JavaOne October 2–6, 2011 San Francisco

MOVING JAVA FORWARD

# JavaONE 2011 Hands-on Lab: 25010

# Using Hudson's Plugin Development Framework to Build Your First Hudson Plug-in


**Winston Prakash**
Consulting Member of Technical Staff, Oracle Corporation

**Susan Duncan**
Product Manager, Oracle Corporation



Oracle is committed to developing practices and products that help protect the environment

ORACLE

# Introduction

The Hudson Continuous Integration system is a highly extensible platform. It consists of core functionality and plugins that extend that functionality. Plugins allow users and developers to do everything from customize the way build results are output to integration with Application Lifecycle Management systems such as SCM, Testing and Analysis tools. A number of plugins are 'bundled' with a standard Hudson install, others can be downloaded and installed as and when needed by users.

Hudson provides a series of extension points that allow developers to extend Hudson's functionality. The Hudson HPI (Hudson Plug-in Interface), a Maven plugin, helps developers to create, build, run and debug plugins.

In this HOL, you will get hands-on experience with

- Creating your first Hudson plugin with the Hudson `HPI`
- Using the NetBeans IDE to run, edit, and debug the plugin
- Extending a Hudson Extension Point to provide your own implementation
- Writing configuration UI for the Extension Point
- How to add Global Configuration for the Extension Point

# Prerequisites

This hands-on lab assumes you have basic experience with the following technologies:

- Java
- Hudson
- Maven

# Software Required

For this JavaOne Hands-on Lab the following software is provided for you:

- JDK 6.0
- Maven 3.0 for building the plugin
- Netbeans 7.0 for editing the plugin

# Lab Exercises

1. Create and run your first Hudson plugin
2. Extend the Builder Extension Point
3. Create UI to configure the extension

# Exercise 1: Create and Run your First Hudson Plugin

In this exercise you will learn how to
- create your first Hudson plugin using the Hudson `HPI` tool
- Open the plugin project in the Netbeans IDE
- Run the plugin project and view it in action in the Hudson Test Server.

## Step 1: Generate the plugin skeleton

**1.1.1** The very first step is to create your plugin skeleton. The Hudson HPI is a Maven plugin (maven-hpi-plugin). It generates the minimal sources for you. In a terminal type the command

```
mvn hpi:create
```

This command tells `maven` (invoked with `mvn` command) to create the necessary sources. Wait until maven downloads all the required jars and plugins to execute the command.

**1.1.2** When prompted provide:

```
Enter the groupId of your plugin: org.sample.hudson
Enter the artifactId of your plugin: javaone-sample
```

When `maven` successfully creates the Hudson plugin, the following information will be printed.

```
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating Archetype: maven-hpi-plugin:2.1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value:  org.sample.hudson
[INFO] Parameter: packageName, Value:  org.sample.hudson
[INFO] Parameter: package, Value:  org.sample.hudson
[INFO] Parameter: artifactId, Value: javaone-sample
[INFO] Parameter: basedir, Value: /home/javaone2012/user12
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ******** End of debug info from resources from generated POM ***********
[INFO] Archetype created in dir: /home/javaone2011/user12/javaone-sample
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

***Troubleshooting:***

If you see the following error

```
mvn: : command not found
```

make sure `maven` is installed.

```
[ERROR] No plugin found for prefix 'hpi' in the current project ..
```

Check $USER_DIR/`.m2/settings.xml` ($USER_DIR is the your home directory where your files are kept) and make sure it has the following entry

```
<pluginGroups>
    <pluginGroup>org.jvnet.hudson.tools</pluginGroup>
</pluginGroups>
```

**1.1.3** Once your Hudson plugin project is successfully created, examine the generated source folder. It will have the following layout

*pom.xml* - Maven POM file which is used to build your plugin
*src/main/java* - Java source files of the plugin
*src/main/resources* - Jelly view files of the plugin.
*src/main/webapp* - Static resources of the plugin, such as images and HTML files.

**Note:** `Jelly` is a Server Side View technology used by Hudson. Jelly files are XML files that are rendered to client side view pages such as HTML, CSS, Javascript etc.

### Step 2: Build the plugin project

**1.2.1** The plugin project created in Step 1 is fully functional. You can build it and run it without any modification. Build the project using `maven`

```
cd javaone-sample
mvn package
```

The `package` command tells maven to build the project and create the plugin packages needed by the Hudson server. On successful build of the plugin project the following message is displayed

```
[INFO] --- maven-hpi-plugin:2.1.0:hpi (default-hpi) @ javaone-sample ---
[INFO] Exploding webapp...
[INFO] Copy webapp webResources to /home/javaone2012/user12/javaone-sample/target/javaone-sample
[INFO] Assembling webapp javaone-sample in /home/javaone2012/user12/javaone-sample/target/javaone-sample
[INFO] Generating hpi /home/javaone2011/user12/javaone-sample/target/javaone-sample.hpi
[INFO] Generating /home/javaone2011/user12/javaone-sample/target/javaone-sample/META-INF/MANIFEST.MF
[INFO] Targeting Hudson-Version: 2.1.0
[INFO] Building jar: /home/javaone2011/user12/javaone-sample/target/javaone-sample.hpi
```

```
[INFO] Building jar: /home/javaone2011/user12/javaone-sample/target/javaone-sample.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

The next step is to run the plugin project and view the result. Netbeans will be used for the remainder of the exercise

## Step 3: Open the plugin project using Netbeans IDE

This step requires `Netbeans` IDE 7.0 or higher. In the JavaONE lab environment, `Netbeans` is pre-installed for you.

**1.3.1** Click on the `Netbeans` icon on your desktop to start the `Netbeans` IDE.

**1.3.2** Open the created plugin project using **`File -> Open project`** or click on the `Open project` icon in the toolbar

**1.3.3** In the resulting dialog browse and choose the plugin project to open it in the `Netbeans` IDE as in Figure 1

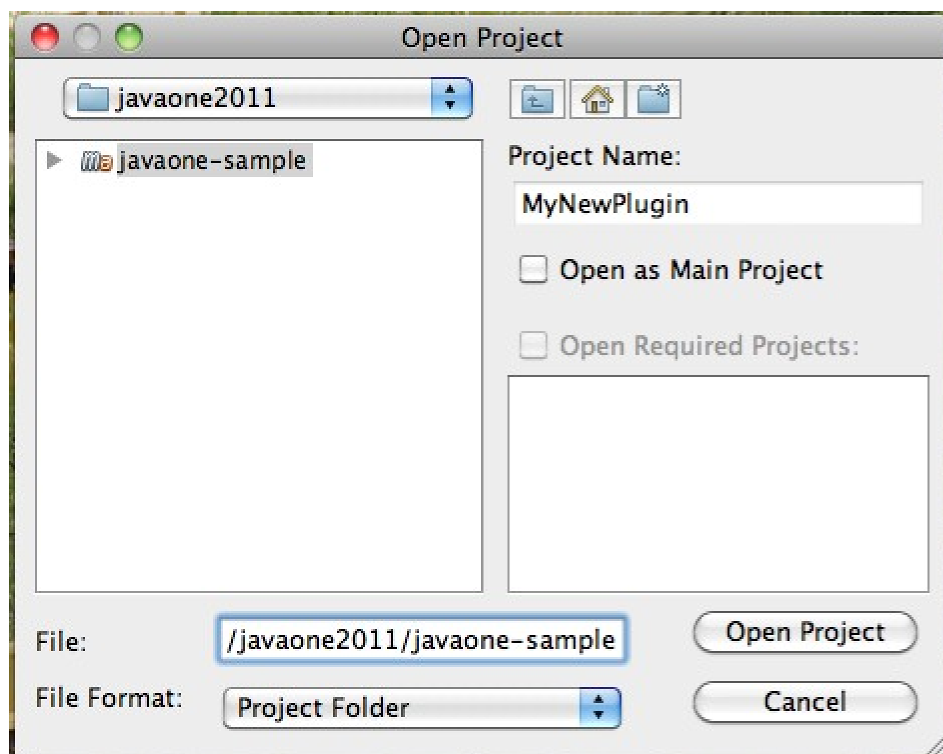

Figure 1: Open the plugin project in Netbeans

The project opens in the IDE with the display name “`MyNewPlugin hpi`”.

**Note:** You can rename the project to your desired name by right clicking on the project name and selecting **Rename...**. In this Lab Tutorial we will refer the plugin projectt as as "**MyNewPlugin**"

## Step 4: Run the Plugin Project

In this step you run the project and see the result of the extension added by plugin.

**1.4.1** Right click on the project in the `Projects Explorer` window and select **`Run`** as shown in Figure 2.

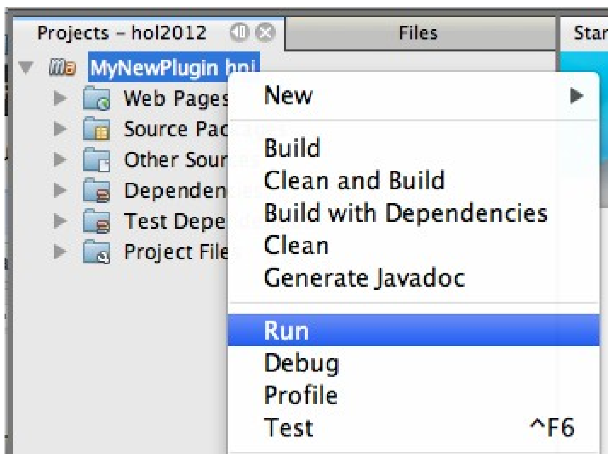

Figure 2: Run the plugin project

**1.4.2** Output of your `Run` command will be displayed in the `Output` window (Figure 3) in the bottom right side of your IDE. Any errors encountered will be displayed here.

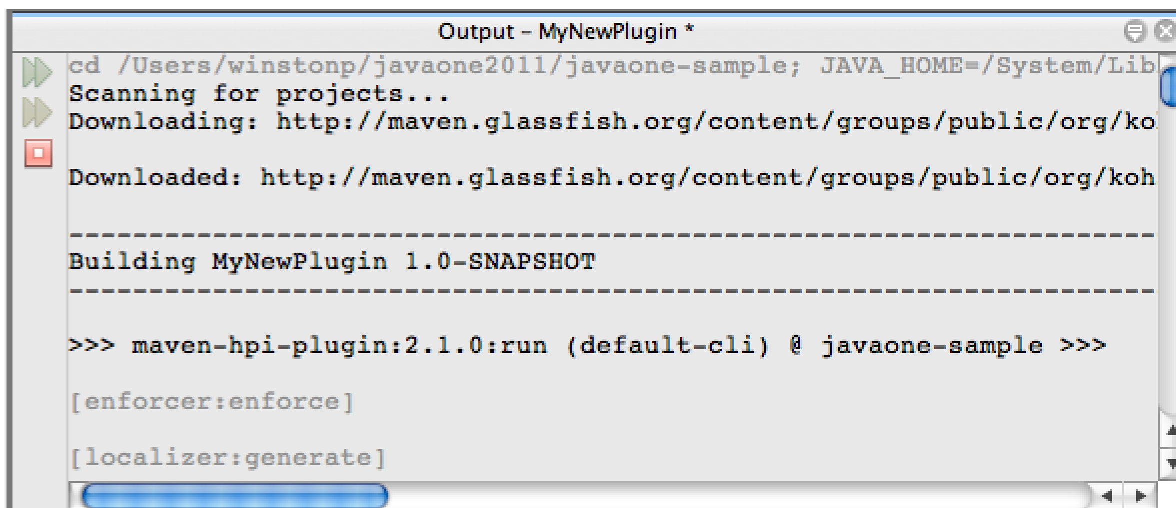

Figure 3: Output window

**1.4.3** For a successful plugin project `Run`, a browser window opens as the Hudson server starts up. Initially it displays the message “`Please Wait Hudson is getting ready to work`”.

**1.4.4** Once the Hudson server is fully started the browser window refreshes to the Hudson main page as in Figure 4

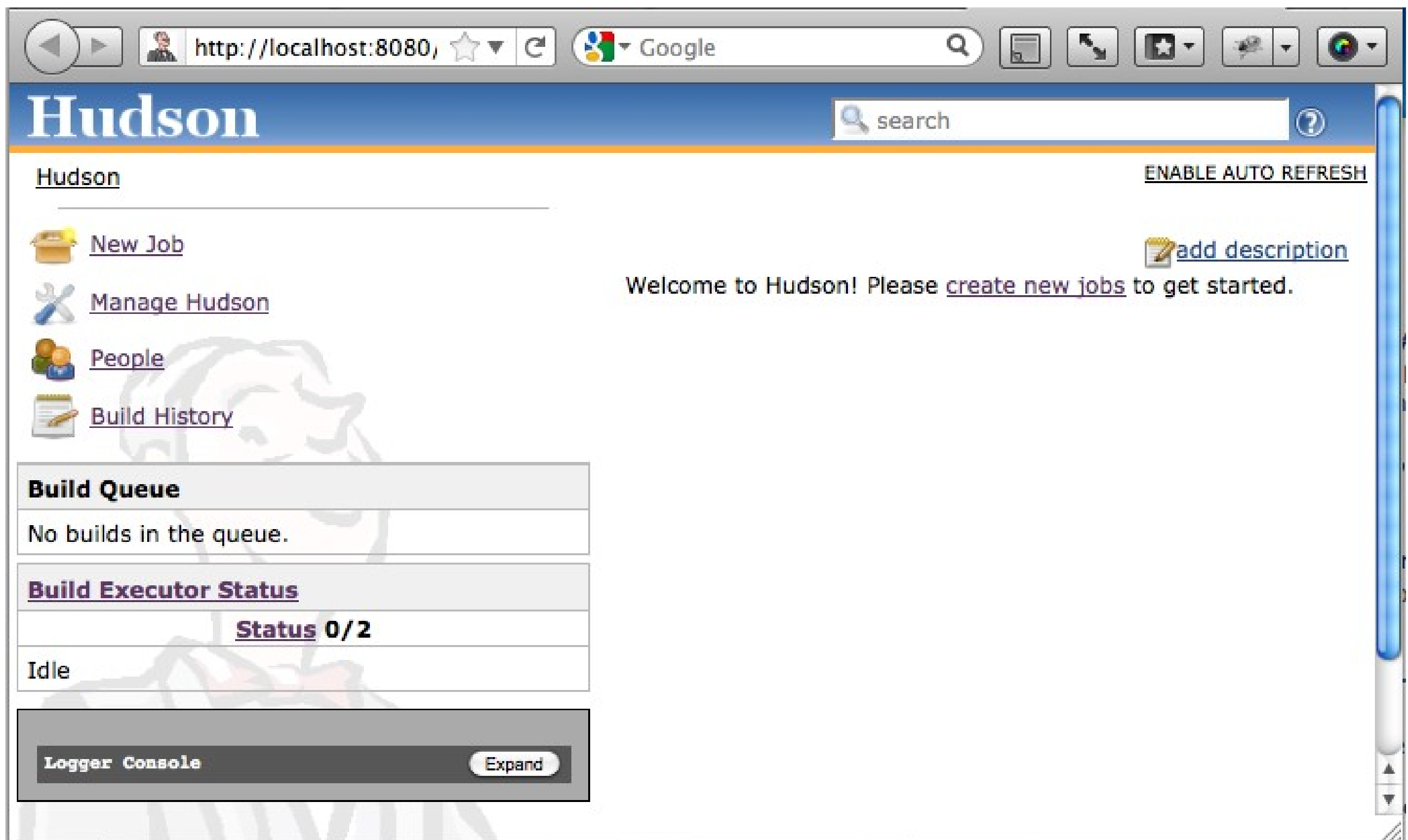


Figure 4: Hudson Main Page

**1.4.5** Click on `Manage Hudson` link on the left hand side

**1.4.6** Click the link `Manage Plugins`. This action opens the `Plugin Manager` page.

**1.4.7** Click on the `Installed` tab to see the list of all installed plugins. One of the installed plugin is your newly created plugin “`MyNewPlugin`” version `1.0-SNAPSHOT` as shown in Figure 4a

Updates | Available | **Installed** | Advanced

| Enabled | Name ↓ | Version |
| --- | --- | --- |
| ☑ | CVS Plugin | 2.0.1 |
| ☑ | Git Plugin<br>This plugin integrates GIT with Hudson. | 2.0.1 |
| ☑ | MyNewPlugin<br>This plugin is a sample plugin to explain how to write a Hudson plugin. | 1.0-SNAPSHOT (private-09/12/2011 08:36-winstonp) |

Figure 4a: Installed Plugins displayed by Hudson Plugin manager

### Step 5: Looking under the hood of the project `Run` action

You might be thinking:

- What happened in the `Run Project` action?
- How was the Hudson server was started?
- How ddid the new plugin get installed on the Hudson server?
- Which Hudson Home was used by the Hudson Server?

**1.5.1** For this you need to look at the messages printed in the `Output Window` of Netbeans IDE. The plugin project is a maven project. So the configuration of the project is defined in the POM.xml. This is where the magic happens.

The `Run Action` simply invokes Maven with hpi:run as the goal. This is made obvious by the message in the `Output Window`

```
<<< maven-hpi-plugin:2.1.0:run (default-cli) @ javaone-sample <<<
```

`hpi:run` is responsible for several of the tasks including starting the Jetty Server, adding the `Hudson Web Application` to the Jetty Server. The corresponding messages in the Output Window are

```
Starting jetty 6.1.1 ...
2011-08-26 18:14:39.476::INFO:  jetty-6.1.1
2011-08-26 18:14:39.496::INFO:  Extract
jar:file:/home/javaone2011/user12/.m2/repository/org/jvnet/hudson/main/hudson-war/2.1.0/hudson-war-
2.1.0.war!/ to /home/javaone2011/user12/javaone-sample/target/work/webapp
Aug 26, 2011 6:14:43 PM hudson.WebAppMain contextInitialized
INFO: Home directory: /home/javaone2011/user12/javaone-sample/work
```

**Note:** Jetty is a light weight HTTP Server and Servlet container to run Java Web Applications.

**Tip:** If you want to run the plugin project from command line with out using any IDE, use the command  mvn hpi:run from with the plugin project folder.

The "work" sub-folder in the plugin project folder is set  as Hudson Home.

1.5.2 To find out how the  currently built plugin itself is added to the Hudson Server, expand the "work/plugins" sub-folder in the plugin project folder. This folder contains the list of .hpi files of the various plugins that are bundled in a default Hudson install. The only notable difference is

```
-rw-r--r--   1 user12  javaone2011  10400 Aug 26 18:14 javaone-sample.hpl
```

Note all other .hpi files are jar files where the entire  contents of the plugin is packages. But javaone-sample.hpl (it is not a .hpi file but a .hpl file), is a simple text file which contains metadata describing all the files (classes, jars and resources) associated with the it. This file is re-generated by the hpi  tool every time the plugin project is run using hpi:run.

```
[hpi:run]
Generating ./work/plugins/javaone-sample.hpl
```

Hudson knows how to interpret this file and load the entire plugin without packaging the plugin to a .hpi package. This makes it easy to debug during development time. During plugin release, the .hpi is generated.

### Step 6: Examining the `HelloBuilder` extension provided by the generated Plugin

As stated before the generated plugin is fully functional. By adding this plugin to Hudson server, it

- Added an extension to the builder interface. The custom builder called HelloBuilder  does not do anything fancy, but simply prints out the text "Hello <name>" in the build console log.
- Provided a UI to configure the HelloBuilder.
- Added a Global configuration for the HelloBuilder

**1.6.1** To see the HelloBuilder Extension in action, first  create a Hudson project. In the Hudson main page click on the **create new jobs** link or the **New Job**  link on the left hand side to start creating a new project.

**1.6.2** As shown in Figure 5 , in the New Job page enter "**TestProject**" in the Job Name field and select the **Build a free-style software project** radio button.

**1.6.3** Click the **OK** button at the bottom to create the Hudson project.

Job name TestProject

Build a free-style software project

This is the central feature of Hudson. Hudson will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Build a Maven 2/3 project (Legacy)

To build a Maven 2/3 project (Legacy) Hudson takes advantage of your POM files and drastically reduces the configuration. WARNING: This job type is deprecated - Maven builds are supported under the Build section of both free-style software project and multi-configuration project job types with Maven invocation build steps.

Figure 5: New Job Creation

**1.6.4** Once the project is created, you will be taken to the project configuration page. In the configuration page you could configure the project to do multitude of tasks. But for the sake of simplicity of this exercise, you will confine yourself to selecting the builder for the project only

**1.6.5** Scroll down to the `Build` section of the configuration page and click on the "**`Add Build Step`**" dropdown button. From the dropdown menu select "**`Say hello world`**", which is the `Extension` added by our new plugin as shown in Figure 6

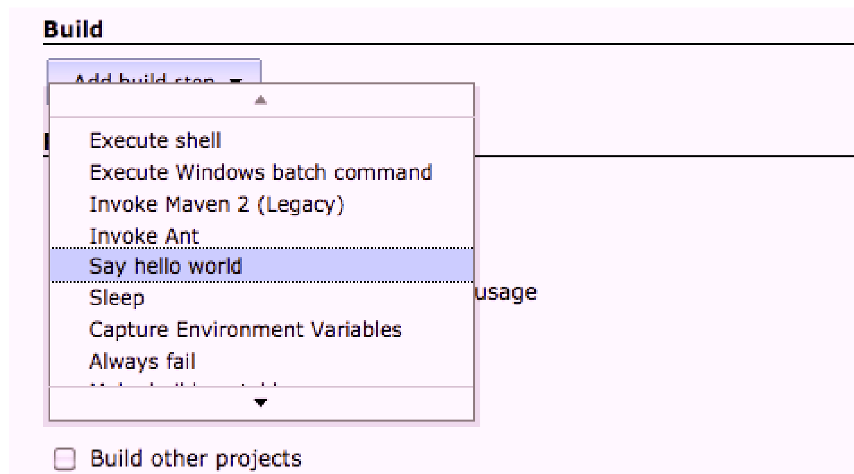


Figure 6: Add the 'Say hello world' build step

Once the `HelloBuilder` is set as the builder for the project, the Build section will display `HelloBuilder` as one of the `Builder` as shown in Figure 7

Figure 7: HelloBuilder Build Step

**1.6.6** The task of the HelloBuilder is to say “Hello <name>”. So HelloBuilder need to know your name. Type in your name in the Name field.

**1.6.7** Save the project configuration by clicking on the **Save** button at the bottom of the page.

Since we set HelloBuilder as the only builder of the project, when we start a build of the Hudson project, HelloBuilder will be asked to perform its tasks. The only task of the HelloBuilder is to print out the message “Hello <name>” to the console log.

**1.6.8** Start a build and see the message in the console log. Voila! the build console log contains the expected text "Hello <name>" as in Figure 8

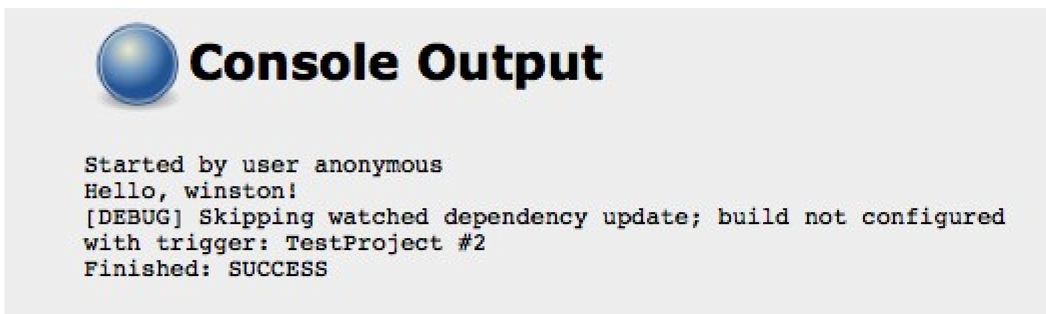


Figure 8 - Console Output Log

## Exercise 2: Extending the Builder Extension Point

In this exercise we will be looking at some of the code that extends the Builder Extension Point. In this exercise you will learn how to

- Extend an Extension Point
- Implement the methods to extend the functionality encapsulated by the Extension Point?

Hudson provides the concept of `Extension Points` and `Extensions` to facilitate plugins to contribute to some of the functionalities of the core platform. Extension points are interfaces that encapsulate entry points to extend certain services or functionalities of services provided by the core platform.

Amongst various services provided by Hudson, the foremost is building a `Job`. A `Job`, which is a `build-able project` consists of several configurable areas and `build steps`. They can be extended via Extension Points. Some of the build steps are

- **`SCM checkout`** `- Based on SCM type, source code check out`
- **`Pre-build`** `- Invoked to indicate that the build is starting`
- **`Build wrapper`** `- Prepare an environment for the build`
- **`Builder runs`** `- Actual building: calling Ant, Make, etc.`
- **`Recording`** `- Record the output from the build, such as test results.`
- **`Notification`** `- Send out notifications, based on the results determined so far.`

**`Builder`** `is another and very` important build step. The responsibility of a builder is to build a job. The Extension Point provided by Hudson to contribute to this builder run step is aptly called **`Builder`**.

Hudson comes bundled with two of the most popular builders - `Ant` and `Maven`. They are in fact `Extensions` to the `Builder` Extension Point. So it is possible for a plugin to provide its own `Builder` extension as a Builder of Jobs. Several external plugins have been contributed for other popular builders such as `make`, `gradle`, `rake` etc.

`HelloBuilder`, our example Builder Extension is a contrived example to understand how extensions are built. Far sophisticated `Builder extensions` are possible with the `Builder` Extension Point. First you will examine the source to understand how the extension mechanism works.

> **Note:** Even though we keep calling the Extension as `HelloBuilder` (*intention of the builder is to say hello to you, not to the world :)* ), the Java Class corresponding to it is called `HelloWorldBuilder.java`.

## Step 1: Examine the `HelloBuilder` Extension

**2.1.1** Open the `HelloWorldBuilder.java` with Netbeans Java Editor using the `Project Explorer` Window as shown in Figure 9

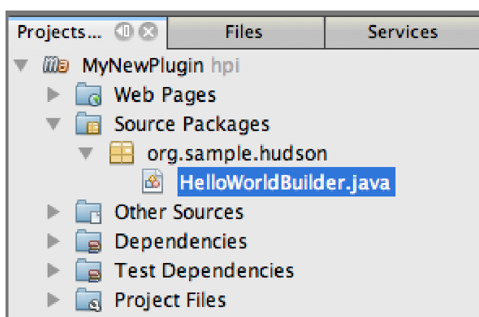


Figure 9: Project Explorer

In order for Hudson to understand your class as an Extension, it must

- Extend a class that advertise itself as an `Extension Point`
- Implement the required `abstract methods` to extend the functionality
- Tell Hudson that your class is an `Extension`

**2.1.2** Look at the line that declares the HelloWorldBuild class. Notice, the class `HelloWordBuilder` extends the class `Builder` which is the `Extension Point` for the `Builder` interface.

```
public class HelloWorldBuilder extends Builder {
```

**2.1.3** Navigate to skeleton of `Builder` class by doing a Ctr-Alt-Click (Ctr-Cmd on Mac) on the text **Builder**. In the opened skeleton class, notice the Builder Class implements `ExtensionPoint` which tells Hudson any Class extending Builder is a potential Extension

```
public abstract class Builder extends BuildStepCompatibilityLayer implements BuildStep,
Describable<Builder>, ExtensionPoint {
```

**2.1.4** Also notice `Builder` itself is a subclass of `BuildStep`, which defines the abstract method needed to be implemented by the `Extensions` to contribute to the functionality.

**2.1.5** Open the skeleton of `BuildStep` by clicking Ctr-Alt (Ctr-Cmd on Mac) on it. The abstract method needed to be implemented by Builder Extension is

```
public boolean perform(AbstractBuild<?, ?> ab, Launcher lnchr, BuildListener bl) throws
InterruptedException, IOException;
```

`BuildStep.perform(..)` overridden by `HelloBuilder` will be called by Hudson to include the `BuildStep` functionality extended by `HelloBuilder` extension.

**2.1.6** Finally to tell Hudson, your class is an `Extension` to some `Extension Point`, it must be annotated with the annotation `@Extension`. The annotation `@Extension` at the inner class `DescriptorImpl` tells Hudson your class is an `Extension`.

In the next exercise you will explore why the `@Extension` is at the inner class `DescriptorImpl`, rather than at the outer class `HelloWorldBuilder` itself.

```
@Extension
public static final class DescriptorImpl extends BuildStepDescriptor<Builder> {
```

Using the Java Annotation Processor, Hudson discovers and keeps a list of classes that are annotated with the annotation `@Extension` categorized by the Extension Point. When a particular service or functionality is extended by an `Extension Point`, all Extensions categorized by that Extension point are obtained and its hook methods are called. In

the case of Builder Extension Point, when the Job is built, all the Extensions corresponding to Builder extensions are obtained and those Extesnions configured in the job are selected and their `perform()` method is called.

### Step 2: Examine the Builder abstract method `perform()`

The `BuildStep.perform(..)` abstract method gives access to three objects

- `Build - Object representing the build of the job being performed. Build in turn give access to important model objects like`
  - `Project - The buildable Job`
  - `Workspace - The folder where the build happens`
  - `Result - Result of the build until this build step`
- `Launcher - Launcher which is used to launch the build of this job`
- `BuildListener - An interface to communicate the status of the build steps being performed in this Builder and send any console message from this Build Step to Hudson`

**2.2.1** `HelloBuilder` uses the `BuildListener` model object to print the Hello message to the console. Look at the code:

```
@Override
public boolean perform(AbstractBuild build, Launcher launcher, BuildListener listener)
{
    if(getDescriptor().useFrench())
        listener.getLogger().println("Bonjour, "+name+"!");
    else
        listener.getLogger().println("Hello, "+name+"!");
    return true;
}
```

**2.2.2** Give attention to **`listener.getLogger()`** that gets the `Logger` whose output goes to the console. Your plugin simply prints “Hello <name>” via the logger. You will look at the rest of the code in the next exercise.

### Step 3: Modify HelloBuilder perform(..) method

In this step you will add code to the perform(..) method to see

- How to use the launcher to execute an external executable
- How to send the result of execution to the console

**2.3.1** Add the following code to the perform(..) method

```
public boolean perform(AbstractBuild build, Launcher launcher, BuildListener listener) {
    if (getDescriptor().useFrench()) {
        listener.getLogger().println("Bonjour, " + name + "!");
    } else {
        listener.getLogger().println("Hello, " + name + "!");
    }

    List<Cause> buildStepCause = new ArrayList();
    buildStepCause.add(new Cause() {
        public String getShortDescription() {
             return "Build Step started by Hello Builder";
        }
    });

    listener.started(buildStepCause);

 ArgumentListBuilder args = new ArgumentListBuilder();
    if (launcher.isUnix()) {
        args.add("/bin/ls");
        args.add("-la");
    } else {
         args.add("dir");  //Windows
    }

    String homeDir = System.getProperty("user.home");
    args.add(homeDir);

    try {
        int r;
        r = launcher.launch().cmds(args).stdout(listener).join();
        if (r != 0) {
            listener.finished(Result.FAILURE);
            return false;
        }
    } catch (IOException ioe) {
        ioe.printStackTrace(listener.fatalError("Execution" + args + "failed"));
        listener.finished(Result.FAILURE);
        return false;
    } catch (InterruptedException ie) {
        ie.printStackTrace(listener.fatalError("Execution" + args + "failed"));
        listener.finished(Result.FAILURE);
        return false;
    }

    listener.finished(Result.SUCCESS);

    return true;
}
```

You also need to include the import statements

```
import hudson.util.ArgumentListBuilder;
import java.util.ArrayList;
import java.util.List;
import hudson.model.Cause;
import hudson.model.Result;
```

**2.3.2** Run the plugin project to see the HelloBuilder in action again.

**Note:** Before starting the re-run of the plugin project, you must always shut down the jetty server started by the previous run. Otherwise, the port 8080 required by the server will not be available for the next run.

**2.3.3** To stop the server started in the previous run click on the close button at the bottom of Netbeans IDE as show in Figure 10

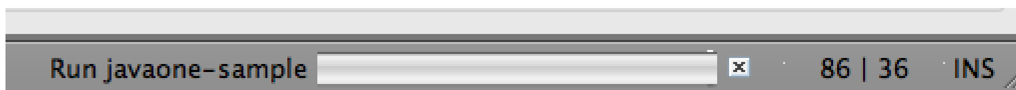


Figure 10: Stop the server in NetBeans

**2.3.4** Once the server is fully shutdown, re-run the project (see 1.4). Once Hudson has restarted, go to the TestProject dashboard and click on the `Build Now` link and open the console log. See the result of the build as

```
Started by user anonymous
Hello, winston!
Build Step started by Hello Builder
$ /bin/ls -la /Users/winstonp
total 320
drwxr-xr-x   16 winstonp  staff    544 Nov 10  2010 Adobe MAX
drwx------+  31 winstonp  staff   1054 Aug 31 14:54 Desktop
drwx------+  38 winstonp  staff   1292 Aug 25 13:02 Documents
drwx------+ 111 winstonp  staff   3774 Aug 31 14:00 Downloads
drwxr-xr-x  118 winstonp  staff   4012 Aug 25 16:46 Images
drwxr-xr-x   17 winstonp  staff    578 Nov  4  2010 JDK-src
drwx------+  50 winstonp  staff   1700 Aug 17 17:44 Library
 ..
drwxr-xr-x    4 winstonp  staff    136 Jul 29 11:14 tools
drwxr-xr-x   10 winstonp  staff    340 Jan 26  2011 webdesigner
Finished: SUCCESS
```

If there is an error, the exception corresponding to the error is displayed as

```
Started by user anonymous
Hello, winston!
Build Step started by Hello Builder
$ "/bin/ls -la" /Users/winstonp
FATAL: Execution[/bin/ls -la, /Users/winstonp]failed
java.io.IOException: Cannot run program "/bin/ls -la": error=2, No such file or directory
       at java.lang.ProcessBuilder.start(ProcessBuilder.java:460)
       at hudson.Proc$LocalProc.<init>(Proc.java:192)
       at hudson.Proc$LocalProc.<init>(Proc.java:164)
       at hudson.Launcher$LocalLauncher.launch(Launcher.java:639)
       at hudson.Launcher$ProcStarter.start(Launcher.java:274)
       at hudson.Launcher$ProcStarter.join(Launcher.java:281)
       at org.sample.hudson.HelloWorldBuilder.perform(HelloWorldBuilder.java:99)
       at hudson.tasks.BuildStepMonitor$1.perform(BuildStepMonitor.java:19)
       at hudson.model.AbstractBuild$AbstractRunner.perform(AbstractBuild.java:631)
        ...
Finished: FAILURE
```

### Step 4: Analyzing HelloBuilder perform(..) method

The code you have added to `perform(..)` is still contrived, but explains some of the important concepts

**2.4.1** When a build step is `started` or `stopped` let Hudson know about it. This is done via the `Job Build Listener` interface.

```
listener.started(buildStepCause);
       ..
       ..
listener.finished(Result.SUCCESS);
```

This is important for two reasons.

- Hudson heuristically shows the progress of the overall build of the job
- If the build step fails and by letting Hudson know the failed status of the builds, Hudson would stop the overall progress of the build and marks the build as FAILED.

**2.4.2** Use the Launcher interface to launch your external executable. Send the console outputs of your execution to Hudson

```
int r;
r = launcher.launch().cmds(args).stdout(listener).join();
if (r != 0) {
      listener.finished(Result.FAILURE);
      return false;
}
```

`Launcher` correctly launches the application in the Master or Slave node the job is running. Always use the return status of the `Launcher` to find out if the execution was successful. The standard output of the Launcher is hooked to the listener. This sends console output of the execution to Hudson. This is how the output of the command to list the `User Directory` is displayed in the build console.

**2.4.3** Notify Hudson of any failure of the Build Step

```
} catch (IOException ioe) {
      ioe.printStackTrace(listener.fatalError("Execution" + args + "failed"));
      listener.finished(Result.FAILURE);
      return false;
}
```

The StackTrace of the exception is sent to Hudson via `Exception.printStackTrace(lister.fatalError(..))`. It may be possible to further annotate your console log using the ConsoleAnnotator interface (not covered in this tutorial)

## Exercise 3: Configure the Extension

There are two ways to configure your Extension. One is local to the area of the functionality the plugin extends and the other via the Hudson wide Global Configuration.  In this exercise you will learn how to configure your Extension in the project configuration page. In this exercise you will learn:

- How to add a UI to get input from user
- How to give feedback to the user on their input
- How to configure the Extension with the user input

### Step 1: Understanding the configuration file convention

Hudson uses a UI technology called `Jelly`. The `Jelly` UI technology is a Server Side rendering technology which uses a Rendering engine to convert XML based Jelly definitions (tags) to client-side code: HTML, Javascript and Ajax. Hudson provides a number of Jelly tags for your convenience.

The model objects are bound to these tag `attributes` via an Expression Language called `Jexl`. When the tags are rendered into HTML and Javascript, the rendered code includes information from the model objects to which their attributes are bound to. This makes it very powerful to express your view with simple jelly tags, rather than writing lots of HTML , Javascript and Ajax.

The jelly files you use to render the UI has the extension `.jelly`. They reside in the `resources` directory of the plugin. Hudson uses a heuristic convention to find these jelly files. The folder under which these jelly files must reside should have a path hierarchy similar to the package name of the model class, plus the name of model class itself.

Hudson use the same namespace of the Class package as the folder hierarchy plus model name. In your example the `HelloWordBuilder` model class has the package name `org.sample.hudson.` So the configuration file must reside under the folder

```
org/sample/hudson/HelloWordBuilder
```

Hudson uses another convention to tell if the configuration file is meant for local configuration or global configuration. If the configuration is named as `config.jelly` it is used as a local configuration file and its content is included in the configuration of the functionality that this Extension extends. Since `HelloWordBuilder` extends the `Builder` build step of a Hudson Job, any `Jelly` content put in the configuration file

```
org/sample/hudson/HelloWordBuilder/config.jelly
```

is included in the Job configuration page to configure the `HelloWordBuilder` extension in the Builder section of the Job Configuration.

As you saw in Exercise 1.5, `HelloBuilder` Extension provides a UI for the user to configure. The UI provided by the `HelloBuilder` Extension is a simple TextBox for the user to input their name.

**3.1.1** Open the `config.jelly` in the Netbeans editor by exploring the resource folder in the MyNewPlugin project and double clicking on the `config.jelly` file as shown in Figure 11

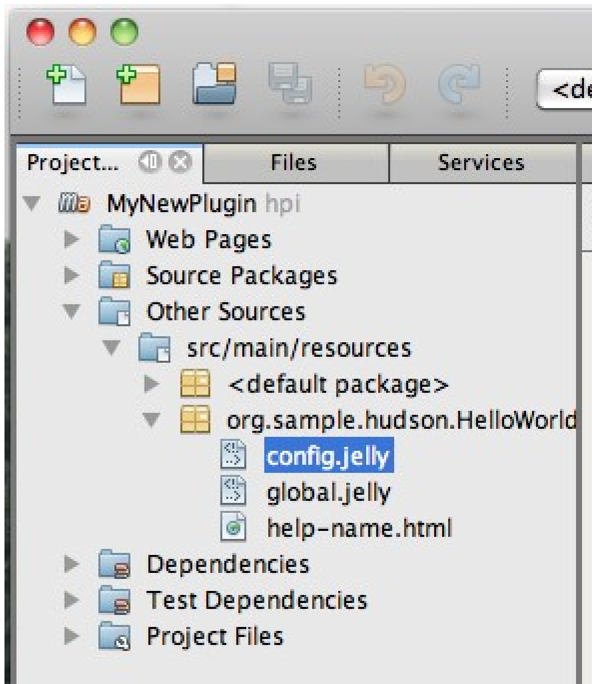


Figure 11: config.jelly in Netbeans editor

The content of the file is very simple. It is a pure XML file with jelly syntax

```
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:d="jelly:define"
xmlns:l="/lib/layout" xmlns:t="/lib/hudson" xmlns:f="/lib/form">
   <!--
      Creates a text field that shows the value of the "name" property.
      When submitted, it will be passed to the corresponding constructor parameter.
   -->
  <f:entry title="Name" field="name">
      <f:textbox />
  </f:entry>
</j:jelly>
```

There are two main tags playing the role of user interaction

- entry - tells hudson the enclosing tags are considered as user interaction elements and submitted via HTML form
- textbox - renders simple HTML text field whose value will be send back to the server

## Step 2: Understanding the UI rendering

In this step you will take a closer look at UI rendering from the jelly file.

**3.2.1** Open the `TestProject` Job configuration Page by clicking on the "`Configure`" link. This should open the Job configuration page.

**3.2.2** Scroll down to the `Build` section and and view the `Say Hello World` Builder and its configuration. Click on the "`Question`" icon on the right hand side of the TextField. It displays Help Text as shown in Figure 12.

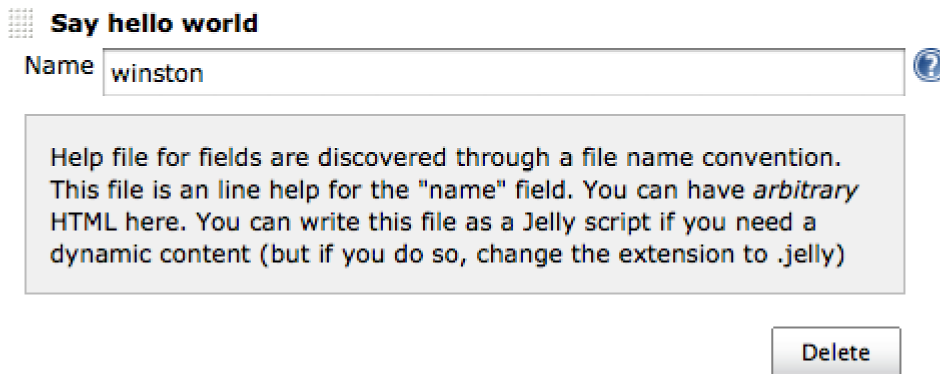


Figure 12: Help Text for Say hello world

Where does this help text come from? If you look at the content of `config.jelly`, you'd notice there is no such Help text. However, Hudson still displays some Help. Once again convention comes in to play.

**3.2.3** Go to the the Netbeans project explorer and click on the file named "help-name.html". The editor displays the text you see in the Help Text above.  How does Hudson know to get the content from this file and display it as Help content for the field? The trick is in the name of the file. By convention Hudson look for a file name in the folder path as the config file. The name of the file should be

```
help-{fieldName}.html
```

In the `config.xml` we have

```
<f:entry title="Name" field="name">
      <f:textbox />
</f:entry>
```

`field="name"`, indicates the `TextBox` should be used as an entry field with the name "`name`". So based on convention, the help text for that field should exist in a file with name "`help-name.html`".

The content of the file `help-name.html` is pure HTML . You can include image, text and hyperlinks in the content to emphasise and enhance your Help Text. As mentioned in the help text, if you want to use information from Hudson model objects, then you should have jelly content in the field Help file and the extension of the file name should be `.jelly` instead of `.html`.

**3.2.4** See that in action. Delete `help-name.html` file. To delete the file in Netbeans project explorer, select the file, right click to bring up the popup menu and select the menu item delete.

**3.2.5** Next, create the file `help-name.jelly`. To create the file, right click on the explorer node and select `New -> Empty File`. In the resulting dialog give the file name `help-name.jelly`. Add the following content to the file

```
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:d="jelly:define" >
  <div>
      Welcome to ${app.displayName}.
      Enter your name in the Field.
  </div>
</j:jelly>
```

**3.2.6** Stop and run the project again. Go to the Build section of the Page configuration page and click on the Help button on the right side of the TextBox. You should see the help text as in Figure 13

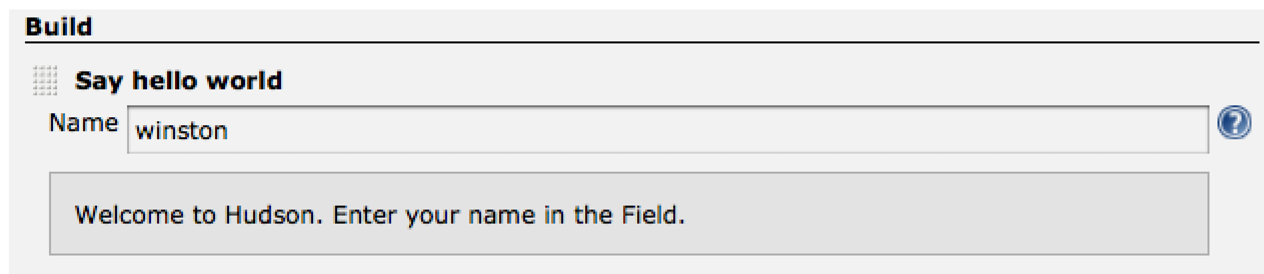


Figure 13: Help field output

Note `${app.displayName}` is replaced as "`Hudson`" which is the name of the application.

## Step 3: Understanding the interaction between UI and model

In this step you will explore how the UI interacts with Hudson model objects. `HelloBuilder` is a Hudson model object. It encapsulate data. UI can interact with this model to get and display its data or get information from user via fields in the UI and update the model data. Now examine how this happens.

**3.3.1** In 3.2 you created help file `help-name.jelly` and included a `Jexl` expression `${app.displayName}` in the content. When the Server side of the Hudson application received the request for Job configuration page, it included the `HelloBuilder` snippet in to the Job configuration page. Since the Help itself is a `jelly` file, it was given to the Jelly

renderer to render in to client side code. The Jelly renderer is responsible for substituting the corresponding value for the `Jexl` expression after evaluating it. The first part of the expression evaluates to the model object, then to the method name of the model object.

**3.3.2** By default Hudson registers three identifiers for the model objects to the Jexl expression evaluator, they are

`app` - The Hudson Application itself
Ex. `${app.displayName}` evaluates to `Hudson.getDisplayName()`
`it` - The model object to which the Jelly UI belongs to
Ex. `${it.name}` evaluates to `HelloWorldBuilder.getName()`
`h` - A global utility function (called `Functions`) which provides static utility methods
Ex. `${h.clientLocale}` evaluates to `Functions.getClientLocale()`

Since the expression `${app.displayName}` evaluates to “Hudson”, the name of the Hudson application, that is what you see in the Field Help text.

While the UI displays the data of a model, the input of the user in the UI must update the model data when the configuration page is submited. In this case, the value of the name the user enters in the UI must be updated in the model.

**3.3.3** When the UI is submitted, Hudson re-creates the model by passing the corresponding value via the constructor. Hence the constructor of the model Object must have a parameter whose name matches the name of the field. In the configuration you have

```
<f:entry title="Name" field="name">
```

**3.3.4** So the constructor of your `HelloBuilder` must have a parameter with name `“name”`. If you look at the constructor of the class HelloWorldBuilder, it does indeed have a parameter `“name”`

```
@DataBoundConstructor
      public HelloWorldBuilder(String name) {
        this.name = name;
      }
```

The annotation `@DataBoundConstructor` hints to Hudson that this Extension is bound to a field and on UI submission, it must be reconstructed using the value of the fields submitted.

**3.3.5** Also it must have a getter with the name of the field for the `config.xml` to get the data for the second time around when the project is configured again.

```
      public String getName() {
            return name;
      }
```

**3.3.6** This information is persisted along with the project configuration. To view the project configuration go to the Files Explorer Window in the Netbeans IDE and drill down to the project config file as in Figure 14

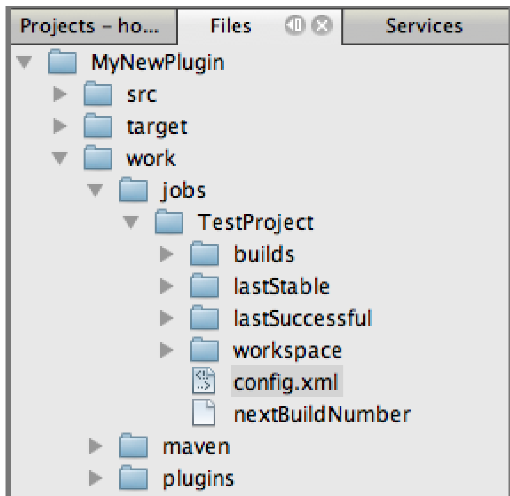


Figure 14: config.xml in Files Explorer

Note that the value of the `Name` field is saved as a HelloBuilder configuration

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <actions/>
  <description></description>
  <keepDependencies>false</keepDependencies>
  <creationTime>1314407794225</creationTime>
  <properties>
      <watched-dependencies-property/>
  </properties>
  <scm class="hudson.scm.NullSCM"/>
  <advancedAffinityChooser>false</advancedAffinityChooser>
  <canRoam>true</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false</blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <triggers class="vector"/>
  <concurrentBuild>false</concurrentBuild>
  <cleanWorkspaceRequired>false</cleanWorkspaceRequired>
  <builders>
      <org.sample.hudson.HelloWorldBuilder>
      <name>Winston</name>
      </org.sample.hudson.HelloWorldBuilder>
  </builders>
  <publishers/>
  <buildWrappers/>
</project>
```

## Step 4: Examining the UI validation methodology

**3.4.1** In the Job configuration page go to `Build Section -> HelloBuilder UI` and remove the name in the text field. Then click else where on the page. You will see the error message as in Figure 15

Figure 15: Error message

**Note:** Do not press enter or return key. This will submit the configuration page.

**3.4.2** Now enter a two letter word (say "`xy`") for name and enter elsewhere. Now you see the information message as in Figure 16

Figure 16: Information message

**3.4.3** Where does this error message or info come from? If you examine your config.xml or any of the corresponding Field Help files, no such message exists. The magic is in the `Jelly` file rendering. Some Ajax code is rendered, which behind the scenes contacts the Hudson server and asking what message it should display.

**3.4.4** You can easily observe these Ajax requests using Firefox and Firebug. When `config.jelly` was rendered by Hudson, the `jelly` tag `<f:textbox />` the Ajax code required to do the checking is also rendered. Firebug displays the Ajax request info as in Figure 17

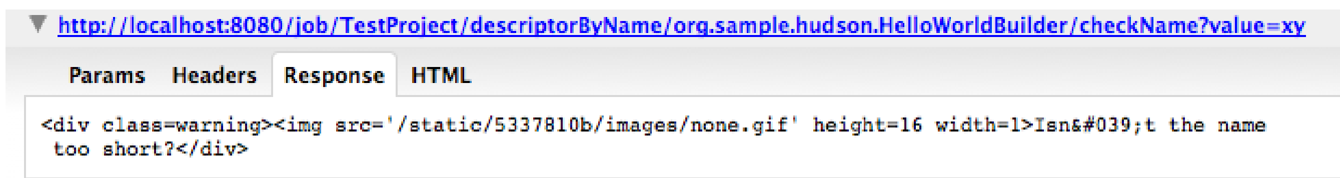


Figure 17: Firebug output

An Ajax request is sent to Hudson Server as

```
GET /job/TestProject/descriptorByName/org.sample.hudson.HelloWorldBuilder/checkName?value=xy HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:6.0.1) Gecko/20100101 Firefox/6.0.1
Accept: text/javascript, text/html, application/xml, text/xml, */*
```

Hudson evaluates this request, finds the extension `HelloWorldBuilder` then executes the method `checkName()` and returns the result. Again Hudson uses the convention `doCheck + {nameOfTheField}` as part of the Ajax URL. In order for the check to occur

**Note:** By default for every `f:textbox` tag in the `Jelly` config file, Hudson will render the Ajax check. However, if your Extension Class does not include the corresponding method (in this case `docheckName()`), then Hudson will silently ignore the check request.

**3.4.5** The `doCheckName()` method is straight forward. The Ajax request specifies the `checkName` method with the parameter `"value"` as `{..}/checkName?value="xy"`. Hence the parameter `value` of `doCheckName` must be annotated with the annotation `@QueryParameter`. Also the method must return a `FormValidation` Object which determines the outcome of the check.

```
public FormValidation doCheckName(@QueryParameter String value)
                                              throws IOException, ServletException {
         if (value.length() == 0) {
             return FormValidation.error("Please set a name");
         }
         if (value.length() < 4) {
             return FormValidation.warning("Isn't the name too short?");
         }
         return FormValidation.ok();
}
```

**3.4.6** As you see in this method, `FormValidation.error(..) is` returned if an error occurs. The HTML which is sent back to the client displays the text in red and with an error icon. If `FormValidation.warning()` is returned then the HTML sent back displays the message in brownish Yellow with a warning icon.

## Step 4: Adding your Extension's Global configuration

You learned how to configure the Extension per Job basis. You may be able to configure the Extension (in this case HelloBuilder) of each Job to do different things. However, some of the configuration Extension could be global. For example, if you use `Git` as your `SCM` in a project, you might want to configure two different projects to force Git to check

out from a different repository. So in both the projects, `Git SCM Extension` must be configured to use two different repository URLs.

If the `Git SCM Extension` needs to know the Git native binaries, then placing the UI to configure the Git binary location in the project makes little sense, because it doesn't vary project to project. It makes sense to put such configuration in Hudson's Global Configuration Page.

For Hudson to include the Global configuration of an Extension in its Gloabl Configuration, it must be placed in a file called `global.jelly.` The namespace convention for this file is similar to local config. For the `HelloBuilder` the global config file is

```
org/sample/hudson/HelloWordBuilder/global.jelly
```

In the HelloWorldBuilder.perform(..) method you saw

```
if (getDescriptor().useFrench()) {
             listener.getLogger().println("Bonjour, " + name + "!");
       } else {
             listener.getLogger().println("Hello, " + name + "!");
       }
```

**3.4.1** HelloBuilder can be configured to say hello either in French or English. The configuration of the language to use is done globally. Once set, all the builds of various jobs which use HelloBuilder would either say hello in French or English based on this global configuration.

To configure this global configuration open the Hudson's Global Configuration Page. To open the Global Configuration Page, click on the link `Manage Hudson` in the main Dashboard and then click on the `Configure System` link. In the configuration page scroll down to the Hello World Builder as shown in Figure 17

Figure 17: Language configuration example

**3.4.2** Here you can set the global configuration to use French as the hello language. In the `global.jelly` the checkbox is defined as

```
<f:section title="Hello World Builder">
    <f:entry title="French" description="Check if we should say hello in French"
       help="/plugin/javaone-sample/help-globalConfig.html">
      <f:checkbox name="hello_world.useFrench" checked="${descriptor.useFrench()}" />
    </f:entry>
</f:section>
```

**3.4.3** The decision whether this checkbox should be checked or not comes from the Extension itself. The Jexl expression `${descriptor.useFrench()}` would resolve to HelloBuilder.DescriptorImpl.useFrench() which is defined as

```
public boolean useFrench() {
            return useFrench;
      }
```

`useFrench` is a field in `HelloBuilder`. This field should be set to true if the user checks the CheckBox. Once the global configuration is submitted, by convention Hudson would call the HelloBuilder.Descriptor.config passing a JSONObject. It is up to the Extension to find its submitted value and then use it. `HelloBuilder` defines this method as

```
@Override
        public boolean configure(StaplerRequest req, JSONObject formData) throws FormException {
                useFrench = formData.getBoolean("useFrench");
                save();
                return super.configure(req, formData);
        }
```

In this method the boolean value of the field `useFrench` is obtained as set to `HelloBuilder.useFrench`. The next time `HelloBuilder.perform()` is called during the Build of the Job, `HelloBuilder.useFrench` is consulted and based on its value either the hello message is either in French or English

# Further reading:

1. Hosting your plugin - http://wiki.hudson-ci.org/display/HUDSON/Hosting+Hudson+Plugins
2. Releasing your plugin - http://wiki.hudson-ci.org/display/HUDSON/Releasing+Hudson+Plugin
3. Extension Point  Javadoc - http://wiki.hudson-ci.org/display/HUDSON/Extension+points
4. List of available Plugins - http://wiki.hudson-ci.org/display/HUDSON/All+Plugins+by+Topic