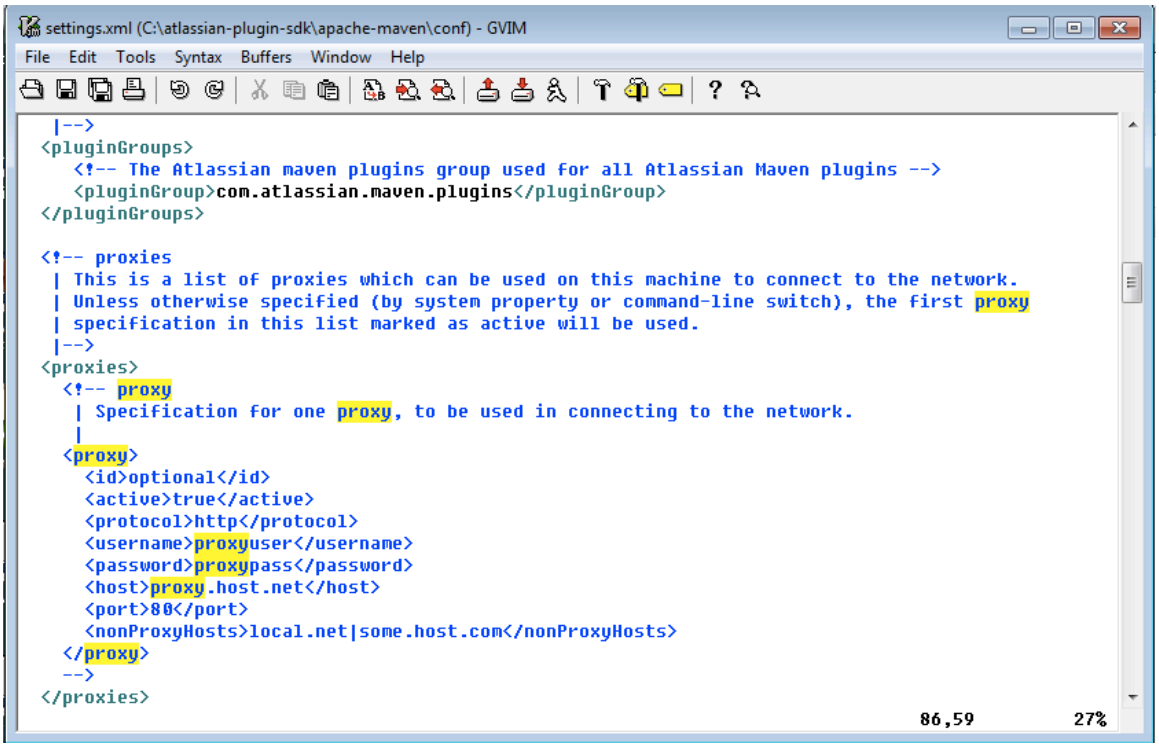
3. Take some time to list the contents of each of the three sub-directories.

Maven relies on the ability to navigate to external repositories URLs and obtain source packages required by your code. When you build an add-on such as a plugin using the SDK's `atlas-` commands, Maven is behind the scenes retrieving the latest Atlassian source files from Atlassian's public repositories. Atlassian preconfigures the location of these repositories in the `ATLAS_HOME/apache-maven/conf/settings.xml` file.

Step 2: (Optional) Configure a <proxy> in the Maven `settings.xml`

If you're doing your development behind a corporate firewall, you may need to connect to the Internet through an HTTP proxy. If you know you do not have a firewall, skip this procedure. If you know your company requires you to use an HTTP proxy, you need to configure proxy settings in the `ATLAS_HOME/apache-maven/conf/settings.xml` file. Do the following:

1. Ask your system administrator to provide you with the following information:
   - The proxy address `protocol://host:port`, for example `https://our.company:8080`.
   - a username/password required for access
   - a list of hosts that don't require a proxy
2. Edit the `ATLAS_HOME/apache-maven/conf/settings.xml` file in your favorite editor.
3. Locate the `<proxies>` section.



4. Uncomment the `proxy` element.
5. Edit the `proxy` element and add the values provided by your system administrator.
   When you are done, the element looks similar to the following:

```
<proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
```

6. Close and save the file.

Have an Existing settings.xml File in Your .m2 Directory?

If you have an existing local `settings.xml` file, you may encounter problems resolving dependencies required by the SDK commands. To prevent these problems, add the following `<pluginRepository>` block to your local `settings.xml` profile:

```
<pluginRepository>
    <id>atlassian-plugin-sdk</id>
    <url>file://${env.ATLAS_HOME}/repository</url>
     <releases>
        <enabled>true</enabled>
        <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
 </pluginRepository>
```

Step 3: Try an atlas command

At this point, you've configured your environment and you are set to begin developing. This tutorial shows you how to add a very simple `helloworld` plugin to JIRA. Before you begin writing code, you should use an `atlas` command to run JIRA in standalone mode. You use the atlas-run-standalone command to start JIRA.

If you haven't already done so, open a command window and do the following:

1. Navigate to the root of your drive (Windows) or home directory (Linux).

Windows	Linux/Mac
`cd C:`	`cd ~`

2. Create a directory called `atlastutorial`.

```
mkdir atlastutorial
```

3. Change directory to your newly created directory.

```
cd atlastutorial
```

4. Start latest version of JIRA on default port of 2990, by entering the following:

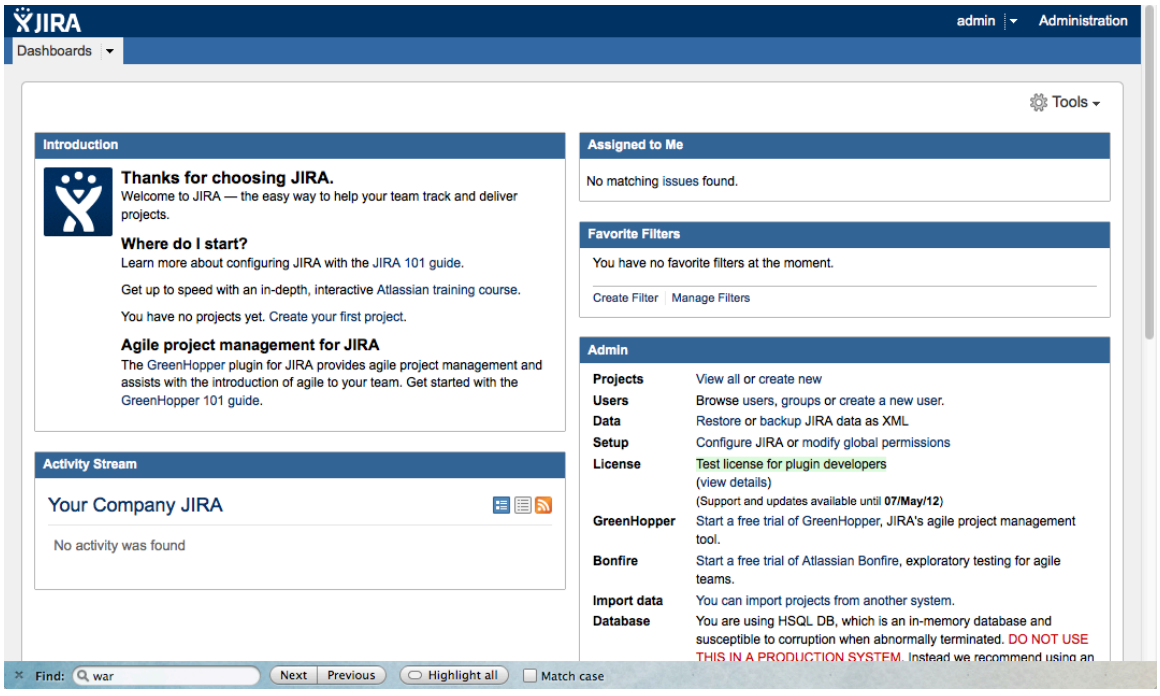
```
atlas-run-standalone --product jira
```

After the command completes successfully you see a message similar to the following:

```
[INFO] Starting jira on the tomcat6x container on ports 2990 (http)
and 52641 (rmi)
[INFO] using codehaus cargo v1.2.3
[INFO] [cargo:start]
[INFO] [2.ContainerStartMojo] Resolved container artifact
org.codehaus.cargo:cargo-core-container-tomcat:jar:1.2.3 for
container tomcat6x
[INFO] [stalledLocalDeployer] Deploying
[/Users/manthony/atlastutorial/amps-standalone/target/jira/jira.war
] to
[/Users/manthony/atlastutorial/amps-standalone/target/container/tom
cat6x/cargo-jira-home/webapps]...
[INFO] [talledLocalContainer] Tomcat 6.x starting...
[INFO] [talledLocalContainer] Tomcat 6.x started on port [2990]
[INFO] jira started successfully in 249s at
http://localhost:2990/jira
[INFO] Type Ctrl-D to shutdown gracefully
[INFO] Type Ctrl-C to exit
```

The output message tells you the URL where JIRA was started.

5. Open a browser and enter the JIRA URL.
   You should see the URL for your installation displayed in the run output. For example, you might see `http://myhost.local:2990/jira` or `http://localhost:2990/jira` depending on your environment. On successful launch, the browser displays the JIRA login page.
6. Enter `admin` for both the username and password.
   Your browser displays the JIRA dashboard:



You'll notice in the **Admin** section the **License** that you are running is a `Test license for plugin developers`.

Go behind the scenes of atlas-run-standalone

When you issue the `atlas-run-standalone` command, it creates a directory called `amps-standalone` in the current directory – in this case `atlastutorial/amps-standalone`.

JIRA runs as a Tomcat web application on your `localhost` — the system on which you issue the command. So, the command actually runs a Maven build on your system. This build

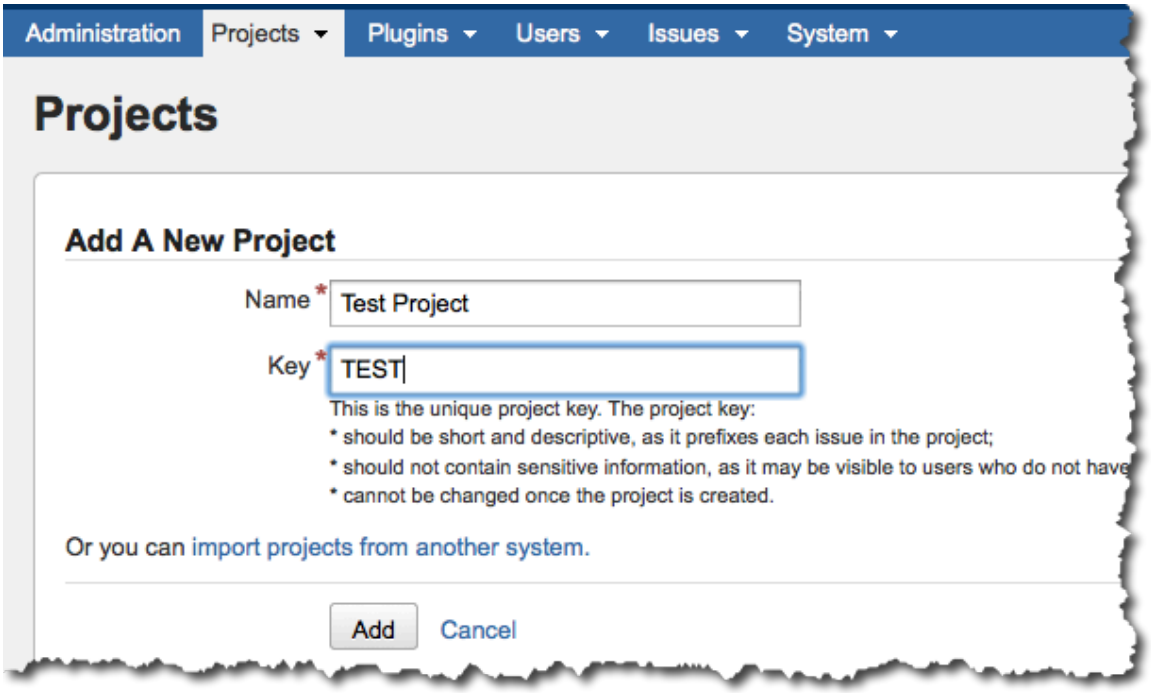
- creates a directory called `amps-standalone/target` in the current `atlastutorial` directory
- downloads the latest JIRA product files to your machine
- downloads any dependencies JIRA needs to run this version of JIRA
- builds the JIRA application
- creates a Tomcat 6 container
- deploys JIRA in the container

The `amps-standalone/target/jira-5.0.log` file contains the details such as the environment variables set and used during this process. You'll notice from this log file that the JIRA test instance relies on an HSQL database. You would never use an HSQL database in a production system.

Step 4: State is saved between runs

When you run a standalone instance, you can make changes in the instance. The system remembers what you do between sessions. At this point, you should still have your standalone JIRA instance up and running. Make sure you are logged in and do the following:

1. Locate the Admin section on Dashboard.
2. Choose the **create new** link from the **Projects** section.
   The system opens the **Create New Project** page.
3. Name your project `Test Project` and give it a key of `TEST`.
   When you are done, the dialog looks similar to the following:



4. Click **Add** to add the project.
   Your standalone JIRA now has a TEST project. Test what happens to the project when you restart this standalone instance.
5. Return to the command line where you started JIRA.
6. Gracefully shutdown JIRA by pressing CTRL-Z (Windows) or CTRL-D (Linux).
7. Browse to the default JIRA URL (`http://myhost.local:2990/jira` or `http://localhost:2990/jira`) with your browser.
   Your browser should inform you that it could not get a connection to the server.
8. Return to the command line (DOS prompt for Windows users).

9. Make sure you are in the `atlastutorial` directory.
10. Restart JIRA standalone.

```
atlas-run-standalone --product jira
```

11. Log back in and locate your project.
    The standalone instance has retained the data you created between restarts. This can be very useful when testing. You'll learn more about this later.

Extra Exploration

You can use the `atlas-run-standalone` command to run a particular version of a product. This is useful if, for instance, you want to test your code on an earlier version of JIRA or Confluence. Review the command's reference page and try running an earlier version of JIRA.

Next steps

At this point, you have some basic understanding of the SDK. Enough to go ahead in the next tutorial section to create your own plugin project.

Create a HelloWorld Plugin Project

At this point, you have set up your environment and run a test with a standalone version of JIRA. In this section, you learn how to create a plugin module. For now, you are just going to use the command line tools. Later, you'll learn how to run the tools from an IDE. This page contains the following sections:

Step 1: Create your first project

If you haven't already done so, go ahead and open a terminal window and then do the following:

1. Change directory to the `atlastutorial` folder you created earlier at the root of your home directory.
2. Once you are in the folder, enter the `atlas-create-jira-plugin` command to create the plugin.

```
atlas-create-jira-plugin
```

   The command prompts you for the JIRA version.
3. Enter `1` for `JIRA 5` and press RETURN.
   The command prompts you for the basic information each plugin needs.
4. Respond to the prompts using the information in the following table:

Define value for groupId	`com.atlassian.tutorial`
Define value for artifactId	`helloworld`
Define value for version	`1.0-SNAPSHOT`
Define value for package	`com.atlassian.tutorial.helloworld`

   The system prompts you to confirm the configuration you entered:

```
Confirm properties configuration:
groupId: com.atlassian.tutorial
artifactId: helloworld
version: 1.0-SNAPSHOT
package: com.atlassian.tutorial.helloworld
```

5. Press `y` to continue.
   The system creates a `helloworld` project folder. This initial project folder contains the basic *skeleton* of a plugin.

Step 2: Examine the contents of the plugin skeleton

With a single command, you have a skeleton plugin project containing the following source:

Contents	Description
`LICENSE`	A placeholder for a license file.
`README`	Simple hints for running the atlas- commands.
`pom.xml`	Maven configuration file for your project.
`src`	The generated source for the plugin.

Take a closer look at the code created for you in the `src` directory. The `src/test/java` contains a generated class and some placeholders for testing your plugin. You'll learn more about this later. The `src/main` folder contains an initial `/com/atlassian/tutorial/helloworld/MyPluginComponent.java` file and a `/com/`

atlassian/tutorial/helloworld/MyPluginComponentImpl.java. The src/main/resources folder contains a single atlassian-plugin.xml file — this file is the descriptor. It defines the *plugin modules* your plugin uses.

At this point, your descriptor file should not define any modules. Let's test this, by looking in the descriptor file:

1. Open your favorite text editor.
2. Browse to and open the atlastutorial/helloworld/src/main/resources/atlassian-plugin.xml file.
   At this point the contents of the file should look like this:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
name="${project.name}" plugins-version="2">
    <plugin-info>
        <description>${project.description}</description>
        <version>${project.version}</version>
        <vendor name="${project.organization.name}"
url="${project.organization.url}" />
        <param name="plugin-icon">images/pluginIcon.png</param>
        <param name="plugin-logo">images/pluginLogo.png</param>
    </plugin-info>
    <!-- add our i18n resource -->
    <resource type="i18n" name="i18n" location="helloworld"/>

    <!-- add our web resources -->
    <web-resource key="helloworld-resources" name="helloworld Web Resources">
        <dependency>com.atlassian.auiplugin:ajs</dependency>

        <resource type="download" name="helloworld.css"
location="/css/helloworld.css"/>
        <resource type="download" name="helloworld.js"
location="/js/helloworld.js"/>
        <resource type="download" name="images/" location="/images"/>
        <context>helloworld</context>
    </web-resource>

    <!-- publish our component -->
    <component key="myPluginComponent"
class="com.atlassian.tutorial.helloworld.MyPluginComponentImpl" public="true">

<interface>com.atlassian.tutorial.helloworld.MyPluginComponent</interface>
    </component>

    <!-- import from the product container -->
    <component-import key="applicationProperties"
interface="com.atlassian.sal.api.ApplicationProperties" />

</atlassian-plugin>
```

   Several of the entries in the <plugin-info> should look familiar. If you recall, the atlas-create-jira-plugin command asked you for a groupId and an artifactId. These values landed in another file all together.

   Your plugin includes resources that allow you to control look and feel. These appear under the web resources section. You'll learn more about these in another tutorial. For now, focus on the relatively simple <plugin-info> section.
3. Close the atlassian-plugin.xml file.

4. Open the `atlastutorial/helloworld/pom.xml`.
   This file is a Maven project object model file. This file contains project and dependency information that Maven uses to build your plugin. This tutorial isn't going into the finer points of Maven or its files.
5. Search for the `artifactId` value.
   You should find the `artifactId` you entered when you created the skeleton. The `project.artifactId` you saw in the `atlassian-plugin.xml` file references this value.
6. Familiarize yourself with the file a bit by looking for other values you entered through the command such as the `groupId` and `version`.
7. Close the file when you are done.

Step 3: Load the helloworld plugin into JIRA

Even though you haven't written any code, you can still load the skeleton plugin into JIRA. When you load a plugin into JIRA, it is visible in the Universal Plugin Manager (UPM). The UPM is in every Atlassian application. It allows you to install, view, and update plugins in your *host application*. The host application in this case is JIRA. You load a plugin using the `atlas-run` command. Try this command and see what happens:

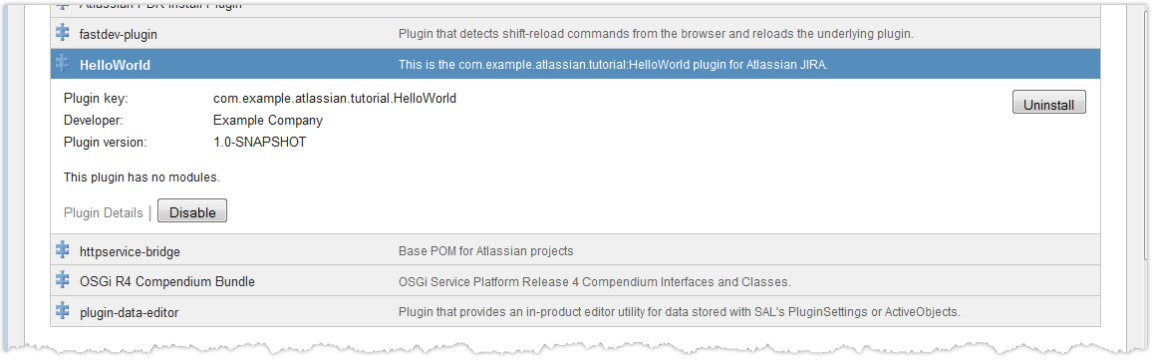
1. Open a command line (DOS prompt for Windows users).
2. Change directory to the root of your plugin project.

   ```
   cd atlastutorial/helloworld
   ```

3. Enter the `atlas-run` command.
   The command creates a `target` sub directory under your project root. You will examine this directory a bit more later. When the command completes successfully, you'll see some output that should be that looks very similar to output of the `atlas-run-standalone` command.

```
[WARNING] [talledLocalContainer] INFO: Deploying web application
archive cargocpc.war
[WARNING] [talledLocalContainer] May 9, 2012 8:15:56 AM
org.apache.coyote.http11.Http11Protocol star
t
[WARNING] [talledLocalContainer] INFO: Starting Coyote HTTP/1.1 on
http-2990
[WARNING] [talledLocalContainer] May 9, 2012 8:15:56 AM
org.apache.jk.common.ChannelSocket init
[WARNING] [talledLocalContainer] INFO: JK: ajp13 listening on
/0.0.0.0:8009
[WARNING] [talledLocalContainer] May 9, 2012 8:15:56 AM
org.apache.jk.server.JkMain start
[WARNING] [talledLocalContainer] INFO: Jk running ID=0 time=0/130
config=null
[WARNING] [talledLocalContainer] May 9, 2012 8:15:56 AM
org.apache.catalina.startup.Catalina start
[WARNING] [talledLocalContainer] INFO: Server startup in 100008 ms
[INFO] [talledLocalContainer] 2012-05-09 08:15:59,176
QuartzWorker-0 INFO ServiceRunner    Backup Se
rvice [jira.bc.dataimport.DefaultExportService] Data export
completed in 781ms. Wrote 622 entities t
o export in memory.
[INFO] [talledLocalContainer] 2012-05-09 08:15:59,186
QuartzWorker-0 INFO ServiceRunner    Backup Se
rvice [jira.bc.dataimport.DefaultExportService] Attempting to save
the Active Objects Backup
[INFO] [talledLocalContainer] 2012-05-09 08:15:59,487
QuartzWorker-0 INFO ServiceRunner    Backup Se
rvice [jira.bc.dataimport.DefaultExportService] Finished saving the
Active Objects Backup
[INFO] [talledLocalContainer] Tomcat 6.x started on port [2990]
[INFO] jira started successfully in 161s at
http://manthony-PC:2990/jira
[INFO] Type Ctrl-D to shutdown gracefully
[INFO] Type Ctrl-C to exit
```

4. Open your browser and log into the JIRA instance.
   (Remember, the username and password are both `admin`.)
5. Select the  cog (**Administration**) icon in the right corner.
6. Choose **Add-ons** from the menu.
   The system places you on the **Administration** page.
7. Choose **Manage Add-ons** from the left-hand menu.
   The system opens the **Manage Add-ons** page.
8. Locate the **helloworld** plugin listings in **User-installed Add-ons** category.
   You'll find two listings for your plugin. One for the plugin itself and one for the plugin tests.
9. Expand each entry by click it.
   You should see the following for the plugin alone:



The plugin has just the basic modules that you get for "free" when you create a plugin. These modules don't do much yet.

10. Close the JIRA browser window.
11. Return to the terminal window where you started `atlas-run` and shutdown the process with `CTRL-D`.

Step 4: Make a change and see it reflected in the application

In this step, you'll make a small change in your plugin's `pom.xml` file. Then, you'll rebuild your plugin with the `atlas-run` command.

1. Open the `pom.xml` file in your favorite editor.
2. Locate the `<organization>`element.
3. Update the element to look like the following:

```
<organization>
    <name>HelloGoodby Inc.</name>
    <url>http://www.helloworldgoodbye.com/</url>
</organization>
```

4. Save and close the file.
5. Run the `atlas-run` command in the terminal window.
6. Open the plugins page in your browser using the following path http://localhost:2990/jira/plugins/servlet/upm#manage.
   JIRA still prompts you for the username/password.
7. Expand the **helloworld** plugin to see your changes.

The Next Steps

So far, you've used the SDK from a command line. However, most programmers working in complex code prefer the help of an integrated development environment (IDE). One of the most popular IDE is Eclipse. In the next section, you install and configure the Eclipse IDE on Windows or for Linux/Mac. If you are using IntelliJ, please see this page.

Set Up the Eclipse IDE for Windows

So far, you've configured your environment, installed the Atlassian SDK, and created a `HelloWorld` plugin project. You've tweaked the plugin project a bit but so far your plugin does nothing. Of course, you've done all this on the command line --- most programmers prefer the advantages of an IDE. So, on this page, you configure the Eclipse IDE to work with the SDK.

Linux or Mac User?

See this page.

IntelliJ User?

See this page.

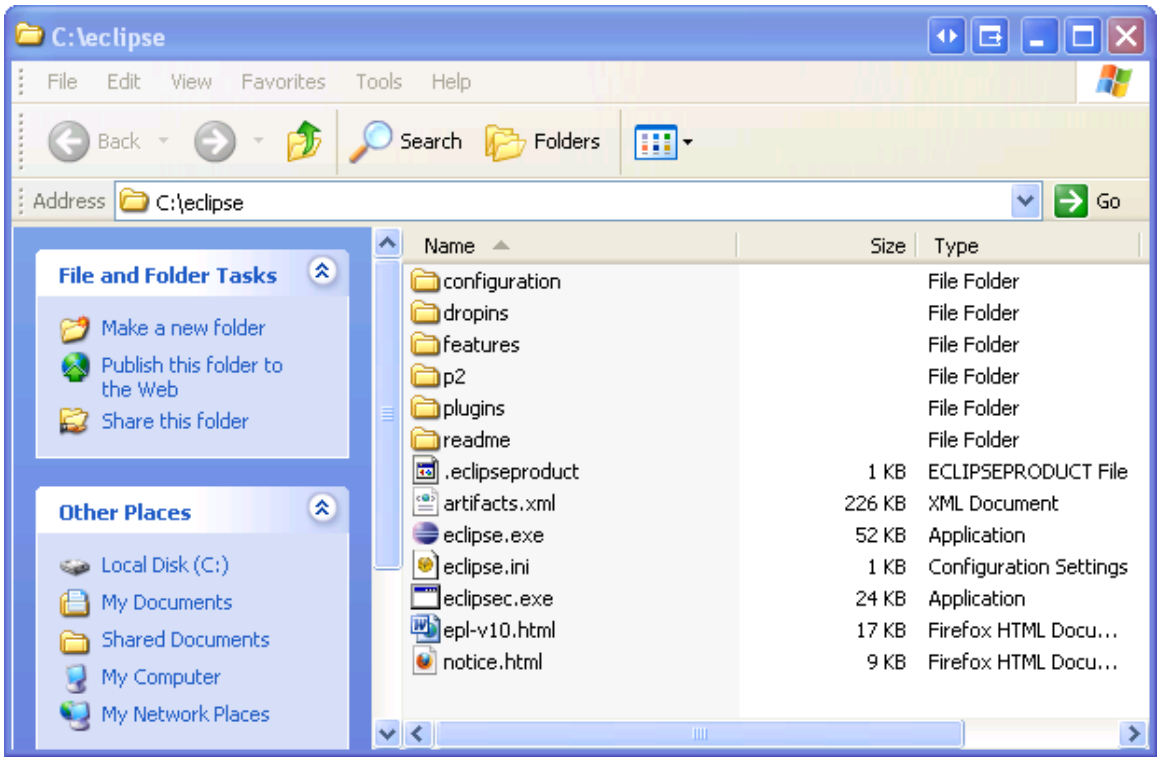
Do I have to Use Eclipse?

If you already have a version of Eclipse installed you don't need to do this step. Note that the tutorial assumes a particular version of Eclipse, so you should anticipate that you may need to adjust procedures if you choose to use another version of it. If you aren't interested in using Eclipse and prefer another IDE, you are perfectly free to skip this page. However, the rest of this tutorial assumes you are using Eclipse.

Step 1: Install the Eclipse IDE

In this step, you download and install the Eclipse IDE for Java EE Developers (Indigo). This version of Eclipse comes with most of the Maven dependencies already installed. Do the following to install Eclipse in your system:

1. Download the Eclpse IDE for Java EE developers.
   This IDE has many of the dependencies required by the Maven Eclipse plugin.
2. Expand the ZIP file into the root of your hard drive.
   When you are done, if your hard drive root is the C:\ drive you will have the following folder on your hard drive:



Step 2: Configure the Eclipse Plugin to start under JDK 1.6

In this step, you edit the Eclipse initialization file. Do the following:

1. Make a note of the location of your JDK 1.6 installation.
   Your root should be similar to: `C:\Program Files\Java\jdk1.6.0_32`
2. Navigate to the root of the Eclipse installation.
3. Make a copy of the `eclipse.ini` file.
   This is good practice any time you are about to edit a configuration file. It allows you to get back to the

original if you make a mistake.

4. Name the copy `eclipse.ini.original`.
5. Edit the `eclipse.ini` file with your favorite text editor.
6. Add a `-vm` entry to file before any `-vmargs` entry.
   The entry should point to the `bin` directory of your JDK. The `eclipse.ini` file requires that you reverse the slashes from back to forward slashes. When you are done the file will look similar to the following:

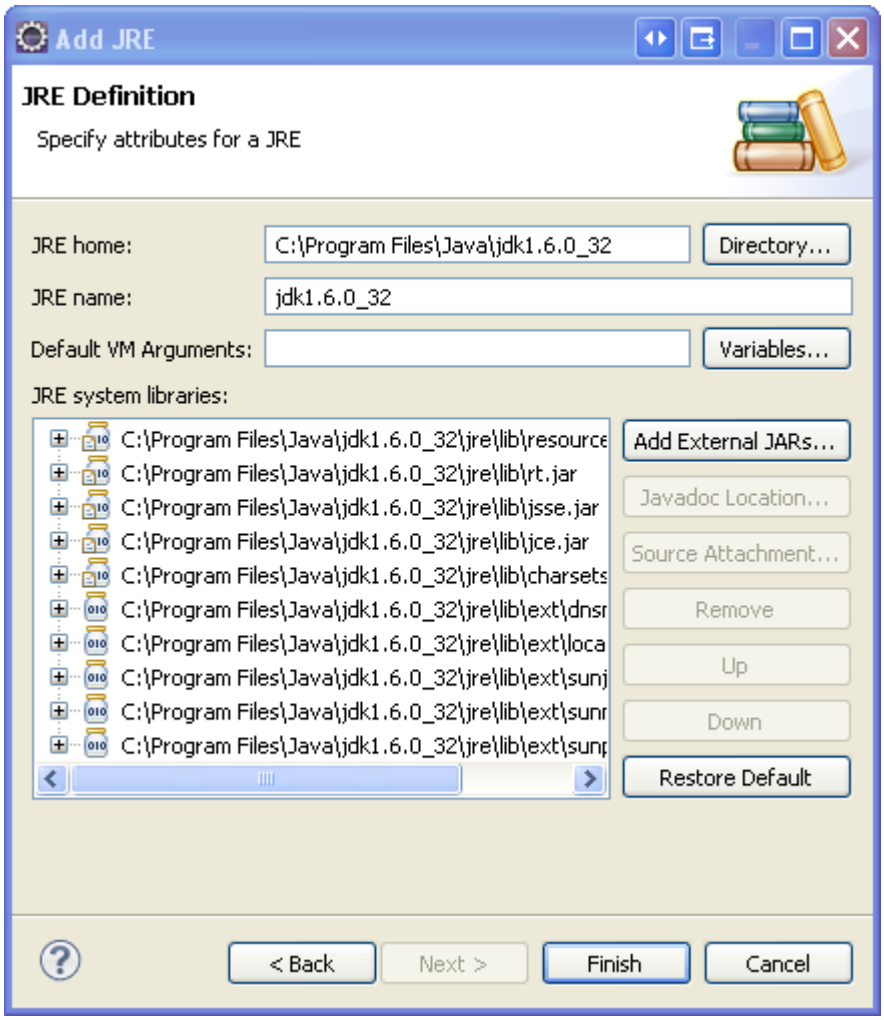
```
-startup
plugins/org.eclipse.equinox.launcher_1.2.0.v20110502.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.1.100.v20110
502
-product
org.eclipse.epp.package.jee.product
--launcher.defaultAction
openFile
--launcher.XXMaxPermSize
256M
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize
256m
--launcher.defaultAction
openFile
-vm
C:/Program Files/Java/jdk1.6.0_32/bin
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xms40m
-Xmx512m
```

7. Close and save the file.

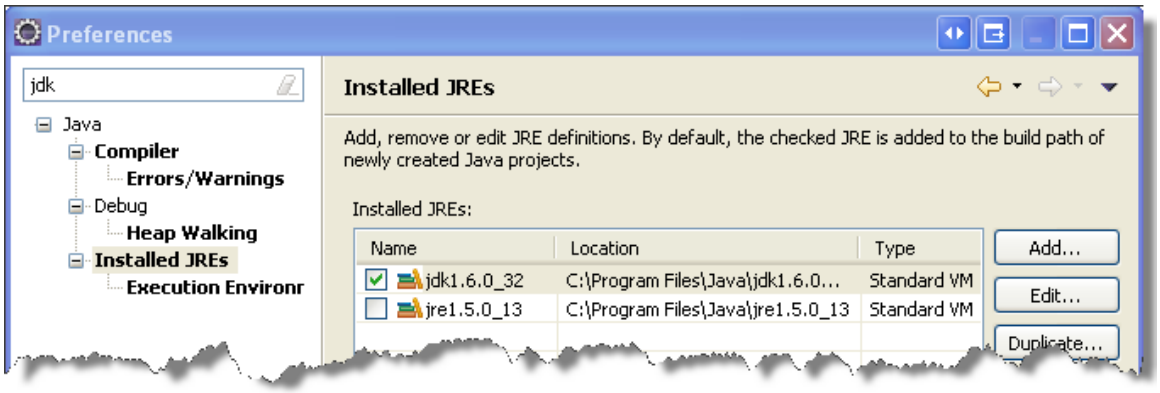
Step 3: Start Eclipse and update the Installed JREs

Start Eclipse and do the following:

1. Choose **Windows > Preferences** from the Eclipse menu bar.
   The system displays the preferences dialog.
2. Filter for or navigate to the **Installed JREs** page.
3. Click the **Add** button.
   The **Add JRE** wizard displays.
4. Make sure **Standard VM** is selected and press **Next**.
5. Press **Directory**.
   The **Browse For Folder** dialog appears.
6. Navigate to your JDK installation.
7. Press **OK**.
   Eclipse locates all the libraries. At this point the dialog should look similar to the following:



8. Press **Finish**.
   The system returns you to the **Installed JREs** page.
9. Check the JDK you just added.
   The dialog should look similar to this:



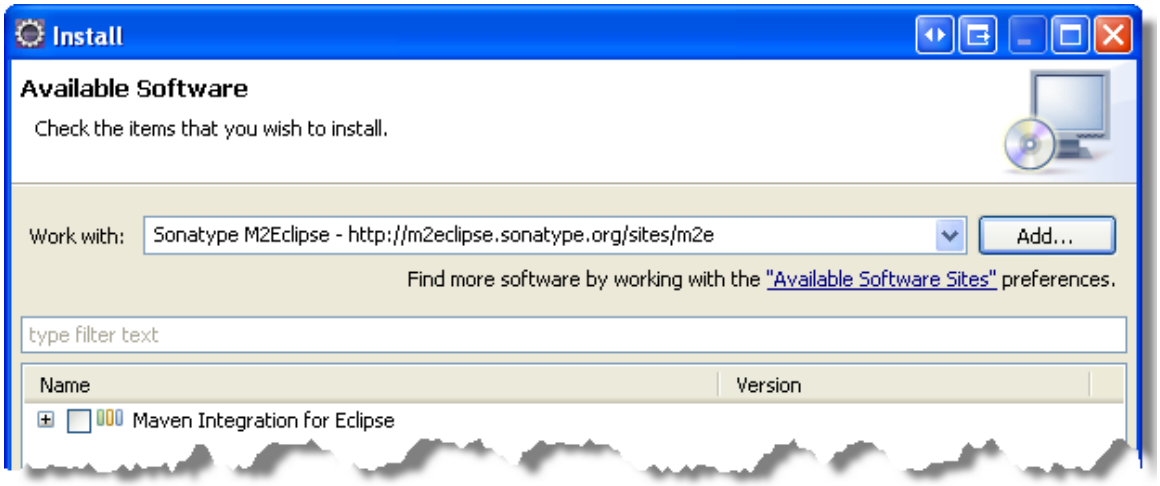
10. Press **OK** to close the dialog.

Step 4: Install the Maven Eclipse Plugin

Start Eclipse and do the following:

1. Choose **Help > Install New Software**.
   The **Available Software** dialog appears.
2. Click the **Add** button.
   The **Add Repository** dialog appears.
3. Enter **Sonatype M2Eclipse** in the **Name** field.

4. Enter `http://download.eclipse.org/technology/m2e/releases` in the **Location** field.
5. Press **OK** to close the dialog.
   The system searches the site for the plugin. After a moment, the **Name** field fills with the **Maven Integration for Eclipse** as the following:

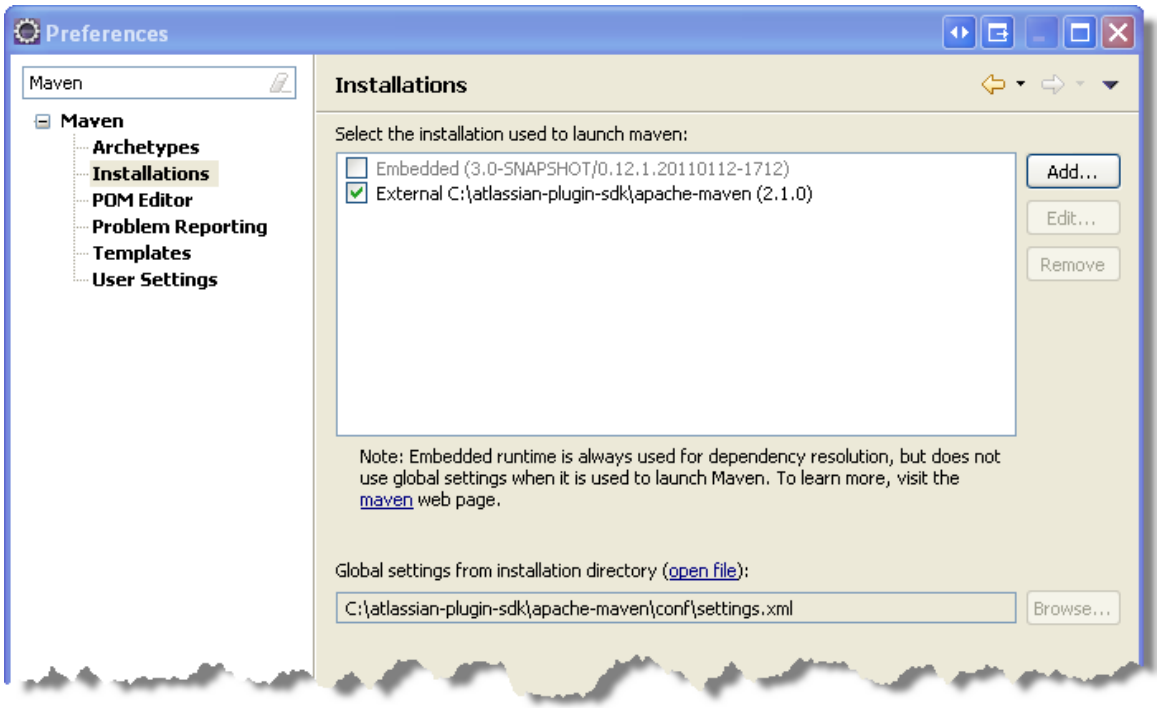


6. Check the box and press **Next**.
7. Select **Maven Integration for Eclipse**.
8. Press **Next** and **Next** again.
9. Accept the terms of the license agreement and press **Finish**.
   Eclipse calculates the dependencies and space.
10. Press **Next**.
11. Accept the License agreement and press **Next**.
    The installation procedure runs.
12. Restart Eclipse when prompted.

Step 5: Configure the Maven Plugin

After the Eclipse restarts, you need to ensure that the M2E plugin is configured:

1. Choose **Windows > Preferences** from the Eclipse menu bar.
   The system displays the preferences dialog.
2. Filter for or navigate to the **Maven > Installations** page.
3. Click the **Add** button.
   The Maven Installation dialog displays.
4. Browse to your `c:\atlassian-plugin-sdk\apache-maven` installation.
5. Press **OK**.
   The system sets this external repository for you. The dialog should look like the following:



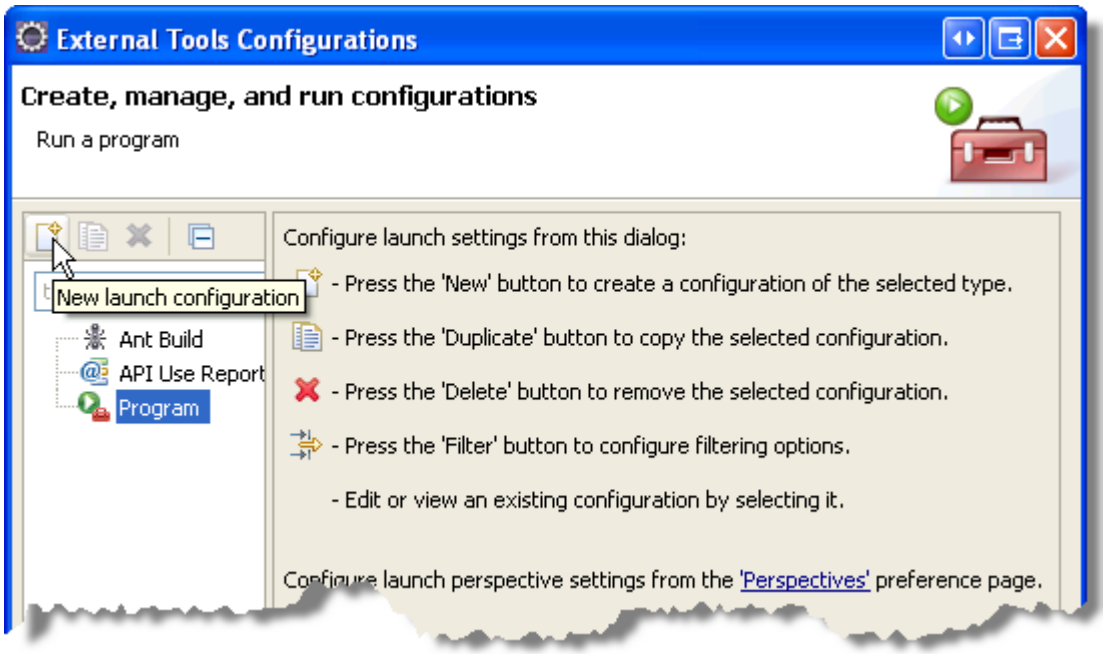
6. Ensure the **Global settings** are coming from the installation directory.
7. Press **Apply**.
8. Click the Maven root.
9. Uncheck Download repository index updates on startup.
   This prevents Maven from updating on Eclipse startup which can be time consuming. The atlas-commands all update the repositories for you.
10. Press **OK** to close the dialog.

Step 6: Set Up a DOS Shell in Eclipse

Once you have Eclipse configured to use the Atlassian SDK, you would still need to keep a DOS command prompt open in which to run each command. This is very handy if you want an "all in one" workspace. In this step, you create an external tool configuration that opens the DOS command prompt in an Eclipse console window. In this window, you can enter the atlas commands.

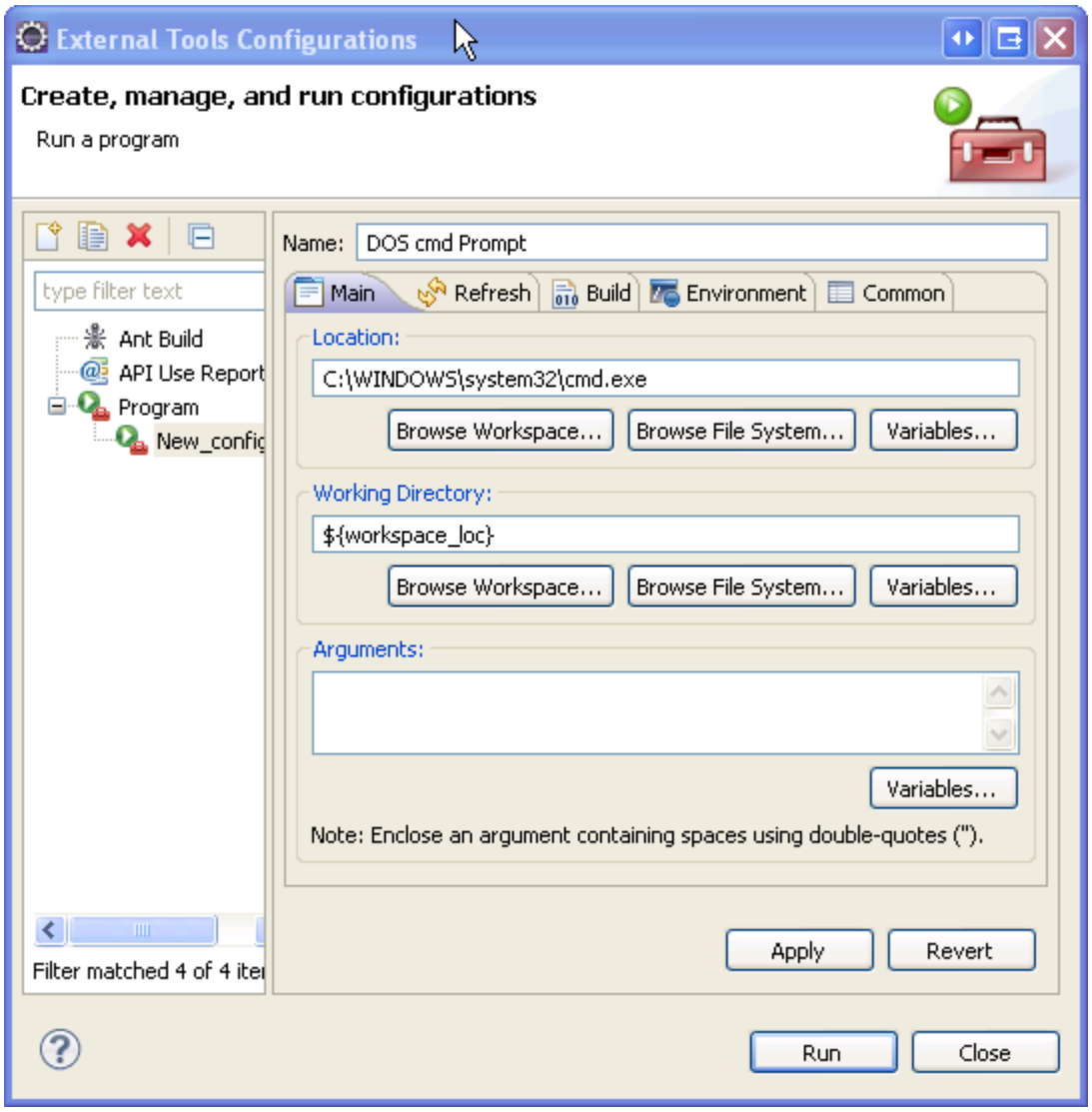
If you haven't already done so, start Eclipse and then do the following:

1. Make sure your workspace is set to your `atlastutorial` project.
   You can use **File > Switch Workspace > Other** to switch if you need to. You need to do this because run configurations are associated with a workspace.
2. Click to **Run > External Tools > External Tools Configuration...** from the Eclipse menu bar.
   The **External Tools Configuration** dialog appears.
3. Select **Programs** and press **New launch configuration**.



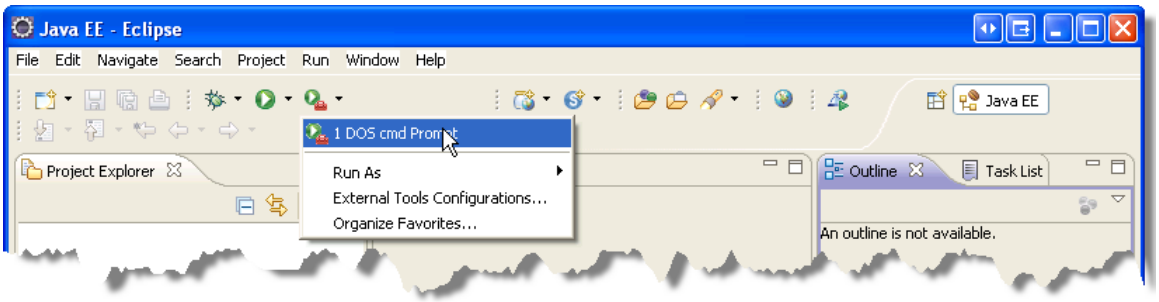
The system creates a new configuration and places you in a configuration dialog.

4. Name the new configuration **DOS cmd Prompt**.
5. Click **Browse File System...**.
6. Navigate to the location of the `cmd.exe` program.
   This should be in the `C:\WINDOWS\system32\cmd.exe` directory.
7. Add the `${workspace_loc}` variable to the **Working Directory** section.
   When you are done, the dialog will appear as follows:

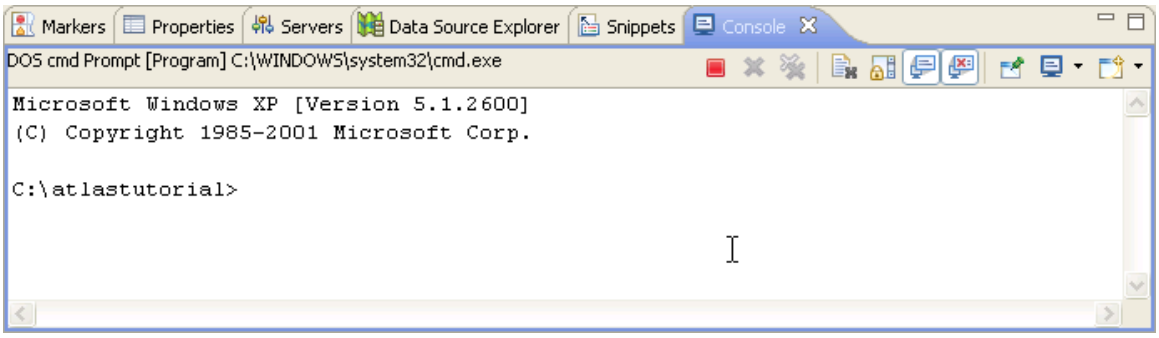


8. Click the **Common** tab.
9. Check **External Tools** under **Display** in **Favorites** menu.
10. Make sure the **Allocate console** (necessary for input) option is checked.
11. Press **Apply** to save your configuration.
12. Press **Close** to close the dialog.

Go ahead and try your new configuration. Launch the DOS cmd Prompt from within Eclipse:



You should see the prompt appear in an Eclipse console:



The console doesn't support everything a regular DOS window does, but it does everything you need for most of your work with the SDK.

The Next Steps

You've got Eclipse set up and configured. Now, you can import your project into Eclipse and use the Atlassian SDK with it.

Set Up the Eclipse IDE for Linux

So far, you've configured your environment, installed the Atlassian SDK, and created a `HelloWorld` plugin project. You've tweaked the plugin project a bit but it has no modules so other than live in the UPM, it does nothing. Of course, you've done all this on the command line — most programmers prefer the advantages of an IDE. So, on this page, you configure the Eclipse IDE to work with the SDK, import `HelloWorld`, and add some functionality.

Windows User?

See this page.

IntelliJ User?

See this page.

Do I have to use Eclipse?

If you already have a version of Eclipse installed, you don't need to do this step. Note that the tutorial assumes a particular version of Eclipse, so you should anticipate that you may need to adjust procedures if you choose to use another version of it. If you aren't interested in using Eclipse and prefer another IDE, you are perfectly free to skip this page. However, the rest of this tutorial assumes you are using Eclipse.

Step 1: Install the Eclipse IDE

In this step, you download and install the Eclipse IDE for Java EE Developers (Indigo). This version of Eclipse comes with most of the Maven dependencies already installed. Do the following to install Eclipse in your system:

1. Download the Eclpse IDE for Java EE developers.
   This IDE has many of the dependencies required by the Maven Eclipse plugin.
2. Move the tar file into your home directory.
3. Expand the tar file.

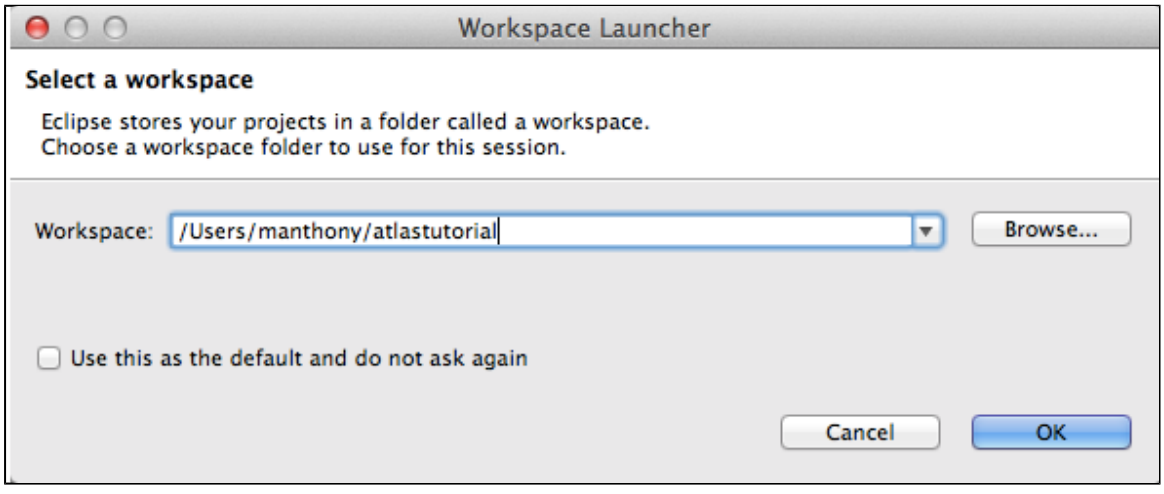
```
tar -xvzf eclipse-jee-indigo-SR2-macosx-cocoa.tar.gz
```

   When you are done, you will have an `eclipse` directory in your home directory.

Step 2: Start Eclipse and update the Installed JREs

Now you are ready to start Eclipse.

1. Change directory to `~/eclipse`.
2. Run the `eclipse` program by typing:
   `./eclipse`
   Eclipse prompts you to identify the location of your workspace.
3. Enter the location of the `atlastutorial` directory.



4. Open the **Eclipse > Preferences** dialog.
5. Filter for or navigate to the **Installed JREs** page.
6. Make sure that the checked JRE comes from the JDK 1.6.

Step 3: Install the Maven Eclipse Plugin

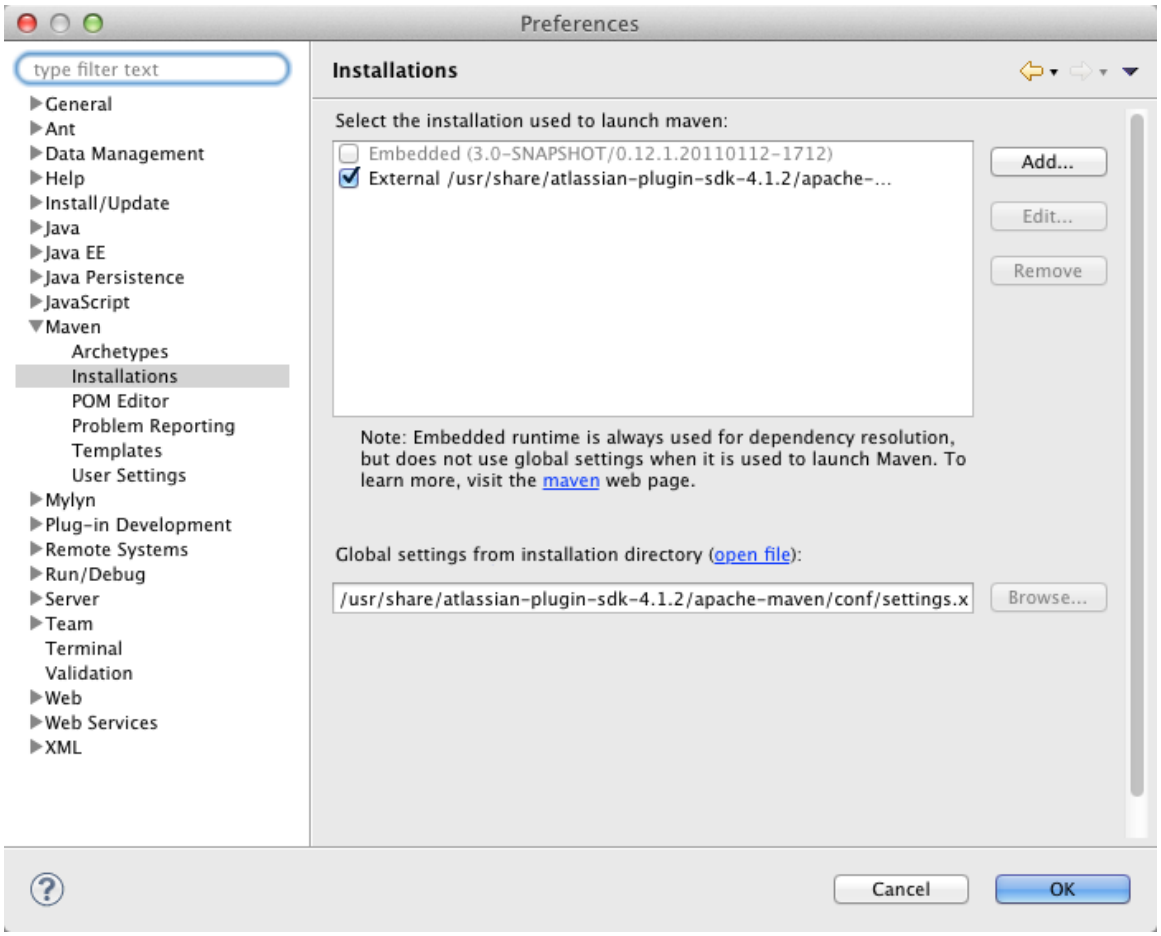
Start Eclipse and do the following:

1. Choose **Help > Install New Software** from the menu bar.
   The **Available Software** dialog appears.
2. Click the **Add** button.
   The **Add Repository** dialog appears.
3. Enter `Sonatype M2Eclipse` in the **Name** field.
4. Enter `http://download.eclipse.org/technology/m2e/releases` in the **Location** in the field.
5. Press **OK** to close the dialog.
   The system searches the site for the plugin. After a moment, the **Name** field fills with the **Maven Integration for Eclipse**.
6. Check the box and press **Next**.
7. Select the **Maven Integration for Eclipse**.
8. Press **Next** and **Next** again.
9. Accept the terms of the license agreement and press **Finish**.
   Eclipse calculates the dependencies and space
10. Press **Next**.
11. Accept the License agreement and press **Next**
12. Press **Finish**.
    The installation procedure runs.
13. Restart Eclipse when prompted.

Step 4: Configure the Maven Plugin

After Eclipse has restarted, configure it to use the Maven Eclipse plugin you just installed.

1. Choose **Window > Preferences** from the Eclipse menu bar.
   The system displays the preferences dialog.
2. Filter for or navigate to the **Maven > Installations** page.
3. Click the **Add** button.
   The **Maven Installation** dialog displays.
4. Browse to your `/usr/share/atlassian-plugin-sdk-<version>/apache-maven` installation.
5. Press **OK**.
   The system sets this external repository for you. The dialog should look like the following:



6. Ensure the Global settings are coming from the installation directory.
7. Press **Apply**.
8. Click the **Maven** root.
9. Uncheck **Download repository index updates on startup**.
   This prevents Maven from updating on Eclipse start up, which can be time-consuming. The **atlas-** commands update the repositories for you.
10. Press **OK** to close the dialog.

The Next Steps

You've got Eclipse set up and configured. Now, you can import your project into Eclipse and use the Atlassian SDK with it.

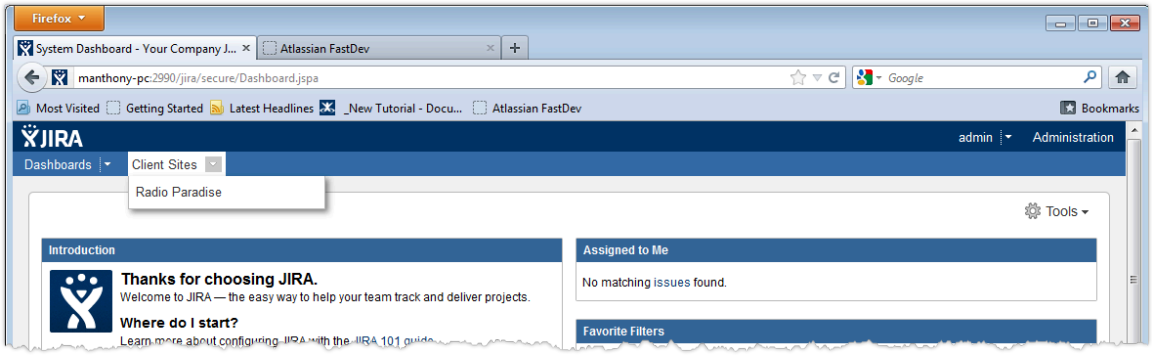
Put the Final Polish on the Project in Eclipse

At this point, you have created an Atlassian Plugin project and edited that project. You have also configured Eclipse to work with Atlassian projects. Let's go ahead and bring the helloworld project you created into Eclipse and work on it there. The following topics are covered:

Step 1: Learn a little about modules

After you bring your helloworld plugin into your project, you will add some modules to it to build up a custom menu. Each Atlassian product exposes a number of plugin modules. There are a set of modules that are common to all plugins and some modules are defined by and specific to the individual applications. For example, all products support a servlet plugin module, but only Confluence provides a macro plugin module. Later, as you become more knowledgeable, you can create your own modules.

You need to use just two module types to create your custom menu. The finished custom menu will look like the following:



You use a `web-item` module to define the menu itself and another one for a menu item. You use a `web-section` to hold the menu. You define all of these using `atlas-` commands.

`helloworld` Source

If you want to check your work when you are done, you can find the `helloworld` source code on Atlassian bitbucket. bitbucket serves a public Git repository containing the tutorial's code. To clone the repository, issue the following command:

```
git clone https://bitbucket.org/atlassian_tutorial/helloworld.git
```

If you aren't sure how to use Git, see the bitbucket getting started guide. Alternatively, you can download the source using the **get source** option here: https://bitbucket.org/atlassian_tutorial/helloworld/overview.

Step 2: Generate Eclipse configuration files

The `helloworld` project you created has everything you need for a Maven project. However, it is not ready to be imported into Eclipse. For that, the first step is to use the `atlas-mvn` command to generate Eclipse configuration files in the project.

1. Go to your system's command line (DOS prompt for Windows, shell prompt for Linux).
2. Navigate to the root of your `helloworld` project.
   This is the directory where the `pom.xml` file is. If you have followed this tutorial exactly up to this point, the directory is `atlastutorial/helloworld`.
3. Enter the following command:

   ```
   atlas-mvn eclipse:eclipse
   ```

   The command will print some informational messages and then return something like the following:

```
[INFO] Wrote settings to
C:\atlastutorial\helloworld\.settings\org.eclipse.jdt.core.prefs
[INFO] Wrote Eclipse project for "helloworld" to
C:\atlastutorial\helloworld.
[INFO]
[INFO]
------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
-----
[INFO] Total time: 37 seconds
[INFO] Finished at: Sun May 20 17:24:27 PDT 2012
[INFO] Final Memory: 52M/125M
[INFO]
------------------------------------------------------------------------
-----
```

This means you are ready for the next step.

Step 3: Import helloworld into Eclipse

Now, import your project into the Eclipse IDE. Start Eclipse and do the following:

1. Select **File > Import**.
   Eclipse starts the Import wizard.
2. Expand the **General** folder tree.
3. Filter for **Existing Projects into Workspace** and press **Next**.
4. Choose **Select root directory** and then **Browse** to the root directory of your workspace.
   Your Atlassian plugin folder should appear under **Projects**.
5. Select your plugin project and click **Finish**.
   Eclipse imports your project.
6. Navigate to your `atlassian-plugin.xml` file and open it.
   At this point, your file will look similar to the following:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
name="${project.name}" plugins-version="2">
    <plugin-info>
        <description>${project.description}</description>
        <version>${project.version}</version>
        <vendor name="${project.organization.name}"
url="${project.organization.url}" />
    </plugin-info>
</atlassian-plugin>
```

Step 4: Create the Menu

In this step, you add a new menu called **Client Sites** to your project using the `atlas-create-jira-plugin-module` command. The menu will contain a link to Radio Paradise, an online radio station.

1. Open a command prompt in Eclipse using the launcher on your desktop.
   You can also just go to a command line outside of Eclipse.

2. Navigate to your project directory.
3. Enter the `atlas-create-jira-plugin-module` command.
   The command prompts you for a module to add.
4. Select the 30 `Web Section` item by entering `30`.
   The system prompts you with a series of questions.
5. Answer each question as follows:

Question	Value	Description
Enter Plugin Module Name:	`mySection`	A human-readable name for the section. This is only visible to administrators working with the Universal Plugin Manager (UPM).
Enter Location:	`client-sites-link`	A unique location representing this web section.
Show Advanced Setup?	N	Not applicable for this tutorial

   The system displays information about what kinds of changes the additional module generates.

```
Enter Plugin Module Name [My Web Section]: mySection
Enter Location (e.g. system.admin/mynewsection): client-sites-link
Show Advanced Setup? (Y/y/N/n) [N]: n
[INFO] Adding the following items to the project:
[INFO]   [dependency: org.mockito:mockito-all]
[INFO]   [module: web-section]
[INFO]   i18n strings: 3
Add Another Plugin Module? (Y/y/N/n) [N]:
```

6. Press Y to add another Plugin Module.
   A web section simply provides a container for web items. You must add a web-item representing the menu.
7. Choose `25: Web Item`

Question	Value	Description
Enter Plugin Module Name	`Client Sites`	This is the name that will appear on the new menu.
Enter Section	`system.top.navigation.bar`	The location in the JIRA menu where you want your custom menu to appear.
Enter Link URL	`deleteMe`	More about this later.
Show Advanced Setup?	N	Not applicable for this tutorial

   The system displays something similar to the following:

```
[INFO] Adding the following items to the project:
[INFO]   [dependency: org.mockito:mockito-all]
[INFO]   [module: web-item]
[INFO]   i18n strings: 3
```

8. Answer `N` when prompted to add another module.
   The system completes its operation and provides you with the following informational messages:

```
[INFO]
------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
-----
[INFO] Total time: 15 minutes 15 seconds
[INFO] Finished at: Mon May 14 14:29:58 PDT 2012
[INFO] Final Memory: 41M/117M
[INFO]
------------------------------------------------------------------------
-----
```

Step 5: Tweak the module description

At this point, you have used the SDK to generate a new module. The generation tools are intended for general cases. This means, that you may occasionally need to edit what was created by a generation command. This is the case for this tutorial. If you haven't already done so, open Eclipse and do the following:

1. Refresh the Eclipse project.
   Eclipse picks up the dependency changes in the `pom.xml` and the modules added to the `atlassian-plugin.xml`. The command also added an i18n resource in `src\main\resources\com\atlassian\tutorial` directory called `helloworld.properties`.
2. Open the `atlassian-plugin.xml` in the **Source** view and look for the web-section and web-item entries.
   These entries were added for you when you added the modules. You should see something similar to the following:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
name="${project.name}" plugins-version="2">

  ... code not shown ...

  <!-- publish our component -->
  <component key="myPluginComponent"
class="com.atlassian.tutorial.helloworld.MyPluginComponentImpl" public="true">
    <interface>com.atlassian.tutorial.helloworld.MyPluginComponent</interface>
  </component>
  <!-- import from the product container -->
  <component-import key="applicationProperties"
interface="com.atlassian.sal.api.ApplicationProperties"/>
  <web-section name="mySection" i18n-name-key="my-section.name"
key="my-section" location="client-sites-link" weight="1000">
    <description key="my-section.description">The mySection
Plugin</description>
    <label key="my-section.label"/>
  </web-section>
  <web-item name="Client Sites" i18n-name-key="client-sites.name"
key="client-sites" section="system.top.navigation.bar" weight="1000">
    <description key="client-sites.description">The Client Sites
Plugin</description>
    <label key="client-sites.label"></label>
    <link linkId="client-sites-link">deleteMe</link>
  </web-item>
</atlassian-plugin>
```

(If you don't see this, make sure you are looking at the **Source** view of the file, not the **Design** view.)
By default, the plugin generator adds a `<label>` element to the `web-section`. You won't need this.

3. Remove the `<label>` element from the `web-section`.
4. Locate the `<link>` element in the `web-item`.
   This `web-item` becomes the menu label for your menu. You don't want that item to actually link anywhere. So the `deleteMe` URL value is not necessary.
5. Remove the URL from the `<link>` element but leave the link item.
   When you are done the `<link>` element should look like this:

```
<link linkId="client-sites-link"></link>
```

6. Save the `atlassian-plugin.xml` file.
   Your `atlassian-plugin.xml` descriptor file should look like this:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
name="${project.name}" plugins-version="2">

  ... code not shown ...
  <web-section name="mySection" i18n-name-key="my-section.name"
key="my-section" location="client-sites-link" weight="1000">
    <description key="my-section.description">The mySection
Plugin</description>
  </web-section>
  <web-item name="Client Sites" i18n-name-key="client-sites.name"
key="client-sites" section="system.top.navigation.bar" weight="1000">
    <description key="client-sites.description">The Client Sites
Plugin</description>
    <label key="client-sites.label"></label>
    <link linkId="client-sites-link"><link/>
  </web-item>
</atlassian-plugin>
```

Step 6: View the Menu in JIRA

In this step, you view the menu you just created in JIRA. At this point the menu has no items in it. We'll add one in a moment.

1. Open a command prompt in the root of your project.
2. Enter the following to build your plugin JAR:

   ```
   atlas-run
   ```

   The command builds your JAR and the JIRA container in the target directory. When it completes successfully, the command echoes back something similar to the following:

   ```
   [WARNING] [talledLocalContainer] INFO: Server startup in 695 ms
   [INFO] [talledLocalContainer] Tomcat 6.x started on port [2990]
   [INFO] jira started successfully in 46s at
   http://manthony-PC:2990/jira
   [INFO] Type Ctrl-D to shutdown gracefully
   [INFO] Type Ctrl-C to exit
   ```

3. Go to a browser and open JIRA.
   You should see your new "Client Sites" menu immediately.
4. Click on the menu.
   Nothing happens except that the page refreshes. Let's fix this.
5. Leave JIRA up and running and start another command prompt.
6. Go to the root of your project and add another `web-item` module:

   ```
   atlas-create-jira-plugin-module
   ```

7. Enter `25: Web Item` when prompted.
8. Complete the module as follows:

Enter Plugin Module Name My Web Item:	`Radio Paradise`
Enter Section (e.g. system.admin/globalsettings):	`client-sites-link/my-section`
Enter Link URL (e.g. /secure/CreateIssue!default.jspa):	http://www.radioparadise.com
Show Advanced Setup? (Y/y/N/n) N:	`n`

9. Enter `n` when prompted to add another module.
10. Go back to Eclipse and refresh the `atlassian-plugin.xml` file.
    At this point, your file should look like the following:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
name="${project.name}" plugins-version="2">

  ... code not shown ...

<web-section name="mySection" i18n-name-key="my-section.name" key="my-section"
location="client-sites-link" weight="1000">
    <description key="my-section.description">The mySection
Plugin</description>
  </web-section>
  <web-item name="Client Sites" i18n-name-key="client-sites.name"
key="client-sites" section="system.top.navigation.bar" weight="1000">
    <description key="client-sites.description">The Client Sites
Plugin</description>
    <label key="client-sites.label"/>
    <link linkId="client-sites-link"/>
  </web-item>
  <web-item name="Radio Paradise" i18n-name-key="radio-paradise.name"
key="radio-paradise" section="client-sites-link/my-section" weight="1000">
    <description key="radio-paradise.description">The Radio Paradise
Plugin</description>
    <label key="radio-paradise.label"></label>
    <link linkId="radio-paradise-link">http://www.radioparadise.com</link>
  </web-item>
</atlassian-plugin>
```

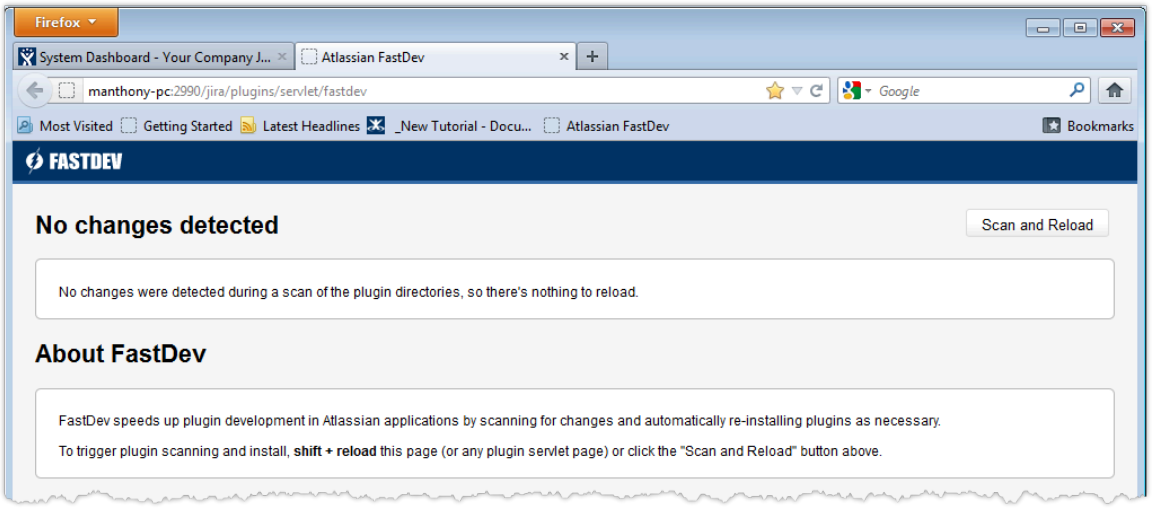
Step 7: Use the FastDev (fast development) feature

At this point, you left code running and made some changes to your code. Rather than rebuild the JAR, use the FastDev feature to reload your changes while JIRA is running.

1. Return to the browser.
2. Navigate to the FASTDEV page for your project. The URL for this page has a format of `http://HOSTNAME:PORT/jira/plugins/servlet/fastdev`.
   For example, you would enter something similar to the following:

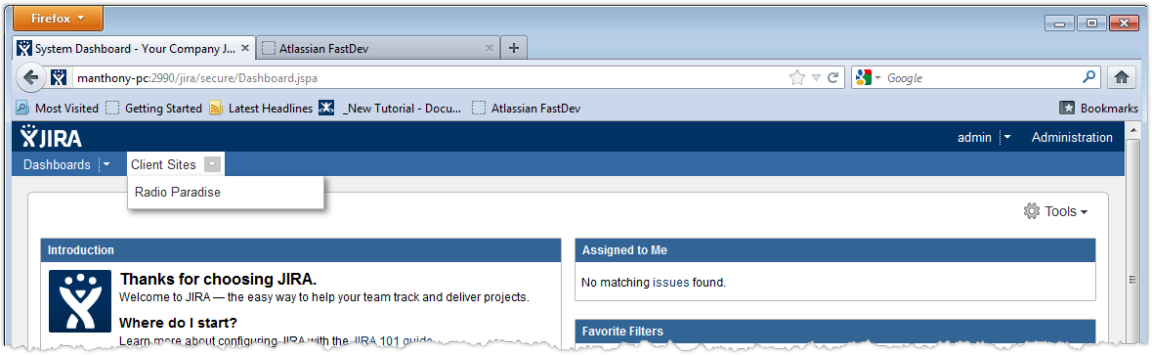
```
http://manthony-pc:2990/jira/plugins/servlet/fastdev
```

   The FASTDEV dialog appears.



3. Press **Scan and Reload**.
   The system scans your projects for changes and reloads them into the JIRA instance. When it completes successfully, you see the following:

4. Return to the JIRA dashboard and reload it.
   After the reload your secondary menu should appear:



Next steps

Congratulations, you have completed the first Atlassian tutorial and added your own custom menu to JIRA.

Where you go now depends upon your interests, but there are many more tutorials to explore. To continue learning with tutorials, follow one of these links:

- JIRA tutorials
- Confluence Tutorials
- Cross-Product Tutorials

Writing and Running Plugin Tests

This tutorial describes the concepts you need to understand, the tools you use, and processes you follow for testing a plugin in the Atlassian Plugin Framework. You can use all of the concepts, tools, and processes described in this section with any plugin regardless of the plugin's intended host application (for example, JIRA, Confluence, or Stash). This section contains the following topics:

- Generate and Examine Skeleton Tests
- Create and Run Unit Tests
- Create and Run Traditional Integration Tests
- Run Wired Tests with the Plugin Test Console
- Create Test Data and a Test Fixture
- Staff README for Foundation Test Docs

How to work through the tutorial

Beginners should work through each page sequentially as each new page builds on the material from the previous page. Each page concludes with a **Next Steps** heading that recaps and tells you where to go next. Beginners should allow two hours to work through the entire tutorial.

Experienced programmers or readers with previous knowledge of Atlassian plugin development processes, may choose to just skim or skip around. Each page encapsulates the core knowledge for each part of the testing process.

Check Your Work

We encourage you to work through this tutorial. If you want to skip ahead or check your work when you are done, you can find the plugin source code on Atlassian Bitbucket. Bitbucket serves a public Git repository containing the tutorial's code. To clone the repository, issue the following command:

```
git clone
https://atlassian_tutorial@bitbucket.org/atlassian_tutorial/testtutorial
.git
```

Alternatively, you can download the latest source here: https://bitbucket.org/atlassian_tutorial/testtutorial/downloads.

Generate and Examine Skeleton Tests

When you generate a new plugin with an `atlas-create-`*`application`*`-plugin` command, the generated plugin skeleton includes skeleton tests. This page walks you through the process of generating the plugin skeleton and its tests. It also introduces you to the project components the tests rely on. The following topics are covered:

Backgrounder: Supported Plugin Test Types

You can and should test plugins using the same types of tests as you would for other software. In general, the different types of tests fall into these categories:

Type	Description
unit	Test on a distinct unique of work within the plugin such as a single method or function.
integration	Tests the interactions between the plugin and the plugin's target environment. In the case of an Atlassian plugin, the target environment is the host application. Integration tests can also include or rely on other external, services. Atlassian supports traditional integration tests and wired integration tests. Wired integration tests are bundled as a plugin and run directly in the host application. You'll learn more about creating wired integration tests later in this tutorial.
functional	Tests the functionality of the application's features and functions.
stress	Tests how the application performs under a large number of requests within a given period.
acceptance	Tests how well the application meets a customer's needs.

Unit and integration tests test your plugin's internal structures or functions. Atlassian provides you with a plugin test skeleton and a tools for unit and integration testing. Often, you can combine functional tests with your integration tests. In some cases, Atlassian host applications supply functional test libraries that you can leverage in developing your own tests.

The stress and acceptance tests require you to write specifications or build systems specific to your plugin and its requirements. Atlassian does not provide tools, processes, or infrastructures for these test types but we do encourage you to do them on your own.

Step 1. Create a Plugin Skeleton

In this step, you create a simple plugin skeleton and import it into the Eclipse IDE. You can create a skeleton for any of the Atlassian host applications. This tutorial uses JIRA.

Do I have to Use Eclipse?
If you aren't interested in using Eclipse and prefer another IDE, you are perfectly free to use another. However, the tutorial assumes you are using Eclipse.

Do the following to generate the plugin skeleton and its tests resources:

1. Open a terminal and navigate to your Eclipse workspace directory.
2. Enter the following command to create a JIRA plugin skeleton:

```
atlas-create-jira-plugin
```

When prompted, enter the following information to identify your plugin:

Create a plugin for?	Enter a JIRA version
group-id	`com.example.plugins.tutorial.jira`
artifact-id	`testTutorial`
version	`1.0-SNAPSHOT`
package	`com.example.plugins.tutorial.jira.testTutorial`

3. Confirm your entries when prompted.
4. Change to the `testTutorial` directory created by the previous step.
5. Run the command:

```
atlas-mvn eclipse:eclipse
```

You should repeat this command after you add a dependency to your `pom.xml` file. It ensures that Eclipse populates your project dependencies correctly.

6. Start Eclipse.
7. Select **File->Import**.
   Eclipse launches the **Import** wizard.
8. Filter for **Existing Projects into Workspace** (or expand the **General** folder tree).
9. Press **Next** and enter the root directory of your workspace.
   Your Atlassian plugin folder should appear under **Projects**.
10. Select your plugin and click **Finish**.
    Eclipse imports your project.

Step 2. Review the Generated Test Structure

The `atlas-create-jira-plugin` command creates test directories and skeleton test files. Most `atlas-create-`*`application`*`-plugin` commands create the following directories:

Directory	Description
`PLUGIN_HOME/src/test/java/it`	Place integration tests here. You must package all your integration tests in a package that begins with the `it` prefix.
`PLUGIN_HOME/src/test/java/ut`	Place unit test here. You must package all your unit tests in a package that begins with the `ut` prefix.
`PLUGIN_HOME/src/test/resources`	Place resource test files here.

Not all `atlas-create-`*`application`*`-plugin` commands generate these directories. If you use a command that does not generate these directories, create them yourself.

Along with the test directory structure, most `atlas-create-`*`application`*`-plugin` commands generate

skeleton test files. For this tutorial, the generation step created these additional files:

- `PLUGIN_HOME/src/test/java/it/com/example/plugins/tutorial/jira/testTutorial/MyComponentWiredTest.java`
- `PLUGIN_HOME/src/test/java/ut/com/example/plugins/tutorial/jira/testTutorial/MyComponentUnitTest.java`
- `PLUGIN_HOME/src/test/resources/atlassian_plugin.xml`

The wired integration tests rely on the descriptor file (`atlassian_plugin.xml`). You'll learn more about this later in the tutorial.

Take some time and open the generated test files. The Java files contain simple classes that you can expand on as you work. If you use the `atlas-create-`*`application`*`-plugin-module` commands to add plugin modules, you may find the commands generate additional files in these folders.

Step 3. Review the pom.xml Test Dependencies

Typically, your `pom.xml` file contains three elements related to test dependencies.

The JUnit Dependency

In the previous step, you may have noticed that all of your plugin test files import a `org.junit.Test` class. JUnit is a widely-used testing framework supported for testing Atlassian plugins. When you generate a plugin, the `pom.xml` file includes a dependency on JUnit 4.10:

```
<dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.10</version>
       <scope>test</scope>
</dependency>
```

The scope of this dependency limits the availability of JUnit only to the project test compilation and execution phases. If you are packaging a plugin, you need not remove your tests files as the test dependencies do not load at runtime in a production system.

Testrunner Dependencies

Look for the wired test runner dependencies further down the page. These look like the following:

```
<!-- WIRED TEST RUNNER DEPENDENCIES -->
  <dependency>
     <groupId>com.atlassian.plugins</groupId>
     <artifactId>atlassian-plugins-osgi-testrunner</artifactId>
     <version>${plugin.testrunner.version}</version>
     <scope>test</scope>
  </dependency>
  <dependency>
     <groupId>javax.ws.rs</groupId>
     <artifactId>jsr311-api</artifactId>
     <version>1.1.1</version>
     <scope>provided</scope>
  </dependency>
  <dependency>
     <groupId>com.google.code.gson</groupId>
     <artifactId>gson</artifactId>
     <version>2.2.2-atlassian-1</version>
  </dependency>
 </dependencies>
```

Your integration tests can make use of these dependencies to write in-product, wired integration tests.

Product-specific Dependencies

Depending on which host application you are using, Jira, Confluence, etc., your `pom.xml` file may contain dependencies in addition to JUnit. The dependencies can be on third-party test frameworks, like Mockito, or to Atlassian test suites. The JIRA generated `pom.xml` includes two additional dependencies:

```
<dependency>
      <groupId>com.atlassian.jira</groupId>
      <artifactId>jira-tests</artifactId>
      <version>${jira.version}</version>
      <scope>test</scope>
</dependency>
<dependency>
      <groupId>com.atlassian.jira</groupId>
      <artifactId>jira-func-tests</artifactId>
      <version>${jira.version}</version>
      <scope>test</scope>
</dependency>
```

The `jira-tests` artifact contains the JIRA Unit tests. The `jira-func-tests`, like the name sounds, contains the JIRA functional tests. When you write your tests, you can access these artifacts from your own unit, integration, and functional tests. The application-specific sections of this site contain information on these resources. The last page of this tutorial explains where those are.

Next Step

So far, you've learned about the generated structure created for you when you run an `atlas-create-`*`application`*`-plugin` command. This structure includes test directories, files, and dependencies. This is code that Atlassian generates for all plugin developers automatically. In the next section, you write a simple unit test, execute the test in your plugin, and review the results of the test.

Create and Run Unit Tests

This page explains how to create and run unit tests in a plugin. You should you have already worked through Generate and Examine Skeleton Tests. The Atlassian Plugin Framework requires that you use JUnit 4.10 or higher. By default, the SDK 4.1 (and higher) `atlas-` commands all generate a `pom.xml` file with a dependency on the appropriate JUnit version. If you are working with an older plugin, you should make sure to update JUnit `<version>` value to 4.10 before continuing. On this page, you do the following:

JUnit Quick Reminders

You use JUnit to annotate test methods in your plugin. These annotations instruct and inform the JUnit framework how to interpret your test. The following table provides a short refresher of some common JUnit annotations:

Annotation Type	Description
`@After`	Release external resources allocated in the `@Before` method.
`@AfterClass`	Release expensive external resources shared among tests. These are allocated in the `@BeforeClass` met hod.
`@Before`	Create resources or prepares an environment before each `@Test`.
`@BeforeClass`	Performs expensive environment setup or creates expensive resources shared among tests.
`@Ignore`	Ignore the `@Test` method.
`@Test`	Identifies a test method.

Recall that JUnit tests can run in any order. In fact, the order varies across different Java versions. So, remember to keep each unit test independent of another. If your tests are dependent, and they run in a different order than you expect, you may find it difficult to debug test failures. Test failure is another reason to keep your tests independent. If your tests are dependent, the failure of one can cause the remaining tests to fail.

Step 1. Run the Generated MyComponentUnitTest

When you created your plugin, the system generated a skeleton `MyPluginComponent.java` class. This class has a `getName()` method whose implementation relies on the Shared Access Layer (SAL) `com.atlassian.sal.api.ApplicationProperties` API. You'll find a unit test of this method in your `MyComponentUnitTest.java` class:

```
@Test
public void testMyName()
{
     MyPluginComponent component = new MyPluginComponentImpl(null);
     assertEquals("names do not match!", "myComponent",component.getName());
}
```

When you next run your plugin with `atlas-run`, the command also runs your unit tests. Often, you only want to run your unit tests, you use the `atlas-unit-test` command to do this. The command executes only the `test` scope defined by the project `pom.xml` file. Try the `atlas-unit-test` command now:

1. Go to a command line.
2. Change to your plugin's top level `PLUGIN_HOME` directory.
3. Enter the `atlas-unit-test` command:

```
atlas-unit-test
```

The command output should contain test output similar to the following:

```
...

[INFO] [surefire:test]
[INFO] Surefire report directory:
/Users/manthony/atlastutorials/testTutorial/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running
ut.com.example.plugins.tutorial.jira.testTutorial.MyComponentUnitTe
st
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.111 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
-----
[INFO] Total time: 29 seconds
[INFO] Finished at: Thu Nov 15 09:57:19 PST 2012
[INFO] Final Memory: 69M/123M
[INFO]
------------------------------------------------------------------------
-----
```

You can see that the generated unit test was successful.

Step 2. Create a Failing Unit Test

The build fails when one of your tests fails. Let's see how this work by adding a faillling file. If you haven't already done so, start Eclipse and open the `testTutorial` you created.

1. Edit the `PLUGIN_HOME/src/main/java/com/atlassian/plugins/tutorials/jira/testTutorial/MyPluginComponent.java` file.
2. Add a new method to the file:

```
int addNumbers(int x, int y);
```

3. Edit the `PLUGIN_HOME/src/main/java/com/atlassian/plugins/tutorials/jira/testTutorial/MyPluginComponentImpl.java` file.
4. Add the following code to the file:

```
@Override
public int addNumbers(int x, int y) {
 return x + y;
}
```

The plugin method adds two numbers together.

5. Save and close both files.
6. Locate the `PLUGIN_HOME/src/test/java/ut/com/atlassian/plugins/tutorials/jira/testTutorial/MyComponentUnitTest.java` file.
7. Edit the file and add the following unit test code:

```
@Test
public void testAdd()
{
    MyPluginComponent component = new MyPluginComponentImpl(null);
    assertEquals("Result", 8, component.addNumbers(8, 8));
}
```

You should now have one passing test and a new one that should fail.

8. Close and save the file.
9. Go to a command line.
10. Make sure you are in at the top level `PLUGIN_HOME` directory for you plugin.
11. Enter the `atlas-unit-test` command.

The failing test returns output similar to the following:

```
Failed tests:
testAdd(ut.com.example.plugins.tutorial.jira.testTutorial.MyComponentUnitTest): Result expected:<8> but was:<16>
```

With a failing test, the Maven build reports an error. If you were to run `atlas-run` at this point, it would fail also because you have a build error. If you have some time, try the `atlas-run` command.

Step 3. Surefire Reports for Your Tests

The Atlassian Plugin SDK uses the Maven Surefire Plugin during the `test` phase to run your tests. Running `atlas-unit-test` creates a `PLUGIN_HOME/target/test-classes` directory containing your compiled test. Surefire also creates a `PLUGIN_HOME/target/surefire` directory to hold temporary files during the test generation. After the test phase completes, Surefire generates a `PLUGIN_HOME/target/surefire-reports` directory.

Take a moment and examine the Surefire report generated by your testing:

1. Change directory to `PLUGIN_HOME/target/surefire-reports` directory.
2. List the directory contents.
   You should see something similar to the following:

```
TEST-ut.com.example.plugins.tutorial.jira.testTutorial.MyComponentU
nitTest.xml
ut.com.example.plugins.tutorial.jira.testTutorial.MyComponentUnitTe
st.txt
```

Both files are Surefire report files. Each time your tests you overwrite these files.

3. Take a minute and review the contents of each file.
   The XML report includes a set of environmental properties used in the test.

The `@Ignore` annotation is useful if you want to skip a test you know is obsolete or you just aren't ready to use. Sometimes, during development, you may encounter tests that fail for other reasons. Remember the `atlas-` commands that start a host application with you plugin installed, also runs your tests. Failing test are inconvient if you just want to do `atlas-run` and see your test in a host application. Go ahead and ignore the failing test in your sample code:

1. Edit the `MyComponentUnitTest.java`.
2. Add an import for the JUnit @Ignore annotation.

```
import org.junit.Ignore;
```

3. Add the `@Ignore` annotation to the `testAdd()` method:

```
@Ignore
@Test
public void testAdd()
{
    MyPluginComponent component = new MyPluginComponentImpl(null);
    assertEquals("Result", 8, component.addNumbers(8, 8));
}
```

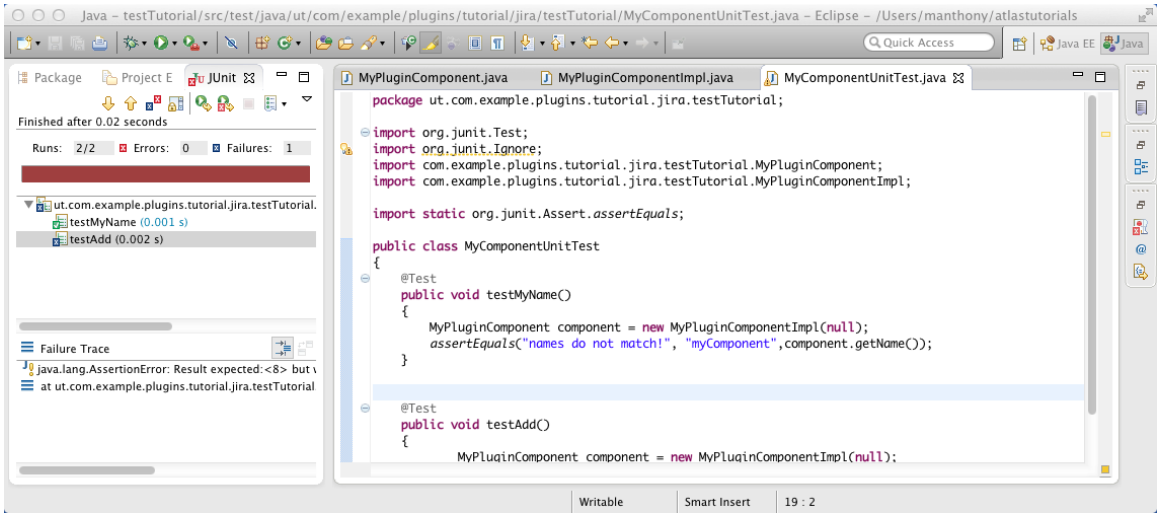
4. Save and close the file.
5. Rerun your unit tests.
   You should see that only a single test is run, the first, the framework skips the other:

```
[INFO] [surefire:test]
[INFO] Surefire report directory:
/Users/manthony/atlastutorials/testTutorial/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running
ut.com.example.plugins.tutorial.jira.testTutorial.MyComponentUnitTe
st
Tests run: 2, Failures: 0, Errors: 0, Skipped: 1, Time elapsed:
0.09 sec
Results :
Tests run: 2, Failures: 0, Errors: 0, Skipped: 1
[INFO]
------------------------------------------------------------------------
-----
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
-----
[INFO] Total time: 28 seconds
[INFO] Finished at: Thu Nov 15 10:34:32 PST 2012
[INFO] Final Memory: 69M/123M
[INFO]
------------------------------------------------------------------------
-----
```

Step 4. Alternative Ways to Run or to Skip Tests

Both Eclipse and IDEA allow you to run JUnit tests from within your IDE. If you are following along with this tutorial exactly, you should be using Eclipse. Go ahead and try this now:

1. Edit the `PLUGIN_HOME/src/test/java/ut/com/atlassian/plugins/tutorials/jira/testTutorial/MyComponentUnitTest.java` file.
2. Remove the `@Ignore` annotation you added in the previous procedure.
3. Save the file.
4. Select **Run As > JUnit** Test.
   You should see output similar to the following:



Attempt an `atlas-run` now:

1. Go to the command line.
2. Change to your `PLUGIN_HOME` directory:
3. Enter the `atlas-run` command:

```
atlas-run
```

   Very quickly the `testAdd()` method fails, the build does not complete, and `atlas-run` exits without launching the host application.
4. Rather than editing your tests to fix the failure, which can be time consuming when you many tests, skip all tests from the command line by entering the following:

```
atlas-run -DskipTests=true
```

   It may have crossed your mind that skipping tests in general may be a good idea if you just want to run the host application and your project has a lot of tests!

Next Steps

In the next section, you learn about the tools and processes for running traditional integration tests.

Create and Run Traditional Integration Tests

This page explains the tools and processes you use in the Atlassian Plugin SDK, to create and run traditional integration tests for your plugin. The material on this page assumes you have already worked through or otherwise understand the information in Create and Run Unit Tests. On this page, you do the following:

Overview of Integration Testing

Integration testing validates the interactions between an Atlassian host application (JIRA, Confluence, Stash, etc.) and your plugin. Traditional integration tests typically used Mockito to replicate or mock-up functions in the host. Now, Atlassian offers the Wired Test Framework for integration and unit testing.

The Wired Test Framework allows you to test your plugin in the actual host application environment rather than against mocked functions. Thus, for example, if your plugin creates an issue in JIRA by invoking the JIRA API, the Wired Test Framework lets you invoke your plugin and then test whether an issue was actually created in JIRA.

If you have existing integration or functional tests, you can convert these to the new framework. You also have the choice of continuing to use your traditional integration tests. And, of course, you can use both the traditional and wired integration test in the same plugin.

There are several ways to execute integration tests and review the results—from the command line or from the test console in the host application's UI. The command line approach as described here compiles the project, starts up the application and executes all the tests. When writing and tweaking the test code itself, this may not be the most efficient manner of working.

As an alternative, if you are using the Wired Test Framework, you can use the test console in the application. The test console lets you modify test code and rerun the individual test you are working on and see the results of the test without having to recompile the plugin and restart the application. This is usually the most efficient manner of working when you are developing and tweaking test code. (The test console is described in the next page in this tutorial).

The atlas-integration-test Command

You use the `atlas-integration-test` command to run integration tests explicitly. The command does the following:

- executes your unit tests (unit tests are part of the `<test>` scope)
- starts the host application configure in your `pom.xml` file's `<build>` section (for this tutorial JIRA)
- loads the application with some default data
- executes your integration tests

Just as with unit tests, the integration tests use Maven's Surefire plugin. Unlike unit tests, the `atlas-integration-test` command starts a host application environment.

Test Configuration

By default, the system runs all the tests defined in your project's `it` package with the host application. You can configure additional information in the `<build>` block of your `pom.xml` file through the `<configuration>` element. You can specify one or more <products> to test against. You can also use the `<testGroups>` element to group integration tests. The following example illustrates both elements:

```
<build>
  <plugins>
   <plugin>
    <groupId>com.atlassian.maven.plugins</groupId>
    <artifactId>maven-jira-plugin</artifactId>
    <version>${amps.version}</version>
    <extensions>true</extensions>
    <configuration>
     <products>
      <product>
       <id>jira</id>
       <instanceId>jiraExpected</instanceId>
       <productVersion>${jira.version}</productVersion>
       <productDataVersion>${jira.version}</productDataVersion>
      </product>
     </products>
     <testGroups>
      <testGroup>
       <id>jira-integration</id>
       <productIds>
        <productId>jira</productId>
       </productIds>
       <includes>
        <include>it/**/*Test.java</include>
       </includes>
      </testGroup>
     </testGroups>
    </configuration>
   </plugin>
...
```

You can specify one or more <product> children in <products> element. A <product> has the following fields:

Field	Descriptoin
id	Can be one of: • jira • confluence • bamboo • fecru • crowd • refapp • stash
instanceId	Unique identifier for this product definition.
productVersion	Version of the product to use.
productDataVersion	Version of test data to use.

You can specify one or more <testGroup> children in <testGroups> element. A <testGroup> has the following fields:

Tag	Definition

`id`	Unique identifier for the test group.
`productIds`	Contains one or more `<productId>` values: • `jira` • `confluence` • `bamboo` • `fecru` • `crowd` • `refapp` • `stash` Alternatively, you can refer to a defined `<instanceId>` value from a `<product>` element.
`includes`	Contains one or more `<include>` values.

Through this configuration you can specify which tests to include in your integration tests and which products to run them against. You can even configure the build to run multiple versions of the host application or more than one host application. For example, you can run Confluence and JIRA together with your plugin tests. Or, you could run JIRA 5.0 and JIRA 4.0.

Click to view a complex configuration with multiple products and test groups...

```
<configuration>
    <products>
        <product>
            <id>confluence</id>
            <instanceId>confluence-3.3.1</instanceId>
            <productVersion>3.3.1</productVersion>
            <productDataVersion>3.0</productDataVersion>
        </product>
        <product>
            <id>confluence</id>
            <instanceId>confluence-3.4.0</instanceId>
            <productVersion>3.4.0</productVersion>
            <productDataVersion>3.0</productDataVersion>
        </product>
  <product>
   <id>jira</id>
   <instanceId>jiraExpected</instanceId>
   <productVersion>${jira.version}</productVersion>
   <productDataVersion>${jira.version}</productDataVersion>
  </product>
    </products>
 <testGroups>
        <testGroup>
            <id>allConfluence</id>
            <productIds>
                <productId>confluence-3.3.1</productId>
                <productId>confluence-3.4.0</productId>
            </productIds>
            <includes>
                <include>it/**/confluence/*Test.java</include>
            </includes>
        </testGroup>
        <testGroup>
            <id>jira</id>
            <productIds>
                <productId>confluence-3.3.1</productId>
                <productId>confluence-3.4.0</productId>
            </productIds>
            <includes>
                <include>it/**/confluence/*Test.java</include>
            </includes>
        </testGroup>
  <testGroup>
   <id>jira-integration</id>
   <productIds>
    <productId>jira</productId>
   </productIds>
   <includes>
    <include>it/**/jira/*Test.java</include>
   </includes>
   </testGroup>
    </testGroups>
</configuration>
```

Accessing Test Instances

During each test execution, the build system sets the following system properties for each product definition:

Property	Description

`baseurl.instanceId`	The base url of the application, e.g. http://localhost:8990/jira
`http.instanceId.port`	The HTTP port that the application is listening on, e.g. `8990`
`context.instanceId.path`	The context path of the application, e.g. `/jira`

Use these values to co-ordinate communication between your integration tests and running application instances.

Step 1. Run the Skeleton Integration Tests

When you created the plugin, the system generated an integration test class for you, the `MyComponentWiredTest.java` file. You can find this file in your project's `PLUGIN_HOME/src/test/java/it /com/atlassian/plugins/tutorials/jira/testTutorial` folder. As the file's name implies, the generated test files assume you plan to use the Wired Test Framework rather than traditional integration tests. This is really the recommended approach during the development phase.

However, you may have situations where you want to run your integration tests using traditional tools, from the command line. One such situation would be if you are building your plugin with Bamboo. In that case, you can still use the traditional Atlassian `atlas-integration-test` command-line interface. The command executes only the `test` scope as defined in your `pom.xml` file (which also includes any unit tests you may have). Try the command now to run the generated test files:

1. Go to a command line.
2. Make sure you are in at the top level `PLUGIN_HOME` directory for you plugin.
3. Enter the `atlas-integration-test` command:

```
atlas-integration-test
```

Notice in command's output, the system runs the unit tests, starts the host application, and then runs your integration tests. The command output should succeed with test output similar to the following:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running
it.com.example.plugins.tutorial.jira.testTutorial.MyComponentWiredT
est
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for
further details.
[INFO] [talledLocalContainer] Nov 15, 2012 11:32:50 AM
com.sun.jersey.server.impl.application.WebApplicationImpl _initiate
[INFO] [talledLocalContainer] INFO: Initiating Jersey application,
version 'Jersey: 1.8-atlassian-6 03/12/2012 02:59 PM'
[INFO] [talledLocalContainer] Nov 15, 2012 11:32:51 AM
com.sun.jersey.api.wadl.config.WadlGeneratorLoader
loadWadlGenerator
[INFO] [talledLocalContainer] INFO: Loading wadlGenerator
com.sun.jersey.server.wadl.generators.WadlGeneratorApplicationDoc
[INFO] [talledLocalContainer] Nov 15, 2012 11:32:51 AM
com.sun.jersey.api.wadl.config.WadlGeneratorLoader
loadWadlGenerator
[INFO] [talledLocalContainer] INFO: Loading wadlGenerator
com.sun.jersey.server.wadl.generators.WadlGeneratorGrammarsSupport
[INFO] [talledLocalContainer] Nov 15, 2012 11:32:51 AM
com.sun.jersey.api.wadl.config.WadlGeneratorLoader
loadWadlGenerator
[INFO] [talledLocalContainer] INFO: Loading wadlGenerator
com.atlassian.plugins.rest.doclet.generators.resourcedoc.AtlassianW
adlGeneratorResourceDocSupport
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
3.185 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

When you next run your plugin with `atlas-run`, the command also runs both your unit and integration tests. If the build were to fail on unit or integration tests, the application does not start.

Step 2. Create a Traditional Integration Test

You may have existing traditional integration tests you want to maintain for some time. You can still do so. The difference between traditional tests and wired tests is the use of the `AtlassianPluginsTestRunner`. If your tests omit the `@RunWith(AtlassianPluginsTestRunner.class)` annotation, they are treated as traditional integration tests. Try adding a traditional integration test. If you haven't already done so, start Eclipse and open the testTutorial project you created.

1. Go to the `PLUGIN_HOME/src/test/java/it/com/atlassian/plugins/tutorials/jira/testTutorial` folder.
2. Create a `MyComponentTrdTest.java` file.
3. Edit the file and add the following code:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MyComponentTrdTest
{
    @Test
    public void testSomeFailure()
    {
        System.out.println("I RAN But failed...");
        assertEquals("something failed","blah","boo");
    }
}
```

4. Save and close the file.

Step 3. Configure Test Groups

When your tests run, Surefire creates a `PLUGIN_HOME/target/`*`testgroupname`* directory to hold temporary files during the test generation and reports. At this point, you haven't defined any test groups. So, Surefire created a `PLUGIN_HOME/target/group-__no_test_group__` directory. Test groups are useful if you are running a lot of integration tests. You can create test groups that represents categories in your tests. Try this now.

While still in Eclipse and do the following to define a `<testGroup>` for your integration tests.

1. Open your `PLUGIN_HOME/pom.xml` file.
2. Locate the `<build>` section.
   This section lists each plugin you are building. You are using a built-in Atlassian Maven plugin to run your plugin build. Locate the plugin with the `<groupId>` of `com.atlassian.maven.plugins` you should see the following:

```
<plugin>
 <groupId>com.atlassian.maven.plugins</groupId>
 <artifactId>maven-jira-plugin</artifactId>
 <version>${amps.version}</version>
 <extensions>true</extensions>
 <configuration>
  <productVersion>${jira.version}</productVersion>
  <productDataVersion>${jira.version}</productDataVersion>
 </configuration>
</plugin>
```

3. Define a `<testGroups>` section with two `<testGroup>` elements inside the configuration:

```
<plugin>
 <groupId>com.atlassian.maven.plugins</groupId>
 <artifactId>maven-jira-plugin</artifactId>
 <version>${amps.version}</version>
 <extensions>true</extensions>
 <configuration>
  <productVersion>${jira.version}</productVersion>
  <productDataVersion>${jira.version}</productDataVersion>
  <testGroups>
   <testGroup>
    <id>wired-integration</id>
    <productIds>
     <productId>jira</productId>
    </productIds>
    <includes>
     <include>it/**/*WiredTest.java</include>
    </includes>
   </testGroup>
   <testGroup>
    <id>traditional-integration</id>
    <productIds>
     <productId>jira</productId>
    </productIds>
    <includes>
     <include>it/**/*TrdTest.java</include>
    </includes>
   </testGroup>
  </testGroups>
 </configuration>
</plugin>
```

You defined a test group consisting of all the test files ending with `WiredTest` and another for traditional tests `TrdTest`. If you have multiple `<testGroups>` defined, you can run a subset of those groups by using the `-DtestGroup=`*`groupname`* flag on the command line.

4. Save and close the file.
5. Re-run `atlas-integration-tests` but only for the traditional tests.

```
atlas-integration-test -DtestGroup=traditional-integration
```

Surefire creates the `PLUGIN_HOME/target/group-traditional-integration` directory

6. Change directory to `PLUGIN_HOME/target/group-traditional-integration/tomcat6x/surefire-reports` directory.
7. List the directory contents.
   You should see something simliar to the following:

```
TEST-it.com.example.plugins.tutorial.jira.testTutorial.MyComponentTrdTest.xml
it.com.example.plugins.tutorial.jira.testTutorial.MyComponentTrdTest.txt
```

Take some time and browse the contents of each file.

8. Try the command again without the `-D` flag.
   Surefire creates a directory for each group.

Next Steps

In this page and the previous, you learned how to build, configure, and run tests. All the testing is run from the command line, this can make the development process time consuming and awkward. In the next section, you le arn how to use wired test together with the test console to test your code and speed up development.

Run Wired Tests with the Plugin Test Console

In previous pages in this tutorial, you learned how to use JUnit to construct unit tests, how to configure and run your traditional integration tests, and how to use the `atlas-` commands to run tests. On this page, you learn how to speed up and simplify plugin development using the Atlassian Wired Test Framework and the Plugin Test Console.

Overview of the Atlassian Wired Test Framework

While the `atlas-unit-test` and `atlas-integration-test` commands allow you to run tests from the command line, neither allowed for a fast development process. The Atlassian Wired Test Framework lets you test your plugin in the context of the host Atlassian product. (In contrast, the traditional plugin testing framework required the use of a host-mocking framework, such as Mockito.)

An additional advantage of the framework is that it facilitates faster test development. Instead of recompiling your plugin and restarting the host application every time you make a change to a test, you can use the test console to re-run the test, without having to recompile and restart the host.

Running as a plugin in product has several advantages:

- Test code is deployed as a plugin bundle in the same OSGI-fied Tomcat container as the host application.
- Your code can use all the JUnit functionality and any additional functionality you would be able to do in non-test plugin.
- Dependencies in your test code are "wired" by Spring inside the container.
- Test code runs in the product container and reports back to the locally running JUnit.
- Tests results display in the Plugin Test Console in addition to Surefire reports.

Finally, since your tests are running in the product, you can test your plugins in a more realistic context.

JUnit Enhancements for Wired Tests

The Atlassian Wired Test Framework includes some enhancements that causes difference in how you use JUnit annotation inside of Atlassian Wired Tests. The following table lists these differences.

Standard JUnit Test	Atlassian Wired Test
Requires a single zero-argument, public constructor.	Can use constructor for dependency injection.
Tests are stateless and every method is run on its own instance of the test class.	Tests are stateful. All methods run on the same test class instance. You must be careful to clean up any data at the end of your methods!
`@BeforeClass` and `@AfterClass` must annotate a `public static void` method.	`@BeforeClass` and `@AfterClass` must ***not*** annot ate a static method. These methods ***should*** annotate a `public void` method.

The `@Before` and `@After` annotations remain the same both in standard and wired tests; they annotate `publi c void` methods.

Step 1. Check for the Wired Dependencies

If you have followed along through this tutorial, you should already have the correct dependencies in your `pom.xml`. If you are upgrading a `pom.xml` file you used from a pre-existing project, you may need to add dependencies to it. Regardless, it is good to check your dependencies.

1. Navigate to the top of your project's `PLUGIN_HOME`.
2. Open the `pom.xml` file.
3. Make sure you have the following wired test dependencies:

```
<dependencies>
....

   <!-- WIRED TEST RUNNER DEPENDENCIES -->
  <dependency>
    <groupId>com.atlassian.plugins</groupId>
    <artifactId>atlassian-plugins-osgi-testrunner</artifactId>
    <version>${plugin.testrunner.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>jsr311-api</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
   </dependency>
   <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.2.2-atlassian-1</version>
  </dependency>
</dependencies>
```

   Specifying the `jrs311-api` and `gson` dependencies overrides the transitive versions Maven would have fetched.
4. Make sure that your JUnit dependency is set to 4.10 or higher:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.10</version>
  <scope>test</scope>
</dependency>
```

5. Make your project `<properties>` include the following values:

```
<amps.version>4.1</amps.version>
<plugin.testrunner.version>1.1</plugin.testrunner.version>
```

6. Save any changes you made and close the file.

Step 2. Examine the atlassian-plugin.xml File for Your Tests

Remember, your wired tests get bundled together as a plugin. Like any other Atlassian plugin, your test suite needs a plugin descriptor file (`atlassian-plugin.xml`). The descriptor lists any components or modules

needed by your integration tests. At runtime, Spring examines this plugin file and wires in the modules needed by your tests.

The Plugins Test Console always runs and displays results from wired tests. It can also run and display the results of unit tests and traditional integration tests. It can do this only if your descriptor file also includes any resources need by these tests as well. For example, unit tests typically have dependencies on the base plugin. So, your descriptor should make sure to include any base plugin dependencies.

If you are following along with this tutorial, the `atlassian-plugin.xml` descriptor file was created for you in the `PLUGIN_HOME/src/test/resources` directory. To convert, a pre-4.1 SDK existing project over to using the Wired Test Framework, you'll need to add this file yourself. Take a moment to familiarize yourself with the test descriptor file and how it differs from your standard descriptor.

1. Navigate to your project's `PLUGIN_HOME/src/test/resources` directory.
2. `Open the atlassian-plugin.xml` file.
   It should look similar to the following:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}-tests"
name="${project.name}" plugins-version="2">
   <plugin-info>
        <description>${project.description}</description>
        <version>${project.version}</version>
        <vendor name="${project.organization.name}"
url="${project.organization.url}" />
    </plugin-info>
  <!-- from our base plugin -->
  <component-import key="myComponent"
interface="com.example.plugins.tutorial.jira.testTutorial.MyPluginComponent"/>
  <!-- from the product container -->
   <component-import key="applicationProperties"
interface="com.atlassian.sal.api.ApplicationProperties" />

</atlassian-plugin>
```

3. Note that your test project `key` is different from your project's `key`.
   The test `key` has a `-tests` identifier at the end of the plugin key as follows:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}-tests"
name="${project.name}" plugins-version="2">
 ...
```

   This identifier distinguishes your test bundle from your project bundle in the OSGI container.
4. Locate the `component-import` from the base plugin.
   You'll recall that your `MyComponentWiredTest.java` file tests a method on your base plugin. This import tells Spring that your test bundle needs an interface from another bundle, the plugin.
5. Locate the `component-import` from the product container.
   Your wired test relies on the Shared Access Layer (SAL). This is a component common to all Atlassian applications. So, here you are just telling Spring to wire in SAL as well.
6. Close the file.

You may want specific modules in your test plugin that you don't have in your main plugin. If that is the case, you would add them to this descriptor rather than your main descriptor. Any dependencies you need to declare to support the component-imports would go into your `pom.xml` file.

Step 3. Launch the Application and the Plugin Test Console

Now, you are ready to run the host application and view your tests in the Plugin Test Console.

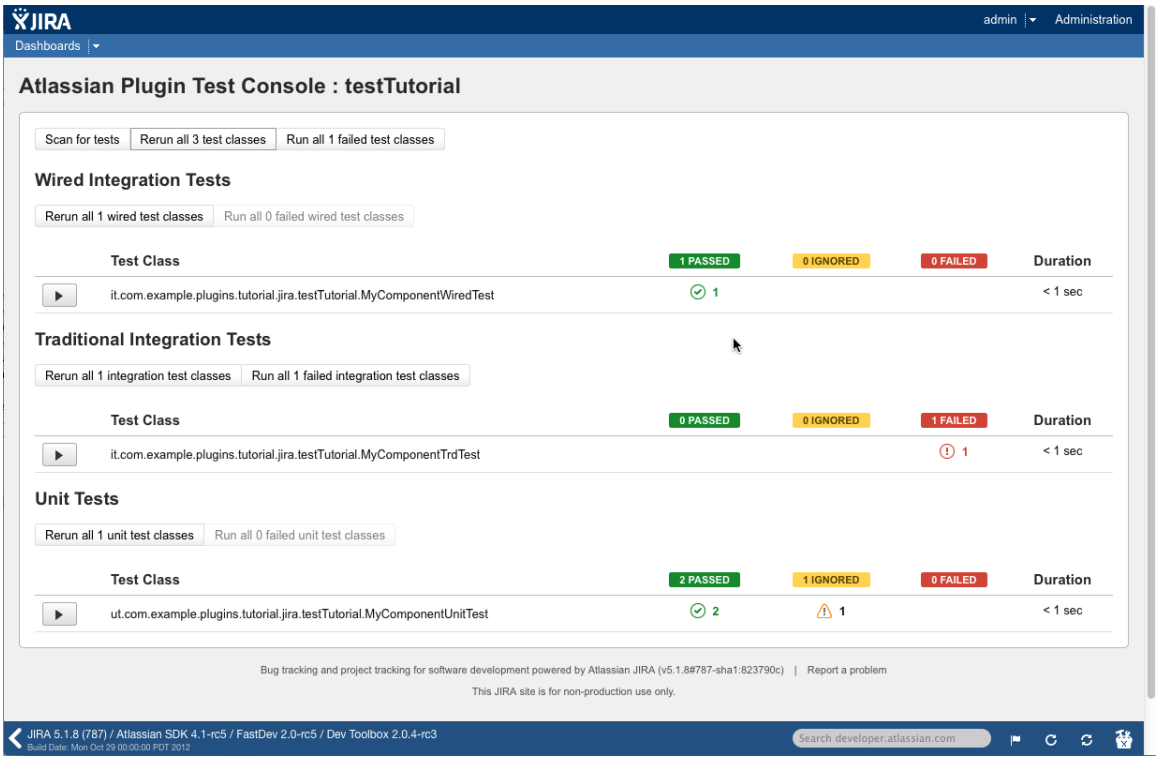
1. Open a command or terminal window.
2. Change directory to the `PLUGIN_HOME` directory.
3. Start the application with the `atlas-debug` command.

```
atlas-debug
```

When the command succeeds, it displays the URL for the host application:

```
...
[INFO] jira started successfully in 106s at
http://localhost:2990/jira
[INFO] Type Ctrl-D to shutdown gracefully
[INFO] Type Ctrl-C to exit
```

4. Copy and paste the URL into your browser's address field.
5. The browser displays the host application, in the case of this tutorial, JIRA.
6. Enter `admin` for both the **User** and the **Password**.
7. Display the Developer Toolbar by clicking on the arrow in the lower left corner of your browser.
8. Click **Toolbox > Plugin Test Console**.
   The test console appears. The console lists the type of tests it found in your project.
9. Press **Rerun all 3 test classes** to run your tests and display their results:



Step 4. Make a Change to Your Running Test Suite

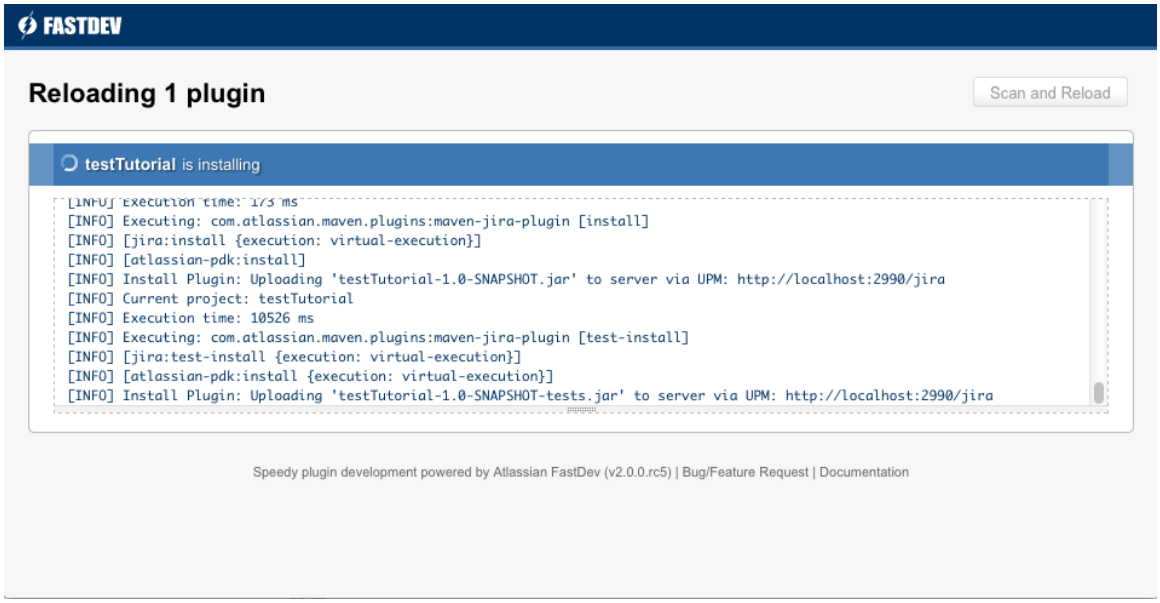
Like any other Atlassian plugin, you can use the FastDev feature with your test plugin. Try this now:

1. Leave the **Plugin Test Console** running in your browser.
2. Edit the `MyComponentWiredTest` class.

3. Add the `@Ignore` annotation to the `testMyName()` method.
4. Add the Ignore import to the test file:

```
import org.junit.Ignore;
```

5. Save the test file.
6. Return to the **Plugin Test Console**.
7. Press the play button next to the test you changed.
   The system recognizes that you changed the underlying tests. It launches FastDev to rebuild your test plugin:



   When the build completes, the system returns you to the Plugin Test Console.
8. Now try changing a unit test or a traditional integration tests.
   Just as with wired tests, the system recognizes the changes and launches FastDev.

Next Steps

This page taught you how to code using the Atlassian Wired Test Framework. Tests that use this framework are plugins that use Spring dependency injection to run inside an Atlassian host application. When you use the framework, you have access to the Plugin Test Console. This console allows you to run test and view their results right in the application. When your underlying test code changes, the system recognizes the change and launches FastDev to rebuild your tests.

At this point, all you really need is some test data. In the next section, you learn how to seed your host application with test data.

Create Test Data and a Test Fixture

Most tests, and in particular integration tests, rely on sample data for testing. You need sample data in the host application and may also need sample data specific to your plugin. This page explains how to populate your test instance with data for testing.

Initial Test Application Configuration

For testing purposes, Atlassian stores a configuration for each host application in Atlassian's Maven repository. This data is a zip archive of a fully configured application's home directory. The `home` directory includes

database properties, search indexes, caches, and more. If you are using the embedded HSQLDB database supplied for evaluation purposes, the database files are also stored in this directory.

The host configuration your plugin uses is specified in the `pom.xml` file through the `<productDataVersion>` value. This value is set automatically for you when you generate a project. When you first start the application and load your plugin, the `atlas-` commands download the specified `<productDataVersion>`, unzip it, and use it to seed the `PLUGIN_HOME/target/application/home` directory.

The following table lists the repository location for each test configuration:

Product	Artifact ID with Link to Maven Repo
JIRA	jira-plugin-test-resources
Confluence	confluence-plugin-test-resources
Bamboo	bamboo-plugin-test-resources
Crowd	crowd-plugin-test-resources
FishEye/Crucible	amps-fecru
Stash	stash-plugin-test-resources

The product configuration data does not *have* to match the application version you are testing. In many cases, you can use an older `<productDataVersion>`. For example, you can specify a `<productDataVersion>` of 3.1 but write a plugin for Confluence 3.2. When you start the Confluence 3.2 with an `atlas-` command, the system automatically updates the 3.1 data to 3.2.

To the initial host configuration, you may want to add a set of test data that you use in testing. You might store some test data your in your project's `PLUGIN_HOME/src/test/resources` directory. Other tests, especially integration and functional tests, can require data to exist in the host application itself. For example, a JIRA project and issues or pages and a space for Confluence.

The combination of the host application and an initial data set would form the basis of a *test fixture*. You use a test fixture to ensure that your plugin is tested against a well known and fixed environment every time the test suite runs. Then, problems that occur in a test run are either the result of problems in your code — plugin or test code. The rest of this article, explains how to create an initial set of test data.

Step 1. Know your Project Data Version

In this step, you check your `pom.xml` and verify the data version you want to use for your tests. If you are following along the tutorial exactly, go ahead and launch the Eclipse IDE. Otherwise, use your favorite editor to do the following:

1. Edit the `pom.xml` file.
2. Locate the `<build>` section of the file.
3. Locate the `<configuration>` for the `com.atlassian.maven.plugins`.
4. Make sure you have a `<productDataVersion>` defined.
   If you have followed along with this tutorial, you should seem something like the following:

```
<build>
      <plugins>
         <plugin>
            <groupId>com.atlassian.maven.plugins</groupId>
            <artifactId>maven-jira-plugin</artifactId>
            <version>${amps.version}</version>
            <extensions>true</extensions>
            <configuration>
               <productVersion>${jira.version}</productVersion>
               <productDataVersion>${jira.version}</productDataVersion>
               <testGroups>
                  <testGroup>
                     <id>wired-integration</id>
                     <includes>
                        <include>it/**/*WiredTest.java</include>
                     </includes>
                  </testGroup>
                  <testGroup>
                     <id>traditional-integration</id>
                     <includes>
                        <include>it/**/*TrdTest.java</include>
                     </includes>
                  </testGroup>
               </testGroups>
            </configuration>
         </plugin>
         <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
               <source>1.6</source>
               <target>1.6</target>
            </configuration>
         </plugin>
      </plugins>
   </build>
```

5. Check the `<productDataVersion>` against the `<productVersion>`.
   The your product and data version don't have to be the same. It is a good idea that they within one or two point releases of each other.
6. Close the file.

Step 2. Create the Test Data Archive

Now, you are ready to create some in-product test data for your tests. If you have complex tests that need a lot of data, you may import a subset of data from a production instance of a host application. For smaller plugins, this might not be necessary. Your set of test data may be very small. Only you can decide what is appropriate for your plugin and your tests. The technique in this tutorial is to manually create some test data. Go to the command window or terminal on your system and do the following:

1. Navigate to your `PLUGIN_HOME`.
2. Start the application passing it the parameter to skip running any tests.

```
atlas-run -DskipTests=true
```

3. Start your browser and log into JIRA.
4. Create a project and add one or two issues to it.
5. Return to the command line and stop the application.
6. Make sure your current directory is your `PLUGIN_HOME`.
7. Create a zip of the application home directory in your target.

```
atlas-create-home-zip
```

   This command creates a `PLUGIN_HOME/target/application/generated-test-resources.zip` file. Since the target directory is often cleaned or removed, you want to copy this file to another location.
8. Again, make sure your current directory is your `PLUGIN_HOME`.
9. Copy the test resources to your `PLUGIN_HOME/src/test/resources` directory.

```
cp target/jira/generated-test-resources.zip
PLUGIN_HOME/src/test/resources
```

   This directory is not cleared out when you run the `atlas-clean` command.

Step 3. Configure Your Plugin with the Test Data

The ZIP archive you created in the previous step is now part of your plugin test data. In this step, you configure your `pom.xml` file to load this data into the host application. Then, you'll run the application and confirm the data was loaded.

1. Return to the command line.
2. Make sure your current directory is your `PLUGIN_HOME`.
3. Go ahead and remove the target directory by entering the following command:

```
atlas-clean
```

   Cleaning the target directory from your project is not something you have to do often. You do it at this point because you want to make sure that your project picks up the test data from your resources rather than anything left behind in the `target` directory.
4. Edit your `pom.xml` file.
5. Locate the `<build>` section of the file.
6. Locate the `<configuration>` for the `com.atlassian.maven.plugins`.
7. Add a `<productDataPath>`specification and point it to your test data ZIP archive.
   When you are done your configuration will look similar to the following:

```
<configuration>
    <productVersion>${jira.version}</productVersion>
    <productDataVersion>${jira.version}</productDataVersion>


<productDataPath>${basedir}/src/test/resources/generated-test-resou
rces.zip</productDataPath>
...
```

8. Save and close the file.

9. Return to the command line and start the application in debug mode.

```
atlas-debug
```

   You could also use `atlas-run -DskipTests=true` it is entirely your choice.
10. After the build completes, log into the application.
    You should find that your test data also now is loaded into the application.

Next Steps

You've completed the Writing and Running Plugin Test tutorial. If you want to check your work, you can find the plugin source code on Atlassian Bitbucket. Bitbucket serves a public Git repository containing the tutorial's code. To clone the repository, issue the following command:

```
git clone
https://atlassian_tutorial@bitbucket.org/atlassian_tutorial/testtutorial
.git
```

Alternatively, you can download the latest source here: https://bitbucket.org/atlassian_tutorial/testtutorial/downloads.

The information in this tutorial was foundation information. Meaning, you can use what you learned here regardless of which host application your plugin is for. Documentation for each host application (available from this site) contains additional and more specific information about testing in those applications.

Staff README for Foundation Test Docs

> This page is restricted to **atlassian-staff** by design. Do not remove the restriction on this page.

This page documents the status and suggestions for improvements to the Foundation test documentation.

Information Related to Working with these Pages

This tutorial is designed so each page builds on the next. It also provides readers with a sample code project. Readers that follow the tutorial exactly should be able to check their final code agains the Bitbucket project here:

https://bitbucket.org/atlassian_tutorial/testtutorial

Wish List for Improvements to this Tutorial

- A more robust sample application that illustrates real-world page actions. So, perhaps a servlet or page in the administration area, Javascript, and a template. How would a user test each major component? Since this tutorial is a foundation tutorial, the sample application should be cross-functional.
- Include a section on Javascript testing for Qunit. This should follow after updating the sample application. Reference page for how to is on EAC here: https://extranet.atlassian.com/pages/viewpage.action?pageId=2020278642. Make sure to look at the comments because they reference other aspects of this work.
- Currently, all the code assumes an HSQL database. In practice, testing with that database is not robust enough. We should provide a page instructions for using another more production-viable database such as MySQL
- Running tests manually, using the test console are both demonstrated. Running tests from a Bamboo project is not. The considerations are different but applicable. We should be encourage plugin

programmers to automate their testing and to use our product — Bamboo.
This text is for AOD THIS IS TEXT FOR BITBUCKET

this text that appears after