Prepared (Subject resp)		No.		
Morhaf Alaraj, Viktor Bostrand, Andres Vazques				
Approved (Document resp)	Checked	Date	Rev	Reference
		2015-06-10	PA1	

JAT

Prepared (Subject resp)		No.		
Morhaf Alaraj, Viktor Bostrand, Andres Vazques				
Approved (Document resp)	Checked	Date	Rev	Reference
		2015-06-10	PA1	

Contents

Contents

1 System Design 3
1.1 Post-build action plugin 4
1.1.1 Post-build action plugin implementation 4
1.1.2 Known Shortcomings 6
1.1.3 Installing JAT post-build action plugin in Jenkins 6
1.2 Database 6
1.3 Web application 7
1.3.1 Back end 8
1.3.2 Grails Specific 10
1.3.3 Front end 11
1.4 Using JAT post-build action plugin 11

2 Installation 12
2.1 Building JAT post-build action plugin 12
2.2 Building JAT web application 12

3 Tools 13

Introduction

Continuous integration (CI) is an important part of Ericsson's software infrastructure. CI gives developers the ability to set up and monitor complex build chains that are essential for the production line. However, Jenkins, the tool at hand, lacks certain desirable features and properties that could enhance the CI system and give developers a more intuitive way of visualizing job executions. The Jenkins test statistic collector and analysis tool (JAT) is a tool that provides Ericsson developers with a flexible and straightforward way of monitoring the results of Jenkins job executions.

The JAT combines a Jenkins plugin solution together with a web application to visualize Jenkins build and test executions in intuitive graphs for a quick overview but also the ability to dive deep into test execution specifics for more detail.

1 System Design

The architecture of the JAT can roughly be split up into two different components; the post-build action plugin and the web-application. The first mentioned actor operates within the Jenkins application and is configured as a post-build action. This action fetches the data associated with a particular build once the build is completed. The web-application components serve multiple purposes, most important of which is to receive data from the post-build action plugin, store this data, and present the data upon user request.

The system design of the JAT tool is presented in Figure 1. The design can be divided up into three main parts:

- The Jenkins post build action plugin
- The web-application
- The relational database

The arrows in the system design represent the information flow between the different components. The dashed arrows represent the data flow from a user-request, and the un-dashed lines represent the data flow when the post build action plugin transfers data to the database. All communication is handled through HTTP requests.

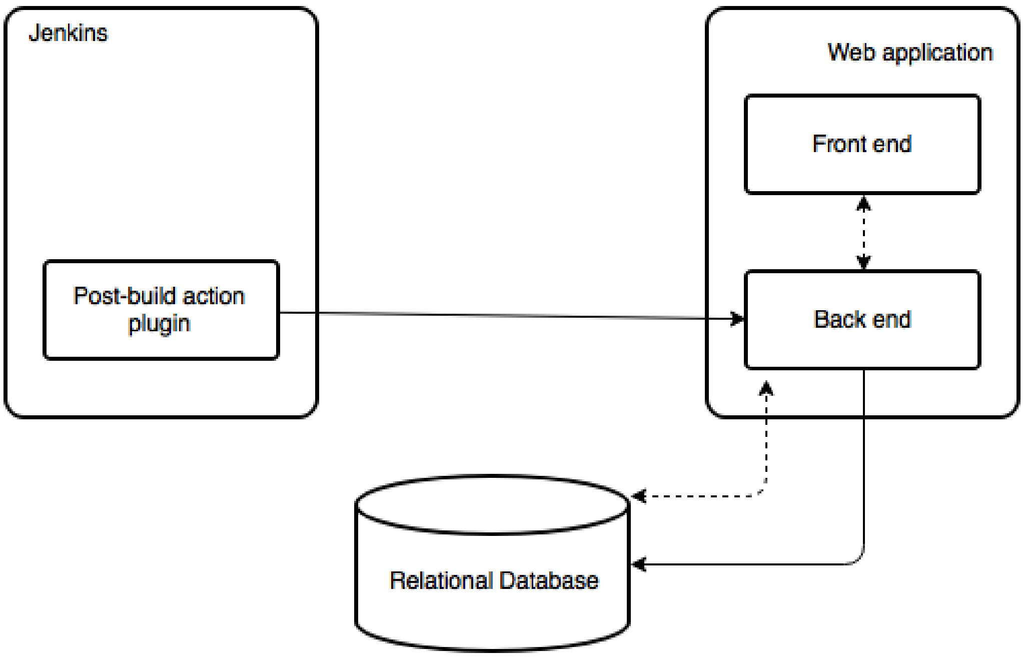


Figure 1 JAT System Design

1.1 Post-build action plugin

The post-build action plugin of the JAT operates as any other Jenkins post-build action plugin and is invoked as soon as a build is completed. When invoked, the plugin fetches the URL that uniquely identifies the build and requests its JSON data by performing an HTTP request to this URL. This JSONObject, in turn, contains all the particularities associated with the build such as test results, execution time, depending projects, timestamp etc. In order to fetch particular test execution result another HTTP request is invoked. Similar to the previously described request, this operation utilizes the JSON API of the test report, fetches it and appends this object into the previously fetched object. When the data has been collected, the plugin simply dispatches it to the web application.

To utilize the JAT for fetching test executions result, users must first configure the job with the **Junit test result report** post-build action plugin. This is due to the fact that the JAT utilizes the API from this report to retrieve test execution particularities.

1.1.1 Post-build action plugin implementation

The post-build action plugin is mainly implemented in Java with some additional properties written Jelly. The Jelly built parts of the system holds the web GUI, whilst the Java part handles all business logic and communication between other components. Figure 2 illustrates a class diagram of the parts written in Java. The system consists, as the figure suggests, of four parts; the JSONDataFetcher, the JSONDataDispatcher, the JatDescriptor and the JAT.

The JAT class together with the JatDescriptor class is the main entry point of the application. In the actual implementation both of these classes are written in the same file, where the JatDescriptor is wrapped within the JAT class, which seems to be the convention when writing Jenkins plugins, but for ease of understanding they have been separated in Figure 2. The JatDescriptor is marked as an extension point which can be thought of as a hook into the Jenkins system, meaning that when the plugin is installed in Jenkins, the application can establish a connection to the plugin through this entry point. Here, the plugin also receives user specified data from the Jelly configured GUI.

The JAT class is the main class of the plugin. This class extends the Hudson Notifier object which is used for configuring post-build action plugins, so that the application is invoked properly. The object is instantiated as soon as a build configured with the JAT finishes, and thus triggers the plugin. The object then calls its descriptor, fetches the instance data of the descriptor which is used to initialize the instance fields of the object. After this, the perform method is called automatically which is the type specific method for the Notifier object. Here the object accesses the absolute URL pointing to the Jenkins build that have been executed. With this URL, the JSONDataFetcher can be created. The task for this object is, as the name suggests, fetching the data pointed to by the URL. This procedure will create a JSONObject representation of the relevant collected data and with this object a JSONDataDispatcher can be created. This object will be called from the JAT and its data will be forwarded to the JAT web application back end.

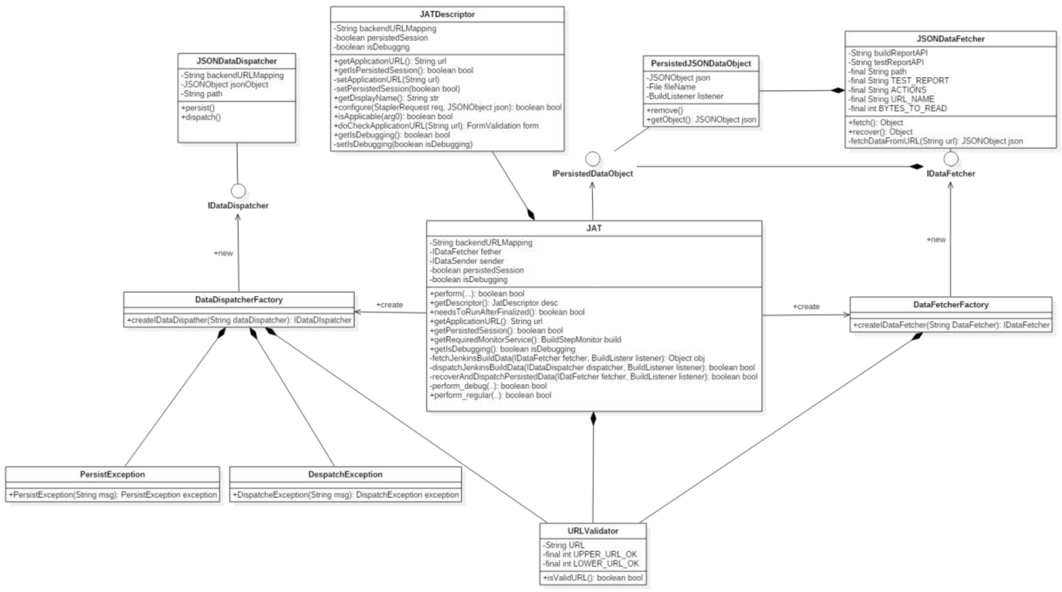

JATDescriptor
-String backendURLMapping
-boolean persistedSession
-boolean isDebuggng
+getApplicationURL(): String url
+getIsPersistedSession(): boolean bool
-setApplicationURL(String url)
-setPersistedSession(boolean bool)
+getDisplayName(): String str
+configure(StaplerRequest req, JSONObject json): boolean bool
+isApplicable(arg0): boolean bool
+doCheckApplicationURL(String url): FormValidation form
+getIsDebugging(): boolean bool
-setIsDebugging(boolean isDebugging)
JSONDataDispatcher
-String backendURLMapping
-JSONObject jsonObject
-String path
+persist()
+dispatch()
PersistedJSONDataObject
-JSONObject json
-File fileName
-BuildListener listener
+remove()
+getObject(): JSONObject json
JSONDataFetcher
-String buildReportAPI
-String testReportAPI
-final String path
-final String TEST_REPORT
-final String ACTIONS
-final String URL_NAME
-final int BYTES_TO_READ
+fetch(): Object
+recover(): Object
-fetchDataFromURL(String url): JSONObject json
IPersistedDataObject
IDataFetcher
IDataDispatcher
+new
JAT
-String backendURLMapping
-IDataFetcher fether
-IDataSender sender
-boolean persistedSession
-boolean isDebugging
+perform(...): boolean bool
+getDescriptor(): JatDescriptor desc
+needsToRunAfterFinalized(): boolean bool
+getApplicationURL(): String url
+getPersistedSession(): boolean bool
+getRequiredMonitorService(): BuildStepMonitor build
+getIsDebugging(): boolean isDebugging
-fetchJenkinsBuildData(IDataFetcher fetcher, BuildListenr listener): Object obj
-dispatchJenkinsBuildData(IDataDispatcher dispatcher, BuildListener listener): boolean bool
-recoverAndDispatchPersistedData(IDatFetcher fetcher, BuildListener listener): boolean bool
-perform_debug(..): boolean bool
+perform_regular(..): boolean bool
+new
+create
DataDispatcherFactory
+createIDataDispather(String dataDispatcher): IDataDIspatcher
+create
DataFetcherFactory
+createIDataFetcher(String DataFetcher): IDataFetcher
PersistException
+PersistException(String msg): PersistException exception
DespatchException
+DispatcheException(String msg): DispatchException exception
URLValidator
-String URL
-final int UPPER_URL_OK
-final int LOWER_URL_OK
+isValidURL(): boolean bool

Figure 2 JAT Class Diagram

Prepared (Subject resp)		No.		
Morhaf Alaraj, Viktor Bostrand, Andres Vazques				
Approved (Document resp)	Checked	Date	Rev	Reference
		2015-06-10	PA1	

1.1.2 Known Shortcomings

Since the JAT post-build action plugin fetches its instance data from its descriptor due to the global configuration of the JAT web application back end URL, one must be careful when updating this URL. Every job that is configured with the JAT must be re-saved when altering the back end URL mapping since this URL lives within each job locally. If this is not done, previously defined job will continue to send data to the previously configured back end URL. The same applies when reconfiguring the plugin to run in debug mode and/or persisted mode.

1.1.3 Installing JAT post-build action plugin in Jenkins

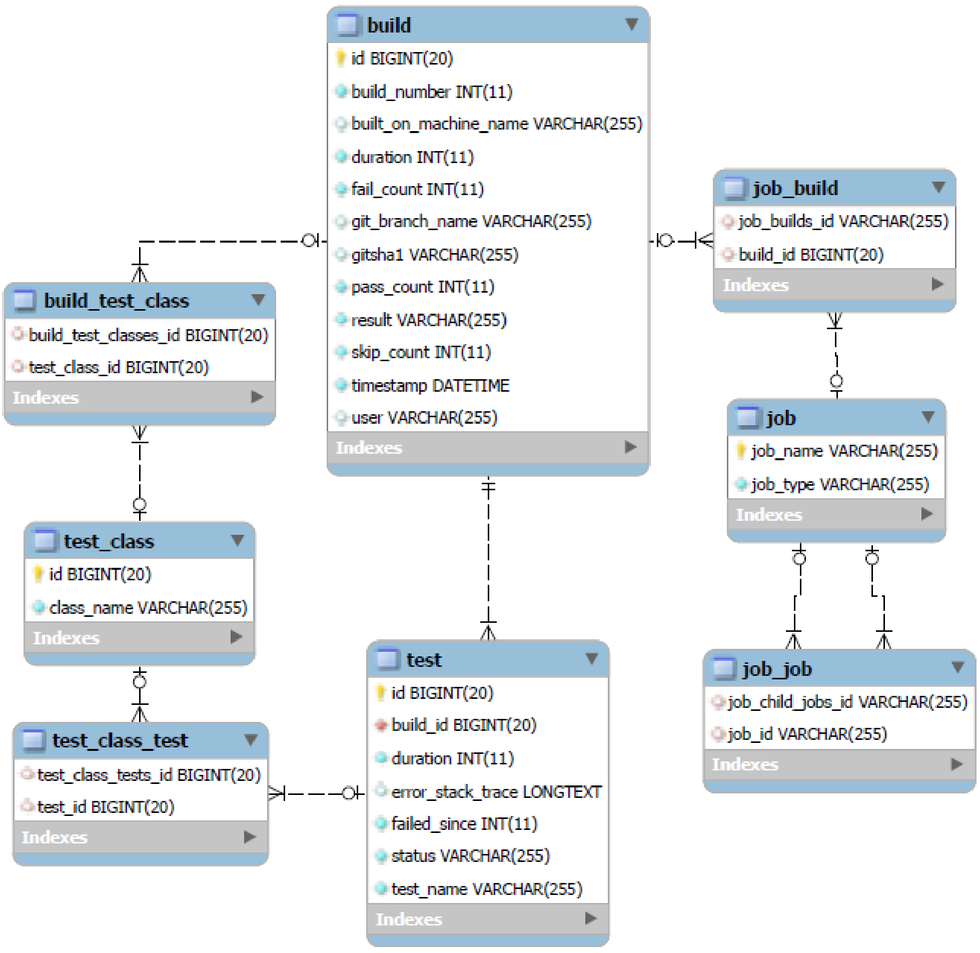
To install the JAT post-build action plugin, the following is required:

- The JAT.hpi-file (See section above)
- A running Jenkins instance

The plugin is installed in Jenkins by navigating to *Jenkins* > *Manage Jenkins* > *Manage Plugins* and then selecting the *Advanced* tab and finally uploading the jat.hpi-file. After this, Jenkins will install the plugin and it’s ready for action immediately. Enjoy!

1.2 Database

The JAT utilizes a relational MySQL database.



1.3 Web application

As illustrated in the system design (Figure 1), the web application can be divided up into two logical parts, the back end and the front end. The front end is the presentation layer while the backend handles server side business logic. The backend delivers data to the front end upon request. The web application is built using a REST-based architecture. The Grails Web Framework is used for developing the back end.

“Grails is a powerful web framework, for the Java platform aimed at multiplying developers’ productivity thanks to a Convention-over-Configuration, sensible defaults and opinionated APIs. It integrates smoothly with the JVM, allowing you to be immediately productive whilst providing powerful features”

The front end is developed with HTML, CSS and JavaScript. Due to the size of the front end application and need of a MVC structure, the JavaScript framework AngularJS was used.

“HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.”

Bootstrap is used for styling the front end. It contains predefined classes for almost all HTML elements, including input fields and buttons. JQuery Flot is used for plotting. It is a responsive and extendable JavaScript library that suits the project requirements.

1.3.1 Back end

The back end is written in Grails 2.4.4 and utilizes a MySQL database. The back end has two primary functions. The first one is to receive build data from the Jenkins plugin, process it and finally insert it into the database. These tables have been set up in grails by creating domain classes where the class name is the name of the table. To create desired columns in the database tables, attributes are simply added in the domain classes as variable. The relationship between two tables is also defined within the domain classes by using e.g. a hasMany or belongsTo relationship.

Example domain: Job

```
class Job {
        String jobName
        String jobType
        static hasMany = [childJobs : Job, builds : Build]
        static mapping = {
                version false
                id generator: 'assigned', name: 'jobName'
                childJobs cascade: "all-delete-orphan"
                builds cascade: "all-delete-orphan"
        }
        static constraints = {
                jobName unique: true, nullable: false
                jobType nullable: false
        }

}
```

This class will result in a database table named “Job”, with two attributes. Two relationship tables will also be created (hasMany relationship). The relation is between the Job class and itself, but also with the Build class. The build class is also declared as a domain. The cascade functionality inherits from the SQL language. It specifies that the child will also be deleted once a parent is deleted.

The other function is to provide information to the presentation layer/front end upon a HTTP request. This is done by setting up endpoints in the URL mapping configuration in Grails, so Grails know which controller and method to call when a certain request is performed.

URLMapping.groovy example

"/jenkins"(controller:"jenkins", action:"run")

This simply defines that incoming requests to the URL “/jenkins” should be forwarded to the controller “jenkins” and the action/function “run”. For REST API’s, the grails specific “resources” can be used. This automatically maps HTTP methods towards the correct function.

"/job"(resources: 'job')

HTTP POST -> save() in controller
HTTP GET -> index() in controller
HTTP PUT -> update() in controller
HTTP DELETE -> delete() in controller

As mentioned earlier, the back end has two primary functions. One part handles the data processing from the post build plugin and insertion to the database, while the other part handles the data delivery to the front end upon request.

The main entry point for the web application is the JenkinsController. It fetches the JSON data that is delivered from the post build plugin and delegates it to different Grails services (business logic handlers) for handling the data.

The flow in JenkinsController (simplified)

1. Job jobObj = processJobDataService.processJobData(buildJsonData) call the processJobDataService and pass the buildJsonData to the function in that service that will process the data. This will return the job instance that is created.

2. Build buildObj = constructBuildService.processBuildData(buildJsonData, jobObj) We will pass the buildJsonData together with the job. The result of this call is the build instance, that is used to verify e.g. if the build already exist within the job.

3. job.addToBuilds(buildObj) This will bind the build to the job.
4. testReportService.processTestReportData(testReportJsonData, buildObj) This will create the testreport and bind the build to it.

1.3.2 Grails Specific

Domain

The domain classes compose the core of the application and each of them are usually mapped to a separate table in the database which acquire the same name as the respective domain class. The columns in a database table are mapped from each of the attributes in the related domain. So if a domain class contains the attributes String name and int age, then two columns will be created in the table. A relationship between two tables in the database is easily created by specifying in the domain class if it has many (hasMany), has one (hasOne) or belongs to (belongsTo) an instance of another domain class. If it's desired, constraints for the domain properties can also be defined.

Controller

A controller is responsible for the handling of web requests and creating or preparing responses for the requests. The responses are either generated directly or delegated to views. The default URL mapping configurations in grails ensures that the first part of the controllers name is mapped to a URI. A controller can contain one or several actions which are methods, defined within the controller, and each one mapped to a URI within the controller name URI. All the actions/methods, are specified in 'allowedMethods' , usually at the top of the controller, and with what kind of HTTP request each one is associated with.

```
static allowedMethods = [getTests: 'GET', compare: 'GET']
```

In some cases, it's easier to render snippets of text or code to the response directly from the controller by using the render method. In this project 'render' was used to render the responses in JSON format, since the response format type is specified in the beginning of the controller:

```
static responseFormats = ['json']
```

If you're planning on moving all or some of the logic to some service/-s, then these will have to be defined within the controller so you can call the methods containing that logic within that service.

```
def GetTestsService
```

Service

The services in grails are used for moving the core application logic away from the controllers, because this promotes reuse and a clean separation of the code. This is done so the controllers will only direct to the correct action to use and render responses from the services.

1.3.3 Front end

The front end is a web page, built on HTML and JavaScript, which the user can interact with to analyze data of interest. The start/homepage gives the user a general view of the status in each of the jobs which are clickable to go further into detail and look at what tests are succeeding or failing and where the problems appear. To illustrate how the builds have been doing over time, graphs are implemented to provide an easy-to-read view. These graphs are also clickable and will fetch and present the data for the specific build you chose below the graph.

When the user have chosen some data, the front end will send a HTTP request to the back end which in turn fetch the data from the database and send it to the front end in JSON format.

1.4 Using JAT post-build action plugin

Once installed in Jenkins, the JAT post-build action plugin is ready for action. For the tool to operate desirably, its settings must first be configured. The plugin is built such that all settings for the plugin are configured globally for the running Jenkins instance. This implies that all jobs that will be configured with the JAT will carry the same JAT configuration. The configurations are managed in Jenkins > Manage Jenkins > Configure System, illustrated by Figure 3.

In order for the plugin to work properly it needs to be provided with an absolute URL pointing to the JAT web application back end. This URL is, as one might have guessed, is used to pass collected build data from the Jenkins plugin to the web application, where it can be processed and stored.

The plugin also offers an option to persist build data locally if the HTTP forward from the JAT post-build action plugin to the JAT web application back end should fail due to some exception. If such an event should occur, and the plugin is configured with this option selected, the JAT will persist the data locally in the Jenkins in work/persistedBuildData/. Every build that the plugin fails to forward to the JAT web application will be given a unique name and its JSON data will be persisted in a file with this unique name. When some new build triggers the plugin yet again, the every persisted build data instance will be fetched from its file and resent to the web application and after this the file is removed.

When the configuration is done, the JAT post-build action plugin can be used within any Jenkins job. It is selected, intuitively, as a post-build action, see Figure 4.

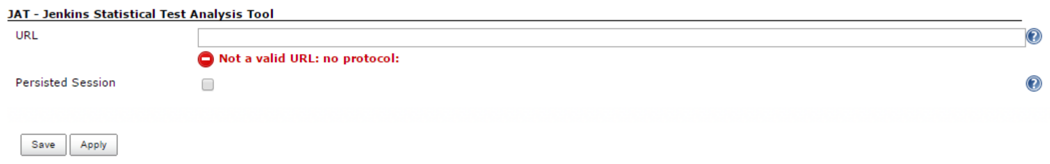

JAT - Jenkins Statistical Test Analysis Tool
URL
Not a valid URL: no protocol:
Persisted Session

Figure 3 JAT Configuration

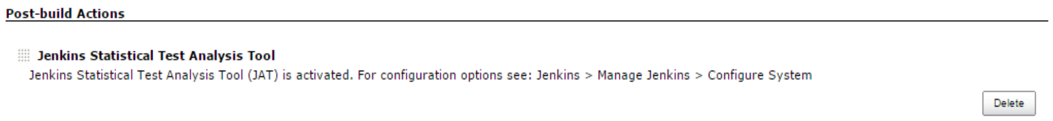

Post-build Actions
Jenkins Statistical Test Analysis Tool
Jenkins Statistical Test Analysis Tool (JAT) is activated. For configuration options see: Jenkins > Manage Jenkins > Configure System

Figure 4 JAT Job configuration

2 Installation

In order to use the JAT, both the web application and the Jenkins plugin must be built and installed.

2.1 Building JAT post-build action plugin

To install the JAT post-build action plugin the following tools are needed:

- Maven 3 or newer
- Java JDK 1.6 or newer

In addition to these tools, a Jenkins instance needs to be installed running in some web container of your choice.

The plugin is installed from the command line:

```
$ mvn package

$ mvn install
```

These commands will generate the file /target/JAT.hpi, which essentially is just a jar-file that can be executed by Jenkins.

2.2 Building JAT web application

To build the application for deployment is done from the command line with the following command:

```
$ grails war
```

This will generate a *.war file in /target/. This *war-file should then be renamed to ROOT.war and can then be uploaded to the JAT web server. NOTE: When building the JAT web application make sure to set the variable JATSERVER = "http://10.64.87.159/" in:

```
 assets > javascripts > angularinit.js
```

To install the web application on the running tomcat server simply shut down the server from command line:

```
  $ sudo service tomcat7 stop
```

Remove the ROOT.war file and the ROOT folder in:

```
  > var/lib/tomcat7/webapps/
```

Paste the new ROOT.war file in the same directory.From the command line type:

```
  $ sudo service tomcat7 start
```

Browse to http://10.64.87.159/, enjoy!

3 Tools

The JAT has been implemented using the following tools:

- Maven
- Java
- Grails
- HTML
- JavaScript
- CSS
- MySQL
- Jelly
- Bootstrap
- Tomcat
- Jenkins
- Bootstrap Glyphicons
- AngularJS