Wisdom Framework

The Wisdom Framework Team - 0.8.0

> By three methods we may learn wisdom: First, by reflection, which is noblest; Second, by imitation, which is easiest; and third by experience, which is the bitteREST.
>
> — Confucius

1. The Wisdom Framework

1.1. Build Web applications with Java and JavaScript.

Wisdom is a lightweight framework enforcing a stateless, web-friendly architecture. It's modular, and dynamic.

Wisdom is based on the NIO and asynchronous framework VertX. It is integrally built on top of OSGi, to enable modularity, and iPOJO, to make dynamism easy as pie.

With Wisdom, you web application is not monolithic anymore, but divided into a set of components *deployable* on the fly, and enabled / disabled dynamically, remotely or locally.

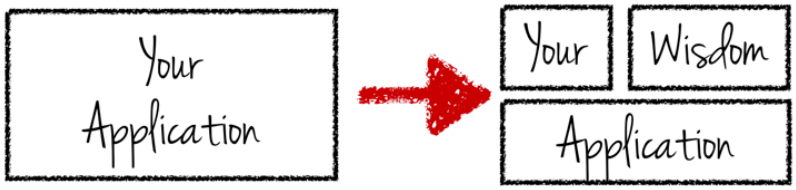

Figure 1. Make your applications modular

1.2. Simple development model, Fast turn-over

Wisdom proposes a very simple development model where you focus on your application logic and not on how things are working.

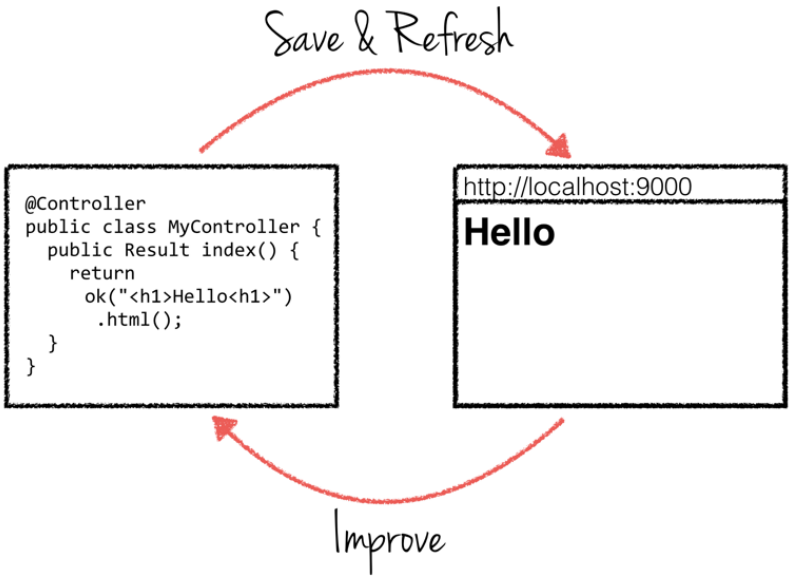

Figure 2. Save and Refresh

When developing an application, don't spend you time to re-package and re-deploy your app. Save your files, hit refresh in your browser. Everything is done for you.

Wisdom relies on the Apache Maven build tool. Don't run away. We made it very easy. It integrates everything you need from testing, to packaging and obviously natural continuous integration.

1.3. Modularity as a core value, dynamism everywhere

You may say: 'well, what's the difference with Grails or Play Framework'. Wisdom is intrinsically modular and dynamic. That means:

- Your application is no longer monolithic, but modular
- Each part of your application is autonomous, can be developed, tested and deployed individually
- Your application is naturally dynamic, every part can come and leave at anytime, and more importantly can be updated without downtime
- Wisdom itself is modular and dynamic, so extend it, replace technical services… on the fly

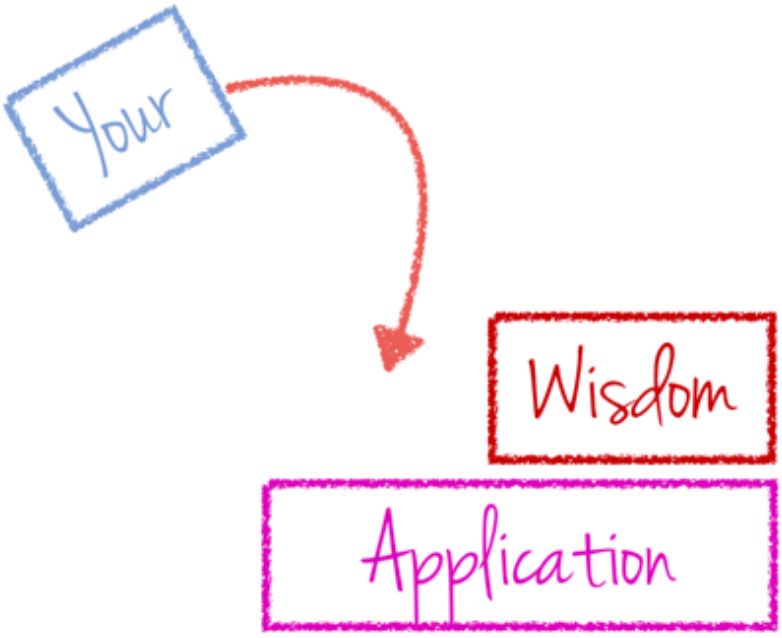

Figure 3. Dynamic Modular Application

A full web stack for Java

Wisdom proposes a full web stack:

- RESTful by default
- Asset optimization, CoffeeScript, Less compilation, etc.
- Data Validation
- JSON
- Web Sockets
- HTML5 template system
- HTML 5

Wisdom runs on the Java Virtual Machine. Wisdom applications can run on any Cloud provider supporting Java.

1.4. Why Wisdom?

We developed dozens of web applications in the past five years, using lots of different technologies: JavaEE, Grails, Lift, Play Framework (1 and 2), pure-JavaScript… We learned a lot from all these great frameworks, but for us, something was missing: the dynamism.

Our applications live in ever-changing (and expanding) environments, face ever-shifting (and increasing) demands, and must live up to ever-escalating expectations. As Niels Bohr said: “Prediction

is very difficult, especially if it's about the future".

To face this situation, we need modularity and dynamism. Modularity to decompose applications into well-defined autonomous blocks, managed individually. Dynamism to manage the bindings between these components, their runtime environment and others services.

2. Getting started

2.1. Installing Wisdom

2.1.1. Prerequisites

To run Wisdom, you need:

1. Java 7+ (Java Development Kit)
2. Apache Maven. Wisdom requires a recent version of Maven (3.2.1+)

In production, Wisdom just requires the Java 7+ virtual machine.

> **TIP** Java 8 is also supported. So you can use lambda expressions !

2.2. Nothing to install ?

You don't need to install anything ? The Wisdom build process will download everything.

3. Your first Wisdom project

> **IMPORTANT** Be sure you have a recent version of Maven (3.2.1 `).

3.1. Create a Wisdom project

Wisdom is based on the Apache Maven build tools. However don't worry, Wisdom makes it easy. However, first, be sure you have Maven installed.

Once installed, you can create your first Wisdom project by launching:

```
mvn org.wisdom-framework:wisdom-maven-plugin:0.8.0:create \
 -DgroupId=YOUR_GROUPID \
 -DartifactId=YOUR_ARTIFACTID \
 -Dversion=1.0-SNAPSHOT
```

On Windows use:

```
mvn.bat org.wisdom-framework:wisdom-maven-plugin:0.8.0:create -DgroupId="YOUR_GROUPID"
-DartifactId="YOUR_ARTIFACTID" -Dversion="YOUR_VERSION"
```

TIP You can also customize the generated `package` with `-Dpackage=org.acme.wisdom`

Once generated, you can already run your wisdom application. Navigate to the generated folder, and launch:

```
mvn wisdom:run
```

Then, open your browser to: http://localhost:9000.

That's it you have your first Wisdom application.

3.2. Using the Maven Archetype

This section is only for Maven users willing to use an *archetype*. The very same project can be generated using the `wisdom-default-project-archetype`:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:generate \
 -DarchetypeArtifactId=wisdom-default-project-archetype \
 -DarchetypeGroupId=org.wisdom-framework \
 -DarchetypeVersion=0.8.0 \
 -DgroupId=YOUR_GROUPID \
 -DartifactId=YOUR_ARTIFACTID \
 -Dversion=1.0-SNAPSHOT
```

The archetype can be used from your IDE to generate a new project directly.

3.3. What was generated?

The previous commands have generated a set of folders. Following the Maven conventions, all files are placed in the `src` directory, while the generated artifacts are in `target`.

The `src/main` directory is structured as follows:

- `src/main/java` contains the Java sources of your application
- `src/main/resources` contains the assets and templates included in your application *package*
- `src/main/configuration` contains the configuration file (application, runtime, logger...)
- `src/main/assets` contains assets that are copied to Wisdom but not included in application *package*

- `src/main/templates` contains assets that are copied to Wisdom but not included in application *package*

The `src/test` directory contains the tests (unit and integration tests).

3.4. You first controller

The generated application contains a *controller*. A *controller* is a Wisdom component containing *actions*, *i.e.* methods called using HTTP requests.

```
package sample;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.*;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
public class WelcomeController extends DefaultController {

    @View("welcome")
    private Template welcome;

    @Route(method = HttpMethod.GET, uri = "/")
    public Result welcome() {
        return ok(render(welcome));
    }

}
```

① First, declare the Java class as Wisdom's controller

② Extend `DefaultController` to inherit from the useful method

③ Specify the HTTP route connected to your action method

④ The body of your action, returning a `Result` object

In the previous example, the controller is using a template (`welcome`), but things can be much simpler:

```
/**
 * Your first Wisdom Controller.
 */
@Controller
public class SampleController extends DefaultController {

    @Route(method = HttpMethod.GET, uri = "/hello")
    public Result hello() {
        return ok("hello Wisdom !");
    }
}
```

① First, declare the Java class as Wisdom's controller

② Extend `DefaultController` to inherit from the useful method

③ Specify the HTTP route connected to your action method

④ The body of your action, returning a `Result` object

3.5. Wisdom API and Maven Plugin

- The Wisdom JavaDoc is available on the API page.
- The Wisdom-Maven-Plugin goals and configuration are described on the Maven Site.

4. Package, Test, Run your application

Wisdom's build system is based on Apache Maven, so the build process is following the *default* Maven lifecycle.

4.1. Package your application

`mvn package` creates two files: a jar file containing your application, and a zip file containing the whole distribution, ready to be deployed.

NOTE The jar file is an OSGi bundle.

4.2. Adding dependencies

To add a dependency to your application, just add a `<dependency>` in the `pom.xml`. Wisdom copies all dependencies from the `compile` scope (default Maven scope):

```
<dependency>
<groupId>asm</groupId>
<artifactId>asm-all</artifactId>
<version>3.3.1</version>
</dependency>
```

However notice two important rules:

1. It does not copy transitive dependencies, so don't forget to declare all your dependencies.
2. Only OSGi bundles are copied, so check that your dependencies are valid OSGi bundles.

IMPORTANT | Most libraries are OSGi bundles already.

4.3. Finding bundles

Unfortunately, you may want to use a library that is not a bundle. When facing this situation, you have 3 choices:

1. Trying to find a *bundlelized* version of the library
2. Use the *non-OSGi dependency* support of Wisdom - Check the Using non-OSGi dependencies section
3. Make your own bundle (that you can contribute to the project) - Check the Some bundelization rules section

You may want to check the bundles available on:

- Service Mix Bundles - Apache Service Mix have created lots of bundles of all the dependencies they use
- Spring Source OSGi Repository - Spring Source has also created a bunch of bundles you can find on their *Enterprise Bundle Repository*

4.4. Using non-OSGi dependencies

Unfortunately, not all libraries are OSGi bundles. Fortunately, Wisdom supports non-bundle dependencies. These dependencies are copied to the `libs` directory of the Wisdom server and are called *libraries*. These dependencies does not need to be OSGi bundles, and can be used by the Wisdom applications.

First add your Maven dependency as usual such as:

```
<dependency>
  <groupId>org.pegdown</groupId>
  <artifactId>pegdown</artifactId>
  <version>1.4.2</version>
</dependency>
```

Then, you need to select explicitly the libraries from the set of dependencies. Only selected dependencies are copied. This selection is made as follows:

```
<plugin>
  <groupId>org.wisdom-framework</groupId>
  <artifactId>wisdom-maven-plugin</artifactId>
  <version>0.8.0</version>
  <extensions>true</extensions>
  <configuration>
    <!-- Defines the set of 'libraries' -->
    <libraries>
      <includes>
        <!--
        defines the set of dependencies to select as library (mandatory).
        it uses the following syntax: groupId:artifactId:type[:classifier]:version
        -->
        <include>:pegdown</include>
      </includes>

      <!--
      whether or not transitive dependencies of the selected libraries should also
      be copied (default to true). only 'compile' dependencies are copied.
      -->
      <resolveTransitive>true</resolveTransitive>

      <!--
      whether or not the selected artifacts should be excluded from the 'application'
       directory (default to false).
      -->
      <excludeFromApplication>true</excludeFromApplication>

      <!--
      allow to exclude some artifacts. This feature is useful when transitive support
      is enabled to filter out undesirable artifacts.
      -->
      <excludes>
        <exclude>:asm-tree</exclude>
      </excludes>
    </libraries>
  </configuration>
</plugin>
```

The `libraries` element configure the selection and behavior. It must contains an `includes` element with at least one `include` sub-element. If none are set, no dependencies are considered as libraries.

Using libraries allows you to rely on non-bundle dependencies, but it comes with a couple of limitations:

- the set of libraries is static, so cannot change after the wisdom server has started.

- there is no dependency resolution at runtime, meaning that all dependencies required by selected libraries must also be available from the 'libs' directory.

In other word, libraries are very useful when you can't find a suitable OSGi bundle, but need to be used carefully to not break the modularity and dynamism of your applications.

NOTE	The content of the jar files contained in the `libs` directory are exported by the OSGi framework.
IMPORTANT	This feature only works when using Apache Felix.

4.5. Some bundelization rules

If you want to create your own OSGi bundle, here are some rules:

- Inspect the library to separate the public API from the private / implementation parts - this is particularly important to avoid exposing implementation details.
- Detect ClassLoader.loadClass calls, ClassLoader.findResource, and more generally all `ClassLoader` access
- Check whether the code rely on the *Thread Context ClassLoader*, if so, wrap the calls with a TTCL Switch:

```
final ClassLoader original = Thread.currentThread().getContextClassLoader();
try {
    Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
    // your code using the library.
} finally {
    Thread.currentThread().setContextClassLoader(original);
}
```

- Detect SPI usage, and try to see if you can create the instance manually and inject it
- Last but not least, design a service interface, to avoid leaking the whole API - and if it's possible, make the service interface independent of the library (meaning that you can substitute by another one someday)

4.6. Run tests

Tests are situated in the `src/test/java` directory.

`mvn test` executes all the unit tests following the Surefire convention. So, it executes all tests from classes starting or finishing by `Test`, such as `MyClassTest` or `TestMyClass`.

`mvn integration-test` executes all the integration tests following the Surefire convention. So, it executes

all tests from classes starting or finishing by `IT`, such as `MyComponentIT` or `ITMyComponent`..

Unit tests are also executed in *watch mode*. It re-executes the unit tests when you change either a Java class or a test. By default, all test are re-executed. However, to reduce the amount of test, use:

```
<testSelectionPolicy>SELECTIVE</testSelectionPolicy>
```

With the `SELECTIVE` policy, only related tests are executed. Notice that, test failures are reported in the browser. You can skip tests adding the `-DskipTests` flag to the Maven command line.

4.7. Watch mode

To run the application while developing, launch:

```
mvn wisdom:run
```

It packages your application and starts the Wisdom server. It also *watches* changes you make on your files and redeploys *things* automatically. For instance, if you edit a Java file, it compiles, packages and redeploys the application.

WARNING Modifying the `pom.xml` file relaunches the server automatically.

The wisdom watch mode allows you to use a *remote* Wisdom server. Using this feature, you can *watch* several project at the same time. To enable this feature launch the watch mode with `-DwisdomDirectory=location`.

So, imagine you have two projects, `P1` and `P2`, and `P2` depends on `P1`. You want to enjoy the *watch* mode on both `P1` and `P2` projects, but, as `P2` is your *final* / *main* project, you also want to immediately copy the `P1` output to `P2`. In this context, just launch the *watch* mode twice as follows:

```
# P2 - regular watch mode (it starts the server we are going to use in P1)
mvn wisdom:run
```

```
# P1 - watch mode with remote server
mvn wisdom:run -DwisdomDirectory=../p2/target/wisdom
```

TIP Launch the main server first, if not, it may override the copied resources.

WARNING	When you launch the *watch* mode with the `wisdomDirectory`, some copy functionality is disabled (configuration and external resources). So in our previous example, modifying the `P1` configuration, external resources, or templates do not copy them to `P2`. However, as `P2` is launched without the parameter, all features are provided.

TIP	The watch mode is based on a File Alteration Monitor (FAM). The default polling period is set to 2 seconds. You can configure it using `-Dwatch.period=5`. The set time is in **seconds**.

4.8. Debugging

In watch mode, you can enable the remote debugging by using the `-Ddebug=port`:

```
mvn wisdom:run -Ddebug=5005
```

Then, just launch the debug mode of your IDE using the set port (5005 in the previous example).

4.9. Run the application

The application is assembled in the `target/wisdom` directory. You can run it without the watch mode using:

```
./chameleon.sh
```

TIP:Why Chameleon? Because Chameleon is a kind of OSGi distribution with some additional features on which Wisdom relies.

4.10. Accessing the OSGi shell

If you are an OSGi expert and want to access the *shell* launch the application with:

```
./chameleon.sh --interactive
```

The provided shell is OW2 Shelbie.

4.11. The distribution

`mvn package` generates an OSGi bundle (a jar file) and a zip file. The jar file can be deployed into any other wisdom application. The zip file contains the whole distribution and can be run using the same shell script as previously explained.

4.12. Customizing applications packaging

Wisdom applications are packaged inside OSGi bundles. By default it:

- includes all classes (from `src/main/java`) and resources (from `src/main/resources`)
- imports all required packages (deduced by analyzing the classes)
- exports all packages containing `service(s)`, `api(s)`, `model(s)`, `entity` or `entities`

However, sometimes you want to customize this default packaging policy. Wisdom relies on BND to build bundles. BND is building the OSGi bundle by following a set of *instructions* such as:

```
Import-Package: com.library.*; version = 1.21
```

To customize the bundle of your application, create a `osgi.bnd` file in `src/main/osgi/`, and write the BND instructions there.

For example, the Wisdom Hibernate Validation bundle has the following instructions:

```
Private-Package: org.wisdom.validation.hibernate;-split-package:=merge-first,  \
    org.jboss.logging.*, \
    com.fasterxml.classmate.*, \
    org.hibernate.validator*, \
    com.sun.el*, \
    javax.el*
Import-Package: javax.validation.*;version="[1.1.0,2.0.0)", \
                !org.apache.log4j, \
                org.jsoup*;resolution:=optional, \
                javax.persistence*;resolution:=optional, \
                org.jboss.logmanager*;resolution:=optional, \
                *
Export-Package: org.hibernate.validator.constraints*
```

TIP The `osgi.bnd` file can use Maven properties.

When used, the `osgi.bnd` file is merged with the default instructions used by Wisdom (`Import-Package`, `Export-Package` (based on the package name), `Private-Package`, and `Include-Resource`). In other words, every instructions specified in the `osgi.bnd` replaces default value provided by Wisdom for this instruction. For example, if your `osgi.bnd` file contains `Export-Package: my.package.to.be.exported`, it won't compute the default exported packages, but use your value (and only your value).

You can completely disable the Wisdom defaults, by adding the following instruction in the `osgi.bnd` file:

```
-no-default: true
```

4.13. Embedding Maven dependencies inside the application package

You can declare a set of (Maven) dependency to be packaged within the application bundle. The Wisdom Maven Plugin supports embedding of selected project dependencies inside the bundle by using the `Embed-Dependency` instruction:

```
Embed-Dependency: dependencies
```

where:

```
dependencies ::= clause ( ',' clause ) *
clause ::= MATCH ( ';' attr '=' attribute | ';inline=' inline | ';transitive='
transitive)
attr ::= 'scope' | 'type' | 'classifier' | 'optional'
transitive ::= 'true' | 'false'
inline ::= 'true' | 'false' | PATH ( '|' PATH ) *
attribute ::= <value> ( '|' <value> ) *
MATCH ::= <Maven Pattern Dependency Filter ([groupId]:[artifactId]:[type]:[version])>
PATH ::= <Ant-style path expression>
```

The plugin uses the `Embed-Dependency` instruction to transform the project dependencies into `Include-Resource` and `Bundle-ClassPath` clauses, which are then appended to the current set of instructions and passed onto BND.

Let's see some examples:

```
# embed all compile and runtime scope dependencies
Embed-Dependency: *;scope=compile|runtime

# embed any dependencies with artifactId junit and scope runtime
Embed-Dependency: junit;scope=runtime

# embed all compile and runtime scope dependencies, except those with artifactIds in the
given list
Embed-Dependency:
*;scope=compile|runtime;inline=false;exclude=:cli|:lang|:runtime|:tidy|:jsch

# inline contents of selected folders from all dependencies
Embed-Dependency: *;inline=images/**|icons/**
```

Normally the plugin only checks direct dependencies, but this can be changed to include the complete set of transitive dependencies with the following option:

```
Embed-Transitive: true
```

It can also be enabled using the `transitive=true` attribute.

IMPORTANT	Be aware that Wisdom is using 1) a different syntax than the Maven-Bundle-Plugin to select the set of artifacts, 2) Inline artifacts by default

4.14. Configuring the Java source and target

By default, Wisodn compiles your code using Java 1.7. This convention was made to stick with the Maven convention. However you can override these settings by adding the following properties in your `pom.xml`:

```
<properties>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.testSource>1.8</maven.compiler.testSource>
        <maven.compiler.testTarget>1.8</maven.compiler.testTarget>
</properties>
```

5. HTTP Programming

5.1. Actions, Controllers and Results

5.1.1. What is an Action?

Most of the requests received by a Wisdom application are handled by an *action*. An action is basically a Java method that processes the received request parameters, and produces a `Result` to be sent to the client.

```
@Route(method = HttpMethod.POST, uri = "/")
public Result index() {
    return ok("Got request " + request() + "!");
}
```

An action returns a `org.wisdom.api.http.Result` value, representing the HTTP response to send to the web client. In this example ok constructs a `OK` (HTTP Code 200) response containing a *text/plain* response body.

Actions are declared with the `org.wisdom.api.annotations.Route` annotation (`@Route`) indicating the HTTP method and *uri* used to invoke the action.

5.1.2. Controllers

Controllers are Wisdom components containing the application logic. A controller is nothing more than a class annotated with the `org.wisdom.api.annotations.Controller` (`@Controller`) that groups several action methods.

```
package snippets.controllers;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
public class Simple extends DefaultController {

    @Route(method= HttpMethod.GET, uri = "/works")
    public Result index() {
        return ok("Follow the path to Wisdom");
    }
}
```

WARNING	Be sure to import `org.wisdom.api.annotations.Controller` when you use the `@Controller` annotation and not `org.wisdom.api.Controller`.

The simplest syntax for defining an action is a (non-static) method with no parameters that returns a `Result` value, as shown above.

An action method can also have parameters:

```
package snippets.controllers;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Parameter;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
public class Name extends DefaultController {

    @Route(method= HttpMethod.GET, uri = "/hello/{name}")
    public Result index(@Parameter("name") String name) {
        return ok("Hi " + name + ", follow us on the Wisdom path");
    }
}
```

These parameters (indicated with the `@Parameter` annotation) will be resolved by Wisdom and will be filled with values from the request. The parameter values can be extracted from either the URL path or the URL query. The annotation value, `"name"` in the previous example, indicates the name of the parameter.

5.1.3. Results

Let's start with basic results: an HTTP result with a status code, a set of HTTP headers and a body to be sent to the client. These results are defined by `org.wisdom.api.http.Result`, and the `org.wisdom.api.http.Results` class provides several helpers to produce standard HTTP results, such as the `ok` method we used in the previous examples:

```
package snippets.controllers;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
public class Simple extends DefaultController {

    @Route(method= HttpMethod.GET, uri = "/works")
    public Result index() {
        return ok("Follow the path to Wisdom");
    }
}
```

Here are several examples that create various results:

```
Result ok = ok("Hello wisdom!");
Result page = ok(render(template));
Result notFound = notFound();
Result pageNotFound = notFound("<h1>Page not found</h1>").as(MimeTypes.HTML);
Result badRequest = badRequest(render(template, "error", errors));
Result oops = internalServerError("Oops");
Result exception = internalServerError(new NullPointerException("Cannot be null"));
Result anyStatus = status(488).render("Strange response").as(MimeTypes.TEXT);
```

All of these helpers can be found in the `org.wisdom.api.http.Results` class.

5.1.4. Redirects are simple results too

Redirecting the browser to a new URL is just another kind of simple result. However, these result types don't have a response body.

```
@Route(method= HttpMethod.GET, uri="/redirect")
public Result redirectToIndex() {
    return redirect("/");
}
```

The default is to use a `303 SEE_OTHER` response type, but you can also specify a more specific status code:

```
@Route(method= HttpMethod.GET, uri="/tmp")
public Result redirectToHello() {
    return redirectTemporary("/hello/wisdom");
}
```

5.1.5. Using the request scope

The request scope is a map shared by all the entities participating to the request resolution (i.e. response computation). It includes filters, interceptors, action methods… So it let these entities to share data.

You can write or retrieve data from the request scope as follows:

```
int echo = (int) request().data().get("echo");
request().data().put("echo", echo + 1);
```

The request scope can contain any types of objects, and is cleared once the response is sent back to the client.

Objects from the request scope can be injected as action's method parameters:

```
@Route(method=HttpMethod.GET, uri="/test")
public Result doIt(@HttpParameter("echo") int echo) {
    // ...
}
```

IMPORTANT	The request scope must be used to avoid using `Thread Local` object. `Thread Local` do not conform to the Wisdom execution model, and would not be available in async result processing. So, instead of using a `Thread Local`, put the object into the request scope. By this way, you ensure the object to be available until the end of the request processing. == The HTTP request router

The router is the component that translates each incoming HTTP request to an action call. The HTTP request contains two major pieces of information:

1. the request path (such as `/foo/1234`, `/photos/`), including the query string (the part after the `?` in the url).
2. the HTTP method (GET, POST, …).

Routes are defined directly on the action method with the `@Route` annotation.

5.2. The @Route annotation

The `@Route` annotation contains two attributes: the HTTP method and the URI pattern determining on which URLs the action is associated.

Let's see a couple of examples:

```
@Route(method=HttpMethod.GET, uri="/photos")
@Route(method=HttpMethod.GET, uri="/photos/{id}")
@Route(method=HttpMethod.POST, uri="/photos/")
@Route(method=HttpMethod.DELETE, uri="/photos/{id}")
```

5.3. The HTTP method

The HTTP method can be any of the valid methods supported by HTTP (mainly GET, POST, PUT, DELETE). They are defined in `org.wisdom.api.http.HttpMethod`.

5.4. The URI pattern

The URI pattern defines the route's request path. Some parts of the request path can be dynamic.

5.4.1. Static path

For example, to exactly match `GET /photos` incoming requests, you can define this route:

```
@Route(method=HttpMethod.GET, uri="/photos")
```

5.4.2. Dynamic parts

If you want to define a route that, say, retrieves a photo by id, you need to add a dynamic part, filled by the router:

```
@Route(method=HttpMethod.GET, uri="/photos/{id}")
```

NOTE URI patterns may have more than one dynamic part such as in `/foo/{id}/{name}`.

The default matching strategy for a dynamic part is defined by the regular expression defined as `{id}` will match exactly one URI path segment.

5.4.3. Dynamic parts spanning several path segments

If you want a dynamic part to capture more than one URI path segment, separated by slashes, you can

define a dynamic part using the `{path*}` syntax:

```
@Route(method=HttpMethod.GET, uri="/assets/{path*}")
```

Here, for a request like `GET /assets/stylesheets/style.css`, the name dynamic part will capture the `stylesheets/style.css` value.

> **NOTE** `{path*}` scheme accepts empty input. To avoid this, use `{path+}`.

5.4.4. Dynamic parts with custom regular expressions

You can also define your own regular expression for a dynamic part, using the `{id<regex>}` syntax:

```
@Route(method=HttpMethod.GET, uri="/photos/{id<[0-9]+>}")
```

5.5. Controller's path

Often the actions of one controller starts with a common prefix, such as `/photos`. To avoid the repetition, you can use the `@Path` annotation indicating the common prefix prepended to all uris from the `@Route`.

```
package snippets.controllers;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Parameter;
import org.wisdom.api.annotations.Path;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
@Path("/photo")
public class PhotoController extends DefaultController {

    @Route(method= HttpMethod.GET, uri="/")
    public Result all() {
        return ok();
    }

    @Route(method= HttpMethod.POST, uri="/")
    public Result upload() {
        return ok();
    }

    @Route(method= HttpMethod.GET, uri="/{id}")
    public Result get(@Parameter("id") String id) {
        return ok();
    }

    // ...
}
```

5.6. Action Parameters

Action methods can receive parameters. Several types of parameters are supported:

1. parameter from the query part of the url or from the dynamic part of the uri. These parameters are annotated with `@Parameter`, `@QueryParameter` and `@PathParameter`
2. fields from form are annotated with `@FormParameter`. It's also used for file upload.
3. object built from the request body (for `POST` requests). These parameters are annotated with `@Body`
4. `@HttpParameter` injects an object extracted from the incoming request such as a header, request scope data, the HTTP context, the request and cookies.

5. `@BeanParameter` injects an instance of a class using the other annotations.

All action method parameters must be annotated with one of the above annotations.

5.6.1. Types

Before going further, let's see what types are supported. The type is extracted from the annotated argument (i.e. formal parameter). So in `@Parameter("param") String p`, it would be `java.lang.String`, while in `@Parameter("params") List<String> p`, it would be a `java.util.List<String>`.

Wisdom supports:

- String
- Primitive types
- Classes having a constructor with a single String argument
- Classes having a `valueOf` static method with a single String argument (so support enumeration)
- Classes having a `from` static method with a single String argument
- Classes having a `fromString` static method with a single String argument
- Any type with a `org.wisdom.api.content.ParameterConverter` service handling it
- Array, List and Set containing previously listed types

Boolean's values are extended. Are considered as 'true': "true", "on", "yes", "1". Other strings are considered as 'false'.

If you want to extend such support, you just have to implement `org.wisdom.api.content.ParameterConverter` and expose it as a service.

5.6.2. @Parameter, @QueryParameter and @PathParameter

The `@Parameter` annotation indicates that the marked method parameter takes a value from either the query part of the method, or a dynamic part of the URI.

In the following example, the `id` value is computed from the URL. For instance, `http://localhost:9000/parameters/foobar` would set id to `foobar`.

```
    @Route(method= HttpMethod.GET, uri="/{id}")
    public Result get(@Parameter("id") String id) {
        // The value of id is computed from the {id} url fragment.
        return ok(id);
    }
```

Using query parameter is also simple:

```
    @Route(method= HttpMethod.GET, uri="/")
    public Result get2(@Parameter("id") String id) {
        // The value of id is computed from the id parameter of the query.
        // In /?id=foo it would get the "foo" value.
        return ok(id);
    }
```

With the previous snippet, and the URL `http://localhost:9000/parameters/?id=foo`, `id` would be set to `foo`.

When bound to an array type, Wisdom collects the values of the given parameter name in the query part of the request. It builds an array injectable as follows:

```
@Parameter("ids") List<Integer> ids,
@Parameter("names") String ns[]
```

If the parameter has no values, an empty array or list is injected.

`@Parameter` can receive a *default value* (except if the parameter is deduced from the path):

```
@Parameter("id") @DefaultValue("0") int id,
@Parameter("ids") @DefaultValue("0, 1") int[] ids
```

The default value is a String representation of the value to use if there are no values. For collections and arrays, the default value is interpreted as a comma-separated list.

If you want to enforce the origin of the value you can use the `@PathParameter` and `@QueryParameter` annotation to select the value respectively from the path or query.

IMPORTANT `@DefaultValue` cannot be used with `@PathParameter`.

5.6.3. @FormParameter

the `@FormParameter` annotation retrieves values sent from HTML forms.

```
    @Route(method= HttpMethod.POST, uri="/")
    public Result post(@FormParameter("id") String id, @FormParameter("name") String
name) {
        // The values of id and names are computed the request attributes.
        return ok(id + " - " + name);
    }
```

In the previous snippet, two attributes can be received by the action method: `id` and `name`. A form sending such data as `application/x-www-form-urlencoded` will be handled directly.

When bound to an array type or a collection, Wisdom collects the values of the given attribute. It builds an injectable object (array or collection) as follows:

```
@FormParameter("ids") List<Integer> ids,
@FormParameter("names") String ns[]
```

If the attribute has no values, an empty array or collection is injected.

`@FormParameter` can receive a *default value*:

```
@FormParameter("id") @DefaultValue("0") int id,
@FormParameter("ids") @DefaultValue("0, 1") int[] ids
```

The default value is a String representation of the value to use if there are no values. For collections and arrays, the default value is interpreted as a comma-separated list.

5.6.4. @Body

The `@Body` annotation lets you wrap the request into an object. This object must be either a bean or have `public` attributes.

For instance, let's imagine you need to manage an instance of the following class:

```
public class MyData {

    public String name;
    public String id;

    public String toString() {
        return "Data: " + id + " - " + name;
    }

}
```

You don't want to build your instance by yourself, luckily Wisdom handles that for you:

```
@Controller
public class BodyWrap extends DefaultController {

    @Route(method= HttpMethod.POST, uri = "/wrap")
    public Result index(@Body MyData data) {
        return ok(data.toString());
    }
}
```

The `@Body` annotation builds an object of the parameter's type using the request content and parameters. It becomes very handy when handling JSON data, or complex form.

5.6.5. @HttpParameter

The `@HttpParameter` annotation let you inject HTTP related data such as the `Request`, the `Context`, HTTP header's value...

```
@Route(method = HttpMethod.GET, uri = "/parameter/http")
    public Result http(
      @HttpParameter Context context, // HTTP Context
      @HttpParameter Request request, // Incoming Request
      @HttpParameter("X-header") String header, // Header value
      @HttpParameter SessionCookie session, // Session
      @HttpParameter Stuff stuff) { // Custom object (invoke a factory service)
    //...
                        }
```

The type of the parameter determines the injected value:

- the HTTP `Context` - `org.wisdom.api.http.Context`
- the HTTP `Request` - `org.wisdom.api.http.Request`
- a `Cookie` object (cookie's name given in parameter) - `org.wisdom.api.cookies.Cookie`
- the `Session` object (Session cookie) - `org.wisdom.api.cookies.SessionCookie`
- the `Flash` object (Flash cookie) - `org.wisdom.api.cookies.FlashCookie`
- the invoked `Route` - `org.wisdom.api.router.Route`
- a `Reader` object to read the request body

If the type does not match one of the previous case:

1. if a `org.wisdom.api.content.ParameterFactory` service handling the parameter's type is available, it

invokes it.

2. if a HTTP header has the name given in the annotation parameter, it injects the header value
3. otherwise it looks into the *request scope* to check whether a filter or interceptor has provided a value with the name given in the annotation parameter.

5.6.6. @BeanParameter

The last annotation lets you build a bean using any of the other annotations presented here. The bean's class must:

- have a *no-args* constructor, or a constructor using the other annotations.
- have setter methods (starting with *set*), having only one parameter, and this parameter must be annotated with one of the other annotations (it can also use the `@DefaultValue` annotation).

Let's see an example of such a bean:

```
/**
 * A bean injected using the {@link org.wisdom.api.annotations.BeanParameter} annotation.
 */
public class Bean {

    private final String value;

    /**
     * You can add validation constraint on the injected value.
     */
    private @NotNull String value2;

    /**
     * Values can be injected in the constructor, and combined with the
     * {@link org.wisdom.api.annotations.DefaultValue} annotation.
     *
     * @param v the value
     */
    public Bean(@DefaultValue("hello") @QueryParameter("q") String v) {
        this.value = v;
    }

    /**
     * A regular setter used to inject a second parameter.
     *
     * @param v the value
     */
    public void setValue2(@QueryParameter("q2") String v) {
        this.value2 = v;
    }

    public String getValue() {
        return value;
    }

    public String getValue2() {
        return value2;
    }
}
```

An instance of this bean is going to be created when the following action method is invoked:

```
/**
 * Shows how to use the {@link org.wisdom.api.annotations.BeanParameter} annotation.
 */
@Controller
public class BeanExample extends DefaultController {

    /**
     * The {@link org.wisdom.api.annotations.BeanParameter} is used to inject
     * the parameter. It can be combined with the {@link javax.validation.Valid}
     * annotation to ensure the validity of the resulting object.
     *
     * @param bean the instantiated bean
     * @return the json form of the bean
     */
    @Route(method = HttpMethod.GET, uri = "/bean")
    public Result get(@Valid @BeanParameter Bean bean) {
        return ok(bean).json();
    }
}
```

The `@BeanParameter` also handles nested beans.

5.7. Validation

The action method parameter can be validated using the Bean Validation Annotations. It lets you express constraints on object models via annotations.

Here are some examples:

```
package snippets.controllers.validation;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Body;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Parameter;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Controller
public class ValidationController extends DefaultController {

    @Route(method = HttpMethod.GET, uri = "/validation")
    public Result actionWithValidatedParams(
            @NotNull(message = "id must be defined") @Parameter("id") String id,
            @Size(min = 4, max = 8) @Parameter("name") String name) {
        return ok();
    }

    @Route(method = HttpMethod.POST, uri = "/validation/bean")
    public Result actionWithValidatedBody(
            @Valid @Body User user
    ) {
        return ok();
    }

    public static class User {
        @NotNull
        @Size(min = 4, max = 8)
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
    }
}
```

Thanks to these annotation, you don't have to validate your parameter manually. If the constraints a violated, Wisdom returns a `bad request` JSON answer:

```
[{"message":"id must be defined","path":"actionWithValidatedParams.arg0"}]
```

or

```
[{
"message": "size must be between 4 and 8"
"invalid": "xx"
"path": "actionWithValidatedBody.arg0.name"
}]
```

5.7.1. Using Hibernate Validation constraints

Wisdom uses the Hibernate Validation implementation defining more constraints (such as `@Email`). You can rely on these constraints by adding the following dependency in your `pom.xml` file:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.3.Final</version>
    <scope>provided</scope>
</dependency>
```

5.7.2. Manual validation of forms

If you want to run the validation yourself and customize the response, you can decide to run the *validation* process yourself:

```
package snippets.controllers.validation;

import org.apache.felix.ipojo.annotations.Requires;
import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Body;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

import javax.validation.ConstraintViolation;
import javax.validation.Validator;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.util.Set;

@Controller
public class ManualValidationController extends DefaultController {

    @Requires
    Validator validator;

    @Route(method = HttpMethod.POST, uri = "/validation/bean/manual")
    public Result actionWithValidatedBody(
            @Body User user
    ) {
        final Set<ConstraintViolation<User>> violations = validator.validate(user);
        if (violations.isEmpty()) {
            return ok();
        } else {
            return badRequest("Oh no, this is not right: "
                    + violations.iterator().next().getMessage());
        }
    }

    public static class User {
        @NotNull
        @Size(min = 4, max = 8)
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
```

```
    }
}
```

5.7.3. Internationalized validation message

Validation constraints have a `message` parameter that lets you specify the message. Instead of a message, you can use a *message key* that will be picked among the available locales. Message keys are wrapped into `{}` as in:

```
    @Route(method = HttpMethod.GET, uri = "/validation/i18n")
    public Result actionWithValidatedParams(
            @NotNull(message = "{id_not_set}") @Parameter("id") String id,
            @Size(min = 4, max = 8, message = "{name_not_fit}") @Parameter("name") String
name) {
        return ok();
    }
```

Then, creates the *resource bundle* files in `src/main/resources/i18n` such as:

constraints.properties

```
id_not_set: The id must be set
name_not_fit: The name '${validatedValue}' must contain between {min} and {max}
characters
```

for the default (english) languages, and files containing the translations as:

constraints_fr.properties

```
id_not_set: L'identifiant n'est pas sp  cifi
name_not_fit: Le nom '${validatedValue}' doit contenir entre {min} et {max} caract  res
```

So, Wisdom is picking the right message according to the request languages (passed into the `ACCEPT_LANGUAGE` header). As you can see, messages can use variables to make the error more informative. For instance, in English the message is *The name 'ts' must contain between 4 and 8 characters*, while on a browser using the French language, the message is *Le nom 'ts' doit contenir entre 4 et 8 caractères*. If the asked language is not available, default (english) is used.

These files are part of the internationalization support. Check Internationalize your application for more details.

5.8. Declaring routes without @Route

If for some reasons, the `@Route` annotation does not meet your requirements, you can override the

`routes()` method. This method lets you define a set of routes. These routes are **not** prefixed by the `@Path` prefix, and can target other controllers.

```
@Override
public List<Route> routes() {
    return ImmutableList.of(
            new RouteBuilder().route(HttpMethod.GET).on("/routes").to(this, "action"),
            new RouteBuilder().route(HttpMethod.POST).on("/routes").to(this, "action")
    );
}
```

Wisdom provides a `RouteBuilder` that lets you define `Route` easily.

5.9. Route Conflict

Because there is no central location where routes are defined (as it breaks modularity), conflict may happen. Wisdom tries to detect them when the application is deployed, however, the detection algorithm is limited. We recommend using different prefixes (one per application) to avoid conflicts.

5.10. Accessing the router

The Wisdom router is exposed as an OSGi service. Don't worry it does not bite. Your controller can access it as follows:

```
    @Requires
    private Router router;

    public void doSomethingWithTheRouter() {
        org.wisdom.api.router.Route route = router.getRouteFor(HttpMethod.GET, "/");
    }
```

The router lets you find routes by yourself, invoke them, check their parameters... In addition, it supports *reverse routing*.

IMPORTANT	Wisdom is based on Apache Felix iPOJO. All iPOJO features are available in Wisdom.

5.11. Reverse routing

The router can be used to generate a URL from within your code. Thus, refactoring is made easier.

```
    @Route(method=HttpMethod.GET, uri="/reverse")
    public Result reverse() {
        // Get the url of another action method (without parameters)
        String all = router.getReverseRouteFor(PhotoController.class, "all");
        // Get the url of another action method, with id = 1
        String get = router.getReverseRouteFor(PhotoController.class, "get", "id", 1);

        return ok(all + "\n" + get);
    }
```

Invoking the defined action returns:

```
/photo/
/photo/1
```

Reverse routing can let you generate redirect results on actions you don't know the url:

```
    @Route(method=HttpMethod.GET, uri = "/reverse/redirect")
    public Result redirectToHello() {
        String url = router.getReverseRouteFor(Name.class, "index", "name", "wisdom");
        return redirect(url);
    }
```

6. Manipulate HTTP Response

6.1. Changing the default Content-Type

The result content type is automatically inferred from the value you specify in the body of the produced result.

For example, in:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/text")
public Result text() {
    return ok("Hello World!");
}
```

Wisdom automatically sets the `Content-Type` header to `text/plain`, while in:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/json")
public Result jsonResult() {
    ObjectNode node = new ObjectMapper().createObjectNode();
    node.put("hello", "world");
    return ok(node);
}
```

it sets the `Content-Type` header to `application/json`.

This is pretty useful, but sometimes you want to change it. Just use the `as(newContentType)` method on a result to create a new similar result with a different `Content-Type` header:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/html2")
public Result htmlResult2() {
    return ok("<h1>Hello World!</h1>").as(MimeTypes.HTML);
}
```

You can also use the methods `html()` and `json()` to set the content type respectively to HTML and JSON.

6.2. Setting HTTP response headers

You can add (or update) any HTTP response header:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/headers")
public Result headers() {
    return ok("<h1>Hello World!</h1>")
            .html()
            .with(CACHE_CONTROL, "max-age=3600")
            .with(ETAG, "xxx");
}
```

The previous action returns a result with the header set to the expected values:

```
Content-Length: 21
Connection: keep-alive
Etag: xxx
Cache-Control: max-age=3600
Content-Type: text/html; UTF-8
```

IMPORTANT Setting an HTTP header will automatically discard any previous values.

6.3. Setting and discarding cookies

Cookies are just a special form of HTTP headers, but Wisdom provides a set method to manipulate them easily. You can easily add a Cookie to the HTTP response:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/cookies")
public Result cookies() {
    return ok("<h1>Hello World!</h1>")
            .html()
            .with(Cookie.cookie("theme", "github").build());
}
```

NOTE You can configure all aspects of the built cookie such as the max age, domain, and path.

Also, to discard a Cookie previously stored on the Web browser:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/remove-cookies")
public Result withoutCookies() {
    return ok("<h1>Hello World!</h1>")
            .html()
            .discard("theme");
}
```

TIP Two methods are available to remove *things* from an existing `Result`. `without(String)` removes a header while `discard(String)` removes a cookie. However, if `without(String)` does not find a matching header, it will try to remove a cookie.

6.4. Specifying the character encoding for text results

For a text-based HTTP response it is very important to handle the character encoding correctly. Wisdom handles that for you and uses UTF-8 by default. The encoding is used to both convert the text response to the corresponding bytes to send over the network socket, and to add the proper `;charset=xxx` extension to the `Content-Type` header.

The encoding can be specified when you are generating the `Result` value:

```
@Route(method = HttpMethod.GET, uri = "/manipulate/charset")
public Result charset() {
    return ok("<h1>Hello World!</h1>")
            .html()
            .with(Charset.forName("iso-8859-1"));
}
```

6.5. Response encoding

Wisdom can automatically encode response content according to the client `Accept_Encoding` header . The default behavior is to automatically encode all traffic which has a size between 1KB and 10MB. Encoders currently available are `gzip` and `deflate`

6.5.1. Activate / Deactivate encoding

You can disable encoding globally by setting the key `encoding.global` to `false` in the configuration file.

Wisdom also provides two annotations allowing you to finely tune encoding behaviors.

- `@DenyEncoding` can be used to disable encoding on `Controllers` and `Routes` ;
- `@AllowEncoding` can be used to enable encoding on `Controllers` and `Routes`.

In case of multiple instructions, priority is as follow : `Route` > `Method` > `Configuration`.

6.5.2. Control encoding activation size

By default, Wisdom encodes content whose size is between 1KB and 10MB. If you want a different behavior, you can tune theses values with :

- `encoding.max.size` and `encoding.min.size` configuration keys (long value in bytes)
- `@AllowEncoding maxSize` and `minSize` parameters (long value in bytes)

6.5.3. MimeType filtering

By default already encoded mime types are not encoded by Wisdom. This includes `audio/`, `video/`, `image/` MimeType groups as well as all common compressed MimeType (`7z`, `rar`, ...)

6.5.4. URL encoding

Content whose size can't be found by Wisdom is not encoded, for performance reasons. Thus `Stream` contents are not encoded.

We have an exception for `URL`, as we are able to determine the mime type. By default, URL content is encoded but you can deactivate this using the global configuration key `encoding.url`. Setting this key to `false` disables `URL` content encoding (this include `assets` and `resources`).

If you want to deactivate URL content encoding on a particular `Route` or `Controller`, you can still use `@DenyEncoding` to not encode URL returned by theses routes.

7. Content negotiation

Content negotiation is a mechanism that makes it possible to serve different representation of a same resource (URI). It is useful e.g. for writing Web Services supporting several output formats (XML, JSON, etc.). Server-driven negotiation is essentially performed using the `Accept*` requests headers. You can find more information on content negotiation in the HTTP specification.

7.1. Language

You can get the list of acceptable languages for a request using the `request().languages()` method that retrieves them from the `Accept-Language` header and sorts them according to their quality value. The return array of locale is sorted from the most 'wanted' language to the less 'wanted' one.

7.2. Content

Similarly, the `request().mediaTypes()` method gives the list of acceptable result's MIME types for a request. It retrieves them from the `Accept` request header and sorts them according to their quality factor.

The *most* preferred type can be retrieved using `request().mediaType()`. In addition, you can test if a given MIME type is acceptable for the current request using the `request().accepts()` method:

```
public Result list() {
  if (request().accepts("text/html")) {
    return ok(result).html();
  } else {
    return ok(result).json();
  }
}
```

If you don't specify the content type of your result, Wisdom tries to serialize your result to an `ACCEPT`-ed content type. For instance, in the following example, the result form depends on the `ACCEPT` header value of the request. If it's `application/json`, `result` is serialized to JSON, if it's `application/xml`, then the XML serializer is applied.

```
public Result list() {
  return ok(result);
}
```

7.3. Using Negotiation methods

Negotiation can become very complex if you have more than 2 possibilities. Fortunately, Wisdom

provides helper methods to ease the development of action methods producing several results. The following example shows how to produce different types of results depending on the `Accept` HTTP header from the request.

```
@Route(method = HttpMethod.GET, uri = "/negotiation/accept")
public Result negotiation() {
    return Negotiation.accept(
            ImmutableMap.of(
                    MimeTypes.JSON, ok("{\"message\":\"hello\"}").json(),
                    MimeTypes.HTML, ok("<h1>Hello</h1>").html()
            )
    );
}
```

Notice that the `accept` method generates a `406` response if the are no suitable possibilities.

TIP To avoid computing all the results, it's recommended to use `async` results:

```
@Route(method = HttpMethod.GET, uri = "/negotiation/accept")
public Result negotiation() {
    return Negotiation.accept(
            ImmutableMap.of(
                    MimeTypes.JSON, async( () -> ok("{\"message\":\"hello\"}").json(); ),
                    MimeTypes.HTML, async( () -> ok("<h1>Hello</h1>").html(); )
            )
    );
}
```

8. Using Templates

8.1. Template as a Service

Just to makes you understand how the template system works, you should read these small paragraphs.

Wisdom allows the integration of any template engine. By default, Thymeleaf is provided. Each template is exposed as a *service* and can be injected into a controller using the `@View` annotation. This pattern is used regardless of the template engine you use.

The `View` annotation is looking for a template with the specified name. It is equivalent to a regular `@Requires` with a filter.

8.2. The Thymeleaf Template Engine

Thymeleaf is an XML / XHTML / HTML5 template engine (extensible to other formats) that can work both in web and non-web environments. It is really well suited for serving XHTML/HTML5 at the view layer of web applications, but it can also process any XML file even in offline environments.

The main goal of Thymeleaf is to provide an elegant and well-formed way of creating templates. It allows you to create powerful natural templates, that can be correctly displayed by browsers and therefore work also as static prototypes. One of the main interests of Thymeleaf is a smooth collaboration with a Web Designer. By using specific attributes on HTML elements, you never interfere with the work of the web designers. They create the HTML page with whatever structure, and CSS tricks, and you just identify the parts where you inject the data.

It looks like this. As you can see, it's a pretty straightforward syntax supporting all the features you expect from a template engine: variables, internationalization, iterations, layouts, fragments, object navigation, and much more.

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price,1,2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

The complete documentation of Thymeleaf is available here.

8.2.1. My First Template

Thymeleaf templates are HTML files with the `.thl.html` extension, for example: `mytemplate.thl.html`. The "mytemplate" is referred as the template name. Templates can be placed in:

- `src/main/resources/templates` and are embedded in the application's Jar.
- `src/main/templates` and are in the distribution but not in the application's Jar file.

Whether or not your application is intended to be reused in several 'bigger' application determines the place of your template files.

Let's look at the following template, named `welcome.thl.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Follow the Wisdom Path</title>
</head>

<body>
    <h1 th:text="${welcome}">Hello</h1>
</body>

</html>
```

The `th:text` attribute instructs the template engine to replace the content of the element by the `welcome` value. This value is given as parameter to the template engine.

So now in your controller, use the `@View` annotation to inject the template:

```
package snippets.controllers.templates;

import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.annotations.View;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;
import org.wisdom.api.templates.Template;

@Controller
public class WelcomeController extends DefaultController {

    @View("doc/welcome")
    Template template;

    @Route(method = HttpMethod.GET, uri = "/templates/welcome")
    public Result welcome() {
        return ok(render(template, "welcome", "Welcome on the Path to Wisdom"));
    }
}
```

① Use the `@View` annotation and indicate the template name

② The injected field must use the `Template` type

③ Use the `render(template, )` methods to render the template

You can render the template using the `render` methods:

- `render(Template template)` asks the template engine to render the template without input variables
- `render(Template template, Map<String, Object> params)` inserts the given parameters to the template
- `render(Template template, Object params)` proposes an easier way to pass parameters. However, one parameter out of two must be a string (the parameter name), while the following is the value:

```
return ok(render(template, "welcome", "hello", "age", 1, "colors", new String[] {"red",
"blue"}));
```

The `render` method is wrapped within an `ok` method indicating the type of response. Obviously, you can use any other type of result.

Some values are automatically given to the template engine:

- the parameters of the request
- the values stored in the session
- the value stored in the flash scope

8.2.2. Some Thymeleaf constructions

Now we have seen how template can be used, let's see some general construction patterns we use in templates:

Variable:

```
<p th:text="${variable}">this will be replace by the variable</p>
<p th:utext="${variable}">this will be replace by the variable (unescaped so won't remove
the
contained HTML elements)
</p>
```

Internationalized variable:

```
<p th:text="#{variable}">this will be replaced by the variable using the request
locale</p>
```

WARNING	Internationalized values are directly retrieved from the internationalization service from Wisdom.

Iteration:

You can iterate on collections, arrays, maps…

```
<!-- Iterate over a list of products -->
<tr th:each="prod : ${prods}">
  <td th:text="${prod.name}">Onions</td>
  <td th:text="${prod.price}">2.41</td>
  <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
```

Conditionals:

```
<td th:text="${prod.inStock}? 'true' : 'false'">
  true or false according to the prod.isStock value
</td>
```

```
<a href="#" th:if="${not #lists.isEmpty(prod.comments)}">display this link comments if
not empty</a>
```

Static Methods:

```
<p th:utext="${@controllers.template.Utils@add(1, 2)}"></p>
<p th:utext="${@org.ietf.jgss.GSSManager@getInstance().toString()}"></p>
```

TIP	Check the Thymeleaf documentation to find more.
TIP	Thymeleaf uses the OGNL expression language. Refer to the OGNL language guide.

8.2.3. Fragments

Templates are generally used to create layouts. Thymeleaf allows creation of *fragments*, and to include them in the *main* template.

Fragments are an identified part of a template. For example, the following snippet defines two fragments: `my-content` and `sub-content`.

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8"/>
    <title>This is a fragment</title>
</head>
<body>

<div th:fragment="my-content">
    <h1>Hello <span th:text="${username}">my name</span></h1>
</div>

<div th:fragment="sub-content">
    <p>This is the first step to Wisdom, follow us...</p>
</div>

</body>
</html>
```

Fragments can be included either using:

- `th:include`: include the fragment 'under' the element using the directive
- `th:replace`: replace the element using this directive with the fragment content

  The value of theses directive is formed by `name_of_the_template_defining_the_fragments

  fragment_name`. For example, if the previous fragment are defined in `content.thl.html`, including them would look like:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8"/>
    <title>My Main Page</title>
</head>
<body>

<!-- th:replace replaces the element with the fragment content -->
<div th:replace="content :: my-content"></div>

<!-- th:include injects the fragment as content of the element (so an element below the
'div') -->
<div th:include="content :: sub-content"></div>

<footer>
    <!-- something else here -->
</footer>

</body>
</html>
```

TIPS: fragments can use any variable given to the template engine.

8.2.4. Layout

Thymeleaf also supports *layout*. This feature is useful to give to your application a common shape. To use this feature, create a first template (called `master`) that is the *layout*, so contains the basic structure of your pages (headers, footers…):

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="utf-8"/>
    <link rel="shortcut icon" href="/assets/images/owl-small.png"/>

    <script src="/libs/jquery.min.js"></script>
    <script src="/libs/js/bootstrap.min.js"></script>
    <link rel="stylesheet" href="/libs/css/bootstrap.css" media="screen"/>
    <link rel="stylesheet" href="/libs/bootswatch-yeti/css/bootstrap.css"
media="screen"/>

    <title layout:title-pattern="$DECORATOR_TITLE :: $CONTENT_TITLE">Wisdom
Monitor</title>
</head>
<body>
<div th:replace="tiles :: topbar"></div>

<div class="container-fluid">
    <div class="row">
        <div th:replace="tiles :: sidebar"></div>
        <div class="col-sm-9 col-sm-offset-3 col-md-10 col-md-offset-2 main"
layout:fragment="content">
            This is the content.
        </div>
    </div>
</div>
</body>
</html>
```

This template is a regular Thymeleaf template (that can use fragment such as `titles::topbar`), but notice the `layout:fragment="content"` indicating that the template using this layout will provide the actual "content" of this element.

Then, create the templates inheriting from the layout:

```
<html layout:decorator="master">
<head lang="en">

    <title>iPOJO</title>

    <link rel="stylesheet" href="/assets/table.css"/>
    <link href="/assets/dashboard.css" rel="stylesheet"/>
</head>
<body>
<div layout:fragment="content">
<!-- the actual content goes there -->
<h1 class="page-header">iPOJO</h1>

<div class="container">
    <!-- .... -->
</div>
```

In the `html` element, notice the `layout:decorator` indicating the layout template (here it's *master*). Then, the *content* fragment is indicated using `layout:fragment="content"` in the `div` element that will contain the content.

More details about the thymeleaf layout system are available here: https://github.com/ultraq/thymeleaf-layout-dialect.

8.2.5. Using the router from the template

A template can use a special object named `#routes` to retrieve the url of action methods. In the following snippet, the `action` attribute is assigned to the url of the `upload` method of the current controller (the controller having requested the template rendering).

```
th:attr="action=${#routes.route('upload')}"
```

You can also ask for the url of an action method receiving parameters:

```
th:attr="action=${#routes.route('upload', 'param', 'value', 'param2', 'value2')}"
```

Finally, you can target methods from other controllers with:

```
th:attr="action=${#routes
    .route(`org.acme.controller.MyOtherController`, 'upload', 'param', 'value')}"
```

If the route cannot be found, the template rendering fails.

The `routes` object is also able to compute the URL of assets:

```
<link th:href="${#routes.asset('css/bootstrap.min.css')}" rel="stylesheet"/>
<link th:href="${#routes.asset('/css/bootstrap-theme.min.css')}" rel="stylesheet"/>
<script th:src="${#routes.asset('/jquery.js')}"></script>
<script th:src="${#routes.asset('js/bootstrap.min.js')}"></script>
```

It locates the assets in the *assets* directories and in *webjars*. With such a functionality, you don't need to write the complete urls. In addition, if an asset cannot be located, the template rendering fails.

8.2.6. Using the HTTP Context, session, flash and request

Like you can access the Wisdom router from the template, you can also inject the following objects:

- the current HTTP context (using the `http` variable)
- the session
- the flash scope
- the request object

```
<div>
    <ul>
        <li><span>From Session</span> = <span
th:text="${#session.get('value')}">VALUE</span></li>
        <li><span>From Flash</span> = <span
th:text="${#flash.get('value')}">VALUE</span></li>
        <li><span>From Request</span> = <span
th:text="${#request.data().get('value')}">VALUE</span></li>
    </ul>
</div>
```

8.2.7. Extending Thymeleaf

The Thymeleaf Template language can be extended using *dialects*. Wisdom tracks these dialects from the service registry and enhanced the template facilities dynamically. So to extend the template language, just implement your own dialect such as:

```
/**
 * An example of custom dialect.
 */
@Service
public class MyDialect extends AbstractDialect {

    @Override
    public String getPrefix() {
        return "my";
    }

    @Override
    public Set<IProcessor> getProcessors() {
        return ImmutableSet.<IProcessor>of(new SayToAttrProcessor());
    }


    private class SayToAttrProcessor
            extends AbstractTextChildModifierAttrProcessor {


        public SayToAttrProcessor() {
            super("sayto");
        }


        public int getPrecedence() {
            return 10000;
        }

        @Override
        protected String getText(final Arguments arguments, final Element element,
                                 final String attributeName) {
            return "Hello, "  + element.getAttributeValue(attributeName) + "!";
        }
    }
}
```

To go further, check the Thymeleaf extension page.

9. Session and Flash scopes

9.1. How it is different in Wisdom

As in the Play Framework, Wisdom makes a distinction between the session and flash scope. If you have to keep data across multiple HTTP requests, you can save them in the Session or the Flash scope. Data stored in the Session is available during the whole user session, and data stored in the flash scope is only available to the next request.

It's important to understand that Session and Flash data are not stored in the server but are added to each subsequent HTTP Request, using Cookies. This means that the data size is very limited (up to 4 KB) and that you can only store string values.

Cookies are signed with a secret key so the client can't modify the cookie data (or it will be invalidated). The Wisdom session is not intended to be used as a cache. If you need to cache some data related to a specific session, you can use the Wisdom Cache Service and use the session to store a unique ID to associate the cached data with a specific user.

There is no technical timeout for the session, which expires when the user closes the web browser. If you need a functional timeout for a specific application, just store a timestamp into the user Session and use it however your application needs (e.g. for a maximum session duration, maximum inactivity duration, etc.).

9.2. Reading a Session value

You can retrieve the incoming Session from the HTTP request:

```
@Route(method= HttpMethod.GET, uri = "/session")
public Result readSession() {
    String user = session("connected");
    if(user != null) {
        return ok("Hello " + user);
    } else {
        return unauthorized("Oops, you are not connected");
    }
}
```

9.3. Storing data into the Session

As the Session is just a Cookie, it is also just an HTTP header, but Wisdom provides a helper method to store a session value:

```
@Route(method= HttpMethod.GET, uri = "/session/login")
public Result login() {
    String user = session("connected");
    if(user != null) {
        return ok("Already connected");
    } else {
        session("connected", "wisdom");
        return readSession();
    }
}
```

The same way, you can remove any value from the incoming session:

```
@Route(method= HttpMethod.GET, uri = "/session/logout")
public Result logout() {
    session().remove("connected");
    return ok("You have been logged out");
}
```

9.4. Discarding the whole session

If you want to discard the whole session, there is special operation:

```
@Route(method= HttpMethod.GET, uri = "/session/clear")
public Result clear() {
    session().clear();
    return ok("You have been logged out");
}
```

9.5. Flash scope

The `Flash` scope works exactly like the `Session`, but with two differences:

1. data are kept for only one request
2. the Flash cookie is not signed, making it possible for the user to modify it.

IMPORTANT	The flash scope should only be used to transport success/error messages on simple non-Ajax applications. As the data is just kept for the next request and because there are no guarantees to ensure the request order in a complex Web application, the Flash scope is subjected to race conditions.

Here are a few examples using the Flash scope:

```
@Route(method= HttpMethod.GET, uri = "/session/flash")
public Result welcome() {
    String message = flash("success");
    if(message == null) {
        message = "Welcome!";
        flash("success", message);
    }
    return ok(message);
}
```

10. Handling and serving JSON requests

10.1. Handling a JSON request

A JSON request is an HTTP request using a valid JSON payload as request body. Its `Content-Type` header must specify the `text/json` or `application/json` MIME type.

An action can retrieve the *raw* body content, or can ask Wisdom to convert it to an object, such as a JSON Node object in the following example:

```
@Route(method = HttpMethod.POST, uri = "/json/hello")
public Result hello() {
    JsonNode json = context().body(JsonNode.class);
    if (json == null) {
        return badRequest("Expecting Json data");
    } else {
        String name = json.findPath("name").textValue();
        if (name == null) {
            return badRequest("Missing parameter [name]");
        } else {
            return ok("Hello " + name);
        }
    }
}
```

Of course, it's much better to use the `@Body` annotation:

```
@Route(method = HttpMethod.POST, uri = "/json/hello2")
public Result helloWithBody(@NotNull @Body JsonNode json) {
    String name = json.findPath("name").textValue();
    if (name == null) {
        return badRequest("Missing parameter [name]");
    } else {
        return ok("Hello " + name);
    }
}
```

In this example, the body is automatically mapped to a `JsonNode`. The `@NotNull` annotation manages the case where the body is empty. A `bad request` result will be sent back.

A third, even better, way to receive Json content is to use a *validated* bean/structure instead of a `JsonNode`:

```
@Route(method = HttpMethod.POST, uri = "/json/hello3")
public Result helloWithBodyUsingBean(@Valid @Body Person person) {
    return ok("Hello " + person.name);
}
```

The structure contains validation annotation, constraining the received data. For any non compliant content, a bad request will be returned. In addition, this way also accepts requests with other types of content (such as form input).

10.2. Serving a JSON response

In our previous examples we handled requests with a JSON body, but replied with a `text/plain` response. Let's change that to send back a valid JSON HTTP response:

```
@Requires Json json;

@Route(method = HttpMethod.POST, uri = "/json/hello4")
public Result helloReturningJsonNode(@Valid @Body Person person) {
    ObjectNode result = json.newObject();
    result.put("name", person.name);
    result.put("message", "hello " + person.name);
    return ok(result);
}
```

① First, access the Wisdom Json service using the `@Requires` annotation.

② Create a new (Jackson) `ObjectNode` from the retrieves Json service

③

Just pass the created object in the `ok` method

Invoked with the `{"name":"wisdom"}` payload, the action returns:

```
{
  "name" : "wisdom",
  "message" : "hello wisdom"
}
```

However, building your own Json object can be very annoying and cumbersome. Fortunately, Wisdom builds a Json representation from an object:

```
private class Response {
    public final String name;
    public final String message;

    private Response(String name) {
        this.name = name;
        this.message = "hello " + name;
    }
}

@Route(method = HttpMethod.POST, uri = "/json/hello5")
public Result helloReturningJsonObject(@Valid @Body Person person) {
    Response response = new Response(person.name);
    return ok(response).json();
}
```

① Define the structure you want to return

② Pass the created object to the `ok` method and invoke the `json` method

10.3. Extending Json support with your own serializer and deserializer

Wisdom relies on Jackson to handle the JSON requests and responses. Despite the provided default serialization and de-serialization processes, you may need to extend it with your own *mapping*. Wisdom lets you extend Jackson by registering Jackson's modules containing serializers and deserializers.

To achieve this, you have two ways:

1. expose a `com.fasterxml.jackson.databind.module.SimpleModule` service
2. use the `JacksonModuleRepository` service (the pre 0.7 way)

10.3.1. Expose a Module service

The first way is straightforward. You just need to implement a `Service` exposing the `com.fasterxml.jackson.databind.module.SimpleModule` class:

```
package snippets.controllers.json;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.databind.module.SimpleModule;
import org.wisdom.api.annotations.Service;

import java.io.IOException;

/**
 * An example of Module service to customize the JSON serialization and de-serialization
process.
 */
@Service(Module.class)
public class MyJsonModuleService extends SimpleModule {


    public MyJsonModuleService() {
        super("My Json Module");
        addSerializer(Car.class, new JsonSerializer<Car>() {
            @Override
            public void serialize(Car car, JsonGenerator jsonGenerator,
                                  SerializerProvider serializerProvider)
                    throws IOException {
                jsonGenerator.writeStartObject();
                jsonGenerator.writeStringField("custom", "customized value");
                jsonGenerator.writeStringField("color", car.getColor());
                jsonGenerator.writeStringField("brand", car.getBrand());
                jsonGenerator.writeStringField("model", car.getModel());
                jsonGenerator.writeNumberField("power", car.getPower());
                jsonGenerator.writeEndObject();
            }
        });
        addDeserializer(Car.class, new JsonDeserializer<Car>() {
            @Override
            public Car deserialize(
                    JsonParser jsonParser,
                    DeserializationContext deserializationContext)
                    throws IOException {
                ObjectCodec oc = jsonParser.getCodec();
                JsonNode node = oc.readTree(jsonParser);
                String color = node.get("color").asText();
                String brand = node.get("brand").asText();
                String model = node.get("model").asText();
                int power = node.get("power").asInt();
```

```
                return new Car(brand, model, power, color);
            }
        });
    }
}
```

① Use the `@Service` annotation and specify that you want to expose the `Module` class as a service

② If you extends `SimpleModule`, call the `super` constructor with your module name

③ Register the serializer

④ Register your deserializer

Unfortunately, this approach lets you register only one module at a time, that's why the second way give you more flexibility.

10.3.2. Using the Jackson Module Repository service

This second way let you register several *modules* from a single component. Your custom serializers and deserializers must be registered within a `module` and then registered to the `JacksonModuleRepository` service (provided by Wisdom itself):

```
package snippets.controllers.json;

import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.ObjectCodec;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.databind.module.SimpleModule;
import org.apache.felix.ipojo.annotations.*;
import org.wisdom.api.content.JacksonModuleRepository;

import java.io.IOException;

/**
 * An example of Jackson module provider.
 */
@Component
@Instantiate
public class MyJsonModuleProvider {

    @Requires
    JacksonModuleRepository repository;

    private final SimpleModule module;

    public MyJsonModuleProvider() {
        module = new SimpleModule("My Json Module");
        module.addSerializer(Car.class, new JsonSerializer<Car>() {
            @Override
            public void serialize(Car car, JsonGenerator jsonGenerator,
                                  SerializerProvider serializerProvider)
                    throws IOException {
                jsonGenerator.writeStartObject();
                jsonGenerator.writeStringField("custom", "customized value");
                jsonGenerator.writeStringField("color", car.getColor());
                jsonGenerator.writeStringField("brand", car.getBrand());
                jsonGenerator.writeStringField("model", car.getModel());
                jsonGenerator.writeNumberField("power", car.getPower());
                jsonGenerator.writeEndObject();
            }
        });
        module.addDeserializer(Car.class, new JsonDeserializer<Car>() {
            @Override
            public Car deserialize(
                    JsonParser jsonParser,
                    DeserializationContext deserializationContext)
                    throws IOException {
                ObjectCodec oc = jsonParser.getCodec();
```

```
                JsonNode node = oc.readTree(jsonParser);
                String color = node.get("color").asText();
                String brand = node.get("brand").asText();
                String model = node.get("model").asText();
                int power = node.get("power").asInt();
                return new Car(brand, model, power, color);
            }
        });
    }

    @Validate
    public void start() {
        repository.register(module);
    }

    @Invalidate
    public void stop() {
        repository.unregister(module);
    }
}
```

① Retrieve the `MyJsonModuleProvider` service

② In your constructor, build your module with a set of serializers and deserializers.

③ Registers your module into the repository

④ Don't forget to unregister the module from the repository

10.4. JSONP

JSON with Padding (JSONP) is a communication technique to request data from a server in a different domain, something prohibited by typical web browsers because of the same-origin policy. To use JSONP, the server must be able to generate a JSONP response, and you know what ? Wisdom knows how to do that.

Basically a JSONP response is a JavaScript code fragment structure as follows: `padding({the data});`. The padding is the name of the method (callback) that will be called in the JavaScript code having requested this response.

There are several ways to generate a JSONP response:

- using the `ok(padding, json node)` method
- using the Json Service

```
@Requires
Json json;

@Route(method = HttpMethod.GET, uri = "/render")
public Result usingRender(@Parameter("callback") String callback) {
    JsonNode node = json.parse("{ \"foo\": \"bar\" }");
    return ok(callback, node);
}
```

The previous snippet uses the first way to get a JSONP response. It builds a JSON Node and creates a JSONP result object. The mime type of the response is set to `text/javascript`. The previous code generates `callback({"foo":"bar"});`.

JSONP responses can be generated from the JSON service too:

```
@Route(method = HttpMethod.GET, uri = "/json")
public Result usingJsonService(@Parameter("callback") String callback) {
    return ok(json.toJsonP(callback, json.newObject().put("foo",
"bar"))).as(MimeTypes.JAVASCRIPT);
}

@Route(method = HttpMethod.GET, uri = "/user")
public Result user(@Parameter("callback") String callback) {
    return ok(json.toJsonP(callback, new User(1, "wisdom", ImmutableList.of("coffee",
            "whisky")))).as(MimeTypes.JAVASCRIPT);
}
```

The `json.toJSONP` method generates the JSONP response from the given callback name (padding) and the given content. This approach is convenient as you can map business objects into JSON directly (as shown in the second example). However it requires setting the JavaScript mime type explicitly.

11. Handling file uploads

11.1. Uploading files in a form using multipart/form-data

The standard way to upload files in a web application is to use a form with a special `multipart/form-data` encoding, which allows to mix standard form data with file attachments.

Start by writing an HTML form:

```
<!DOCTYPE html>
<html>
<head>
    <title>Wisdom - File Upload</title>
</head>
<body>

<form action="#" enctype="multipart/form-data" method="POST"
th:attr="action=${#routes.route('upload')}">
    <fieldset>
        <input type="file" name="upload" />
        <input type="submit" value="Upload"/>
    </fieldset>
</form>

</body>
</html>
```

Now let's define the upload action:

```
@Route(method = HttpMethod.POST, uri = "/")
public Result upload(@FormParameter("upload") FileItem uploaded) {
    return ok("File " + uploaded.name() + " of type " + uploaded.mimetype() +
            " uploaded (" + uploaded.size() + " bytes)");
}
```

The uploaded file is accessed using the `@Attribute("upload") FileItem uploaded`. `upload` is the name of the field used in the upload form. From the file item, you can:

1. get the file name
2. get the file size and mime type
3. get the content of the file as a byte array (will load the content in memory), or as a stream

IMPORTANT	Wisdom may decide to store the uploaded file to a temporary directory, but this depends on the file size and current load of the server. If you want to store the file on disk, you must copy the file by yourself.

11.2. Direct file upload

Another way to send files to the server is to use Ajax to upload files asynchronously from a form. In this case, the request body will not be encoded as `multipart/form-data`, but will just contain the plain file contents.

```
@Route(method = HttpMethod.POST, uri = "/ajax")
public Result ajax() throws IOException {
    byte[] content = IOUtils.toByteArray(context().reader());
    return ok(content.length + " bytes uploaded");
}
```

The `context().getReader()` method lets you access the raw content of the request. `content().body()` returns the raw content as a String.

12. Handling asynchronous results

12.1. Why asynchronous results?

Until now, we have been able to compute the result to send to the web client directly. This is not always the case: the result may depend on an expensive computation or on a long web service call.

Because of the way Wisdom works, action code must be as fast as possible (i.e. non blocking). So what should we return from our action if we are not yet able to compute the result? We should return an asynchronous result, *i.e.* a result that will be computed in the background and sent to the client when it's done!

An asynchronous result will eventually be redeemed with a value of type `Result`. By using a `AsyncResult` instead of a normal `Result`, we are able to return from our action quickly without blocking anything. Wisdom will then serve the result as soon as the promise is redeemed.

The web client will be blocked while waiting for the response but nothing will be blocked on the server, and server resources can be used to serve other clients.

NOTE:*Why is it important to not block the server?* Wisdom is based on a NIO model and uses very few threads. Blocking one of them drastically reduces the velocity of the server.

12.2. How to create an Asynchronous Result ?

To create an `AsyncResult`, you just need to write the *heavy* computation in a `Callable<Result>` object:

```
@Route(method = HttpMethod.GET, uri = "/async")
public Result async() {
    return new AsyncResult(new Callable<Result>() {
        @Override
        public Result call() throws Exception {
            // heavy computation here...
            return ok("Computation done");
        }
    });
}
```

IMPORTANT The method signature must still return a `Result`

You can also use the `@Async` annotation to make a regular action method as an asynchronous method:

```
@Route(method = HttpMethod.GET, uri = "/async")
@Async
public Result regular() {
    return ok("Computation done");
}
```

TIP The `@Async` annotation let you configure a timeout. If the timeout is reached, and the result is still not computed, an error result is returned to the client.

13. Streaming HTTP responses

13.1. Standard responses and Content-Length header

Since HTTP 1.1, to keep a single connection open to serve several HTTP requests and responses, the server must send the appropriate `Content-Length` HTTP header along with the response.

By default, when you send a simple result, such as:

```
public Result hello() {
    return ok("Hello!");
}
```

You are not specifying a `Content-Length` header. Of course, because the content you are sending is well known, Wisdom is able to compute the content size for you and to generate the appropriate header.

Note that for text-based content this is not as simple as it looks, since the `Content-Length` header must be computed according the encoding used to translate characters to bytes.

To be able to compute the `Content-Length` header properly, Wisdom must consume the whole response data and load its content into memory. Obviously it has some drastic limitations.

13.2. Serving files

If it's not a problem to load the whole content into memory for simple content what about a large data set? Let's say we want to send back a large file to the web client.

Wisdom provides methods to ease the manipulation of file:

```
@Route(method = HttpMethod.GET, uri = "/movie")
public Result file() {
    return ok(new File("/tmp/myMovie.mkv"));
}
```

NOTE	Files are sent chunk by chunk avoiding copying the whole file in memory. This means you can serve very large files. See Chunked Transfer Encoding for more details.

Additionally this helper will also compute the `Content-Type` header from the file name.

If you want to force the browser to download the file (and not just display it), use the following version to set the `Content-Disposition` header.

```
@Route(method = HttpMethod.GET, uri = "/pdf")
public Result fileAsAttachment() {
    return ok(new File("/tmp/wisdom.pdf"), true);
}
```

When you passe `true` as second argument, Wisdom sets the `Content-Disposition` header to `Content-Disposition: attachment; filename=wisdom.pdf`.

13.3. Serving URLs

Wisdom also supports serving urls directly. Handling URLs is a bit more complicated as the size in unknown, and so chunk transfer is required. Don't worry Wisdom handles everything for you:

```
@Route(method = HttpMethod.GET, uri = "/perdu")
public Result url() throws MalformedURLException {
    return ok(new URL("http://perdu.com")).html();
}
```

Serving URLs is useful when loading resources packaged with the application:

```
@Route(method = HttpMethod.GET, uri = "/manifest")
public Result manifest() throws MalformedURLException {
    URL manifest = this.getClass()
            .getClassLoader().getResource("/META-INF/MANIFEST.MF");
    return ok(manifest).as(MimeTypes.TEXT);
}
```

As for files, Wisdom tries to deduce the `Content-Type` from the URL. However, as shown, it can be overridden.

NOTE	As the content size in unknown, the web browser is not able to display a proper download progress bar.

13.4. Serving Input Streams

You can also return input streams directly. As for URLs, the content length is unknown and so it will be transferred chunk by chunk. Unlike URLs and Files, Wisdom does not try to determine the `Content-Type` so, you must set it explicitly:

```
@Route(method = HttpMethod.GET, uri = "/manifest-as-stream")
public Result manifestAsStream() throws MalformedURLException {
    InputStream is = this.getClass()
            .getClassLoader().getResourceAsStream("/META-INF/MANIFEST.MF");
    return ok(is).as(MimeTypes.TEXT);
}
```

13.5. Avoiding chunks

If you don't want to transfer the data using chunks, you can use the following variants:

```
ok(new RenderableFile(file, false));
ok(new RenderableURL(url, false));
ok(new RenderableStream(stream, false));
```

IMPORTANT	Avoiding chunks means copying the whole data into memory, this has a major impact on the memory used by your server and may lead to `OutOfMemory` issues.

14. Web Sockets and SockJS

14.1. What are web sockets

WebSocket is a protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C.

When a client (Browser) wants to establish a connection with a server using a web socket, it starts by a handshake. Once done, a tunnel connects the client and the server. So, the server can push data to the client and vice-versa. Such a mechanism paves the way to more reactive web applications, where notifications and up to date data are pushed to the client without having to rely on ajax or long polling.

WebSocket are identified using urls. These urls starts either by `ws://` or `wss://`.

14.2. Receiving data

A controller willing to listen for data sent by clients on a specific web socket has to use the `@onMessage` annotation:

```
@OnMessage("/socket")
public void onMessage(@Body String message) {
    logger().info("Message received : {}", message);
}
```

Every time a client sends data on `ws:///localhost:9000/socket`, the callback is called. Notice the `@Body` annotation parsing the message to the parameter's type (here as String). The `@Body` annotation works the same way as in action methods:

```
@OnMessage("/jsonSocket")
public void onJsonMessage(@Body Data data) {
    logger().info("Data received : name: {}, age: {}", data.name, data.age);
}
```

The web socket URI provided in the `@OnMessage` annotation's parameter can contain a dynamic part as for action methods:

```
@OnMessage("/socket/{name}")
public void messageOnSeveralSockets(@Parameter("name") String name, @Body String message)
{
    logger().info("Message received on {} : {}", name, message);
}
```

The `@Parameter` annotation let you retrieve the dynamic parts.

Finally, you can identify the client sending the data using a special parameter named `client`:

```
@OnMessage("/socket")
public void identifyClient(@Parameter("client") String client, @Body String message) {
    logger().info("Message received from {} : {}", client, message);
}
```

IMPORTANT	Be aware that the `client` identifier changes if the user disconnects and reconnects.

14.3. Send data to a specific client

Now that we can receive data from the client, it would be nice to push data to it.

```
@Requires
Publisher publisher;

@OnMessage("/echo")
public void echo(@Parameter("client") String client, @Body String message) {
    logger().info("Message received : {}", message);
    publisher.send("/echo", client, message.toUpperCase());
}
```

Two important things here:

- The `publisher` is a service (provided by Wisdom) responsible for sending data to web socket clients.
- We use the `send` method pushing data to a specific client

So, the previous snippet would produce such a kind of conversation:

```
client >>> hello >>> server
client <<< HELLO <<< server
```

14.4. Send data to all clients

The previous example sends data specifically to one client. However, data can be broadcast to all clients connected to a specific web socket:

```
@OnMessage("/echo-all")
public void echoAll(@Body String message) {
    logger().info("Message received : {}", message);
    publisher.publish("/echo-all", message.toUpperCase());
}
```

The main difference is the usage of the `publish` method instead of `send`.

14.4.1. Sending Json or binary data

So far, we have only sent String messages. However, you can send or publish binary data too:

```
@Every("1h")
public void binary() {
    byte[] bytes = new byte[5];
    random.nextBytes(bytes);
    logger().info("Message dispatching binary : {}", bytes);
    publisher.publish("/binary", bytes);
}
```

By the way, notice that this method is not an `OnMessage` callback, but a method executed every hour.

You can also send JSON messages directly too:

```
@Requires
Json json;

@Every("1h")
public void sendJson() {
    publisher.publish("/ws/json", json.newObject().put("name", "hello").put("date",
System.currentTimeMillis()));
}
```

14.5. Being notified of client connections and disconnections

In addition to `OnMessage`, there are two other annotations useful to know when clients connect and disconnect from the listened socket:

```
@Opened("/ws/json")
public void opened(@Parameter("client") String client) {
    logger().info("Client connection on web socket {}", client);
}

@Closed("/ws/json")
public void closed(@Parameter("client") String client) {
    logger().info("Client disconnection on web socket {}", client);
}
```

`@Opened` and `@Closed` callbacks can also use URI with dynamic parts too. To retrieve the identifier of the client, just use `@Parameter("client")`.

14.6. SockJS

SockJS is a browser JavaScript library that provides a WebSocket-like object. SockJS gives you a coherent, cross-browser, Javascript API which creates a low latency, full duplex, cross-domain communication channel between the browser and the web server. It lets you emulate web socket like interactions on older browsers. See here to check the compatibility.

Wisdom SockJS integration lets you use the very same development model (same annotations) as for web sockets for SockJS application.

14.6.1. Declaring SockJS prefixes

Unlike WebSockets (and this is the only difference), the path that are handled by SockJS needs to be declared in the `application.conf` file:

```
vertx {
  sockjs {
    prefixes : ["/chat"]
  }
}
```

Wisdom is reading the list of prefixes and create the adequate SockJS handler on the server side.

IMPORTANT As you can see SockJS support is based on Vert.x

14.6.2. Receiving data from a SockJS socket

Once declared, your controllers can receive data sent from SockJS client:

```
@OnMessage("/chat")
public void onMessage(@Body Message message) {
    System.out.println("Receiving message : " + message.message);
}
```

It uses the same annotation as the web sockets.

14.6.3. Notifications on client connections and departures

As for the web sockets, you can use the `@Opened` and `@Closed` annotations to be notified when a client connects or disconnects from a SockJS socket:

```
@Opened("/chat")
public void open(@Parameter("client") String client) {
System.out.println("socket opened on /chat by " + client);
publisher.send("/chat", client, "Welcome on the chat !");
}

@Closed("/chat")
public void close() {
System.out.println("socket closed => /chat");
}
```

As you can see, the `client` parameter is also supported.

14.6.4. Sending data

Sending data to clients connected to SockJS follows the same way as for web socket, and so relies on the `org.wisdom.api.http.websockets.Publisher` service. In the previous example, it sends a message to a specific client using the `send` method, but you can also `publish` to broadcast a message to all connected clients:

```
@OnMessage("/chat")
public void onMessage(@Body Message message) {
    System.out.println("Receiving message : " + message.message);
    publisher.publish("/chat", json.toJson(message.message.toUpperCase()));
}
```

14.6.5. Using a SockJS socket in JavaScript

Now that the server is ready, let's see how the JavaScript client can connect to the SockJS support.

First, you need to import the SockJS JavaScript library. You can either use a CDN or a webjars:

```
<script src="//cdn.jsdelivr.net/sockjs/0.3.4/sockjs.min.js"></script>
<!-- OR using a webjar -->
<script src="/libs/sockjs.min.js"></script>
```

If you choose the WebJar way, don't forget to add the following dependency to you `pom.xml` file:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>sockjs-client</artifactId>
    <version>0.3.4-1</version>
</dependency>
```

Once done, just create and use a SockJS object as follows:

```
var socket = new SockJS(window.location.origin + '/chat');

socket.onopen = function () {
getTextAreaElement().value = "Socket opened!";
};

socket.onmessage = function (event) {
appendTextArea(event.data);
};

socket.onclose = function () {
appendTextArea("Socket closed");
};

function send(event) {
event.preventDefault();
var msg = {};
msg.message = event.target.message.value;
socket.send(JSON.stringify(msg));
}
```

That's all !

14.6.6. Handling clients using SockJS and Web Sockets

As the server side is identical, the same controller can handle clients using Web Sockets and SocketJS. First, don't forget to declare the SockJS prefix in the `application.conf` file. Then, the web socket client uses the `"ws://" + window.location.host + "/chat"` URL while the SockJS client uses the `window.location.origin + '/chat'` URL (`/chat` is the prefix).

15. Action Interception

When Wisdom processes a request, it searches for the route to invoke and, if found, invokes it. However, the invocation can be *intercepted*, letting *filters* and *interceptors* do stuff before and/or after the actual invocation.

15.1. Filters vs. Interceptors

Wisdom proposes two types of interceptors:

- Filters select the request to intercept using urls
- Interceptors are configured by the controllers

Both filters and interceptors are services. Filters are services independent from the running application. They intercept all requests matching a regex (provided by the filter). On the other hand, interceptors are configured by the controllers. The set of interceptors called for an action is specified by the action itself or by the controller using the annotations bound to interceptors. When several interceptors are declared, a chain is computed and is called sequentially:

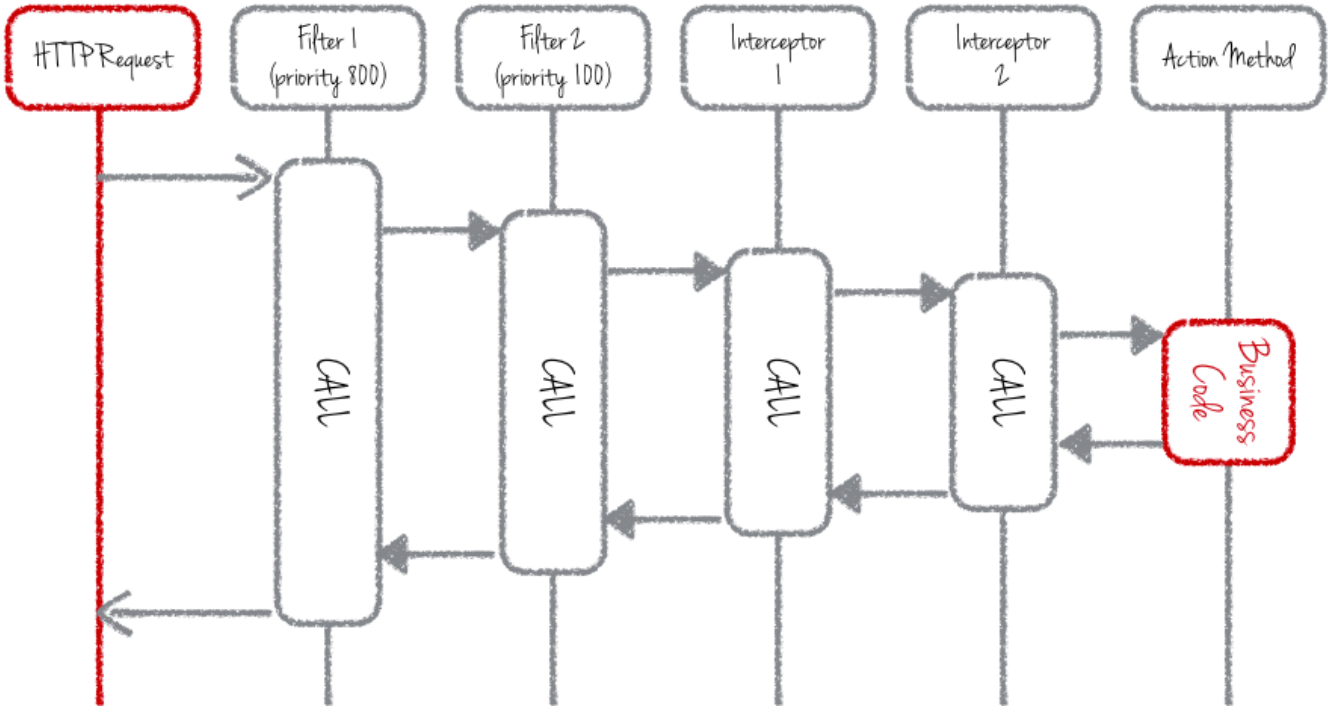

Figure 4. Filter chain

Both filters and interceptors are enqueued in the interception chain. Filters are executed first, and then interceptors. When several filters match the incoming requests, the order is determined using their *priority* (highest first).

Filters can intercept *unbound* requests, *i.e.* requests that do not have a matching action method, or all actions having thrown an exception. Such interceptions are not feasible using interceptors.

15.2. Creating a filters

A filter is a component implementing `org.wisdom.api.interception.Filter` and exposing it as a service. Here is an example of a filter measuring the time spent to compute the response to a request:

```
package snippets.interceptors;

import org.apache.felix.ipojo.annotations.Component;
import org.apache.felix.ipojo.annotations.Instantiate;
import org.apache.felix.ipojo.annotations.Provides;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.wisdom.api.http.Result;
import org.wisdom.api.interception.Filter;
import org.wisdom.api.interception.RequestContext;
import org.wisdom.api.router.Route;

import java.util.regex.Pattern;

/**
 * A filter measuring the time spent to compute the time spent to handle an action
 */
@Component
@Provides
@Instantiate
public class TimeFilter  implements Filter {

    private static final Logger LOGGER =
LoggerFactory.getLogger(TimeFilter.class.getName());

    private static final Pattern REGEX = Pattern.compile("/documentation.*");

    @Override
    public Result call(Route route, RequestContext context) throws Exception {
        final long begin = System.currentTimeMillis();
        try {
            return context.proceed();
        } finally {
            final long end = System.currentTimeMillis();
            LOGGER.info("Time spent to reply to {} {} : {} ms",
                    route.getHttpMethod(), context.context().path(),
                    (end - begin));
        }
    }

    /**
     * Gets the Regex Pattern used to determine whether the route
     * is handled by the filter or not.
     * Notice that the router are caching these patterns and so
     * cannot changed.
     */
    @Override
```

```
    public Pattern uri() {
        return REGEX;
    }

    /**
     * Gets the filter priority, determining the position of the
     * filter in the filter chain. Filter with a high
     * priority are called first. Notice that the router are caching
     * these priorities and so cannot changed.
     * <p/>
     * It is heavily recommended to allow configuring the priority
     * from the Application Configuration.
     *
     * @return the priority
     */
    @Override
    public int priority() {
        return 100;
    }
}
```

The `call` method is the method intercepting the request. It should, in the very high majority of the cases, call the `proceed` method to continue the chain. If it does not, the chain is cut, and no other filters or interceptors will be called.

The `uri` method returns the regex selecting the intercepted urls. In the example, all requests pointing to "/documentation" are intercepted.

The `priority` method allows configuring the position of the filters in the chain if there are several matching filters. Filters with the highest priorities are first. Notice that the default error page filter from Wisdom (displaying the error pages and route not found pages) has a priority of 1000.

Filters can come and go at anytime, promoting a very dynamic model. So, you can intercept requests targeting controllers not developed by you (such as the asset controller), or deploy a temporary filter to understand a specific issue.

15.3. Creating an interceptor's annotation

Developing an interceptor requires an annotation and the interceptor itself. Interceptors are configured using an annotation *bound* to the interceptor. This annotation uses the `@Interception` meta-annotation specifying that an interceptor is consuming the annotation:

```
package snippets.interceptors;

import org.wisdom.api.annotations.Interception;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Just logged requests.
 */
@Interception
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Logged {

    boolean duration() default true;

}
```

The previous snippet shows an interceptor annotation that will be handled by an interceptor. The annotation must be visible at runtime, and can target both types and methods (this means they can be used either on classes or on methods).

NOTE	when used on classes, the interceptor is applied to all contained actions.
IMPORTANT	The package containing the annotation must be exported to be usable by Wisdom components not included in the current project.

15.4. Implementing the interceptor

Interceptors are Wisdom components registered as *OSGi services*. Don't worry, Wisdom makes that easy:

```
@Component
@Provides(specifications = Interceptor.class)
@Instantiate
public class LoggerInterceptor extends Interceptor<Logged> {

    Logger logger = LoggerFactory.getLogger(LoggerInterceptor.class);

    @Override
    public Result call(Logged configuration, RequestContext context)
            throws Exception {
        logger.info("Invoking " + context.context().request().method() +
                " " + context.context().request().uri());
        long begin = System.currentTimeMillis();

        Result result = context.proceed();

        long end = System.currentTimeMillis();
        if (configuration.duration()) {
            logger.info("Result computed in " + (end - begin) + " ms");
        }

        return result;
    }

    @Override
    public Class<Logged> annotation() {
        return Logged.class;
    }
}
```

First, the class is annotated with three annotations:

```
@Component
@Provides
@Instantiate
```

This instructs the framework to expose the OSGi service.

Then, the class contains two main methods:

1. `call` intercepting the action invocation
2. `annotation` returning the class of the handled annotation

The `annotation` method is straightforward. It just returns the class of the handled annotation.

The `call` method is intercepting the action invocation. It invokes the `context.proceed()` method to call the next interceptor. It can also return a `Result` object immediately, shortcutting the chain. The `call` method receives the `RequestContext` object and the actual interceptor configuration. The `RequestContext` let the interceptor to:

- get the current HTTP Context with the `context()` method
- get the current HTTP Request with the `request()` method
- get the current route with the `route()` method
- set data consumed by other interceptor and filters using the `data()` method

15.5. Annotating controllers

As said, interceptor annotation can target either classes or methods:

```
@Controller
@Logged(duration = true)
public class MyController extends DefaultController {
```

```
    @Route(method= HttpMethod.GET, uri = "/intercepted")
    @Logged(duration = true)
    public Result action() {
        //...
        return ok();
    }
```

15.6. Sharing data between filters and interceptor of a chain

Filters and interceptors can share data when they are on the same chain (i.e. intercepting the same request). The `RequestContext` object hold the shared data:

```
context.data().put("my-data", "hello world");
```

This data can be consumed by the filters and interceptors invoked after the entity having set the data.

16. Working with assets

Assets are files delivered by your application or by Wisdom. It can be CSS, JavaScript, HTML files or

resources such as images.

Wisdom supports three types of assets:

1. Assets packaged with the application, they are placed in the `src/main/resources/assets` directory
2. Assets served by wisdom but not included in the application, they are placed in the `src/main/assets` directory, and copied to the `wisdom/assets` directory
3. Assets packages as *webjars*.

The list of all available assets is available on `/assets` (http://localhost:9000/assets). This page is only served in *development* mode.

16.1. Structuring your assets

We strongly recommend that you use the following conventions to organize your assets:

```
assets
     └ javascripts
     └ stylesheets
     └ images
```

16.2. How your assets are packaged and served

The assets from `src/main/resources/assets` are packaged in the application's jar file (don't forget it's an OSGi bundle).The assets from `src/main/assets` are placed in the `wisdom/assets` directory and packaged in the distribution zip file. Despite this difference, both types of assets are accessible from the `/assets/` path.

Let's see some examples:

```
src/main/assets/javascripts/script.js ==> /assets/javascripts/script.js
src/main/assets/stylesheets/style.css ==> /assets/stylesheets/style.css
src/main/resources/assets/javascripts/script.js ==> /assets/javascripts/script.js
src/main/resources/assets/stylesheets/style.css ==> /assets/stylesheets/style.css
```

IMPORTANT	When an asset file is present from both sources, the asset from `src/main/assets` is served. This policy lets you override internal files with external assets.

Wisdom contains a built-in controller to serve the assets. By default, this controller provides caching, ETag, and gzip compression.

16.3. Etag support

The Assets controller automatically manages ETag HTTP Headers. The ETag value is generated from the file's last modification date. (If the resource file is embedded into a file, the JAR file's last modification date is used.)

When a web browser makes a request specifying this Etag, the server can respond with `304 NotModified`, without body. By this method, bandwidth is saved.

You can disable the Etag support by setting `http.useETag` to `false` in the application configuration:

```
http.useETag = false
```

By default, Etag is enabled.

16.4. Cache Control

In addition to the Etag support, Wisdom lets you configure the asset cache time. More precisely the value set by Wisdom in the `Cache-Control` HTTP header. By setting the `http.cache_control_max_age` configuration property in the application configuration, you can set the amount of time (in seconds) the browser can keep the resource in its own cache:

```
http.cache_control_max_age = 3600
```

To disable the cache, set this value to 0:

```
http.cache_control_max_age = 0
```

This configuration instructs Wisdom to use the `no-cache` value, forbidding the browser and other proxies to use caching facilities.

By default, the cache age is set to 3600 seconds.

16.5. Gzip support

IMPORTANT | Not yet implemented

Wisdom lets you serve compressed versions of your assets. For a given resource, if a resource with the same name but using a `.gz` suffix is found, the Assets controller will serve this one by adding the proper HTTP header:

```
Content-Encoding: gzip
```

Of course, Wisdom checks that such encoding is accepted by the browser.

16.6. Asset processing

Before being packaged, assets are *processed*. For example, `CoffeeScript` files are compiled to `JavaScript`, while `Less` files are compiled to `CSS`. This processing is done during the build process (and not at runtime).

16.7. CoffeeScript processing

CoffeeScript is a little language that compiles into JavaScript. Underneath that awkward Java-esque patina, JavaScript has always had a gorgeous heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way. Any `.coffee` files from your assets are compiled to Javascript automatically. The source map is also generated. For example, the file `src/main/assets/javascripts/some-coffee.coffee`:

```
song = ["do", "re", "mi", "fa", "so"]

singers = {Jagger: "Rock", Elvis: "Roll"}

bitlist = [
  1, 0, 1
  0, 0, 1
  1, 1, 0
]

kids =
  brother:
    name: "Max"
    age:  11
  sister:
    name: "Ida"
    age:  9
```

is accessible from `/assets/javascripts/some-coffee.js`:

```
// Generated by CoffeeScript 1.8.0
(function() {
  var bitlist, kids, singers, song;

  song = ["do", "re", "mi", "fa", "so"];

  singers = {
    Jagger: "Rock",
    Elvis: "Roll"
  };

  bitlist = [1, 0, 1, 0, 0, 1, 1, 1, 0];

  kids = {
    brother: {
      name: "Max",
      age: 11
    },
    sister: {
      name: "Ida",
      age: 9
    }
  };

}).call(this);

//# sourceMappingURL=some-coffee.js.map
```

NOTE In *watch* mode, the `.coffee` files are automatically recompiled.

The CoffeeScript compilation relies on the original compiler executed on top of the *node.js* runtime.

16.8. Less processing

LESS extends CSS with dynamic behavior such as variables, mixins, operations and functions. Wisdom automatically compiles any `.less` files from your assets to CSS. For example, the file `src/main/assets/stylesheets/site.less`:

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
  -webkit-box-shadow: @style @c;
  -moz-box-shadow:    @style @c;
  box-shadow:         @style @c;
}
.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha)) {
  .box-shadow(@style, rgba(0, 0, 0, @alpha));
}
.box {
  color: saturate(@base, 5%);
  border-color: lighten(@base, 30%);
  div { .box-shadow(0 0 5px, 30%) }
}
```

is accessible from `/assets/stylesheets/site.css`:

```
.box {
  color: #fe33ac;
  border-color: #fdcdea;
}
.box div {
  -webkit-box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
  -moz-box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
}
```

NOTE In *watch* mode, the `.less` files are automatically recompiled.

The Less version compilation relies on the original less compiler executed on top of the *node.js* runtime.

16.9. Stylesheets Aggregation and Minification

Wisdom integrates Clean-CSS to aggregate and minify stylesheets. By default, it minifies all `.css` file into a `-min.css` file. However this behavior can be configured to aggregate the files and minify the result.

In your `pom.xml`, in the `wisdom-maven-plugin` configuration section, add:

```
<stylesheets>
    <aggregations>
        <aggregation>
            <minification>true</minification> <!-- optional, default to true -->
            <output>my-css.css</output> <!-- optional -->
            <files>
                <file>style.css</file>
                <file>socket/socket.css</file>
            </files>
        </aggregation>
    </aggregations>
</stylesheets>
```

The `stylesheets` element let you configure the aggregation and minification. You can have several `aggregation` element under the `aggregations` element. Each aggregation must list the set of files it aggregates. Only *internal* assets (i.e. from `src/main/resources/assets`) can be aggregated. Each files is looked up from the `target/classes/assets` directory (containing the compiles assets). The extension can be omitted (for example, `style` instead of `style.css`). The aggregation can optionally defined the output file, also related to `target/classes/assets`. By default it builds the name as follows: *artifactid-min.css* (or just *artifactid.css* if minification is disabled).

16.10. JavaScript Aggregation and Minification

Wisdom integrates Google Closure to check, aggregate and minify JavaScript files. For any JavaScript file (even the one generated by the CoffeeScript compiler), a minified version is generated. The minified file name ends with `-min.js`. For example, `my-script.js` will be minified into `my-script-min.js`.

To use the minified version, add the `-min` suffix to your `script` tags in your templates or HTML files.

NOTE Files already minified are not re-minified.

You can configure the optimization level of the Closure Compiler and the *pretty print* option from your `pom.xml` file:

```
<configuration>
    <googleClosureCompilationLevel>ADVANCED_OPTIMIZATIONS</googleClosureCompilationLevel>
    <googleClosurePrettyPrint>true</googleClosurePrettyPrint>
</configuration>
```

The value of the compilation level can be `WHITESPACE_ONLY` (default), `SIMPLE_OPTIMIZATIONS` or `ADVANCED_OPTIMIZATIONS`.

The pretty print option lets you configure how the minified file is formatted . Enabling pretty print impacts the final file size but is much more readable.

Aggregation is configured inside the `<javascript>` element:

```
<javascript>
    <aggregations>
        <aggregation>
            <files>
                <file>coffee/math</file>
                <file>js/log.js</file>
            </files>
            <output>math-min.js</output>
        </aggregation>
    </aggregations>
</javascript>
```

You can have several `aggregation` element under the `aggregations` element. Each aggregation must list the set of files it aggregates. Only *internal* assets (i.e. from `src/main/resources/assets`) can be aggregated. Each files is looked up from the `target/classes/assets` directory (containing the compiles assets). The extension can be omitted (for example, `coffee/math` instead of `coffee/math.js`). The aggregation can optionally defined the output file, also related to `target/classes/assets`. By default it builds the name as follows: *artifactid-min.js*.

You can also disable the Google Closure support with:

```
<configuration>
    <skipGoogleClosure>true</skipGoogleClosure>
</configuration>
```

In *watch mode*, you can disable Google Closure support with: `mvn wisdom:run -DskipGoogleClosure=true`

16.11. WebJar packaging

The internal assets from your project can be packaged into a webjar. This WebJar will contained the process versions of your assets and has a structure conform to the webjar specification. WebJars are really useful to share assets.

WebJar packaging is disabled by default, but can be easily enabled using:

```
<configuration>
    <packageWebJar>true</packageWebJar>
</configuration>
```

By default, all the internal resources of your project are packaged into a webjar. This webjar use the project's artifact id as name, and project's version as version. The created artifact use the `webjar`

classifier.

The webjar can be customized using the `webjar` element in the `pom.xml` file:

```
<webjar>
    <!-- change the webjar name -->
    <name>sample-resources</name>
    <!-- change the webjar version -->
    <version>1.0</version>
    <!-- change the webjar classifier -->
    <classifier>resource</classifier>
    <!-- the selected set of file -->
    <fileset>
        <directory>${project.build.directory}/classes/assets</directory>
        <excludes>
            <exclude>**/js/*</exclude>
        </excludes>
    </fileset>
</webjar>
```

IMPORTANT	The fileset directory must be a full path. By default the asset ouput directory is used.

17. WebJars

WebJars are client-side web libraries (e.g. jQuery & Bootstrap) packaged into JAR (Java Archive) files. It allows to :

- Explicitly and easily manage the client-side dependencies in JVM-based web applications
- Know which client-side dependencies you are using

Wisdom natively supports WebJars: WebJars from your Maven dependencies are detected and copied to the Wisdom Runtime. In addition, Wisdom makes requiring the embedded assets very easy.

Assets included in webjars are listed on the `/assets` page (http://localhost:9000/assets).

17.1. Maven dependencies

Web Jars are plain Jars embedding web assets (such as javascript and css files). The list of WebJars is listed on http://www.webjars.org/.

Search for the library you need and copy the dependency to your project pom file. For example, to use jquery, add:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>2.1.0-2</version>
</dependency>
```

When building your Wisdom project, it unpacks your dependencies to *wisdom/assets/libs*. So you can quickly check what files are available, the different libraries and their versions.

17.2. Importing the embedded assets

Assets embedded in Web Jars are available under "/libs". Let's see some examples:

```
<link rel="stylesheet" href="/libs/css/bootstrap.min.css"/>
<link rel="stylesheet" href="/libs/css/bootstrap-theme.min.css"/>
<script src="/libs/jquery.js"></script>
<script src="/libs/js/bootstrap.min.js"></script>
```

The path to the file depends on how the library is structured.

You can also specify the library name to be sure you get the file from the right library (useful when file names conflict):

```
<link rel="stylesheet" href="/libs/bootstrap/css/bootstrap.min.css"/>
<link rel="stylesheet" href="/libs/bootstrap/css/bootstrap-theme.min.css"/>
<script src="/libs/jquery/jquery.js"></script>
<script src="/libs/bootstrap/js/bootstrap.min.js"></script>
```

Finally, you can also use the library name and version, to be even more specific:

```
<script src="/libs/jquery/2.1.0-2/jquery.js"></script>
```

18. Using the Cache

Caching data is a typical optimization in modern applications, and so Wisdom provides a global cache. An important point about the cache is that it behaves just like a cache should: the data you just stored may just go missing.

For any data stored in the cache, a regeneration strategy needs to be put into place in case the data goes missing. This philosophy is one of the fundamentals behind Wisdom, and is different from Java EE, where the session is expected to retain values throughout its lifetime.

The default cache service uses `EHCache` and it's enabled by default. You can also provide your own implementation via a service.

18.1. Accessing the Cache Service

The cache is available as a service, so accessible using:

```
// Inject the cache service.
@Requires
private Cache cache;

@Route(method = HttpMethod.GET, uri = "/")
public Result action() {
    // Retrieve an object from the cache
    Object cached = cache.get("my.cache.key");
    if (cached == null) {
        cached = new Object();
        // Store an object in the cache for a specific duration (in second)
        cache.set("my.cache.key", cached, 60);
    }
    return ok();
}
```

NOTE The API is intentionally minimal to allow various implementations to be implemented.

Using this simple API you can store data in the cache:

```
cache.set("item.key", news, 60 * 15);
```

You can retrieve the data later:

```
News cached = cache.get("item.key");
```

To remove an item from the cache use the `remove` method:

```
cache.remove("item.key");
```

18.2. Caching HTTP responses

You can easily create an augmented cached action using standard action interception. Wisdom provides a default built-in interceptor for the standard case:

```
@Route(method= HttpMethod.GET, uri = "/cached")
@Cached(key="cached", duration = 60)
public Result cached() {
    String s = DateFormat.getDateTimeInstance().format(new Date());
    return ok(s);
}

@Route(method= HttpMethod.GET, uri = "/cachedForAYear")
@Cached // Use the uri as cache key, and the duration set to 365 days.
public Result cachedWithoutKey() {
    String s = DateFormat.getDateTimeInstance().format(new Date());
    return ok(s);
}
```

NOTE if the duration is not specified, the result is cached for one year.

NOTE if the key is not specified it uses the request's uri (path and query)

19. Scheduling tasks

Wisdom lets you define periodic jobs. To define a job, you need to:

1. implement the `scheduled` interface, and expose it as a service
2. add the `@Every` on a method that will be executed periodically

About the first step, if you are in a controller, you just have to implement the `Scheduled` interface:

```
@Controller
public class ScheduledController extends DefaultController implements Scheduled {

    @Every("30m")
    public void task() {
        System.out.println("Task fired");
    }
}
```

If you are not in a controller, you need to use the `@Provides` annotation to expose the service (or just use the `@Service` annotation).

```
@Component
@Provides
@Instantiate
public class Printer implements Scheduled {

    @Every("1h")
    public void print() {
        System.out.println(new SimpleDateFormat().format(new Date()) + " - Hello ...");
    }

    @Every(period = 1, unit = TimeUnit.DAYS)
    public void cleanup() {
        System.out.println("Cleanup !");
    }
}
```

For each method you want to run periodically, add the `@Every` annotation specifying the period. The period format is very simple:

```
"1s" : every second
"1m" : every minute
"1h" : every hour
"1d" : every day
"2h30m" : every 2 hours and 30 minutes
```

The task is not run immediately but only after the specified period.

TIP As you can see above, you can also use the `period` and `unit` parameters to configure the period.

20. Authentication

Wisdom proposes a very simple security layer letting you secure your endpoints. This model is inspired from the Play 2 Framework security support.

20.1. Implementing a authentication service

The first step is to implement a service checking if the current session is 'authenticated'. Such a class must implement the Authenticator interface.

The following snippet is a basic implementation checking if the property `username` is contained in the session (and if it's 'admin'):

```
@Component
@Provides
@Instantiate
public class MyAuthenticator implements Authenticator {

    @Override
    public String getName() {
        return "my-authenticator";
    }

    @Override
    public String getUserName(Context context) {
        if (context.session().get("username") != null) {
            return context.session().get("username");
        }
        String username = context.parameter("username");
        if (username != null  && username.equals("admin")) {
            context.session().put("username", "admin");
            return "admin";
        } else {
            return null;
        }
    }

    @Override
    public Result onUnauthorized(Context context) {
         return Results.unauthorized("Your are not authenticated !");
    }
}
```

The implementation provides three methods:

- `getUserName` determines the username of the currently authenticated user
- `onUnauthorized` defines the actions to do when the user is not authenticated
- `getName()` returns the identifier of this authenticator service. This identifier is used for selection (see below).

20.2. Annotating your action method or controller

Once you have your authenticator service, you can annotate your action method or controller (this is equivalent to annotating all methods) with the `@Authenticated` annotation:

```
@Controller
public class SecretKeeperController extends DefaultController {

    /**
     * Secured action.
     */
    @Route(method= HttpMethod.GET, uri="/security/secret")
    @Authenticated("Monitor-Authenticator")
    public Result secret() {
        return ok("This is a secret... " + context().request().username());
    }

    /**
     * Not secured action.
     */
    @Route(method= HttpMethod.GET, uri="/security")
    public Result notSecret() {
        String name = context().session().get("username");
        if (name == null) {
            name = "anonymous";
        }
        return ok("Hello " + name);
    }
}
```

Once annotated, access to your method invokes the authenticator service to check if the current context / session / request is 'authenticated'. If not the `onUnauthorized` method is called and its result is used as the response.

20.3. Selecting the right authentication service

You can have several authenticator services at the same time. To select the authenticator to use:

```
@Authenticated("id")
```

The 'id' is the String returned by the `getName()` method. If the specified authenticator is not available, an `unauthorized` response is returned.

21. Internationalize your application

21.1. Externalizing messages

Wisdom lets you externalize your messages by creating *properties* files (containing your messages as explained on this page) in the `src/main/resources/i18n` folder. The name of the file lets you configure the locale/languages of the messages contained in the file. Let's see some examples:

- `src/main/resources/i18n/app.properties` provides messages in the default locale
- `src/main/resources/i18n/app_fr.properties` provides messages in French
- `src/main/resources/i18n/app_en_US.properties` provides messages in American English

The default locale (empty locale) is used when there is no locale matching the request. Generally we use this locale for English messages (but you can configure it using the `application.locale` property in the `application.conf` file).

> **NOTE** Unlike what's said on the Javadoc page, the files containing the messages are read using UTF-8.

You can retrieve messages using the `org.wisdom.api.i18n.InternationalizationService` service:

```
@Requires
private InternationalizationService i18n;

@Route(method= HttpMethod.GET, uri = "internationalization/messages")
public Result retrieveInternationalizedMessages() {
    return ok(
            "english: " + i18n.get(Locale.ENGLISH, "welcome") + "\n" +
            "french: " + i18n.get(Locale.FRENCH, "welcome") + "\n" +
            "default: " + i18n.get(InternationalizationService.DEFAULT_LOCALE, "welcome")
+ "\n"
    ).as(MimeTypes.TEXT);
}
```

Notice that you need to specify the language explicitly.

> **NOTE** The messages do not have to be packaged within the application's bundle, but can be provided in separate bundles, even deployed dynamically.

21.2. Retrieving messages using the request languages

HTTP Requests can specify the expected languages using the `Accept-Language` header. Wisdom parses this header and lets you provide the messages in the most suitable language.

To retrieve the messages using the request accepted languages, just use:

```
@Route(method= HttpMethod.GET, uri = "internationalization/request")
public Result retrieveInternationalizedMessagesFromRequest() {
    return ok(
            i18n.get(request().languages(), "welcome")
    ).as(MimeTypes.TEXT);
}
```

Wisdom determines what's the best language to use for the request. For example, for a request specifying an `Accept-Language` with `en-US,en;q=0.8,de;q=0.6,fr;q=0.4`, Wisdom tries to provide the message in American English, then English, then German, then French. If none of them are provided, the default locale is used.

21.3. Use in templates

Using internationalized messages in a template is pretty simple. It relies on the *Thymeleaf* syntax:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
```

A `utext` instructs Wisdom to use an internationalized message specified using `#{key}`. For example, the following template uses two internationalized messages:

```
<!DOCTYPE html>
<html>
<head>
    <title th:utext="#{app.title}">Title</title>
</head>
<body>

<h1 th:utext="#{app.welcome(${username})}">Welcome Message</h1>

</body>
</html>
```

Notice that the second message is parameterized.

21.4. Formatting messages

Messages can be formatted using the `java.text.MessageFormat` format. For example, if you have defined a message like this:

```
files.summary=The directory {1} contains {0} file(s).
```

You can then specify parameters as:

```
i18n.get(request().languages(), "files.summary", dir.list().length, dir.getName())
```

21.5. Retrieve messages in JavaScript

The internationalization service can be requested using simple HTTP Requests to retrieve the messages. This lets you retrieve the messages from a JavaScript client or anything able to emit a HTTP GET request.

```
GET http://localhost:9000/i18n/welcome  ===> The message with the welcome key (returned
as text)
GET http://localhost:9000/i18n          ===> All messages (returned as JSON)
GET http://localhost:9000/i18n/welcome?locale=en  ===> The message with the welcome key
in the specified language
GET http://localhost:9000/i18n?locales=en  ===> All messages in the specified locales
(returned as JSON)
GET http://localhost:9000/bundles/Messages_fr.properties  ===> Gets a Resource Bundle
(Java Properties) containing
all messages in the specified locale (here fr)
```

Except if specified otherwise, Wisdom computes the language to use according to the request (so according to the `ACCEPT-LANGUAGE` header). When retrieving all messages, the JSON structure is composed as follows:

```
{
  "files.summary":"The directory {1} contains {0} file(s)",
  "welcome":"Welcome"
}
```

21.6. Using the JQuery i18n plugin

To use the JQuery i18n Properties Plugin, just use the last URL form: `http://localhost:9000/bundles/Messages_fr.properties`:

---

jQuery.i18n.properties({ name:'Messages', path:'/i18n/bundles/', mode:'both', language:'fr', callback: function() { // We specified mode: 'both' so translated values will be // available as JS vars/functions and as a map

```
// Accessing a simple value through the map
jQuery.i18n.prop('msg_hello');
// Accessing a value with placeholders through the map
jQuery.i18n.prop('msg_complex', 'John');
```

```
        // Accessing a simple value through a JS variable
        alert(msg_hello +' '+ msg_world);
        // Accessing a value with placeholders through a JS function
        alert(msg_complex('John'));
    }
});
---
```

21.7. Using i18next

Wisdom also provide the support of i18next.

---

var option = { resGetPath: '/i18n/bundles/messages.json?locales=*lng*', dynamicLoad: true }; i18n.init(option, function(t) { // translate nav $(".translation").i18n();

```
var appName = t("app.welcome");
console.log(appName);
```

}); --- == Application configuration

Wisdom configuration system uses the HOCON syntax. This section presents a brief introduction to HOCON and how you application can read its configuration from the `application.conf` file.

the `application.conf` file contains all the configuration data. In your project it's located in the `src/main/configuration` directory. At runtime, this file is copied to the `conf` directory.

21.8. HOCON

HOCON stands for "Human-Optimized Config Object Notation". It's a easy to read, structured syntax that avoid some of the ambiguity of the Java Properties files.

Instead of explaining in details what HOCON is, let's look at some examples:

```
monitor {
  auth {
    enabled = true
    username = admin
    password = admin
  }
  http {
    enabled = true
  }
  jmx {
    enabled = true
  }
  # the update period in second, 10 seconds by default
  # period = 10
}
```

This first example introduces the "structure" syntax using "{}". As you can see, HOCON is very close to JSON. To retrieve a property, you can use `path` such as:

- `monitor.auth.enabled` `true`
- `monitor.auth.username` `admin`

HOCON also support the "Flat" format:

```
documentation.standalone = false
```

However, be aware that it's not compatible with the Java Properties format, even if it's very close. Differences are mostly about quoted / unquoted values:

```
foo {
    value = "this is a quoted String"
}
```

HOCON supports substitution:

```
foo = bar
baz = ${foo} # == bar
```

It also supports *includes*

```
# Include in the root
include "my-include.conf"
sub {
    # Include there
    include "my-sub-conf.conf"
}
```

That's mostly all you need to know about HOCON.

21.9. Retrieving configuration from your application

Your application can retrieve the configuration using the `org.wisdom.api.configuration.ApplicationConfiguration` service. Let's look at a sample.

Let's imagine we have the following configuration in the `application.conf` file:

```
my-application-configuration {
  my-key = "my value"
}
```

To retrieve the "my value" data, just *require* the `ApplicationConfiguration` file and retrieve the data:

```
@Requires
ApplicationConfiguration configuration;

public void readConfiguration() {
    System.out.println(configuration.get("my-application-configuration.my-key"));
    // or
    Configuration conf =
            configuration.getConfiguration("my-application-configuration");
    System.out.println(conf.get("my value"));
}
```

There are other useful methods you can use:

```
String v = configuration.get("key");
v = configuration.getWithDefault("key", "default");
v = configuration.getOrDie("key");

boolean b = configuration.getBoolean("key");
b = configuration.getBooleanWithDefault("key", true);
b = configuration.getBooleanOrDie("key");

int i = configuration.getInteger("key");
i = configuration.getIntegerWithDefault("key", 5);
i = configuration.getIntegerOrDie("key");

long l = configuration.getLong("key");
l = configuration.getLongWithDefault("key", 5l);
l = configuration.getLongOrDie("key");

// The application base directory
File baseDir = configuration.getBaseDir();

// Convert the value to a MyData object, using 'converters'
MyData data = configuration.get("key", MyData.class);
data = configuration.get("key", MyData.class, "data1,data2,data3");
data = configuration.get("key", MyData.class, DEFAULT_DATA);

// Durations
// Durations are converted to the given unit, for instance for:
// key = 1 minute
// key = 2 hours
long duration = configuration.getDuration("key", TimeUnit.SECONDS);
duration = configuration.getDuration("key", TimeUnit.SECONDS, 2);


// Sizes in bytes to avoid the confusion between powers of 1000 and powers of 1024
// For instance for
// key = 1 kB => 1000 bytes
// key = 1 K => 1024 bytes
long size = configuration.getBytes("key");
size = configuration.getBytes("key", 2048);
```

21.9.1. From Properties to HOCON

Historically, Wisdom was using Apache Commons Configuration to handle its configuration. However, we realized it has ambiguities, and we needed a format that handle modularisation. To ease the migration from the former format to hocon, we provide a converter that you run only once. It transforms entry by entry the properties file to the HOCON format:

```
mvn org.wisdom-framework:wisdom-maven-plugin:0.8.0:properties2hocon -Dbackup=false
```

It reads the `src/main/configuration/application.conf` file and transforms it. You can enabled or disable the backup support. This automatic transformation keeps the format close to the initial format to ease the learning. In addition, it may not be perfect, so a review of the output is generally required.

22. Exception Handling

When an action method throws an exception, one of the three following possibilities happens:

- a default *Internal Server Error* is returned to the client
- the exception is an `org.wisdom.api.exceptions.HttpException` or extends `org.wisdom.api.exceptions.HttpException` and so a specific result is sent to the client
- the exception can be handled by a `org.wisdom.api.exceptions.ExceptionMapper` service

22.1. Default Error Handling

By default, when an exception is thrown by a controller, an *internal server error* is returned to the client (status 500). If the request accepts HTML and if the error template is available, the exception is presented as a web page.

22.2. HTTP Exception

`org.wisdom.api.exceptions.HttpException` lets us customize the error returned when such an exception is thrown. When a controller throws an exception of type `org.wisdom.api.exceptions.HttpException` (or a subclass of it), a specific result is built and returned to the client.

```
@Route(method = HttpMethod.GET, uri = "/http_error")
public Result http() {
    throw new HttpException(418, "bad");
}
```

22.3. Exception Mapper

In other cases it may not be appropriate to throw instances of `HttpException`, or classes that extend `HttpException`, and instead it may be preferable to map an existing exception to a result. For such cases it is possible to use a custom exception mapping provider. The provider must implement the `ExceptionMapper<E extends Exception>` interface and exposes it as a service.

```
@Service
public class NoSuchElementExceptionMapper implements
ExceptionMapper<NoSuchElementException> {
    /**
     * Gets the class of the exception instances that are handled by this mapper.
     *
     * @return the class
     */
    @Override
    public Class<NoSuchElementException> getExceptionClass() {
        return NoSuchElementException.class;
    }

    /**
     * Maps the instance of exception to a {@link org.wisdom.api.http.Result}.
     *
     * @param exception the exception
     * @return the HTTP result.
     */
    @Override
    public Result toResult(NoSuchElementException exception) {
        return new Result().status(404).render("nobody there");
    }
}
```

The above class is annotated with `@Service`, so it will be exposed as an OSGi service. When an application throws an `NoSuchElementException` the `toResult` method of the `NoSuchElementExceptionMapper` instance will be invoked.

23. Testing applications

Wisdom provides everything you need to test your application. More specifically, it supports:

- Unit tests - test your classes and action methods
- In-container integration tests - test your controllers and services on a running server
- Blackbox integration tests - emit HTTP request and check the results
- UI tests - check the web page exposed by your application
- JavaScript Testing - check you JavaScript code

To ease the development ot tests, Wisdom provides AssertJ, OSGi Testing Helpers for in-container tests, a HTTP fluent API for blackbox tests, and FluentLenium for UI tests.

JavaScript code relies on Karma and can be executed during the (unit) *test* phase or *integration-test* phase.

IMPORTANT	By default, tests are looking for a free port. So don't worry about configuring the port used by your tests. You can configure the used port by setting it explicitly with the `http.port` system property.

23.1. Unit Tests

The unit test support lets you write Junit tests to check the behavior of your classes. These tests cases are written in Java classes in the `src/test/java` folder. Notice that test case classes must start or end with `Test` (this is a Surefire convention).

Unit tests should extend the `WisdomUnitTest` class to access utility methods.

```
import snippets.controllers.Name;
import snippets.controllers.Simple;
import org.junit.Test;
import org.wisdom.api.http.Result;
import org.wisdom.test.parents.Action;
import org.wisdom.test.parents.Invocation;
import org.wisdom.test.parents.WisdomUnitTest;

import static org.assertj.core.api.Assertions.assertThat;
import static org.wisdom.test.parents.Action.action;

public class UnitTest extends WisdomUnitTest {

    @Test
    public void test() {
        assertThat(1 + 1).isEqualTo(2);
    }

    @Test
    public void testActionWithoutParameter() {
        // This is an instance of my controller
        final Simple simple = new Simple();

        // Call the action method as follows:
        Action.ActionResult result = action(new Invocation() {
            @Override
            public Result invoke() throws Throwable {
                return simple.index();
            }
        }).parameter("name", "clement").header("Accept", "text/html").invoke();

        assertThat(status(result)).isEqualTo(OK);
        assertThat(toString(result)).isEqualTo("Follow the path to Wisdom");
    }

    @Test
    public void testActionWithParameter() {
        // This is an instance of my controller
        final Name name = new Name();

        // Call the action method as follows:
        Action.ActionResult result = action(new Invocation() {
            @Override
            public Result invoke() throws Throwable {
                return name.index("clement");
            }
        }).invoke();
```

```
        assertThat(status(result)).isEqualTo(OK);
        assertThat(toString(result)).isEqualTo("Hi " + "clement" + ", follow us on the
Wisdom path");
    }
}
```

The first test is a really simple test just checking that 1+1=2. Notice the AssertJ syntax.

The second test checks the behavior of one of your action methods. As your action method can access the HTTP method, the call is wrapped into an `Invocation` call:

```
Action.ActionResult result = action(new Invocation() {
    @Override
    public Result invoke() throws Throwable {
        return simple.index();
    }
}).invoke();
```

Such a pattern lets you call your action directly, and retrieve the HTTP result. You can also configure the HTTP headers, parameters, form attributes, cookies, session…. The chain must end with `invoke()`

```
Action.ActionResult result = action(new Invocation() {
    @Override
    public Result invoke() throws Throwable {
        return simple.index();
    }
}).parameter("name", "clement").header("Accept", "text/html").invoke();
```

The `ActionResult` contains the HTTP Context (that may have been modified by your action method, and the HTTP result.

23.2. In-Container Tests

In-container tests let you check the behavior of your controller and services running in the server. The server is automatically launched. These tests cases are written in Java classes in the `src/test/java` folder. Notice that test case classes must start or end with "IT" (this is a Surefire/Failsafe convention).

In-container tests must extend `WisdomTest`, and follow the same rules as unit tests. However, `@Inject` annotated fields are injected.

```
import controllers.Documentation;
import org.junit.Test;
import org.wisdom.api.http.Result;
import org.wisdom.test.parents.Action;
import org.wisdom.test.parents.Invocation;
import org.wisdom.test.parents.WisdomTest;

import javax.inject.Inject;

import static org.assertj.core.api.Assertions.assertThat;
import static org.wisdom.test.parents.Action.action;

public class InContainerIT extends WisdomTest {

    // Inject the controllers, services or template you are testing

    @Inject
    Documentation documentation;

    @Test
    public void testDocumentation() {
        // Call the action method as follows:
        Action.ActionResult result = action(new Invocation() {
            @Override
            public Result invoke() throws Throwable {
                return documentation.doc();
            }
        }).header("Accept", "text/html").invoke();

        // It returns a redirection to the index.html page.
        assertThat(status(result)).isEqualTo(SEE_OTHER);
        assertThat(result.getResult().getHeaders().get(LOCATION)).contains("/index.html")
;
    }

    @Test
    public void test() {
        // Not recommended, but this is also executed
        assertThat(1 + 1).isEqualTo(2);
    }

}
```

The `@Inject` annotation lets you access the real instance of your controller, service or template:

```
// Controller injection:
@Inject
HelloController controller;

// Service injection:
@Inject
private Validator validator;

// The OSGi bundle context
@Inject
private BundleContext context;
```

Once injected your test can invoke them either directly or in an invocation block to configure the HTTP context.

IMPORTANT the same server instance is used for all your tests.

23.3. BlackBox Tests

Black box tests are emitting HTTP requests and retrieving the result. Such tests are not executed in the server. Your application is deployed within the server (it reuses the launched one by in-container tests if it's already running).

As for in-container tests, a black box test class must start or end with "IT". These classes must extend `org.wisdom.test.parents.WisdomBlackBoxTest`.

```
import org.jsoup.nodes.Document;
import org.junit.Test;
import org.wisdom.test.http.HttpResponse;
import org.wisdom.test.parents.WisdomBlackBoxTest;

import static org.assertj.core.api.Assertions.assertThat;

public class BlackBoxIT extends WisdomBlackBoxTest {

    @Test
    public void testDoc() throws Exception {
        HttpResponse<Document> response = get("/documentation").asHtml();
        assertThat(response.code()).isEqualTo(OK);
        assertThat(response.body()
                // JSOUP let us find HTML element very easily
                // and retrieve its content:
                .getElementById("_the_wisdom_framework").text())
                .isEqualTo("1. The Wisdom Framework");
    }

}
```

The test API lets you emit any type of HTTP requests and configure headers, payload, cookies... In addition, the response can be smoothly wrapped to Json Node, HTML Document (JSoup), String, or input stream.

If your blackbox test requires controllers or services from your test sources (i.e. located in the `src/test/java` directory), you need to use `@BeforeClass` and `@AfterClass` as follows:

```
@BeforeClass
public static void init() throws BundleException {
    installTestBundle();
}

@AfterClass
public static void cleanup() throws BundleException {
    removeTestBundle();
}
```

Notice that the methods have to be static. The invoked bundles are located in `org.wisdom.test.parents.WisdomBlackBoxTest`.

23.4. FluentLenium Tests

UI Tests are loading pages using a browser and check the elements present on the page. Wisdom uses FluentLenium to ease the implementation of UI tests.

As for in-container tests, UI test classes must start or end with "IT". These classes must extend `org.wisdom.test.parents.WisdomFluentLeniumTest`.

```
import org.junit.Test;
import org.wisdom.test.parents.WisdomFluentLeniumTest;

import static org.fluentlenium.assertj.FluentLeniumAssertions.assertThat;

public class UIIT extends WisdomFluentLeniumTest {

    @Test
    public void doc() throws Exception {
        goTo("/documentation");
        assertThat(find("#_the_wisdom_framework")).hasText("1. The Wisdom Framework");
    }

}
```

You can select the browser to use by setting the `-Dfluentlenium.browser` property in the command line. Accepted values are `firefox, chrome, ie (Internet Explorer) and safari`. If not set it uses the `HTML Unit Browser` (Fluentlenium's default) relying on Firefox.

Notice that depending on the browser you choose you may have to install additional software or specify addition values. Check the following links:

- Internet Explore Driver.
- Chrome Driver.
- Safari Driver.
- Firefox Driver.

23.5. Stopping the server instance

When building a Maven multi-module project, the same instance of the server is used for all your tests. This can be annoying. Fortunately, Wisdom provides a test listener to stop the instance once the module build is completed:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.18</version>
<executions>
    <execution>
        <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
        </goals>
        <configuration>
            <properties>
                <property>
                    <name>listener</name>
                    <value>org.wisdom.test.WisdomRunListener</value>
                </property>
            </properties>
        </configuration>
    </execution>
</executions>
</plugin>
```

The previous configuration is automatically generated when creating a new project.

23.6. JavaScript tests

To test your JavaScript code, Wisdom integrates Karma, a test runner executing the code in a real or emulated browser . Karma test can be executed either during the *test* phase, or during the *integration-test* phase. In both case you need a Karma configuration file located in the `src/test/javascript` directory, in which your JavaScript tests will also be.

Karma relies on a set of plugins that you must configure from your `pom.xml` file:

```
<karmaPlugins>
    <plugin>karma-jasmine,0.1.5</plugin>
    <plugin>karma-phantomjs-launcher,0.1.1</plugin>
    <plugin>karma-junit-reporter</plugin>
</karmaPlugins>
```

This list indicates the name of the plugin and its version (optional).

23.6.1. Unit Tests

Unit tests are configured in the `src/test/javascript/karma.conf.js` file. Here is an example:

```
module.exports = function(config) {
    config.set({
        basePath: "${basedir}",
        frameworks: ['jasmine'],
        files: [
            '${project.build.outputDirectory}/assets/square.js',
            'src/test/javascript/*Spec.js'
        ],
        exclude: ['src/test/javascript/karma.conf*.js'],
        port: 9876,
        logLevel: config.LOG_INFO,
        browsers: ['PhantomJS'],
        singleRun: true,
        plugins: [
            'karma-jasmine',
            'karma-phantomjs-launcher',
            'karma-junit-reporter'
        ],
        reporters:['progress', 'junit'],
        junitReporter: {
            outputFile: 'target/surefire-reports/karma-test-results.xml',
            suite: ''
        }
    });
};
```

You can find more details about the configuration file here.

TIP The configuration file is filtered, meaning you can use Maven properties such as `/Users/clement/Projects/wisdom/wisdom-framework/documentation/documentation`.

WARNING The set of plugins listed in the configuration must have been listed in the `karmaPlugins` parameter in the `pom.xml` file.

WARNING `singleRun` must be set to `true`.

TIP using the `junitReporter` allow retrieving the Karma tests in your continuous integration server such as Jenkins.

Once you have this configuration file, you can write JavaScript unit test. Here is an example using Jasmine:

```
// squareSpec.js

describe('The square function', function(){
    it('should square a number', function(){
        expect(square(3)).toBe(9);
    });
});

describe('true is true', function() {
   it('should be true', function() {

   });

    it('should be true 2', function() {
         expect(true).toBe(true);
    });
});
```

23.6.2. Integration Tests

Unlike unit tests, integrationt ests need to be enabled in your Maven `pom.xml` file:

```
<executions>
  <execution>
    <id>javascript-it</id>
    <goals>
      <goal>test-javascript-it</goal>
    </goals>
    <phase>integration-test</phase>
  </execution>
</executions>
```

The Karma integration tests are configured in the `src/test/javascript/karma.conf-it.js` file. This file is also filtered, and in addition to the Maven properties can rely on:

- `hostname`: the local server
- `httpPort`: the HTTP port
- `httpsPorts`: the HTTPS port

So, the `proxies` configuration can be like:

```
module.exports = function(config) {
    config.set({
        basePath: "${basedir}",
        frameworks: ['jasmine'],
        files: [
            '${project.build.outputDirectory}/assets/square.js',
            'src/test/javascript/*Spec.js'
        ],
        exclude: ['src/test/javascript/karma.conf*.js'],
        port: 9876,
        logLevel: config.LOG_INFO,
        browsers: ['PhantomJS'],
        singleRun: true,
        proxies: {
            '/': 'http://${hostname}:${httpPort}'
        },
        plugins: [
            'karma-jasmine',
            'karma-phantomjs-launcher',
            'karma-junit-reporter'
        ],
        reporters:['progress', 'junit'],
        junitReporter: {
            outputFile: 'target/surefire-reports/karma-IT-results.xml',
            suite: ''
        }
    });
};
```

To run integration tests, Wisdom starts your application and then launch Karma tests. So, you can load pages and executes Ajax calls.

23.6.3. Watch Mode

Karma **unit** tests are executed, in watch mode, every time you change a `.js` file. Errors are reported in the browser.

23.6.4. Skipping tests

You can skip tests using the `-DskipTests` flag in the Maven command line.

24. Deployment

This section describes how to deploy a Wisdom application on a server. Wisdom is particularly easy to deploy as it's almost self-contained. The only requirements are:

1. a Java 7` virtual machine
2. a writable disk space

24.1. Packaging a distribution

To build a distribution of your application just use: `mvn clean package`. A distribution containing the Wisdom runtime, the application dependencies and external assets are created in the `target` folder as a `zip` file. Wisdom application deployment is based on this distribution file.

24.2. Launching & Stopping

First, unzip the distribution file anywhere. Be aware that the zip file does not include a base directory. To launch a Wisdom application in background, just use the `chameleon.sh` script provided at the root of the unzipped location:

```
./chameleon.sh start
```

Stopping is quite similar:

```
./chameleon.sh stop
```

NOTE **Why `chameleon.sh`?** Actually, Wisdom is built on top of the OW2 Chameleon project, packaging a set of OSGi bundles and providing low-level features, such as logging, dynamic deployment…

To avoid the output on the console, use:

```
./chameleon.sh start > /dev/null
```

NOTE The server's process id is written to the `RUNNING_PID` file. To kill a running Wisdom server, it is enough to send a `SIGTERM` to the process to properly shutdown the application.

NOTE the HTTP port can be set by configuring the `JVM_ARGS` system variable before launching the application: `export JVM_ARGS="-Dhttp.port=9005"  && ./chameleon.sh start`

24.3. Accessing logs

By default the logged messages are written in `logs/wisdom.log`.

24.4. Production deployment

24.4.1. Specifying the HTTP server address and port

You can provide both the HTTP port and address from the `JVM_ARGS` system variable. The default is to listen on port `9000` at the `0.0.0.0` address (all addresses).

```
export JVM_ARGS="-Dhttp.port=8080 -Dhttp.address=127.0.0.1"
./chameleon.sh start
```

NOTE Note that these configurations are only provided for the default embedded Vertx server.

You can configure the HTTPS port using the `https.port` property.

You can disable HTTP or HTTPS using the `-1` value. The `0` value enables a free port lookup.

TIP The free port lookup does not rely on the 'port 0 hack', but actually attempts to bind on random ports.

24.5. Specifying additional JVM arguments

You can specify any JVM arguments to the `JVM_ARGS` system variable. Otherwise the default JVM settings will be used:

```
export JVM_ARGS="-Xms128M -Xmx512m -server"
./chameleon.sh start
```

24.6. Specifying alternative configuration files

The default is to load the `conf/application.conf` file. You can specify an alternative configuration file if needed:

```
export JVM_ARGS="-Dapplication.configuration=conf/application-prod.conf"
./chameleon.sh start
```

TIP The application configuration file uses the HOCON syntax. Unlike properties file, it supports *includes*.

24.7. Wisdom application as a system service

This section explains how to create a system service starting and stopping your application when the machine boots and shutsdown.

NOTE	This section explains the configuration for CentOS, however the same idea can be applied for other systems such as Ubuntu

The service script used to manage a Wisdom application is very basic and relies on the `chameleon.sh` script. In addition, most of the configuration is similar to most Wisdom applications. This script can be downloaded from here.

In this script, you must configure some variables:

```
WISDOM_HOME=/home/wisdom/wisdom => The path of the wisdom application
APPLICATION_MODE="PROD" => The execution mode
export JVM_ARGS="-Dapplication.mode=${APPLICATION_MODE}" => Add the other variables at
the end of this line
JAVA_HOME=/usr/java/latest/ => The Java location
USER=wisdom => The user executing the application
```

WARNING	The set user must be able to write in the application directory.

Once edited, copy the script file to `/etc/init.d`. Then add the *execution* flag with `chmod +x /etc/init.d/filename`. Add the script to the boot and shutdown sequence using: `chkconfig --add /etc/init.d/filename`. Finally, enable the configuration with `chkconfig filename on`. So, if your file is named *wisdom*, the sequence of instructions is the following:

```
cp wisdom.sh /etc/init.d/wisdom
chmod +x /etc/init.d/wisdom
chkconfig --add /etc/init.d/wisdom
chkconfig wisdom on
```

Once done, start the service using `/etc/init.d/wisdom start`. The service can be stopped using `/etc/init.d/wisdom stop`.

25. Setting up a front end HTTP server

You can easily deploy your application as a stand-alone server by setting the application HTTP port to 80:

```
export JVM_ARGS="-Dhttp.port=80"
./chameleon.sh start
```

Note that you probably need root permissions to bind a process to this port.

However, if you plan to host several applications in the same server or load balance several instances of your application for scalability or fault tolerance, you can use a front end HTTP server. Note that using a front end HTTP server will rarely give you better performance than using the Wisdom server directly. However, HTTP servers are very good at handling HTTPS, conditional GET requests and static assets, and many services assume a front end HTTP server is part of your architecture.

25.1. Set up with lighttpd

This example shows you how to configure lighttpd as a front end web server. Note that you can do the same with Apache, but if you only need virtual hosting or load balancing, lighttpd is a very good choice and much easier to configure!

The `/etc/lighttpd/lighttpd.conf` file should define things like this:

```
server.modules = (
      "mod_access",
      "mod_proxy",
      "mod_accesslog"
)

$HTTP["host"] =~ "www.myapp.com" {
    proxy.balance = "round-robin" proxy.server = ( "/" =>
        ( ( "host" => "127.0.0.1", "port" => 9000 ) ) )
}

$HTTP["host"] =~ "www.loadbalancedapp.com" {
    proxy.balance = "round-robin" proxy.server = ( "/" => (
          ( "host" => "127.0.0.1", "port" => 9001 ),
          ( "host" => "127.0.0.1", "port" => 9002 ) )
    )
}
```

25.2. Set up with nginx

This example shows you how to configure nginx as a front end web server. Note that you can do the same with Apache, but if you only need virtual hosting or load balancing, nginx is a very good choice and much easier to configure!

The `/etc/nginx/nginx.conf` file should define things like this:

```
http {

  proxy_buffering    off;
  proxy_set_header   X-Real-IP $remote_addr;
  proxy_set_header   X-Scheme $scheme;
  proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header   Host $http_host;
  # proxy_http_version appeared in nginx 1.1.4
  proxy_http_version 1.1;

  upstream my-backend {
    server 127.0.0.1:9000;
  }

  server {
    server_name www.mysite.com mysite.com;
    rewrite ^(.*) https://www.mysite.com$1 permanent;
  }

  server {
    listen               443;
    ssl                  on;
    ssl_certificate      /etc/ssl/certs/my_ssl.crt;
    ssl_certificate_key  /etc/ssl/private/my_ssl.key;
    keepalive_timeout    70;
    server_name www.mysite.com;
    location / {
      proxy_pass  http://my-backend;
    }
  }
}
```

Note Make sure you are using version 1.2 or greater of Nginx otherwise chunked responses won't work properly.

25.3. Set up with Apache

The example below shows a simple set up with Apache `httpd` server running in front of a standard Wisdom configuration.

```
LoadModule proxy_module modules/mod_proxy.so

<VirtualHost *:80>
  ProxyPreserveHost On
  ServerName www.loadbalancedapp.com
  ProxyPass  /excluded !
  ProxyPass / http://127.0.0.1:9000/
  ProxyPassReverse / http://127.0.0.1:9000/
</VirtualHost>
```

IMPORTANT	Apache does not support Websockets, so you may wish to use another front end proxy (such as haproxy or nginx) that does implement this functionality.

25.4. Advanced proxy settings

When using an HTTP frontal server, request addresses are seen as coming from the HTTP server. In a usual set-up, where you both have the Wisdom app and the proxy running on the same machine, the Wisdom app will see the requests coming from `127.0.0.1`.

Proxy servers can add a specific header to the request to tell the proxied application where the request came from. Most web servers will add an `X-Forwarded-For` header with the remote client IP address as first argument. If the proxy server is running on localhost and connecting from `127.0.0.1`, Wisdom trusts its `X-Forwarded-For` header. If you are running a reverse proxy on a different machine, you can set the `trustxforwarded` configuration item to `true` in the application configuration file, like so:

```
trustxforwarded=true
```

However, the host header is untouched, it'll remain issued by the proxy. If you use Apache 2.x, you can add a directive like:

```
ProxyPreserveHost on
```

The `host:` header will be the original host request header issued by the client. By combining theses two techniques, your app will appear to be directly exposed.

If you don't want this Wisdom app to occupy the whole root, add an exclusion directive to the proxy config (for Apache):

```
ProxyPass /excluded !
```

26. HTTP Configuration

HTTP and HTTPS ports can be configured using:

```
http.port = 9000
https.port = 9001
```

Passing `-1` to one of these value disables the support.

You can also configure:

```
http {
    upload {
        disk.threshold = 16384 # The threshold switching from memory to file storage for
file upload
        max = -1 # The max size in bytes. If an uploaded file exceeds this size, a bad
request is immediately returned
    }
}
request.body.max.size = 102400 # the max body size, the rest is not read.
```

27. Configuring HTTPS

Wisdom can be configured to serve HTTPS. To enable this, simply tell Wisdom which port to listen to using the `https.port` system property. For example:

```
export JVM_ARGS="-Dhttps.port=9005"
./chameleon.sh start
```

The `https.port` property can also be written in the application configuration file.

27.1. SSL Certificates

By default, Wisdom generates itself a self signed certificate, however typically this will not be suitable for serving a website in production. Wisdom uses Java key stores to configure SSL certificates and keys.

Signing authorities often provide instructions on how to create a Java keystore (typically with reference to Tomcat configuration). The official Oracle documentation on how to generate keystores using the JDK keytool utility can be found here.

Having created your keystore, the following **system properties** can be used to configure Wisdom to

use it:

- `https.keyStore`: The path to the keystore containing the private key and certificate, if not provided generates a keystore for you
- `https.keyStoreType`: The key store type, defaults to JKS
- `https.keyStorePassword`: The password, defaults to a blank password
- `https.keyStoreAlgorithm`: The key store algorithm, defaults to the platforms default algorithm

27.2. Turning HTTP off

To disable binding on the HTTP port, set the `http.port` system property to be `-1`, eg:

```
export JVM_ARGS="-Dhttp.port=-1 -Dhttps.port=9005 -Dhttps.keyStore=/path/to/keystore
-Dhttps.keyStorePassword=pwd"
./chameleon.sh start
```

The `http.port` property can also be written in the application configuration file, and as a consequence, HTTP can be disabled from the application configuration file.

28. Logging configuration

Wisdom Framework uses SLF4Jas log API, and LogBack as backend. This section explains how to configure logback, but also how to use other API (JUL, commons-logging, log4j...).

28.1. Configuring Logback

Usually you want to use different logging settings when running in test, dev or on production. The best way to configure Logback is to follow the excellent guide at: http://logback.qos.ch/manual/configuration.html.

There are two main ways how you can configure Logback.

28.2. Using logback.xml

By default Logback will look for a file called `logger.xml` in the `conf` directory of your application. Wisdom provides a default logging configuration generated into `src/main/configuration`.

28.3. Using Java system property to specify link to logback.xml

Logback evaluates a Java system property named `logback.configurationFile`. This approach is handy when you launch your application in a service file, or a script:

```
./chameleon.sh -Dlogback.configurationFile=your/conf/logger.xml
```

This allows you to use one logging configuration for all your instances. More about that approach here: http://logback.qos.ch/manual/configuration.html.

28.4. Using JUL (Java Utils Logger)

JUL is the standard logging API provided in Java. You can use JUL logger without requiring any other configuration or additional bundles. JUL loggers are redirected to logback, so messages are written in the logback logs:

```
@Route(method = HttpMethod.GET, uri = "/log/jul")
public Result jul(@Parameter("message") String message) {
    final Logger logger = Logger.getLogger(LogController.class.getName());
    logger.severe(message);
    return ok();
}
```

28.5. Using Commons-Logging

Apache Commons Logging is a Log API provided by Apache (http://commons.apache.org/proper/commons-logging/). To transfer the logged entry from commons-logging to logback, you need to deploy the `jcl-over-slf4j` bundle. Adding the following dependency adds the bundle:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.9</version>
</dependency>
```

Then, calls to commons-logging is redirected to slf4j and backed using logback:

```
@Route(method = HttpMethod.GET, uri = "/log/jcl")
public Result jcl(@Parameter("message") String message) {
    final Log log =
org.apache.commons.logging.LogFactory.getLog(LogController.class.getName());
    log.error(message);
    return ok();
}
```

28.6. Using log4j

Apache Log4J is another Log API from Apache (http://logging.apache.org/log4j). As for Apache Commons Logging, Log4J requires an additionnal bundle:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
    <version>1.7.9</version>
</dependency>
```

Then, calls to commons-logging is redirected to slf4j and backed using logback:

```
@Route(method = HttpMethod.GET, uri = "/log/log4j")
public Result log4j(@Parameter("message") String message) {
    org.apache.log4j.Logger log =
org.apache.log4j.Logger.getLogger(LogController.class.getName());
    log.error(message);
    return ok();
}
```

29. Cryptographic Functions

Wisdom Framework provided a `org.wisdom.api.crypto.Crypto` service offering functions related to cryptography, security and encoding.

The service can be used to:

- encode/decode using Base64
- compute hashes (MD5, SHA-1, SHA-256, SHA-512)
- encrypt / decrypt message using AES

sign message using HMAC SHA1

- generate random and signed token
- encode Strings to Hexadecimal and decode them to array of bytes

When a function need a *key*, by default it uses the `application.secret` key stored in the `application.conf` file. Be sure to never leak this key.

These functions are also used to sign cookies.

29.1. Using the Crypto service

```
package snippets.controllers.crypto;

import org.apache.felix.ipojo.annotations.Requires;
import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.crypto.Crypto;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

@Controller
public class CryptoController extends DefaultController {

    @Requires
    Crypto crypto;


    @Route(method = HttpMethod.GET, uri = "/crypto")
    public Result aes() {
        String message = crypto.encryptAES("this is a secret");
        return ok(message);
    }
}
```

The complete API of the crypto service is available here

29.2. Configuring the Crypto service

The crypto service can be configured from the `application.conf` file:

```
# Crypto service configuration.
crypto {
 default-hash: MD5 # Can be SHA-1, SHA-256 and SHA-512
 aes {
   key-size: 128 # The size of the AES key
   transformation: "AES/CBC/PKCS5Padding" # The AES transformation to apply
   iterations: 20 # Number of iteration
 }
}
```

30. Validation Service

The validation service lets you *validate* user input, such as forms or Ajax request payloads. The validation service reuses the Bean Validation 1.1 API, meaning it offers the Validator as a service.

30.1. Retrieving the validation service and API

As said above, the validator service is based on Bean Validation 1.1. So it's API is the Bean Validation API. The main entry point is the `Validator` service that lets you validate beans or parameters. The Hibernate Validation documentation provides everything you need to know about the usage of the `validator`.

You can retrieve the `Validator` service as follows:

```
package snippets.controllers.validation;

import org.apache.felix.ipojo.annotations.Requires;
import org.wisdom.api.DefaultController;
import org.wisdom.api.annotations.Body;
import org.wisdom.api.annotations.Controller;
import org.wisdom.api.annotations.Route;
import org.wisdom.api.http.HttpMethod;
import org.wisdom.api.http.Result;

import javax.validation.ConstraintViolation;
import javax.validation.Validator;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.util.Set;

@Controller
public class ManualValidationController extends DefaultController {

    @Requires
    Validator validator;

    @Route(method = HttpMethod.POST, uri = "/validation/bean/manual")
    public Result actionWithValidatedBody(
            @Body User user
    ) {
        final Set<ConstraintViolation<User>> violations = validator.validate(user);
        if (violations.isEmpty()) {
            return ok();
        } else {
            return badRequest("Oh no, this is not right: "
                    + violations.iterator().next().getMessage());
        }
    }

    public static class User {
        @NotNull
        @Size(min = 4, max = 8)
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
```

```
    }
}
```

30.2. Available set of constraints

The Bean Validation specification defines a set of constraints defined in the javax.validation.constraints package:

Annotation	Supported data types	Use
@AssertFalse	Boolean, boolean	Checks that the annotated element is false
@AssertTrue	Boolean, boolean	Checks that the annotated element is true
@DecimalMax(value=,inclusive=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value is less than the specified maximum, when inclusive=false. Otherwise whether the value is less than or equal to the specified maximum. The parameter value is the string representation of the max value according to the BigDecimal string representation.
@DecimalMin(value=,inclusive=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value is larger than the specified minimum, when inclusive=false. Otherwise whether the value is larger than or equal to the specified minimum. The parameter value is the string representation of the min value according to the BigDecimal string representation.
@Digits(integer=,fraction=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value is a number having up to integer digits and fraction fractional digits

Annotation	Supported data types	Use
@Future	java.util.Date, java.util.Calendar, java.time.chrono.ChronoLocalDate, java.time.chrono.ChronoLocalDateTime, java.time.chrono.ChronoZonedDateTime, java.time.Instant, java.time.OffsetDateTime, java.time.Year, java.time.YearMonth	Checks whether the annotated date is in the future
@Max(value=)	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value is less than or equal to the specified maximum
@Min(value=)	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value is higher than or equal to the specified minimum
@NotNull	Any type	Checks that the annotated value is not null.
@Null	Any type	Checks that the annotated value is null
@Past	java.util.Date, java.util.Calendar, java.time.chrono.ChronoLocalDate, java.time.chrono.ChronoLocalDateTime, java.time.chrono.ChronoZonedDateTime, java.time.Instant, java.time.OffsetDateTime, java.time.Year, java.time.YearMonth	Checks whether the annotated date is in the past
@Pattern(regex=,flag=)	CharSequence	Checks if the annotated string matches the regular expression regex considering the given flag match
@Size(min=, max=)	CharSequence, Collection, Map and arrays	Checks if the annotated element's size is between min and max (inclusive)

Annotation	Supported data types	Use
@Valid	Any non-primitive type	Performs validation recursively on the associated object. If the object is a collection or an array, the elements are validated recursively. If the object is a map, the value elements are validated recursively.

The default implementation of the validation service uses Hibernate Bean Validation and proposes additional useful constraints. To have accessed to them you need to add the following dependency to your *pom.xml* file:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.3.Final</version>
    <scope>provided</scope>
</dependency>
```

These additional constraints are:

Annotation	Supported data types	Use	@CreditCardNumber(ignoreNonDigitCharacters=)
CharSequence	Checks that the annotated character sequence passes the Luhn checksum test. Note, this validation aims to check for user mistakes, not credit card validity! See also Anatomy of Credit Card Numbers. ignoreNonDigitCharacters allows to ignore non digit characters. The default is false.	@EAN	CharSequence

Annotation	Supported data types	Use	@CreditCardNumber(ignoreNonDigitCharacters=)
Checks that the annotated character sequence is a valid EAN barcode. type determines the type of barcode. The default is EAN-13.	@Email	CharSequence	Checks whether the specified character sequence is a valid email address. The optional parameters regexp and flags allow to specify an additional regular expression (including regular expression flags) which the email must match.
@Length(min=, max=)	CharSequence	Validates that the annotated character sequence is between min and max included	@LuhnCheck(startIndex = , endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)

Annotation	**Supported data types**	**Use**	**@CreditCardNumber(ignoreNonDigitCharacters=)**
CharSequence	Checks that the digits within the annotated character sequence pass the Luhn checksum algorithm (see also Luhn algorithm). startIndex and endIndex allow to only run the algorithm on the specified sub-string. checkDigitIndex allows to use an arbitrary digit within the character sequence as the check digit. If not specified it is assumed that the check digit is part of the specified range. Last but not least, ignoreNonDigitCharacters allows to ignore non digit characters.	@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)	CharSequence

Annotation	Supported data types	Use	@CreditCardNumber(ignoreNonDigitCharacters=)
Checks that the digits within the annotated character sequence pass the generic mod 10 checksum algorithm. multiplier determines the multiplier for odd numbers (defaults to 3), weight the weight for even numbers (defaults to 1). startIndex and endIndex allow to only run the algorithm on the specified sub-string. checkDigitIndex allows to use an arbitrary digit within the character sequence as the check digit. If not specified it is assumed that the check digit is part of the specified range. Last but not least, ignoreNonDigitCharacters allows to ignore non digit characters.	@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=)	CharSequence	Checks that the digits within the annotated character sequence pass the mod 11 checksum algorithm. threshold specifies the threshold for the mod11 multiplier growth; if no value is specified the multiplier will grow indefinitely. treatCheck10As and treatCheck11As specify the check digits to be used when the mod 11 checksum equals 10 or 11, respectively. Default to X and 0, respectively. startIndex, endIndex acheckDigitIndex and ignoreNonDigitCharacters carry the same semantics as in @Mod10Check.
@NotBlank	CharSequence	Checks that the annotated character sequence is not null and the trimmed length is greater than 0. The difference to @NotEmpty is that this constraint can only be applied on strings and that trailing white-spaces are ignored.	@NotEmpty

Annotation	Supported data types	Use	@CreditCardNumber(ignoreNonDigitCharacters=)
CharSequence, Collection, Map and arrays	Checks whether the annotated element is not null nor empty	@Range(min=, max=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types
Checks whether the annotated value lies between (inclusive) the specified minimum and maximum	@SafeHtml(whitelistType= , additionalTags=, additionalTagsWithAttributes=)	CharSequence	Checks whether the annotated value contains potentially malicious fragments such as <script/>. In order to use this constraint, the jsoup library must be part of the class path. With the whitelistType attribute a predefined whitelist type can be chosen which can be refined via additionalTags or additionalTagsWithAttributes. The former allows to add tags without any attributes, whereas the latter allows to specify tags and optionally allowed attributes using the annotation @SafeHtml.Tag.

Annotation	Supported data types	Use	@CreditCardNumber(ignoreNonDigitCharacters=)
@ScriptAssert(lang=, script=, alias=)	Any type	Checks whether the given script can successfully be evaluated against the annotated element. In order to use this constraint, an implementation of the Java Scripting API as defined by JSR 223 ("Scripting for the Java™ Platform") must part of the class path. The expressions to be evaluated can be written in any scripting or expression language, for which a JSR 223 compatible engine can be found in the class path.	@URL(protocol=, host=, port=, regexp=, flags=)

30.3. Extending the set of constraints

If the set of provided constraints is not enough, you can create your own constraints and validators. To implement such custom constraints, follow the Bean Validation guide. However you should:

- put the annotation and its validator in the same package in the same bundle.
- export the package containing the annotation (and so the validator)

30.4. Internationalization

Constraints define a message. If this message is wrapped into {}, it is interpreted as a message key and the value is looked into the internationalization service (Check Internationalize your application for more details) as in:

```
    @Route(method = HttpMethod.GET, uri = "/validation/i18n")
    public Result actionWithValidatedParams(
            @NotNull(message = "{id_not_set}") @Parameter("id") String id,
            @Size(min = 4, max = 8, message = "{name_not_fit}") @Parameter("name") String
name) {
        return ok();
    }
```

Then the message is looked into the *resource bundles*:

src/main/resources/i18n/constraints.properties

```
id_not_set: The id must be set
name_not_fit: The name '${validatedValue}' must contain between {min} and {max}
characters
```

src/main/resources/i18n/constraints_fr.properties

```
id_not_set: L'identifiant n'est pas sp  cifi
name_not_fit: Le nom '${validatedValue}' doit contenir entre {min} et {max} caract  res
```

So, Wisdom is picking the right message according to the request languages (passed into the `ACCEPT_LANGUAGE` header). As you can see, messages can use variables to make the error more informative. For instance, in English the message is *The name 'ts' must contain between 4 and 8 characters*, while on a browser using the French language, the message is *Le nom 'ts' doit contenir entre 4 et 8 caractères*. If the asked language is not available, default (english) is used.

31. Using Wisdom Executors

Wisdom provides a set of services to handle backend and scheduled tasks:

- `org.wisdom.api.concurrent.ManagedExecutorService` acting as a regular thread pool. It's a enhanced version of the Java `ExecutorServices`.
- `org.wisdom.api.concurrent.ManagedScheduledExecutorService` supporting scheduling. It's a enhanced version of the Java `ScheduledExecutorServices`.

Main difference with the pur Java versions are:

- A management API to know the current state of the thread pool (size, max, core, number of completed tasks...)
- Hung task detection

They return enhanced futures supporting `onSuccess` and `onFailure` callbacks

- Support execution context to migrate data from the caller thread to the background thread

31.1. Using the system executor

By default, Wisdom has a system pool you can use to run backend / asynchronous tasks. Be aware that Wisdom is relying on it, so, it you plan to submit lots of jobs, you should create your own executor (see below).

Once injected, using the service is very straightforward:

```
@Component
@Instantiate
public class ExecutorSample {

    @Requires(filter = "(name=" + ManagedExecutorService.SYSTEM + ")")
    ManagedExecutorService service;

    // Equivalent to this if you don't need the enhanced API

    @Requires(filter = "(name=" + ManagedExecutorService.SYSTEM + ")")
    ExecutorService executor;

    public void doSomething() {
        service.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "heavy computation has been done";
            }
        });

        // With handler
        ManagedFutureTask<?> task = service.submit(new Runnable() {
            @Override
            public void run() {
                System.out.println("Happening in background");
            }
        });

        // Cancel it.
        task.cancel(true);
    }

}
```

31.2. Using the system scheduler

As the system executor, the system scheduler is available as a service. Using it is straightforward:

```
@Component
@Instantiate
public class SchedulerSample {

    @Requires(filter = "(name=" + ManagedScheduledExecutorService.SYSTEM + ")")
    ManagedScheduledExecutorService service;

    // Equivalent to this if you don't need the enhanced API

    @Requires(filter = "(name=" + ManagedScheduledExecutorService.SYSTEM + ")")
    ScheduledExecutorService executor;

    public void doSomething() {
        service.schedule(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "will be done tomorrow";
            }
        }, 1, TimeUnit.DAYS);

        // With handler
        ManagedFutureTask<?> task = service.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                System.out.println("Happening in background");
            }
        }, 1, 5, TimeUnit.MINUTES);

        // Cancel it.
        task.cancel(true);
    }

}
```

31.3. Configuring the system executor and scheduler

The configuration of the system executor and scheduler is made from the `application.conf` file:

```
pools {
    executors {
        wisdom-system-executor { # This is a name of the system executor
            threadType: POOLED # Thread type between POOLED and DAEMON
            hungTime : 60s # Hung threshold
            coreSize : 5 # Number of core threads created on start
            maxSize : 25 # Maximum numbers of threads
            keepAlive : 5s # The idle time before disposing an unused thread
            workQueueCapacity : 2147483647 # Size of the work queue
                                           # (2147483647 = Integer.MAX = unbound)
            priority: 5 # the thread priority
        }
    }

    schedulers {
        wisdom-system-scheduler {
            threadType: POOLED # Thread type between POOLED and DAEMON
            hungTime : 60s # Hung threshold
            coreSize : 5 # Number of threads
            priority: 5 # the thread priority
        }
    }
}
```

The values written above are the default values.

31.4. Creating your own executor or scheduler

You can create another executor or scheduler by adding its configuration in the `application.conf` file:

```
pools {
    executors {
       my-executor { # This is a name of the executor, used in the @Requires filter
            threadType: DAEMON # Thread type between POOLED and DAEMON
            hungTime : 1h # Hung threshold
            coreSize : 3 # Number of core threads created on start
            maxSize : 5 # Maximum numbers of threads
            keepAlive : 1h # The idle time before disposing an unused thread
            workQueueCapacity : 5 # Size of the work queue
                                  # (2147483647 = Integer.MAX = unbound)
            priority: 2 # the thread priority
        }
    }

    schedulers {
        my-scheduler {
            threadType: POOLED # Thread type between POOLED and DAEMON
            hungTime : 10s # Hung threshold
            coreSize : 10 # Number of threads
            priority: 8 # the thread priority
        }
    }
}
```

Once configured, you can retrieve them using:

```
@Requires(filter = "(name=my-executor)")
ManagedExecutorService executor;

@Requires(filter = "(name=my-scheduler)")
ManagedScheduledExecutorService scheduler;
```

31.5. Using Managed Task

As said above, once of the main difference with the 'regular' Java `execution service` is the type of `future` returned by the different methods. Wisdom enhanced the regular feature with administration methods as well as callbacks.

31.5.1. Cancelling a task

First, as usually a task can be cancelled using the `cancel(boolean mayInterrupt)` method.

31.5.2. Task callbacks

You can register callbacks on a submitted tasks:

```
@Component
@Instantiate
public class ExecutorCallbackSample {

    @Requires(filter = "(name=" + ManagedExecutorService.SYSTEM + ")")
    ManagedExecutorService service;

    public void doSomething() {
        service.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "heavy computation has been done";
            }
        }).onSuccess(new ManagedFutureTask.SuccessCallback<String>() {
            @Override
            public void onSuccess(ManagedFutureTask<String> future, String result) {
                System.out.println("Task has returned " + result);
            }
        }).onFailure(new ManagedFutureTask.FailureCallback() {
            @Override
            public void onFailure(ManagedFutureTask future, Throwable throwable) {
                System.out.println("Task has thrown an exception");
            }
        }, MoreExecutors.directExecutor());
    }

}
```

When registering these callbacks, you can pass an 'executor' that will invoke the callback method. By default, callbacks are invoked in the **same** executor as the listened task. To use the thread having submitted the task, use `MoreExecutors.directExecutor()` as for the failure callback on the previous example.

31.6. Detecting hung tasks

Each executor and schedulers has a 'hung threshold' allowing to detect hung task. The executor or scheduler do not do anything when a hung task is detected. It's a only for monitoring and administration support.

To retrieve the hung tasks use: `executor.getHungTasks()`. You can also check on the future returned on submission.

31.7. Execution Context

When submitting a task, Wisdom retrieve all `org.wisdom.api.concurrent.ExecutionContextService` services to build an `org.wisdom.api.concurrent.ExecutionContext`. This context is captured during the submission (in the caller threads), and applied before the execution of the task. After its completion, the context is un-applied. This mechanism let you migrate data that are thread-sensitive to the thread that is going to execute a task.

You can implement your own `org.wisdom.api.concurrent.ExecutionContextService` to extend this support.

TIP Execution context are not supported for periodic and scheduled tasks

32. Vertx

Wisdom is based on Vertx. this section explains how you can benefit from the underlying Vertx infrastructure.

32.1. Accessing the Event Bus

The `Vertx Event Bus` is exposed as an OSGi service. To access it, you first need to add the following dependency to your `pom.xml` file:

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-core</artifactId>
    <scope>provided</scope>
</dependency>
```

TIP The version is inherited from the imported `bom`.

Once done, you can access the Event Bus as following:

```
@Requires
EventBus bus; // org.vertx.java.core.eventbus.EventBus
```

32.2. Sending data on the bus

Once you access the event bus you can send and publish data:

```
@Service
public class MyDataSender implements Scheduled {

    private final static Logger LOGGER = LoggerFactory.getLogger(MyDataSender.class);

    @Requires
    EventBus bus;

    Random random = new Random();

    @Every("1s")
    public void generateAndSendData() {
        int value = random.nextInt(40);
        bus.publish("/data", value);
        LOGGER.info("Has published {} on {}", value, "/data");
    }
}
```

32.3. Receiving data from the bus

You can also register a `handler` to receive data transmitted through the bus:

```
@Component
@Instantiate
public class MyDataReceiver {

    private final static Logger LOGGER = LoggerFactory.getLogger(MyDataReceiver.class);

    @Requires
    EventBus bus;

    private Handler<Message> handler;

    @Validate
    public void start() {
        handler = new Handler<Message>() {
            @Override
            public void handle(Message message) {
                LOGGER.info("Has received " + message.body());
            }
        };
        bus.registerHandler("/data", handler);
    }

    @Invalidate
    public void stop() {
        bus.unregisterHandler("/data", handler);
    }
}
```

32.4. Clustering

Vertx support distributed Event Bus, and so the creation of *clustered* configurations. The cluster is managed using Hazelcast, an In-Memory Data Grid is data management software. So, events sent to the event bus can be received by receivers running on a remote machine.

To enable the clustering you first need to configure Vertx. Edit the `src/main/configuration/application.conf` file and add:

```
vertx {
  cluster-host = "localhost" # Address or IP used for the clustering
  cluster-port = 25501 # The port used for the clustering
}
```

WARNING	The couple host-port must be unique. If not, Vertx cannot bind to the address, and the event bus creation fails.

Once done, create the `src/main/configuration/cluster.xml` containing the Hazelcast configuration. Copy and edit this file is a good start.

> **WARNING** Do not forget to edit the interface on which Hazelcast is bound.

When clustering is enabled, you will see a stack trace on startup. Just ignore it:

```
java.lang.IllegalArgumentException: version string 'unknown' is not valid
at
com.hazelcast.management.ManagementCenterIdentifier.getVersionAsInt(ManagementCenterIdent
ifier.java:44)
at
com.hazelcast.management.ManagementCenterIdentifier.<init>(ManagementCenterIdentifier.jav
a:68)
at
com.hazelcast.management.ManagementCenterService.newManagementCenterIdentifier(Management
CenterService.java:201)
...
```

32.5. HTTP Configuration

Vertx is also used as network stack by Wisdom, so you can configure various aspects of HTTP handling:

```
vertx {
    acceptBacklog: 10000 # The accept backlog
    receiveBufferSize: 4096 # The receive buffer size
    sendBufferSize: 4096 # The send buffer size

    maxWebSocketFrameSize: 65536 # The maximum websocket frame size
    websocket-subprotocols: [] # The list of websocket subprotocols that are allowed

    # sockJS configuration
    sockjs {
        prefixes: [] # The list of path that need to be handled by SockJS
        timeout: 5s # The server sends a close event if a client has not been seen for a
while.
        heartbeat: 5s # Heartbeat time to keep long running HTTP connections opened.
        max: 4kb # Max frame size
    }
}
```

32.6. Core Pool Threads

By default, Vertx uses a limited number of threads, the number of processor you have. You can

configure this number by setting the `vertx.pool.eventloop.size` system property, or by setting it in the `application.conf` file:

```
vertx {
    pool.eventloop.size: 2
}
```

33. Integrating with Akka

Akka uses the Actor Model to raise the abstraction level and provide a better platform to build correct concurrent and scalable applications. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

33.1. The application actor system

Akka can work with several containers called `ActorSystems`. An actor system manages the resources it is configured to use in order to run the actors it contains.

Wisdom defines a special actor system to be used by the applications. You can access the default actor system using the `org.wisdom.akka.AkkaSystemService` service:

```
@Requires
AkkaSystemService akka;

public void doSomethingWithAkka() {
    // Retrieve the default actor system
    ActorSystem system = akka.system();
}
```

33.2. Scheduling asynchronous tasks

Once you have retrieved the actor system, you can schedule messages sending to actors and executing tasks (functions or Runnable instances). You will get a `Cancellable` back that you can call cancel on to cancel the execution of the scheduled operation.

For example, to send a message to the `testActor` every 30 minutes:

```
public void schedule() {
    akka.system().scheduler().schedule(
            //Initial delay 0 milliseconds
            Duration.create(0, TimeUnit.MILLISECONDS),
            //Frequency 30 minutes
            Duration.create(30, TimeUnit.MINUTES),
            receiver,
            "tick",
            akka.system().dispatcher(),
            null
    );
}
```

Alternatively, to run a block of code ten milliseconds from now:

```
public void runOnce() {
    akka.system().scheduler().scheduleOnce(
            Duration.create(10, TimeUnit.MILLISECONDS),
            new Runnable() {
                public void run() {
                    // ...
                }
            },
            akka.system().dispatcher()
    );
}
```

34. Extending the Wisdom Build Process

Wisdom lets you extend the build process and react to file updates to improve the development experience. The Wisdom build system is based on Apache Maven, and so, to extend it you just implement a regular Maven Plugin, aka Mojo.

This section does not explain how to develop a regular Maven plugin. Refer to The Maven Mojo Developer Documentation for further details. However, it explains how your Mojo can participate in the *Wisdom Watch Mode*.

34.1. Rationale

We are often asked why we want a *Watch Mode* based on Maven, and not something separated, as most of frameworks do. This is because duplicating the build process creates a strange experience. The result looks the same, but it may not really be the same. Debugging such kind of differences is almost impossible. So in Wisdom, Maven is the reference and the *Watch Mode* is built on top of it. During the

Watch Mode, Wisdom invokes the Maven Mojos directly.

This also means that using your Wisdom Build extension is just like using any other Maven plugin: * add the plugin in the *pom.xml* file * configure it * you're done!

34.2. From Mojo To Watcher

The *Wisdom Watch Mode* is based on the concept of `Watcher`. A `Watcher` is an object who is notified when a file, *accepted* by the watcher, is created, updated or deleted.

So, first, in your Mojo's project, add the dependency on the `Wisdom-Maven-Plugin`:

```
<dependency>
    <groupId>org.wisdom-framework</groupId>
    <artifactId>wisdom-maven-plugin</artifactId>
    <version>0.8.0</version>
</dependency>
```

To make your Mojo participate in the *Wisdom Watch Mode*, it needs to become a `Watcher` and registers itself in the *watch pipeline*. In order to do that, the easiest way is to make your Mojo extend `AbstractWisdomWatcherMojo` instead of `AbstractMojo`.

TIP	The `Pipeline` is the ordered sequence of *Watchers* invoked in the *Watch Mode*. This list contains Mojo instances (that are also *Watcher*). This list is collected during the *regular* execution of your build. So the order is the same as the Maven plugin's order.

By extending `AbstractWisdomWatcherMojo`, it automatically registers your Mojo to the *Pipeline*.

As a *Watcher* your mojo has to implement the following methods:

```
    /**
     * Checks whether the given file is managed by the current watcher. Notice that
implementation must not check
     * for the existence of the file as this method is also called for deleted files.
     *
     * @param file is the file.
     * @return {@literal true} if the watcher is interested in being notified on an event
     attached to the given file,
     * {@literal false} otherwise.
     */
    public boolean accept(File file);

    /**
     * Notifies the watcher that a new file is created.
     *
     * @param file is the file.
     * @return {@literal false} if the pipeline processing must be interrupted for this
event. Most watchers should
     * return {@literal true} to let other watchers be notified.
     * @throws WatchingException if the watcher failed to process the given file.
     */
    public boolean fileCreated(File file) throws WatchingException;

    /**
     * Notifies the watcher that a file has been modified.
     *
     * @param file is the file.
     * @return {@literal false} if the pipeline processing must be interrupted for this
event. Most watchers should
     * returns {@literal true} to let other watchers to be notified.
     * @throws WatchingException if the watcher failed to process the given file.
     */
    public boolean fileUpdated(File file) throws WatchingException;

    /**
     * Notifies the watcher that a file was deleted.
     *
     * @param file the file
     * @return {@literal false} if the pipeline processing must be interrupted for this
event. Most watchers should
     * return {@literal true} to let other watchers be notified.
     * @throws WatchingException if the watcher failed to process the given file.
     */
    public boolean fileDeleted(File file) throws WatchingException;
```

The `accept` method let you select the file the watcher handles. For instance, if you want to handle `.css`

files, your `accept` method will be similar to:

```
@Override
public boolean accept(File file) {
    return
        (WatcherUtils.isInDirectory(file, WatcherUtils.getInternalAssetsSource(basedir))
            || (WatcherUtils.isInDirectory(file,
WatcherUtils.getExternalAssetsSource(basedir)))
        )
        && WatcherUtils.hasExtension(file, "css");
    }
```

It first checks the location of the file. In this example, it checks that the file is either in the internal assets, i .e. `src/main/resources/assets` or in the global assets (`src/main/assets`). Then, it checks for the file's extension. Notice that you must not check for the existence, as the `accept` method is also called for deleted file.

Once the `accept` method is implemented, you need to implement the:

- `fileCreated(File file)` : called when an *accepted* file is created
- `fileUpdated(File file)` : called when an *accepted* file is updated
- `fileDeleted(File file)` : called when an *accepted* file is deleted

In general, the `fileCreated` and `fileUpdated` methods call the Mojo's main method (`execute`), while the `fileDeleted` method cleans up generated files when the input file is deleted.

These methods return a boolean controlling the pipeline execution. If one of the method returns `false`, the pipeline is interrupted, and none of the following watchers are called. In most cases, you need to return `true`.

34.3. Handling errors : the Watching Exceptions

As you may have noticed, the *Watcher*'s methods can throw `WatchingException`. `Watching Exceptions` let you indicate an issue in the processing. A default error page is generated from this exception with the message, of the guilty file, and if specified, the line and position of the error.

34.4. Initialization and Injection

As your Watcher is a Mojo and the *pipeline* is reusing the Mojo's instances, all of the injected parameters from your Mojo are still available. Notice that the `execute` method is called before the Watcher's methods, meaning that you can do the required initialization there if required.

IMPORTANT The pipeline is executed by one thread only (no parallel execution).

34.5. Disabling the *Watch Mode*

Maybe your Mojo is *skipped*. In that case you need to explicitly unregister it from the *pipeline* by calling the `removeFromWatching` method in your `execute` method.

34.6. Generating a Watcher using the Maven Archetype

We provide a Maven Archetype generating the project to develop your Watcher. Executes the following command to generate it:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:generate \
 -DarchetypeArtifactId=wisdom-simple-watcher-archetype \
 -DarchetypeGroupId=org.wisdom-framework \
 -DarchetypeVersion=0.8.0 \
 -DgroupId=YOUR_GROUPID \
 -DartifactId=YOUR_ARTIFACTID \
 -Dversion=1.0-SNAPSHOT
```

34.7. Using Node and NPMs

Most of the Web Tools developed today are available as *NPM* (i.e. Node's module). Fortunately, Wisdom provides a set of utility classes to install and execute NPM.

IMPORTANT	Wisdom has it's own *node* version installed in `~/.wisdom/node/Node_Version`. So, the NPMs you are installing and executing do not conflict with the user's ones. These NPM are installed in `~/.wisdom/node/Node_Version/lib/node_modules`.
IMPORTANT	The Wisdom's NPM support only work for executable NPM's.

To use a *NPM* tool, declare a `NPM` object as follows:

```
NPM myNPM = npm(this, npm_name, version);
```

In most cases, declare your NPM object as a field, as both the *execute* method and watcher's method will invoke it. If the specified NPM is not install yet, it will install it (from NPM).

It is a good practice to declare the NPM's version as a parameter, so the user can specify the version:

```
@Parameter(defaultValue = "0.3.4")
String version;
```

To execute the NPM, use:

```
myNPM.execute("exec", input.getAbsolutePath(), destination.getAbsolutePath() ...);
```

The `"exec"` argument is the name of the *command* to launch. Available commands are available from the `package.json` files of the NPM, in the *bin* part:

```
 "bin": {
     "myth": "bin/myth"
 },
```

The others arguments are the command's arguments. In the previous example the input and output paths.

So, for instance, here are the lines invoking the *myth* processing:

```
try {
    int exit = myth.execute("myth", input.getAbsolutePath(),
destination.getAbsolutePath());
    getLog().debug("Myth execution exiting with status: " + exit);
} catch (MojoExecutionException e) {
    throw new WatchingException("An error occurred during Myth processing of " +
input.getAbsolutePath(), e);
}
```

The *CoffeeScript* invocation is made by:

```
try {
    int exit = coffee.execute(COFFEE_SCRIPT_COMMAND, "--compile", "--map", "--output",
out.getAbsolutePath(),
            input.getAbsolutePath());
    getLog().debug("CoffeeScript compilation exits with " + exit + " status");
} catch (MojoExecutionException e) {
    if (!Strings.isNullOrEmpty(coffee.getLastErrorStream())) {
        throw buildCompilationError(coffee.getLastErrorStream(), input);
    } else {
        throw new WatchingException(ERROR_TITLE, "Error while compiling " + input
                .getAbsolutePath(), input, e);
    }
}
```

In this last example, we retrieve the error stream (if any) and parse the error message to build a `WatchingException` containing the compilation error, the guilty line and position.

34.8. Publishing your application in the Wisdom Extension Registry.

So you have made your very own extension for Wisdom and you want the world to see it? Great! You can add it to the Wisdom Extension registry located here.

In order to add your extension to the registry you need to create a Json file. This file will contain the details about your extension such as the name, the project's homepage and so on.

It may be helpful to have information on your homepage on how to use your extension. Extensions are generally used by either adding the extension as a dependency in a project's pom file, or as a maven plugin.

Here is a quick overview of the structure for the json file.

```
{
    "name": "wisdom-monitor",
    "version":"0.6.2",
    "description": "The Wisdom monitor provides an administration interface letting you
know the current status of the Wisdom server and deployed applications. This interface is
modular and can be easily extended with your own views.",
 "repository": {
        "type": "git",
        "url": "https://github.com/wisdom-framework/wisdom/tree/master/extensions/wisdom-
monitor"
    },

    "author": "The Wisdom Team",
    "license": {
            "type": "Apache",
            "url": "https://github.com/wisdom-framework/wisdom/blob/master/LICENSE.txt"
        },
    "homepage": "https://github.com/wisdom-
framework/wisdom/tree/master/extensions/wisdom-monitor",
    "keywords": [
        "cpu usage",
        "system",
        "monitor" ,
        "memory",
        "threads",
        "bundles"
    ]
}
```

Required fields are: `name` - the name of your extension, this name must be unique; `version`, `description`,

and `homepage`. The other fields are optional.

After you have have saved your Json file to a location accessible on the web, you can then enter its url into the text box at the bottom of the registry page. (If you don't see the input box make sure you're on the developers view version).

TIP	Currently to update your extension you just resubmit the url, in the same process you did for adding it.

35. The Wisdom Project

The Wisdom Framework is an open source project licensed under the Apache License 2.0.

All contributions are welcome !

35.1. Source Code

The source code is hosted on GitHub: https://github.com/wisdom-framework/wisdom/.

35.2. Bug Tracker

The Wisdom Project is using the GitHub Issue Tracker: https://github.com/wisdom-framework/wisdom/issues.

35.3. Continuous Integration

The main continuous integration server is provided by Cloudbees (Wisdom adheres to the FOSS program): https://wisdom-framework.ci.cloudbees.com/

35.4. Building Wisdom

To build Wisdom you need Apache Maven 3.2.1+. The Wisdom build is divided in several profiles:

```
mvn clean install -Pcore,\!framework,\!extensions,\!documentation
mvn clean install -P\!core,framework,extensions,documentation
```

To skip the tests, append the `-DskipTests` option to the command line.

As tests are being executed in a Wisdom server, you may experience some timing issues. If it's the case, try setting the `TIME_FACTOR` to extend the *wait* durations:

```
mvn clean install -Pfull-runtime -DTIME_FACTOR=2
```

TIP	**Time Factor**: The time factor extends the time waited to determine whether a service is published or not. It is also used to detect the *Stability* of the platform, *i.e.* when all services and bundles are started.

35.5. Releasing Wisdom

As Wisdom contains a Maven Plugin used by other projects and extending the lifecycle (so called *extension*), the maven-release-plugin cannot be used. This Maven limitation makes the release process a bit cumbersome.

First, check that your `~/.m2/settings.xml` contains you GPG password when the `release` profile is activated:

```
<profile>
    <id>release</id>
    <properties>
        <gpg.passphrase>YOUR_PASSPHRASE</gpg.passphrase>
    </properties>
</profile>
```

Then check that your credential to `Sonatype OSS` are written in your `~/.m2/settings.xml`:

```
<server>
<id>sonatype-nexus-staging</id>
<username>YOUR_USERNAME</username>
<password>SECRET</password>
</server>
```

You also need to be sure you have the permission to push to the Wisdom Git repository.

1. Check that you have the latest version, and declare the code freeze on wisdom-discuss.
2. Use the maven-versions-plugin to update to the release version:

```
# We will use this version everywhere:
export VERSION=enter the released version here
mvn clean
mvn versions:set -DnewVersion=$VERSION  -DgenerateBackupPoms=false
```

3. Be sure that you don't have any `-SNAPSHOT<` anymore.

```
egrep --exclude="**/target*" -ir  "(-SNAPSHOT)<" .
```

Notice that you will have matches, and that's normal. You need to verify by yourself.

4. Let's now check that everything is fine:

```
mvn install -Prelease,core,\!framework,\!extensions,\!documentation
-Dmaven.repo.local=$HOME/tmp/repo-for-$VERSION
mvn install -Prelease,\!core,framework,extensions,documentation
-Dmaven.repo.local=$HOME/tmp/repo-for-$VERSION
mvn clean -Prelease -Dmaven.repo.local=$HOME/tmp/repo-for-$VERSION
```

If everything is fine, continue, else rollback. It's generally recommended to check that the artifact are correctly signed with GPG.

5. Commit the updated pom files, create a tag, and push.

```
git add -A
git commit -m "[RELEASE] update to release version: $VERSION"
git push origin master
git tag -a wisdom-framework-$VERSION -m "[RELEASE] Create tag for $VERSION"
git push origin wisdom-framework-$VERSION
```

6. The tag is created, we are going to bump to the version to the next development version:

IMPORTANT The new version must ends with `-SNAPSHOT`

```
export NEW_VERSION=enter the new version here

mvn versions:set -DnewVersion=$NEW_VERSION
-Pcore,\!framework,\!extensions,\!documentation -DgenerateBackupPoms=false
mvn clean install -DskipTests -Pcore,\!framework,\!extensions,\!documentation
mvn versions:set -DnewVersion=$NEW_VERSION -P\!core,framework,extensions,documentation
-DgenerateBackupPoms=false
mvn clean install -P\!core,framework,extensions,documentation -DskipTests
git add -A
git commit -m "[RELEASE] update to development version: $NEW_VERSION"
git push origin master
```

What you did so far is the equivalent to the `mvn release:prepare`. It's now time to `perform` the release, so building the released artifacts.

7. Checkout the tag we created

```
git fetch
git tag -l
git checkout tags/wisdom-framework-$VERSION
```

8. Let's build and deploy the release, as everything has been checked before we can safely skip the tests:

```
mvn deploy -Prelease,core,\!extensions,\!framework,\!documentation -DskipTests
mvn deploy -Prelease,framework,extensions,documentation,\!core -DskipTests
```

9. When done, open a browser to https://oss.sonatype.org/, and log in.

10. On the left sidebar, go to 'Staging Repositories', and find the created repository.

11. Select it, and click on the `close` button. A couple of checks are performed, such as checking that all the expected artifacts are available, and that all artifacts are correctly signed.

12. Once done, re-check the repository and click on the 'release' button.

13. Wait until the artifacts lands in Maven Central.

14. Announce the release on the `wisdom-discuss` mailing list. Don't forget to integrate the release notes. These releases notes can be computed from the Github Issue Tracker.

15. On the Github Issue Tracker, don't forget to create a new milestone for the next release, and decide which issues are going to be fixed in this new release.

10. Almost done, now we have to generate the 'documentation artifacts'

```
mkdir ~/Desktop/Wisdom-$VERSION
# Copy the source code archive
cp target/*-source-release.zip ~/Desktop/Wisdom-$VERSION
# Copy the documentation jar
cp documentation/documentation/target/documentation-$VERSION.jar ~/Desktop/Wisdom-
$VERSION
# Generate and copy the javadoc
mvn javadoc:aggregate-jar -Pcore,framework,extensions,documentation
cp target/*$VERSION-javadoc.jar ~/Desktop/Wisdom-$VERSION
# Generate the wisdom-maven-plugin site
cd core/wisdom-maven-plugin
mvn site site:jar
cp target/*$VERSION-site.jar ~/Desktop/Wisdom-$VERSION
```

11. On Github, go to the release section (https://github.com/wisdom-framework/wisdom/releases). A

new release should have been created. Edit its name to be 'Wisdom Framework $VERSION' (replace $VERSION) and add a link to the release notes. In addition upload the `-source-release.zip` artifacts as binaries.

12. Send the 3 others artifacts to wisdom-commit so they can be uploaded on the web site. The upload need to wait until the artifact reach Maven Central.

13. Send a mail to wisdom-discuss to announce the release

14. Update the docker images, they will be rebuilt automatically by DockerHub. The image are built from https://github.com/wisdom-framework/wisdom-docker. You need to update to the release version, then launch `mvn clean install`, and push the result as to a tag and to the master branch.

If you reach this point, you made it ! Congratulations !

36. F.A.Q.

36.1. IDE Integration

36.1.1. Eclipse

Maven integration in Eclipse requires a m2e connector to fully use plugins in the native Eclipse's build process. An overview of this issue can be found at here. For now there is no plan to add such a connector for Wisdom. To remove warnings and errors, we recommend to set an lifecycle mapping file in your Eclipse.

Here is the minimal lifecycle mapping file to remove wisdom's errors in Eclipse.

Eclipse compiler may generate classes in `target\classes`. If you see issue in the watch mode, change this folder to be `target/classes-eclipse`, and let Wisdom compile your classes.

36.2. Using Eclipse Equinox

By default Wisdom uses Apache Felix as OSGi framework. You can also use Eclipse Equinox. To switch to Equinox, edit your project's `pom.xml` file and add the `<wisdomRuntime>` parameter of the `wisdom-maven-plugin`:

```
<plugin>
    <groupId>org.wisdom-framework</groupId>
    <artifactId>wisdom-maven-plugin</artifactId>
    <version>0.8.0</version>
    <extensions>true</extensions>
    <configuration>
        <wisdomRuntime>equinox</wisdomRuntime>
    </configuration>
</plugin>
```