# Javache Web Server 1.3 documentation

This documentation will help you understand how to deploy applications on the Javache Web Server and how to configure your web server.

This documentation will cover.

* **Javache Overview**
* **How to Run an app**
* **App libraries**
* **Configurations**

The latest version of Javache and all required libraries will be included in the examples folder under the name **00JavacheLatest.zip**

For building an application, check out other documentation like the Summer MVC one where a step-by-step guide is made on how to create a project, import required libraries and start building your app.

# **Javache Overview**

Javache Web Server is a Java based application that listens for a connection on a given post, accepts a connection and sends the read data from the Socket’s InputStream to a set of request handlers. Those request handlers are Broccolina and Toyote.

The connection between Javache and the request handlers is done through an interface “RequestHandler” that has multiple implementations across all of the request handlers.

The main request handlers are currently limited to only 2, but the idea is that more request handlers could be added so in order for Javache to know which Request Handlers to use, there needs to be a configuration file. In this case, the configuration file is called “request-handlers.ini” and inside this file we specify which .jar files represent a request handler.

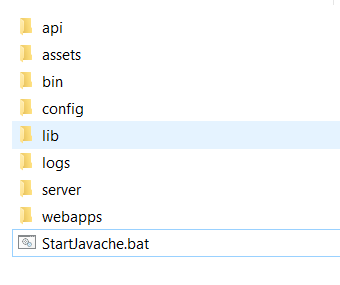
Example:

broccolina-1.1, toyote-1.1

The order of the request handlers is set from a config file (more on that later) so it doesn’t matter in which order they are put in this file.

The main folders in Javache:

From those folders **api**, **assets**, **lib**, **logs**, **webapps** are configurable (Their names can be changed, but more on that in the Configurations section).



## Api

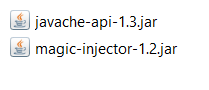
Here are located libraries that are used both by request handlers and the user. Http and Solet APIs are example of those libraries. As of version 1.3 they are the only ones in this folder.

## Assets

A place to store files. The idea is that in the **webapps** folder applications are reloaded on every start of the server so we need to store files elsewhere. Hence, in the **assets** folder. Great example for usage of the folder is when you want to store an uploaded multipart file.

## Bin

The folder that contains **Javache’s dependencies**. As of version 1.3, Javache depends on two libraries – **javache-api** and **magic-injector**.



The name of the files is important for running the application (more of that in How to run the app section).

## Config

This is where the configuration files are stored . As of version 1.3, there are 2 files inside.



Read more about config.ini In the **Configurations** section.

## Lib

This is where you put the **request handlers** as well as the **global** **application** libraries. Request handlers should be in that folder (Broccolina and Toyote). More about libraries in the **App Libraries** section.

## Temp

This is where Toyote stores temporary uploaded multipart files. (Files are removed on every request).

## Logs

This is where log files from javache are saved.

## Server

This folder contains the compile output from Javache Web Server. There is the main class that is called to start the server.

Webapps – This is where you put your applications. You can have many apps loaded at the same time. The main app is ROOT.jar by default, but this is **configurable**.

StartJavache.bat – Contains the command for starting the server on **WINDOWS**! It calls the main class and adds the files from the **bin** folder as dependencies.

@echo off

cd server

java -cp ".;../bin/javache-api-1.3.jar;../bin/magic-injector-1.2.jar" com.cyecize.StartUp 8001

PAUSE

In that command you can specify the port, which his **8000** by default.

# How To Run an App?

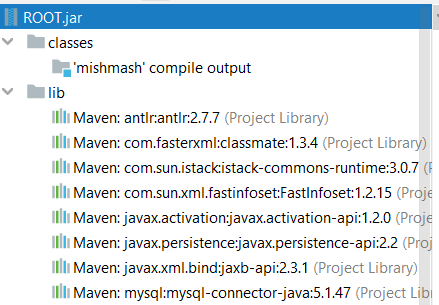
Javache allows you to run multiple applications at the same time. The folder where you want to have your apps is **webapps** (**configurable**). Inside you can put your application.jar file where the main app is by default **ROOT**.jar (**configurable**).

## How should your application jar look like?

### One way

By default, your jar file should contain a folder called “classes” and inside that folder you should put your **compiled output**. This means all classes and all resources.

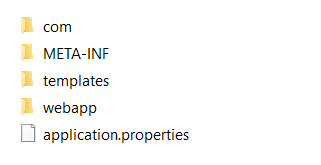
Example jar file structure.



Those two images are taken from the example with Summer MVC, which is in the Examples folder.

Inside **lib** folder there are .jar files that your application might depend on. Read **App Libraries** for additional information.

Inside classes folder is the compiled output, example again from the same app:



### The easiest way

The easiest way is to run mvn package. For libraries you could create a fat jar. This will NOT work by default on Javache. You need to change **app\_compile\_output\_dir\_name**: to “.” From “classes”.

This method is certainly easier, but the convention is that compile output should be in a folder named classes.

### Using maven to generate jar file for the standalone server:

<properties>  
 <maven.compiler.source>11</maven.compiler.source>  
 <maven.compiler.target>11</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
</properties>

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-dependency-plugin</artifactId>  
 <executions>  
 <execution>  
 <id>copy-dependencies</id>  
 <phase>prepare-package</phase>  
 <goals>  
 <goal>copy-dependencies</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>${project.build.directory}/classes/lib</outputDirectory>  
 <overWriteReleases>false</overWriteReleases>  
 <overWriteSnapshots>false</overWriteSnapshots>  
 <overWriteIfNewer>true</overWriteIfNewer>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>  
  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-antrun-plugin</artifactId>  
 <version>3.1.0</version>  
 <executions>  
 <execution>  
 <id>Prepare-For-Jar</id>  
 <phase>compile</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <encoding>${project.build.sourceEncoding}</encoding>  
 <target name="Prepare classes folder for custom jar file">  
 <copy encoding="${project.build.sourceEncoding}"  
 outputencoding="${project.build.sourceEncoding}"  
 todir="${project.basedir}/target/temp/classes/">  
 <fileset dir="${project.basedir}/target/classes/"/>  
 </copy>  
 <delete dir="${project.basedir}/target/classes"/>  
  
 <copy encoding="${project.build.sourceEncoding}"  
 outputencoding="${project.build.sourceEncoding}"  
 todir="${project.basedir}/target/classes/classes/">  
 <fileset dir="${project.basedir}/target/temp/classes/"/>  
 </copy>  
 </target>  
 </configuration>  
 </execution>  
 <execution>  
 <id>Restore-After-Jar</id>  
 <phase>package</phase>  
 <goals>  
 <goal>run</goal>  
 </goals>  
 <configuration>  
 <encoding>${project.build.sourceEncoding}</encoding>  
 <target name="Restore classes folder">  
 <delete dir="${project.basedir}/target/classes/classes"/>  
 <copy encoding="${project.build.sourceEncoding}"  
 outputencoding="${project.build.sourceEncoding}"  
 todir="${project.basedir}/target/classes/">  
 <fileset dir="${project.basedir}/target/temp/classes/"/>  
 </copy>  
 <delete dir="${project.basedir}/target/temp/"/>  
 </target>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

Using these two plugins and then calling mvn clean package, you will result in a jar file with structure that is fitted to work with the Javache standard server out of the box.  
The structure is the same as the first example:

* classes: compile output
* lib: libraries (if any)

Check out the provided file ‘example2-pom.xml’ to see the plugins in action.

## Application Start Process

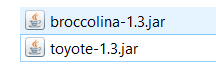
You can click on the StartJavache.bat file or execute the command inside and the application should start. If you application name is different than then main app name (ROOT by default), you will have to access you application like so: <http://localhost:8000/appName/> instead of <http://localhost:8000/>.

# App Libraries

By default, every Javache application depends on those applications: **HttpAPI**, **SoletAPI** and might depend on **Summer MVC**. Also it could depend on any third party library or framework like Hibernate or MySql.

There are 2 jar files that must be in Javache’s lib folder at ANY TIME!

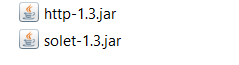
* Broccolina
* Toyote



NOTE that all names except for Broccolina and Toyote DO NOT MATTER (Javache will just add them to the classpath).

There are 2 jar files that must be in Javache’s api folder at ANY TIME!

* SoletApi
* HttpApi



You have 4 ways to approach adding library:

* Create fat jar.
* Include the libraries in the application’s lib folder.
* Include the libraries in Javache’s lib folder.
* Both at the same time.

## Create a fat jar

Using maven and the shade plugin you can create a fat jar having all or some dependencies included in the build.

A nice and easy method since you don’t need to create custom jar file. Dependencies added this way are only available for that **particular application**.

## Include in application’s lib folder.

Similar to fat jar, but you actually put the .jar files and not the extracted classes. You will have to create a folder called **lib** (or different if configuration is changed) and place the jars there. This is a preferred way over fat jar because your application will load faster since broccolina will not have to scan all classes. Dependencies added this way are only available for that **particular application**.

## Include in Javache’s lib folder.

You can put the application dependencies in Javache’s lib folder where those libraries will be accessible by all applications.

## Both at the same time.

JUST DO BOTH. Add global libraries to Javache’s lib folder and do one of the first 2 methods for any unique dependencies for that specific app.

# Sandbox and Class loaders

Since Javache can load multiple apps, it is very important that each application has its own context. Javache creates a new Class Loader for each web application so that even if there are two apps with the same class names or same libraries, but different versions they will be separated in their own sandbox.

So adding a jar file in app A’s lib folder will not be visible to app B. You will need to put it in the global lib folder in order to achieve that.

# Configurations

Javache has a handful of configurations, which can be found in the config/config.ini file.

Those are the possible configurations:

* max\_request\_size
* show\_request\_log
* assets\_dir\_name
* web\_apps\_dir\_name
* app\_compile\_output\_dir\_name
* main\_app\_jar\_name

### max\_request\_size

Defaults to 2 GB if not set. Specify the max size (in bytes) of each request. If the request is exceeded, you will get an error 400 BAD REQUEST.

### show\_request\_log

Defaults to false. If set to TRUE, the raw request content will be printed and logged into the logs file.

### assets\_dir\_name

Defaults to assets. The name of the folder, which stores permanent data for an app.

### web\_apps\_dir\_name

Defaults to webapps. The name of the folder, which contains the applications.

### app\_compile\_output\_dir\_name

Defaults to classes. Sets the folder name for the compiled output for you applications.  
If you are using mvn package, set this to “.” Since the compile output is in the root folder.

### app\_resources\_dir\_name

Defaults to webapp. This is the folder in which the web applications will have their static resources such as html, css, js or images.

### main\_app\_jar\_name

Defaults to ROOT.jar. The name of the main application.

### application\_dependencies\_folder\_name

Defaults to lib. The name of the folder within the web app where the jar files of its libraries will be placed.

### broccolina\_force\_overwrite\_files

Defaults to true. If set to false, broccolina will check if the application JAR has been extracted and if it is, it will not extract it again. This is good for improving startup speeds, but it is risky in case the web app is updated and the extracted folder hasn’t been removed.

### brocollina\_skip\_extracting\_if\_folder\_exists

Defaults to false. This setting is deprecated. It is replaced by **broccolina\_force\_overwrite\_files**.

### broccolina\_track\_resources

Defaults to true. This setting specifies whether broccolina will process a request which is a static resource (contains dot)

### server\_port

Defaults to 8000. The server port can also be passed as an argument in the main method. The setting in config.ini is prioritized, if it is different than the default.

### lib\_dir\_name

Defaults to lib. This setting sets the name of Javache’s global shared libraries folder.

### api\_dir\_name

Defaults to api. This setting sets the name of Javache’s API folder.

### logs\_dir\_name

Defaults to logs. This setting sets the name of Javache’s folder for writing logs.

### javache\_print\_exceptions

Defaults to true. This setting tells if unhandled exceptions should be printed and logged.

### toyote\_resource\_handler\_order

Defaults to 1. This tells Javache the order in which it should execute the request handler for serving static content. Lower means earlier.

### broccolina\_solet\_dispatcher\_order

Defaults to 2. This tells Javache the order in which it should execute the request handler for serving dynamic content. Lower means earlier.

### enable\_resource\_caching

Defaults to true. This setting tells Toyote whether it should add the Cache-Control header when serving static resources.

### resource\_caching\_expression

Defaults to - image/png, image/gif, image/jpeg @ max-age=120 & text/css @ max-age=84600, public & application/javascript @ max-age=7200

This setting provides Toyote with instructions for caching resources. For each one or a group of media types, there is a Cache-Control value that will be returned when serving a request.

The format is the following:

media/type1, media/type2 @ header-value & media/type3 @ header-value

& - separates different header values