

Play 2.0 documentation

Play 2.0 is a high-productivity Java and Scala web application framework that integrates the components and APIs you need for modern web application development.

Play is based on a lightweight, stateless, web-friendly architecture and features predictable and minimal resource consumption (CPU, memory, threads) for highly-scalable applications, thanks to its reactive model, based on Iteratee IO.

1. About
2. Getting started
3. Play 2.0 for Scala developers
 1. HTTP programming
 2. Asynchronous HTTP programming
 3. The template engine
 4. HTTP form submission and validation
 5. Working with JSON
 6. Working with XML
 7. Handling file upload
 8. Accessing an SQL database
 9. Using the cache
 10. Calling web services
 11. Integrating with Akka
 12. Internationalization
 13. The application Global object
 14. Testing your application
 15. Handling data streams reactively
 16. Your first application
4. Play 2.0 for Java developers
 1. HTTP programming
 2. Asynchronous HTTP programming
 3. The template engine
 4. HTTP form submission and validation
 5. Working with JSON
 6. Working with XML
 7. Handling file upload
 8. Accessing an SQL database
 9. Using the cache
 10. Calling web services
 11. Integrating with Akka
 12. Internationalization
 13. The application Global object
 14. Testing your application
 15. Your first application
5. Detailed topics
 1. The build system
 2. Working with public assets
 3. Managing database evolutions
 4. Configuration file syntax and features
 5. Deploying your application
6. Hacking Play 2.0

Introducing Play 2.0

Since 2007, we have been working on making Java web application development easier. Play started as an internal project at Zenexity and was heavily influenced by our way of doing web projects: focusing on developer productivity, respecting web architecture, and using a fresh approach to packaging conventions from the start - breaking so-called JEE best practices where it made sense.

In 2009, we decided to share these ideas with the community as an open source project. The immediate feedback was extremely positive and the project gained a lot of traction. Today - after two years of active development - Play has several versions, an active community of 4,000 people, with a growing number of applications running in production all over the globe.

Opening a project to the world certainly means more feedback, but it also means discovering and learning about new use cases, requiring features and un-earthing bugs that we were not specifically considered in the original design and its assumptions. During the two years of work on Play as an open source project we have worked to fix this kind of issues, as well as to integrate new features to support a wider range of scenarios. As the project has grown, we have learned a lot from our community and from our own experience - using Play in more and more complex and varied projects.

Meanwhile, technology and the web have continued to evolve. The web has become the central point of all applications. HTML, CSS and JavaScript technologies have evolved quickly - making it almost impossible for a server-side framework to keep up. The whole web architecture is fast moving towards real-time processing, and the emerging requirements of today's project profiles mean SQL no longer works as the exclusive datastore technology. At the programming language level we've witnessed some monumental changes with several JVM languages, including Scala, gaining popularity.

That's why we created Play 2.0, a new web framework for a new era.

Built for asynchronous programming

Today's web applications are integrating more concurrent real-time data, so web frameworks need to support a full asynchronous HTTP programming model. Play was initially designed to handle classic web applications with many short-lived requests. But now, the event model is the way to go for persistent connections - though Comet, long-polling and WebSockets.

Play 2.0 is architected from the start under the assumption that every request is potentially long-lived. But that's not all: we also need a powerful way to schedule and run long-running tasks. The Actor-based model is unquestionably the best model today to handle highly concurrent systems, and the best implementation of that model available for both Java and Scala is Akka - so it's going in. Play 2.0 provides native Akka support for Play applications, making it possible to write highly-distributed systems.

Focused on type safety

One benefit of using a statically-typed programming language for writing Play applications is that the compiler can check parts of your code. This is not only useful for detecting mistakes early in the development process, but it also makes it a lot easier to work on large projects with a lot of developers involved.

Adding Scala to the mix for Play 2.0, we clearly benefit from even stronger compiler guarantees - but that's not enough. In Play 1.x, the template system was dynamic, based on the Groovy language, and the compiler couldn't do much for you. As a result, errors in templates were only detectable at run-time. The same goes for verification of glue code with controllers.

In version 2.0, we really wanted to push this idea of having Play check most of your code at compilation time further. This is why we decided to use the Scala-based template engine as the default for Play applications - even for developers using Java as the main programming language. This doesn't mean that you have to become a Scala expert to write templates in Play 2.0, just as you were not really required to know Groovy to write templates in Play 1.x.

In templates, Scala is mainly used to navigate your object graph in order to display relevant information, with a syntax that is very close to Java's. However, if you want to unleash the power of Scala to write advanced templates abstractions, you will quickly discover how Scala, being expression-oriented and functional, is a perfect fit for a template engine.

And that's not only true for the template engine: the routing system is also fully type-checked. Play 2.0 checks your routes' descriptions, and verifies that everything is consistent, including the reverse routing part.

A nice side effect of being fully compiled is that the templates and route files will be easier to package and reuse. You also get a significant performance gain on these parts at run-time.

Native support for Java and Scala

Early in the Play project's history, we started exploring the possibility of using the Scala programming language for writing Play applications. We initially introduced this work as an external module, to be able to experiment freely without impacting the framework itself.

Properly integrating Scala into a Java-based framework is not trivial. Considering Scala's compatibility with Java, one can quickly achieve a first naive integration that simply uses Scala's syntax instead of Java's. This, however, is certainly not the optimal way of using the language. Scala is a mix of true object orientation with functional programming. Leveraging the full power of Scala requires rethinking most of the framework's APIs.

We quickly reached the limits of what we can do with Scala support as a separate module. Initial design choices we made in Play 1.x, relying heavily on Java reflection API and byte code manipulation, have made it harder to progress without completely rethinking some essential parts of Play's internals. Meanwhile, we have created several awesome components for the Scala module, such as the new type-safe template engine and the brand new SQL access component Anorm. This is why we decided that, to fully unleash the power of Scala with Play, we would move Scala support from a separate module to the core of Play 2.0, which is designed from the beginning to natively support Scala as a programming language.

Java, on the other hand, is certainly not getting any less support from Play 2.0; quite the contrary. The Play 2.0 build provides us with an opportunity to enhance the development experience for Java developers. Java developers get a real Java API written with all the Java specificity in mind.

Powerful build system

From the beginning of the Play project, we have chosen a fresh way to run, compile and deploy Play applications. It may have looked like an esoteric design at first, but it was crucial to providing an asynchronous HTTP API instead of the standard Servlet API, short feedback cycles through live compilation and reloading of source code during development, and promoting a fresh packaging approach. Consequently, it was difficult to make Play follow the standard JEE conventions.

Today, this idea of container-less deployment is increasingly accepted in the Java world. It's a design choice that has allowed the Play framework to run natively on platforms like Heroku, which introduced a model that we consider the future of Java application deployment on elastic PaaS platforms.

Existing Java build systems, however, were not flexible enough to support this new approach. Since we wanted to provide straightforward tools to run and deploy Play applications, in Play 1.x we created a collection of Python scripts to handle build and deployment tasks.

Meanwhile, developers using Play for more enterprise-scale projects, which require build process customization and integration with their existing company build systems, were a bit lost. The Python scripts we provided with Play 1.x are in no way a fully-featured build system and are not easily customizable. That's why we've decided to go for a more powerful build system for Play 2.0.

Since we need a modern build tool, flexible enough to support Play original conventions and able to build Java and Scala projects, we have chosen to integrate sbt in Play 2.0. This, however, should not scare existing Play users who are happy with the simplicity of the original Play build. We are leveraging the same simple `play new`, `run`, `start` experience on top of an extensible model: Play 2.0 comes with a preconfigured build script that will just work for most users. On the other hand, if you need to change the way your application is built and deployed, the fact that a Play project is a standard sbt project gives you all the power you need to customize and adapt it.

This also means better integration with Maven projects out of the box, the ability to package and publish your project as a simple set of JAR files to any repository, and especially live compiling and reloading at development time of any depended project, even for standard Java or Scala library projects.

Datastore and model integration

'Data store' is no longer synonymous with 'SQL database', and probably never was. A lot of interesting data storage models are becoming popular, providing different properties for different scenarios. For this reason it has become difficult for a web framework like Play to make bold assumptions regarding the kind of data store that developers will use. A generic model concept in Play no longer makes sense, since it is almost impossible to abstract over all these kinds of technologies with a single API.

In Play 2.0, we wanted to make it really easy to use any data store driver, ORM, or any other database access library without any special integration with the web framework. We simply want to offer a minimal set of helpers to handle common technical issues, like managing the connection bounds. We also want, however, to maintain the full-stack aspect of Play framework by bundling default tools to access classical databases for users WHO don't have specialized needs, and that's why Play 2.0 comes with built-in relational database access libraries such as Ebean, JPA and Anorm.

Installing Play 2.0

Prerequisites

To run the Play framework, you need JDK 6 or later.

If you are using MacOS, Java is built-in. If you are using Linux, make sure to use either the Sun JDK or OpenJDK (and not gcj, which is the default Java command on many Linux distros). If you are using Windows, just download and install the latest JDK package.

Be sure to have the `java` and `javac` commands in the current path (you can check this by typing `java -version` and `javac -version` at the shell prompt).

Download the binary package

Download the Play 2.0 binary package and extract the archive to a location where you have both **read** and **write** access. (Running `play` writes some files to directories within the archive, so don't install to `/opt`, `/usr/local` or anywhere else you'd need special permission to write to.)

Add the play script to your PATH

For convenience, you should add the framework installation directory to your system PATH. On UNIX systems, this means doing something like:

```
export PATH=$PATH:/path/to/play20
```

On Windows you'll need to set it in the global environment variables.

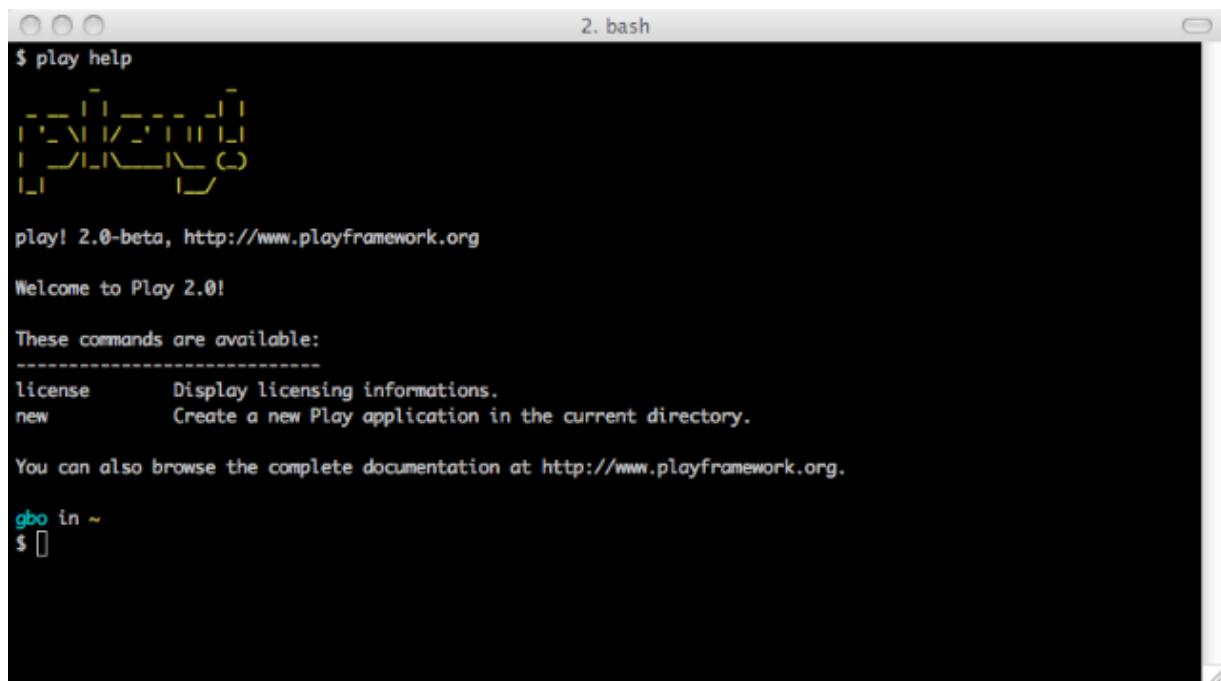
If you're on UNIX, make sure that the `play` script is executable (otherwise do a `chmod a+x play`).

Check that the play command is available

From a shell, launch the `play help` command.

```
$ play help
```

If everything is properly installed, you should see the basic help:



```
$ play help
   __  _ 
  / \ | |
 /  \| |
 /   \| |
 /_ \|_|

play! 2.0-beta, http://www.playframework.org

Welcome to Play 2.0!

These commands are available:
-----
license      Display licensing informations.
new          Create a new Play application in the current directory.

You can also browse the complete documentation at http://www.playframework.org.

gbo in ~
$ 
```

Creating a new application

Create a new application with the play command

The easiest way to create a new application is to use the `play new` command.

```
$ play new myFirstApp
```

This will ask for some information.

- The application name (just for display, this name will be used later in several messages).
- The template to use for this application. You can choose either a default Scala application, a default Java application, or an empty application.

```
$ play new myFirstApp
[Progress Bar]
play! 2.0-beta, http://www.playframework.org
The new application will be created in /Volumes/Data/gbo/myFirstApp
What is the application name?
> My first application
Which template do you want to use for this new application?
1 - Create a simple Scala application
2 - Create a simple Java application
3 - Create an empty project
> 1
OK, application My first application is created.
Type `play` to enter the development console.
Have fun!
```

Note that choosing a template at this point does not imply that you can't change language later. For example, you can create a new application using the default Java application template and start adding Scala code whenever you like.

Once the application has been created you can use the `play` command again to enter the Play 2.0 console .

```
$ cd myFirstApp
$ play
```

Create a new application without having Play installed

You can also create a new Play application without installing Play, by using sbt.

First install sbt 0.11.2 if needed.

Just create a new directory for your new application and configure your sbt build script with two additions.

In `project/plugins.sbt`, add:

```
// The Typesafe repository
resolvers += "Typesafe repository" at "http://repo.typesafe.com/typesafe/
releases/"

// Use the Play sbt plugin for Play projects
addSbtPlugin("play" % "sbt-plugin" % "2.0")
```

Be sure to replace `2.0` here by the exact version you want to use. If you want to use a snapshot version, you will have to specify this additional resolver:

```
// Typesafe snapshots
resolvers += "Typesafe Snapshots" at "http://repo.typesafe.com/typesafe/
snapshots/"
```

In `project/Build.scala`:

```
import sbt._
import Keys._
import PlayProject._

object ApplicationBuild extends Build {

    val appName          = "My first application"
    val appVersion       = "1.0"

    val appDependencies = Nil

    val main = PlayProject(
        appName, appVersion, appDependencies, mainLang = SCALA
    )

}
```

You can then launch the sbt console in this directory:

```
$ cd myFirstApp  
$ sbt
```

sbt will load your project and fetch the dependencies.

Anatomy of a Play 2.0 application

The standard application layout

The layout of a Play application is standardized to keep things as simple as possible. A standard Play application looks like this:

app	→ Application sources
└ assets	→ Compiled asset sources
└ stylesheets	→ Typically LESS CSS sources
└ javascripts	→ Typically CoffeeScript sources
└ controllers	→ Application controllers
└ models	→ Application business layer
└ views	→ Templates
conf	→ Configurations files
└ application.conf	→ Main configuration file
└ routes	→ Routes definition
public	→ Public assets
└ stylesheets	→ CSS files
└ javascripts	→ Javascript files
└ images	→ Image files
project	→ sbt configuration files
└ build.properties	→ Marker for sbt project
└ Build.scala	→ Application build script
└ plugins.sbt	→ sbt plugins
lib	→ Unmanaged libraries dependencies
logs	→ Standard logs folder
└ application.log	→ Default log file
target	→ Generated stuff
└ scala-2.9.1	
└ cache	
└ classes	→ Compiled class files
└ classes_managed	→ Managed class files (templates, ...)
└ resource_managed	→ Managed resources (less, ...)
└ src_managed	→ Generated sources (templates, ...)
test	→ source folder for unit or functional tests

The app/ directory

The `app` directory contains all executable artifacts: Java and Scala source code, templates and compiled assets' sources.

There are three standard packages in the `app` directory, one for each component of the MVC architectural pattern:

- `app/controllers`
- `app/models`
- `app/views`

You can of course add your own packages, for example an `app/utils` package.

Note that in Play 2.0, the controllers, models and views package name conventions are now just that and can be changed if needed (such as prefixing everything with `com.yourcompany`).

There is also an optional directory called `app/assets` for compiled assets such as LESS sources and CoffeeScript sources .

The public/ directory

Resources stored in the `public` directory are static assets that are served directly by the Web server.

This directory is split into three standard sub-directories for images, CSS stylesheets and JavaScript files. You should organize your static assets like this to keep all Play applications consistent.

In a newly-created application, the `/public` directory is mapped to the `/assets` URL path, but you can easily change that, or even use several directories for your static assets.

The conf/ directory

The `conf` directory contains the application's configuration files. There are two main configuration files:

- `application.conf`, the main configuration file for the application, which contains standard configuration parameters
- `routes`, the routes definition file.

If you need to add configuration options that are specific to your application, it's a good idea to add more options to the `application.conf` file.

If a library needs a specific configuration file, try to file it under the `conf` directory.

The lib/ directory

The `lib` directory is optional and contains unmanaged library dependencies , ie. all JAR files you want to manually manage outside the build system. Just drop any JAR files here and they will be added to your application classpath.

The project/ directory

The `project` directory contains the sbt build definitions:

- `plugins.sbt` defines sbt plugins used by this project
 - `Build.scala` defines your application build script.
-

The target/ directory

The `target` directory contains everything generated by the build system. It can be useful to know what is generated here.

- `classes/` contains all compiled classes (from both Java and Scala sources).
 - `classes_managed/` contains only the classes that are managed by the framework (such as the classes generated by the router or the template system). It can be useful to add this class folder as an external class folder in your IDE project.
 - `resource_managed/` contains generated resources, typically compiled assets such as LESS CSS and CoffeeScript compilation results.
 - `src_managed/` contains generated sources, such as the Scala sources generated by the template system.
-

Typical `.gitignore` file

Generated folders should be ignored by your version control system. Here is the typical `.gitignore` file for a Play application:

```
logs
project/project
project/target
target
tmp
```

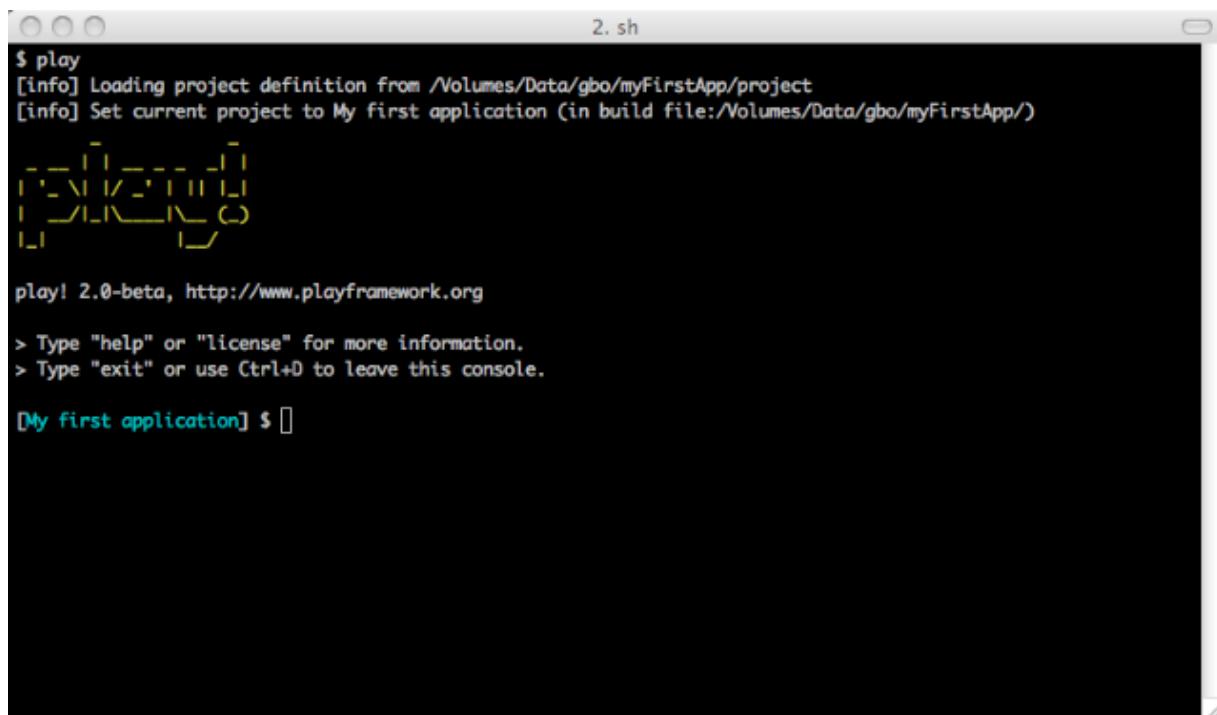
Using the Play 2.0 console

Launching the console

The Play 2.0 console is a development console based on sbt that allows you to manage a Play application's complete development cycle.

To launch the console, enter any existing Play application directory and run the `play` script:

```
$ cd /path/to/any/application  
$ play
```



A screenshot of a terminal window titled "2. sh". The window contains the following text:

```
$ play  
[info] Loading project definition from /Volumes/Data/gbo/myFirstApp/project  
[info] Set current project to My first application (in build file:/Volumes/Data/gbo/myFirstApp/)  
[My first application]$ [REDACTED]
```

The terminal window has a dark background with white text. The title bar is light-colored. The text area shows the command \$ play followed by informational messages about loading the project and setting the current project to "My first application". Below this, there is a prompt [My first application]\$ followed by a redacted area.

Getting help

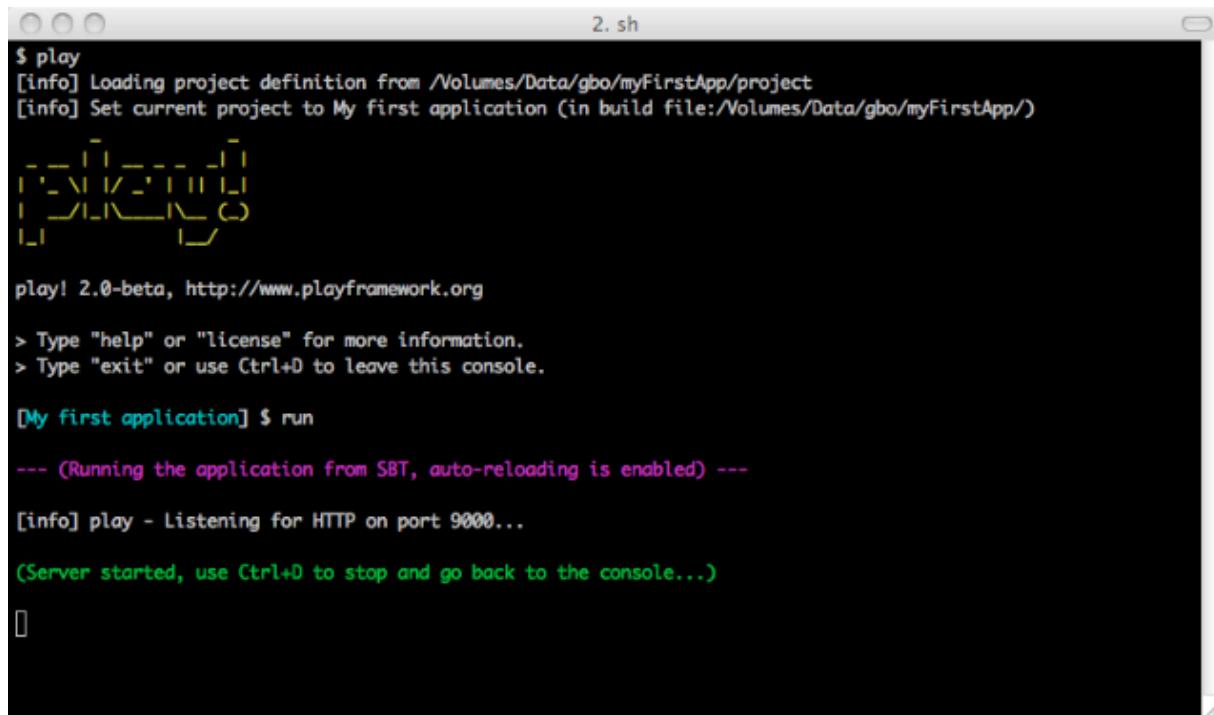
Use the `help play` command to get basic help about the available commands:

```
[My first application] $ help play
```

Running the server in development mode

To run the current application in development mode, use the `run` command:

```
[My first application] $ run
```



```
$ play
[info] Loading project definition from /Volumes/Data/gbo/myFirstApp/project
[info] Set current project to My first application (in build file:/Volumes/Data/gbo/myFirstApp/)

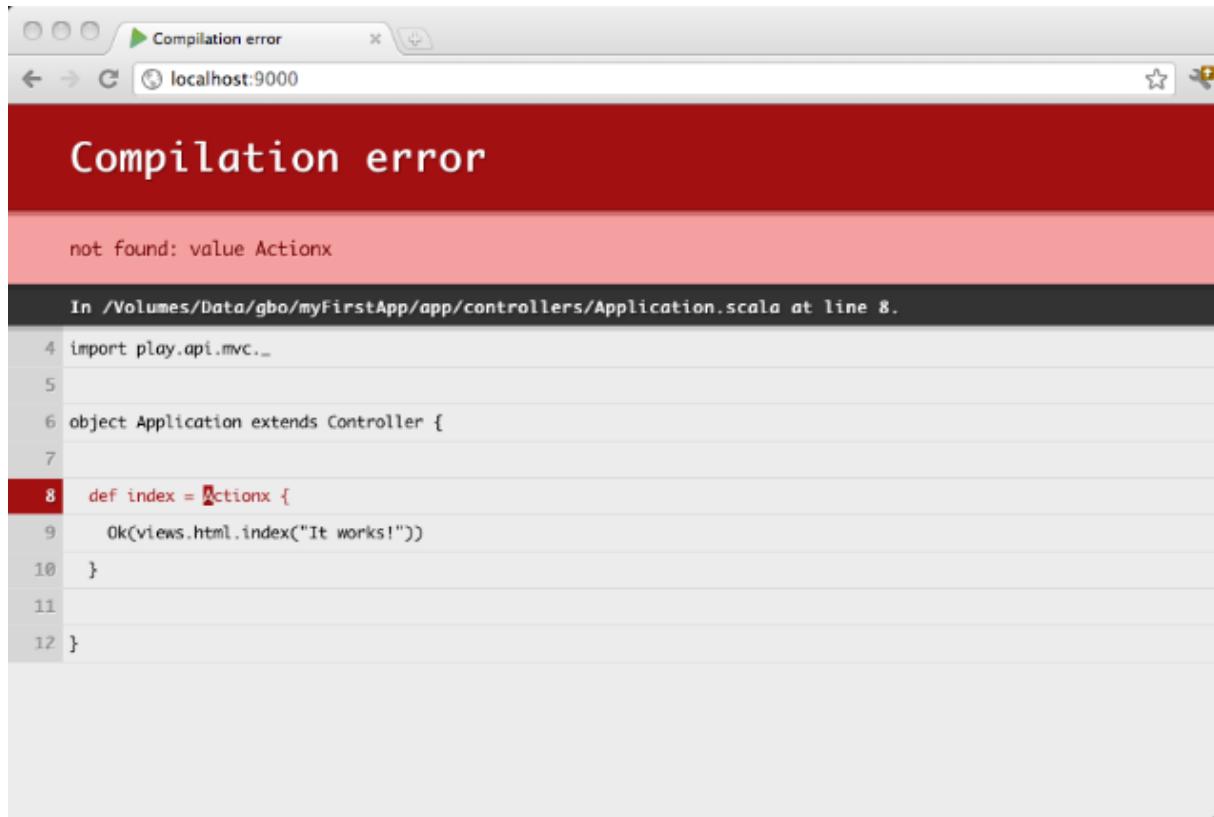
play! 2.0-beta, http://www.playframework.org

> Type "help" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[My first application] $ run
--- (Running the application from SBT, auto-reloading is enabled) ---
[info] play - Listening for HTTP on port 9000...
(Server started, use Ctrl+D to stop and go back to the console...)
```

In this mode, the server will be launched with the auto-reload feature enabled, meaning that for each request Play will check your project and recompile required sources. If needed the application will restart automatically.

If there are any compilation errors you will see the result of the compilation directly in your browser:

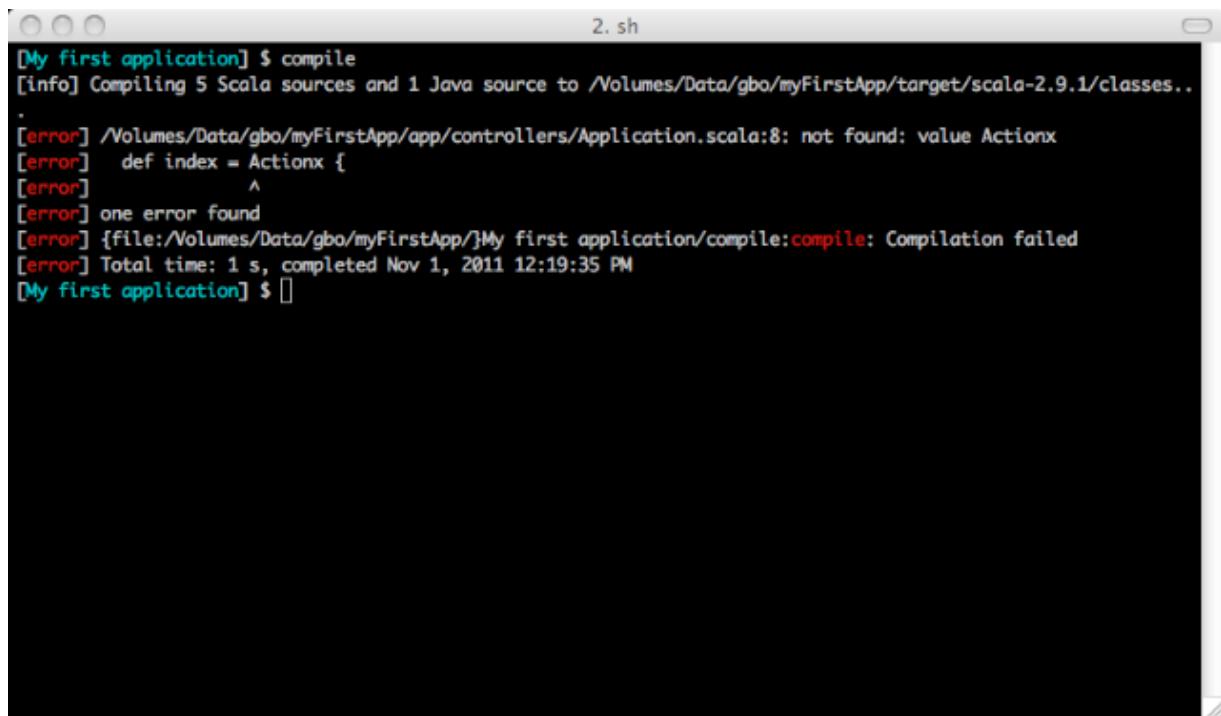


To stop the server, type **Ctrl+D** key, and you will be returned to the Play console prompt.

Compiling

In Play 2.0 you can also compile your application without running the server. Just use the `compile` command:

```
[My first application] $ compile
```



The screenshot shows a terminal window with a dark background and light-colored text. The title bar says "2. sh". The text inside the window is a command-line session:

```
[My first application] $ compile
[info] Compiling 5 Scala sources and 1 Java source to /Volumes/Data/gbo/myFirstApp/target/scala-2.9.1/classes..
.
[error] /Volumes/Data/gbo/myFirstApp/app/controllers/Application.scala:8: not found: value Actionx
[error]   def index = Actionx {
[error]   ^
[error] one error found
[error] {file:/Volumes/Data/gbo/myFirstApp/]My first application/compile:compile: Compilation failed
[error] Total time: 1 s, completed Nov 1, 2011 12:19:35 PM
[My first application] $ []
```

Launch the interactive console

Type `console` to enter the interactive Scala console, which allows you to test your code interactively:

```
[My first application] $ console
```



```
[My first application] $ console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_26).
Type in expressions to have them evaluated.
Type :help for more information.

scala> views.html.index("Hello")
res0: play.api.templates.Html =
```



```
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
    <link rel="stylesheet" media="screen" href="/assets/stylesheets/main.css">
    <link rel="shortcut icon" type="image/png" href="/assets/images/favicon.png">
    <script src="/assets/javascripts/jquery-1.6.4.min.js" type="text/javascript"></script>
  </head>
  <body>

    <section>
```

Debugging

You can ask Play to start a **JPDA** debug port when starting the console. You can then connect using Java debugger. Use the `play debug` command to do that:

```
$ play debug
```

When a JPDA port is available, the JVM will log this line during boot:

```
Listening for transport dt_socket at address: 9999
```

Note: Using `play debug` the JPDA socket will be opened on port `9999`. You can also set the `JPDA_PORT` environment variable yourself using `set JPDA_PORT=1234`.

Using sbt features

The Play console is just a normal sbt console, so you can use sbt features such as **triggered execution**.

For example, using `~ compile`

```
[My first application] $ ~ compile
```

The compilation will be triggered each time you change a source file.

If you are using `~ run`

```
[My first application] $ ~ run
```

The triggered compilation will be enabled while a development server is running.

You can also do the same for `~ test`, to continuously test your project each time you modify a source file:

```
[My first application] $ ~ test
```

Using the play commands directly

You can also run commands directly without entering the Play console. For example, enter `play run`:

```
$ play run
[info] Loading project definition from myFirstApp/project
[info] Set current project to My first application

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)
```

The application starts directly. When you quit the server using `Ctrl+D`, you will come back to your OS prompt.

Force clean

If something goes wrong and you think that the sbt cache is corrupted, use the `clean-all` command for your OS command line to clean all generated directories.

```
$ play clean-all
```

Setting-up your preferred IDE

Working with Play is easy. You don't even need a sophisticated IDE, because Play compiles and refreshes the modifications you make to your source files automatically, so you can easily work using a simple text editor.

However, using a modern Java or Scala IDE provides cool productivity features like auto-completion, on-the-fly compilation, assisted refactoring and debugging.

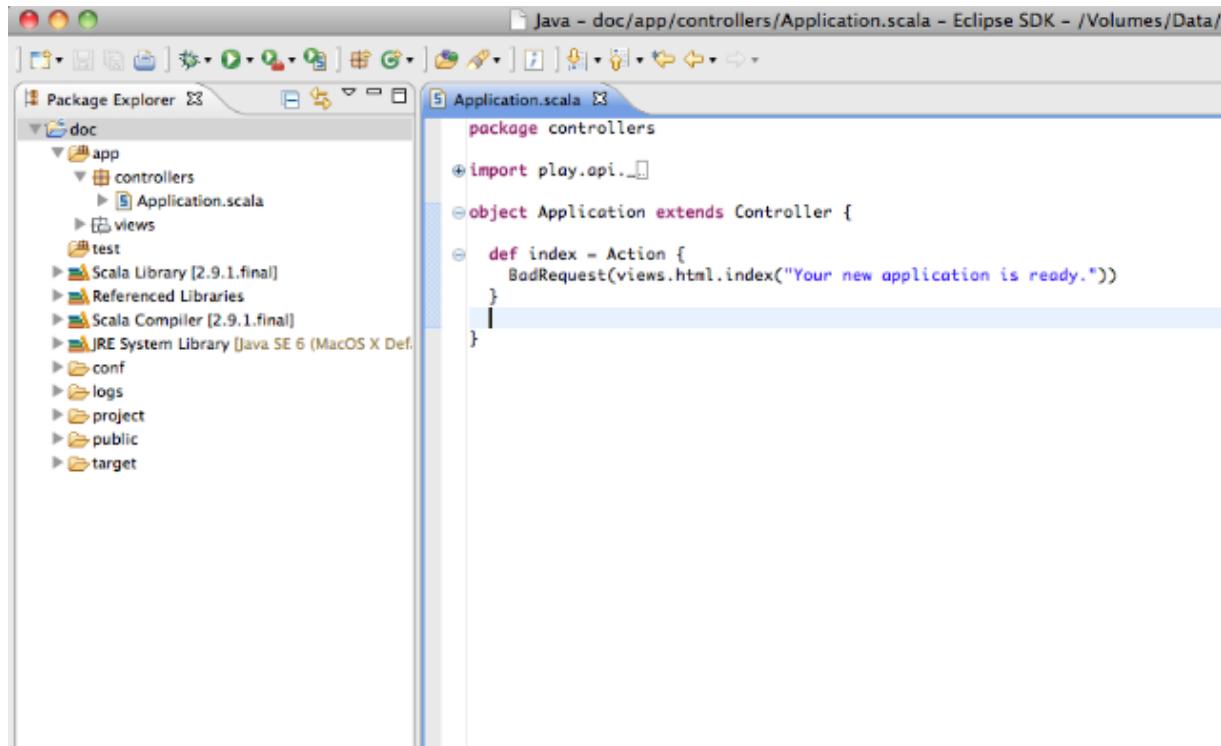
Eclipse

Generate configuration

Play provides a command to simplify Eclipse configuration. To transform a Play application into a working Eclipse project, use the `eclipsify` command:

```
[My first application] $ eclipsify
```

You then need to import the application into your Workspace with the **File/Import/General/Existing project...** menu (compile your project first).



You can also start your application with `play debug run` and then you can use the Connect JPDA launcher using **Debug As** to start a debugging session at any time. Stopping the debugging session will not stop the server.

Tip: You can run your application using `~run` to enable direct compilation on file change. This way scala templates files are auto discovered when you create new template in `view` and auto compiled when file change. If you use normal `run` then you have to hit `Refresh` on your browser each time.

If you make any important changes to your application, such as changing the classpath, use `eclipsify` again to regenerate the configuration files.

Tip: Do not commit Eclipse configuration files when you work in a team!

The generated configuration files contain absolute references to your framework installation. These are specific to your own installation. When you work in a team, each developer must keep his Eclipse configuration files private.

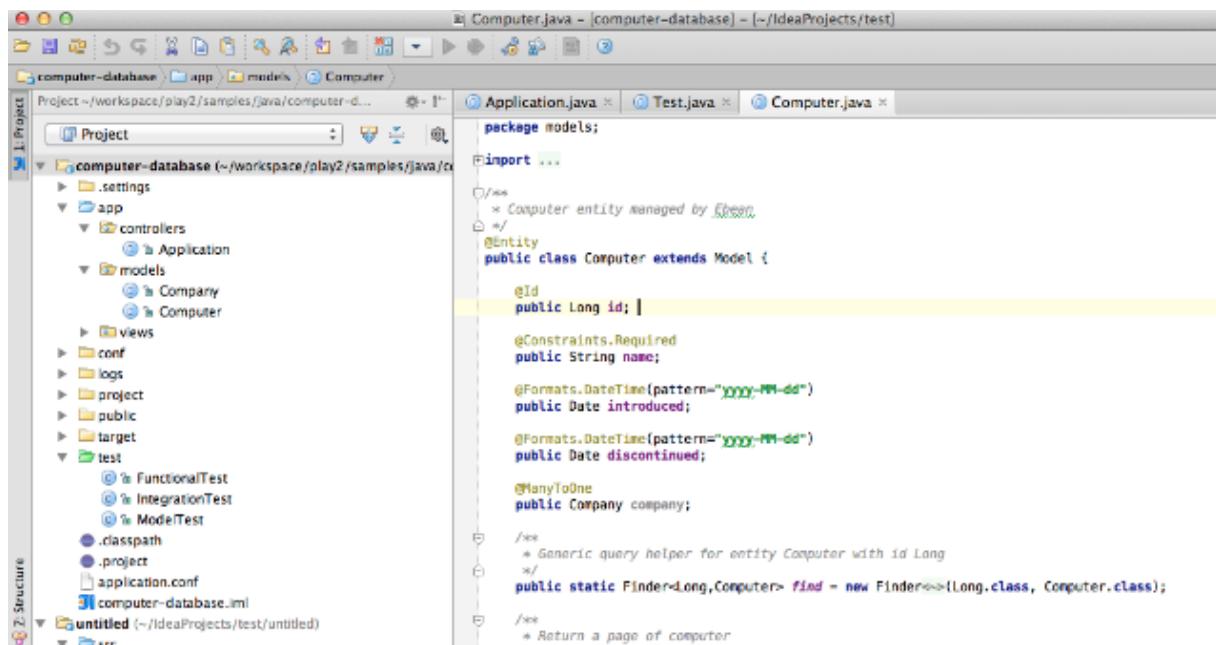
IntelliJ

Generate configuration

Play provides a command to simplify IntelliJ IDEA configuration. To transform a Play application into a working IDEA module, use the `idea` command:

```
[My first application] $ idea
```

You then need to import the application into your project (File->New Module->Import existing Module)



Tip: There is an [IntelliJ IDEA issue](#) regarding building Java based Play2 apps while having the scala plugin installed. Until it's fixed, the recommended workaround is to disable scala plugin.

To debug, first add a debug configuration

- Open Run/Debug Configurations dialog, then click Run -> Edit Configurations
- Add a Remote configuration, then select `Remote`
- Configure it:
 - Set a name
 - Transport : Socket
 - Debugger mode: Attach
 - Host: localhost
 - Port: 9999
 - Select module you imported
- Close dialog - click Apply

Start play in debug mode:

```
$ play debug
```

which should print:

```
Listening for transport dt_socket at address: 9999
```

Set some breakpoints. Run the web app by executing the task `play` (again I had to do this in same terminal I ran `play debug`). Finally, browse `http://localhost:9000`. IntelliJ should stop at your breakpoint.

If you make any important changes to your application, such as changing the classpath, use `idea` again to regenerate the configuration files.

Netbeans

Generate Configuration

Play does not have native Netbeans project generation support at this time. For now you can generate a Netbeans Scala project with the Netbeans SBT plugin.

First edit the `plugins.sbt` file

```
resolvers += {  
    "remeniuk repo" at "http://remeniuk.github.com/maven"  
}  
  
libraryDependencies += {  
    "org.netbeans" %% "sbt-netbeans-plugin" % "0.1.4"  
}
```

Now run

```
$ play netbeans
```

Sample applications

The Play 2.0 package comes with a comprehensive set of sample applications written in both Java and Scala. This is a very good place to look for code snippets and examples.

The sample applications are available in the `samples/` directory of your Play installation.

Hello world

The 'helloworld' application

Configure your 'Hello world':

What's your name?

How many times?

Choose a color

This is a very basic application that demonstrates Play 2.0 fundamentals:

- Writing controllers and actions.
- Routing and reverse routing.
- Linking to public assets.
- Using the template engine.
- Handling forms with validation.

Computer database

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

This is a classic CRUD application, backed by a JDBC database. It demonstrates :

- accessing a JDBC database , using Ebean in Java and Anorm in Scala
- table pagination and CRUD forms
- integrating with a CSS framework (Twitter Bootstrap).

Twitter Bootstrap requires a different form layout to the default layout provided by the Play 2.0 form helper, so this application also provides an example of integrating a custom form input constructor .

Forms

The screenshot shows a web browser window titled "Form samples" with the URL "localhost:9000/signup". The page has a dark header bar with tabs for "Forms", "Sign up", and "Contacts". The main content area is titled "Sign Up" with a link "Or edit an existing user".

Account informations

- Username: An input field with the placeholder "Please choose a valid username."
- Email: An input field with the placeholder "Enter a valid email address."
- Password: An input field with the placeholder "A password must be at least 6 characters."
- Repeat password: An input field with the placeholder "Please repeat your password again."

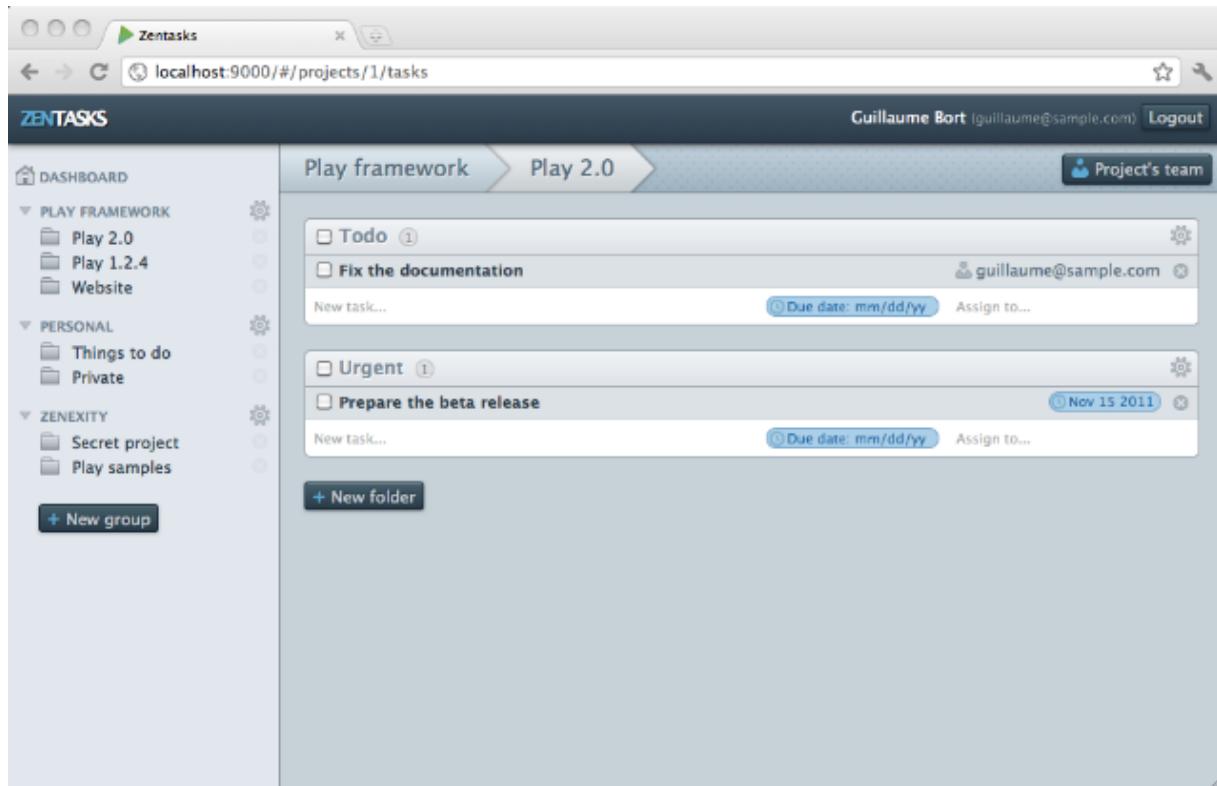
Contact informations

- Country: A dropdown menu with the placeholder "--- Choose a country ---" and the word "Required" below it.
- Address: An input field.

This is a dummy application presenting several typical form usages. It demonstrates :

- writing complex forms with validation
- handling forms with dynamically repeated values.

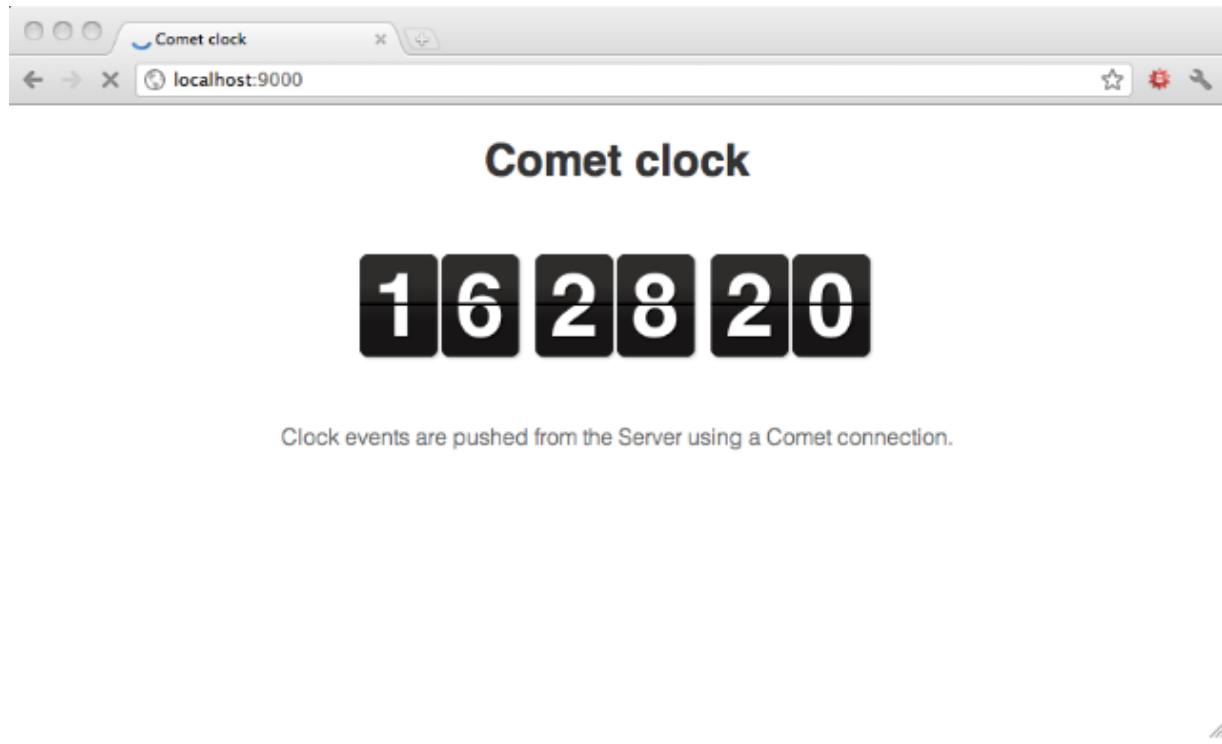
ZenTasks



This advanced todo list demonstrates a modern Ajax-based web application. This is a work in progress, and we plan to add features in the future releases. For now you can check it out to learn how to:

- integrate authentication and security
- use Ajax and JavaScript reverse routing
- integrate with compiled assets - LESS CSS and CoffeeScript.

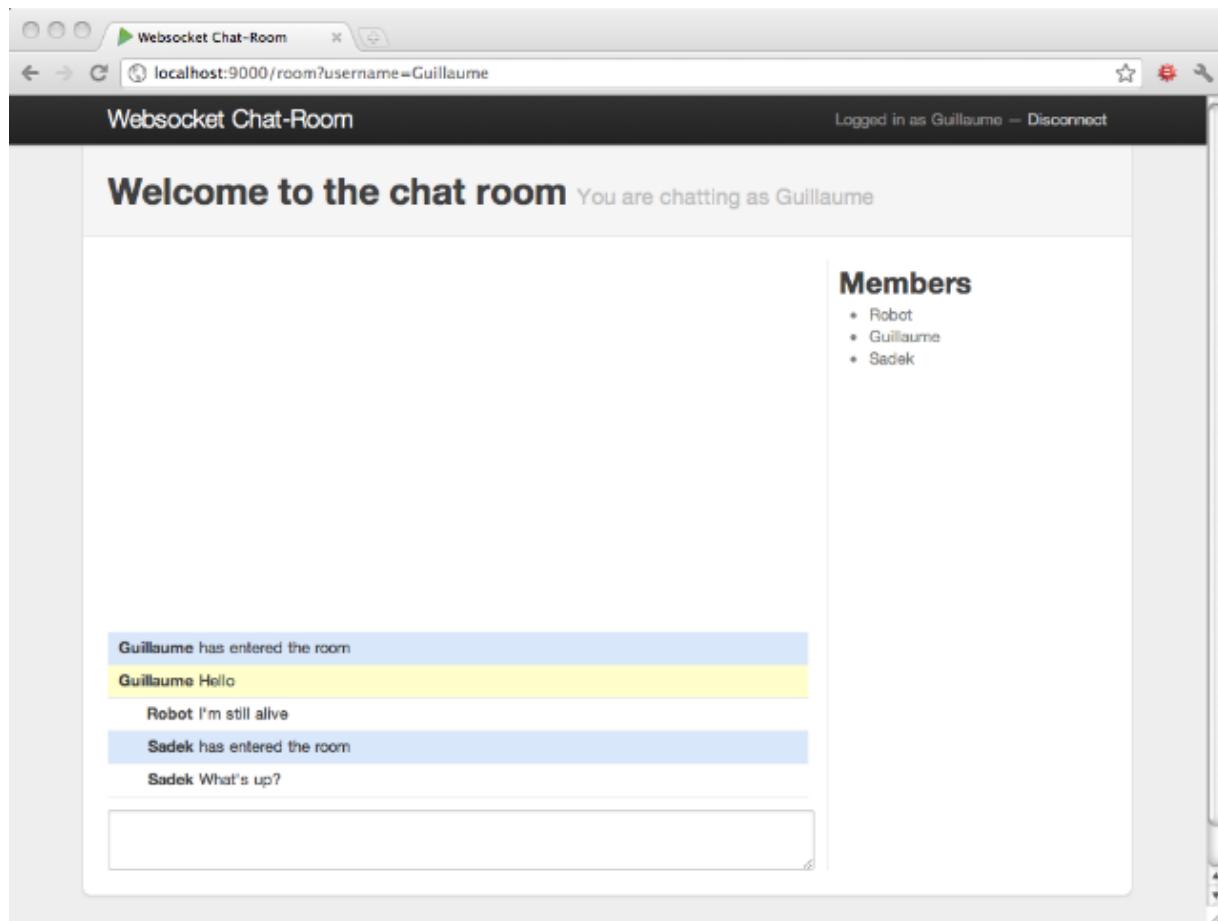
CometClock



This a very simple Comet demonstration pushing clock events from the server to the Web browser using a the forever-frame technique. It demonstrates how to:

- create a Comet connection
- use Akka actors (in the Java version)
- write custom Enumerators (in the Scala version).

WebSocket chat



This application is a chat room, built using WebSockets. Additionally, there is a bot used that talks in the same chat room. It demonstrates:

- WebSocket connections
- advanced Akka usage.

Comet monitoring



This mobile web application monitors Play server performance . It demonstrates :

- advanced usage of Enumerators and Enumeratees .

Play 2.0 for Scala developers

The Scala API for Play 2.0 application developers is available in the `play.api` package.

The API available directly inside the `play` package (such as `play.mvc`) is reserved for Java developers. As a Scala developer, look at `play.api.mvc`.

Actions, Controllers and Results

What is an Action?

Most of the requests received by a Play application are handled by an `Action`.

A `play.api.mvc.Action` is basically a `(play.api.mvc.Request => play.api.mvc.Result)` function that handles a request and generates a result to be sent to the client.

```
val echo = Action { request =>
    Ok("Got request [" + request + "]")
}
```

An action returns a `play.api.mvc.Result` value, representing the HTTP response to send to the web client. In this example `Ok` constructs a **200 OK** response containing a **text/plain** response body.

Building an Action

The `play.api.mvc.Action` companion object offers several helper methods to construct an `Action` value.

The first simplest one just takes as argument an expression block returning a `Result`:

```
Action {
    Ok("Hello world")
}
```

This is the simplest way to create an Action, but we don't get a reference to the incoming request. It is often useful to access the HTTP request calling this Action.

So there is another Action builder that takes as an argument a function `Request => Result`:

```
Action { request =>
    Ok("Got request [" + request + "]")
}
```

It is often useful to mark the `request` parameter as `implicit` so it can be implicitly used by other APIs that need it:

```
Action { implicit request =>
    Ok("Got request [" + request + "]")
}
```

The last way of creating an Action value is to specify an additional `BodyParser` argument:

```
Action(parse.json) { implicit request =>
  Ok("Got request [" + request + "]")
}
```

Body parsers will be covered later in this manual. For now you just need to know that the other methods of creating Action values use a default **Any content body parser**.

Controllers are action generators

A `Controller` is nothing more than a singleton object that generates `Action` values.

The simplest use case for defining an action generator is a method with no parameters that returns an `Action` value :

```
package controllers

import play.api.mvc._

object Application extends Controller {

  def index = Action {
    Ok("It works!")
  }

}
```

Of course, the action generator method can have parameters, and these parameters can be captured by the `Action` closure:

```
def hello(name: String) = Action {
  Ok("Hello " + name)
}
```

Simple results

For now we are just interested in simple results: An HTTP result with a status code, a set of HTTP headers and a body to be sent to the web client.

These results are defined by `play.api.mvc.SimpleResult`:

```
def index = Action {  
    SimpleResult(  
        header = ResponseHeader(200, Map(CONTENT_TYPE -> "text/plain")),  
        body = Enumerator("Hello world!")  
    )  
}
```

Of course there are several helpers available to create common results such as the `Ok` result in the sample above:

```
def index = Action {  
    Ok("Hello world!")  
}
```

This produces exactly the same result as before.

Here are several examples to create various results:

```
val ok = Ok("Hello world!")  
val notFound = NotFound  
val pageNotFound = NotFound(<h1>Page not found</h1>)  
val badRequest = BadRequest(views.html.form(formWithErrors))  
val oops = InternalServerError("Oops")  
val anyStatus = Status(488)("Strange response type")
```

All of these helpers can be found in the `play.api.mvc.Results` trait and companion object.

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 take a response body.

There are several helpers available to create redirect results:

```
def index = Action {  
    Redirect("/user/home")  
}
```

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

```
def index = Action {  
    Redirect("/user/home", status = MOVED_PERMANENTLY)  
}
```

“TODO” dummy page

You can use an empty `Action` implementation defined as `TODO`: the result is a standard ‘Not implemented yet’ result page:

```
def index(name:String) = TODO
```

HTTP routing

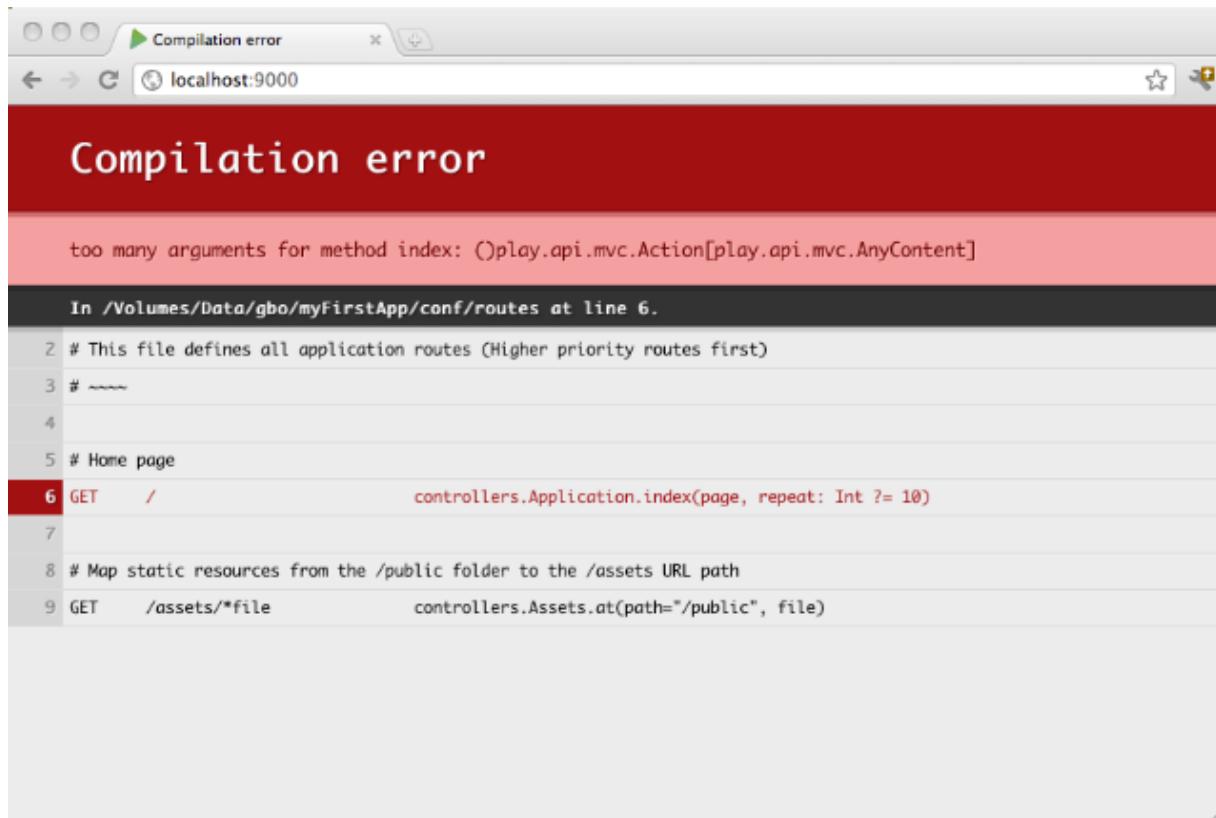
The built-in HTTP router

The router is the component in charge of translating each incoming HTTP request to an Action.

An HTTP request is seen as an event by the MVC framework. This event contains two major pieces of information:

- the request path (e.g. `/clients/1542`, `/photos/list`), including the query string
- the HTTP method (e.g. GET, POST, ...).

Routes are defined in the `conf/routes` file, which is compiled. This means that you'll see route errors directly in your browser:



A screenshot of a web browser window titled "Compilation error". The address bar shows "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below this, the error message "too many arguments for method index: ()play.api.mvc.Action[play.api.mvc.AnyContent]" is displayed. The rest of the page shows the contents of the routes file, with line numbers 2 through 9 visible. Line 6 is highlighted in red, indicating the error occurred there. The code defines routes for the home page and static assets.

```
too many arguments for method index: ()play.api.mvc.Action[play.api.mvc.AnyContent]

In /Volumes/Data/gbo/myFirstApp/conf/routes at line 6.

2 # This file defines all application routes (Higher priority routes first)
3 #
4
5 # Home page
6 GET    /           controllers.Application.index(page, repeat: Int ?= 10)
7
8 # Map static resources from the /public folder to the /assets URL path
9 GET    /assets/*file   controllers.Assets.at(path="/public", file)
```

The routes file syntax

`conf/routes` is the configuration file used by the router. This file lists all of the routes needed by the application. Each route consists of an HTTP method and URI pattern, both associated with a call to an `Action` generator.

Let's see what a route definition looks like:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Each route starts with the HTTP method, followed by the URI pattern. The last element is the call definition.

You can also add comments to the route file, with the `#` character.

```
# Display a client.  
GET /clients/:id controllers.Clients.show(id: Long)
```

The HTTP method

The HTTP method can be any of the valid methods supported by HTTP (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`).

The URI pattern

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

Static path

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

```
GET /clients/all controllers.Clients.list()
```

Dynamic parts

If you want to define a route that retrieves a client by ID, you'll need to add a dynamic part:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Note that a URI pattern may have more than one dynamic part.

The default matching strategy for a dynamic part is defined by the regular expression `[^/]+`, meaning that any dynamic part defined as `:id` will match exactly one URI part.

Dynamic parts spanning several /

If you want a dynamic part to capture more than one URI path segment, separated by forward slashes, you can define a dynamic part using the `*id` syntax, which uses the `.*` regular expression:

GET /files/*name	controllers.Application.download(name)
------------------	--

Here for a request like `GET /files/images/logo.png`, the `name` dynamic part will capture the `images/logo.png` value.

Dynamic parts with custom regular expressions

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

GET /clients/\$id<[0-9]+> controllers.Clients.show(id: Long)
--

Call to the Action generator method

The last part of a route definition is the call. This part must define a valid call to a method returning a `play.api.mvc.Action` value, which will typically be a controller action method.

If the method does not define any parameters, just give the fully-qualified method name:

GET / controllers.Application.homePage()
--

If the action method defines some parameters, all these parameter values will be searched for in the request URI, either extracted from the URI path itself, or from the query string.

Extract the page parameter from the path. GET /:page controllers.Application.show(page)
--

Or:

Extract the page parameter from the query string. GET / controllers.Application.show(page)

Here is the corresponding, `show` method definition in the `controllers.Application` controller:

def show(page: String) = Action { loadContentFromDatabase(page).map { htmlContent => Ok(htmlContent).as("text/html") }.getOrElse(NotFound) }
--

Parameter types

For parameters of type `String`, typing the parameter is optional. If you want Play to transform the incoming parameter into a specific Scala type, you can explicitly type the parameter:

```
GET /client/:id controllers.Clients.show(id: Long)
```

And do the same on the corresponding `show` method definition in the `controllers.Clients` controller:

```
def show(id: Long) = Action {
    Client.findById(id).map { client =>
        Ok(views.html.Clients.display(client))
    }.getOrElse(NotFound)
}
```

Parameters with fixed values

Sometimes you'll want to use a fixed value for a parameter:

```
# Extract the page parameter from the path, or fix the value for /
GET / controllers.Application.show(page = "home")
GET /:page controllers.Application.show(page)
```

Parameters with default values

You can also provide a default value that will be used if no value is found in the incoming request:

```
# Pagination links, like /clients?page=3
GET /clients controllers.Clients.list(page: Int ?= 1)
```

Routing priority

Many routes can match the same request. If there is a conflict, the first route (in declaration order) is used.

Reverse routing

The router can also be used to generate a URL from within a Scala call. This makes it possible to centralize all your URI patterns in a single configuration file, so you can be more confident when refactoring your application.

For each controller used in the routes file, the router will generate a ‘reverse controller’ in the `routes` package, having the same action methods, with the same signature, but returning a `play.api.mvc.Call` instead of a `play.api.mvc.Action`.

The `play.api.mvc.Call` defines an HTTP call, and provides both the HTTP method and the URI.

For example, if you create a controller like:

```
package controllers

import play.api._
import play.api.mvc._

object Application extends Controller {

    def hello(name: String) = Action {
        Ok("Hello " + name + "!")
    }

}
```

And if you map it in the `conf/routes` file:

```
# Hello action
GET  /hello/:name      controllers.Application.hello(name)
```

You can then reverse the URL to the `hello` action method, by using the `controllers.routes.Application.reverse controller`:

```
// Redirect to /hello/Bob
def helloBob = Action {
    Redirect(routes.Application.hello("Bob"))
}
```

Manipulating Results

Changing the default Content-Type

The result content type is automatically inferred from the Scala value that you specify as the response body.

For example:

```
val textResult = Ok("Hello World!")
```

Will automatically set the `Content-Type` header to `text/plain`, while:

```
val xmlResult = Ok(<message>Hello World!</message>)
```

will set the `Content-Type` header to `text/xml`.

Tip: this is done via the `play.api.http.ContentTypeOf` type class.

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:

```
val htmlResult = Ok(<h1>Hello World!</h1>).as("text/html")
```

or even better, using:

```
val htmlResult = Ok(<h1>Hello World!</h1>).as(HTML)
```

Note: The benefit of using `HTML` instead of the `"text/html"` is that the charset will be automatically handled for you and the actual Content-Type header will be set to `text/html; charset=utf-8`. We will see that in a bit.

Manipulating HTTP headers

You can also add (or update) any HTTP header to the result:

```
Ok("Hello World!").withHeaders(  
  CACHE_CONTROL -> "max-age=3600",  
  ETag -> "xx"  
)
```

Note that setting an HTTP header will automatically discard the previous value if it was existing in the original result.

Setting and discarding cookies

Cookies are just a special form of HTTP headers but we provide a set of helpers to make it easier.

You can easily add a Cookie to the HTTP response using:

```
Ok("Hello world").withCookies(  
  Cookie("theme", "blue")  
)
```

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

```
Ok("Hello world").discardingCookies("theme")
```

Changing the charset for text based HTTP responses .

For text based HTTP response it is very important to handle the charset correctly. Play handles that for you and uses `utf-8` by default.

The charset is used to both convert the text response to the corresponding bytes to send over the network socket, and to alterate the `Content-Type` header with the proper `;charset=xxxx` extension.

The charset is handled automatically via the `play.api.mvc.Codec` type class. Just import an implicit instance of `play.api.mvc.Codec` in the current scope to change the charset that will be used by all operations:

```
object Application extends Controller {  
  
  implicit val myCustomCharset = Codec.javaSupported("iso-8859-1")  
  
  def index = Action {  
    Ok(<h1>Hello World!</h1>).as(HTML)  
  }  
  
}
```

Here, because there is an implicit charset value in the scope, it will be used by both the `Ok(...)` method to convert the XML message into `ISO-8859-1` encoded bytes and to generate the `text/html; charset=iso-8859-1` Content-Type header.

Now if you are wondering how the `HTML` method works, here it is how it is defined:

```
def HTML(implicit codec: Codec) = {  
    "text/html; charset=" + codec.charset  
}
```

You can do the same in your API if you need to handle the charset in a generic way.

Session and Flash scopes

How it is different in Play

If you have to keep data across multiple HTTP requests, you can save them in the Session or Flash scopes. Data stored in the Session are available during the whole user Session, and data stored in the Flash scope are available to the next request only.

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

Of course, cookie values are signed with a secret key so the client can't modify the cookie data (or it will be invalidated).

The Play 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 Play built-in cache mechanism and use store a unique ID in the user Session to keep them related to a specific user.

There is no technical timeout for the Session. It 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.).

Reading a Session value

You can retrieve the incoming Session from the HTTP request:

```
def index = Action { request =>
    request.session.get("connected").map { user =>
        Ok("Hello " + user)
    }.getOrElse {
        Unauthorized("Oops, you are not connected")
    }
}
```

Alternatively you can retrieve the Session implicitly from a request:

```
def index = Action { implicit request =>
    session.get("connected").map { user =>
        Ok("Hello " + user)
    }.getOrElse {
        Unauthorized("Oops, you are not connected")
    }
}
```

Storing data in the Session

As the Session is just a Cookie, it is also just an HTTP header. You can manipulate the session data the same way you manipulate other results properties:

```
Ok("Welcome!").withSession(
    "connected" -> "user@gmail.com"
)
```

Note that this will replace the whole session. If you need to add an element to an existing Session, just add an element to the incoming session, and specify that as new session:

```
Ok("Hello World!").withSession(
    session + ("saidHello" -> "yes")
)
```

You can remove any value from the incoming session the same way:

```
Ok("Theme reset!").withSession(
    session - "theme"
)
```

Discarding the whole session

There is special operation that discards the whole session:

```
Ok("Bye").withNewSession
```

Flash scope

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

- data are kept for only one request
- 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 are 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 subject to race conditions.

Here are a few examples using the Flash scope:

```
def index = Action { implicit request =>
  Ok {
    flash.get("success").getOrElse("Welcome!")
  }
}

def save = Action {
  Redirect("/home").flashing(
    "success" -> "The item has been created"
  )
}
```

Body parsers

What is a body parser?

An HTTP PUT or POST request contains a body. This body can use any format, specified in the `Content-Type` request header. In Play, a **body parser** transforms this request body into a Scala value.

However the request body for an HTTP request can be very large and a **body parser** can't just wait and load the whole data set into memory before parsing it. A `BodyParser[A]` is basically an `Iteratee[Array[Byte], A]`, meaning that it receives chunks of bytes (as long as the web browser uploads some data) and computes a value of type `A` as result.

Let's consider some examples.

- A **text** body parser could accumulate chunks of bytes into a String, and give the computed String as result (`Iteratee[Array[Byte], String]`).
- A **file** body parser could store each chunk of bytes into a local file, and give a reference to the `java.io.File` as result (`Iteratee[Array[Byte], File]`).
- A **s3** body parser could push each chunk of bytes to Amazon S3 and give a the S3 object id as result (`Iteratee[Array[Byte], S3ObjectID]`).

Additionally a **body parser** has access to the HTTP request headers before it starts parsing the request body, and has the opportunity to run some precondition checks. For example, a body parser can check that some HTTP headers are properly set, or that the user trying to upload a large file has the permission to do so.

Note: That's why a body parser is not really an `Iteratee[Array[Byte], A]` but more precisely a `Iteratee[Array[Byte], Either[Result, A]]`, meaning that it has the opportunity to send directly an HTTP result itself (typically `400 BAD_REQUEST`, `412 PRECONDITION_FAILED` or `413 REQUEST_ENTITY_TOO_LARGE`) if it decides than it is not able to compute a correct value for the request body

Once the body parser finishes its job and gives back a value of type `A`, the corresponding `Action` function is executed and the computed body value is passed into the request.

More about Actions

Previously we said that an `Action` was a `Request => Result` function. This is not entirely true. Let's have a more precise look at the `Action` trait:

```
trait Action[A] extends (Request[A] => Result) {  
    def parser: BodyParser[A]  
}
```

First we see that there is a generic type `A`, and then that an action must define a `BodyParser[A]`.

With `Request[A]` being defined as:

```
trait Request[+A] extends RequestHeader {  
    def body: A  
}
```

The `A` type is the type of the request body. We can use any Scala type as the request body, for example `String`, `NodeSeq`, `Array[Byte]`, `JsonValue`, or `java.io.File`, as long as we have a body parser able to process it.

To summarize, an `Action[A]` uses a `BodyParser[A]` to retrieve a value of type `A` from the HTTP request, and to build a `Request[A]` object that is passed to the action code.

Default body parser: AnyContent

In our previous examples we never specified a body parser. So how can it work? If you don't specify your own body parser, Play will use the default, which processes the body as an instance of `play.api.mvc.AnyContent`.

This body parser checks the `Content-Type` header and decides what kind of body to process:

- **text/plain**: `String`
- **application/json**: `JsValue`
- **text/xml**: `NodeSeq`
- **application/form-url-encoded**: `Map[String, Seq[String]]`
- **multipart/form-data**: `MultipartFormData[TemporaryFile]`
- any other content type: `RawBuffer`

For example:

```
def save = Action { request =>  
    val body: AnyContent = request.body  
    val textBody: Option[String] = body.asText  
  
    // Expecting text body  
    textBody.map { text =>  
        Ok("Got: " + text)  
    }.getOrElse {  
        BadRequest("Expecting text/plain request body")  
    }  
}
```

Specifying a body parser

The body parsers available in Play are defined in `play.api.mvc.BodyParsers.parse`.

So for example, to define an action expecting a text body (as in the previous example):

```
def save = Action(parse.text) { request =>
    Ok("Got: " + request.body)
}
```

Do you see how the code is simpler? This is because the `parse.text` body parser already sent a `400 BAD_REQUEST` response if something went wrong. We don't have to check again in our action code, and we can safely assume that `request.body` contains the valid `String` body.

Alternatively we can use:

```
def save = Action(parse.tolerantText) { request =>
    Ok("Got: " + request.body)
}
```

This one doesn't check the `Content-Type` header and always loads the request body as a `String`.

Tip: There is a `tolerant` fashion provided for all body parsers included in Play.

Here is another example, which will store the request body in a file:

```
def save = Action(parse.file(to = new File("/tmp/upload"))) { request =>
    Ok("Saved the request content to " + request.body)
}
```

Combining body parsers

In the previous example, all request bodies are stored in the same file. This is a bit problematic isn't it? Let's write another custom body parser that extract the user name from the request Session, to give a unique file for each user:

```
val storeInUserFile = parse.using { request =>
    request.session.get("username").map { user =>
        file(to = new File("/tmp/" + user + ".upload"))
    }.getOrElse {
        error(Unauthorized("You don't have the right to upload here"))
    }
}

def save = Action(storeInUserFile) { request =>
    Ok("Saved the request content to " + request.body)
}
```

Note: Here we are not really writing our own BodyParser, but just combining existing ones. This is often enough and should cover most use cases. Writing a `BodyParser` from scratch is covered in the advanced topics section.

Max content length

Text based body parsers (such as `text`, `json`, `xml` or `formUrlEncoded`) use a maximum content length because they have to load all of the content into memory.

There is a default content length (the default is 100KB), but you can also specify it inline:

```
// Accept only 10KB of data.
def save = Action(parse.text(maxLength = 1024 * 10)) { request =>
    Ok("Got: " + text)
}
```

Tip: The default content size can be defined in `application.conf`:

```
parsers.text.maxLength=128K
```

You can also wrap any body parser with `maxLength`:

```
// Accept only 10KB of data.
def save = Action(maxLength(1024 * 10, parser = storeInUserFile)) { request =>
    Ok("Saved the request content to " + request.body)
}
```

Action composition

This chapter introduce several ways of defining generic action functionality.

Basic action composition

Let's start with the simple example of a logging decorator: we want to log each call to this action.

The first way is not to define our own Action, but just to provide a helper method building a standard Action:

```
def LoggingAction(f: Request[AnyContent] => Result): Action[AnyContent] = {
  Action { request =>
    Logger.info("Calling action")
    f(request)
  }
}
```

That you can use as:

```
def index = LoggingAction { request =>
  Ok("Hello World")
}
```

This is simple but it works only with the default `parse.anyContent` body parser as we don't have a way to specify our own body parser. We can of course define an additional helper method:

```
def LoggingAction[A](bp: BodyParser[A])(f: Request[A] => Result): Action[A] =
  {
    Action(bp) { request =>
      Logger.info("Calling action")
      f(request)
    }
  }
```

And then:

```
def index = LoggingAction(parse.text) { request =>
  Ok("Hello World")
}
```

Wrapping existing actions

Another way is to define our own `LoggingAction` that would be a wrapper over another `Action`:

```
case class Logging[A](action: Action[A]) extends Action[A] {  
  
    def apply(request: Request[A]): Result = {  
        Logger.info("Calling action")  
        action(request)  
    }  
  
    lazy val parser = action.parser  
}
```

Now you can use it to wrap any other action value:

```
def index = Logging {  
    Action {  
        Ok("Hello World")  
    }  
}
```

Note that it will just re-use the wrapped action body parser as is, so you can of course write:

```
def index = Logging {  
    Action(parse.text) {  
        Ok("Hello World")  
    }  
}
```

Another way to write the same thing but without defining the `Logging` class, would be:

```
def Logging[A](action: Action[A]): Action[A] = {  
    Action(action.parser) { request =>  
        Logger.info("Calling action")  
        action(request)  
    }  
}
```

A more complicated example

Let's look at the more complicated but common example of an authenticated action. The main problem is that we need to pass the authenticated user to the wrapped action and to wrap the original body parser to perform the authentication.

```

def Authenticated[A](action: User => Action[A]): Action[A] = {

    // Let's define an helper function to retrieve a User
    def getUser(request: RequestHeader): Option[User] = {
        request.session.get("user").flatMap(u => User.find(u))
    }

    // Wrap the original BodyParser with authentication
    val authenticatedBodyParser = parse(using { request =>
        getUser(request).map(u => action(u).parser).getOrElse {
            parse.error(Unauthorized)
        }
    })

    // Now let's define the new Action
    Action(authenticatedBodyParser) { request =>
        getUser(request).map(u => action(u)(request)).getOrElse {
            Unauthorized
        }
    }
}

```

You can use it like this:

```

def index = Authenticated { user =>
    Action { request =>
        Ok("Hello " + user.name)
    }
}

```

Note: There is already an `Authenticated` action in `play.api.mvc.Security.Authenticated` with a better implementation than this example.

Another way to create the Authenticated action

Let's see how to write the previous example without wrapping the whole action and without authenticating the body parser:

```

def Authenticated(f: (User, Request[AnyContent]) => Result) = {
    Action { request =>
        request.session.get("user").flatMap(u => User.find(u)).map { user =>
            f(user, request)
        }.getOrElse(Unauthorized)
    }
}

```

To use this:

```
def index = Authenticated { (user, request) =>
    Ok("Hello " + user.name)
}
```

A problem here is that you can't mark the `request` parameter as `implicit` anymore. You can solve that using currying:

```
def Authenticated(f: User => Request[AnyContent] => Result) = {
    Action { request =>
        request.session.get("user").flatMap(u => User.find(u)).map { user =>
            f(user)(request)
        }.getOrElse(Unauthorized)
    }
}
```

Then you can do this:

```
def index = Authenticated { user => implicit request =>
    Ok("Hello " + user.name)
}
```

Another (probably simpler) way is to define our own subclass of `Request` as `AuthenticatedRequest` (so we are merging both parameters into a single parameter):

```
case class AuthenticatedRequest(
    val user: User, request: Request[AnyContent]
) extends WrappedRequest(request)

def Authenticated(f: AuthenticatedRequest => Result) = {
    Action { request =>
        request.session.get("user").flatMap(u => User.find(u)).map { user =>
            f(AuthenticatedRequest(user, request))
        }.getOrElse(Unauthorized)
    }
}
```

And then:

```
def index = Authenticated { implicit request =>
    Ok("Hello " + request.user.name)
}
```

We can of course extend this last example and make it more generic by making it possible to specify a body parser:

```
case class AuthenticatedRequest [A] (
  val user: User, request: Request [A]
) extends WrappedRequest (request)

def Authenticated [A] (p: BodyParser [A]) (f: AuthenticatedRequest [A] =>
Result) = {
  Action(p) { request =>
    request.session.get("user").flatMap(u => User.find(u)).map { user =>
      f(AuthenticatedRequest (user, request))
    }.getOrElse(Unauthorized)
  }
}

// Overloaded method to use the default body parser
import play.api.mvc.BodyParsers._
def Authenticated (f: AuthenticatedRequest [AnyContent] => Result): Action
[AnyContent] = {
  Authenticated (parse.anyContent) (f)
}
```

Handling asynchronous results

Why asynchronous results?

Until now, we were able to generate the result to send to the web client directly. However, this is not always the case: the result might depend on an expensive computation or of a long web service call.

Because of the way Play 2.0 works, the action code must be as fast as possible (ie. non blocking). So what should we return as result if we are not yet able to generate it? The response is a promise of result!

A `Promise[Result]` will eventually be redeemed with a value of type `Result`. By giving a `Promise[Result]` instead of a normal `Result`, we are able to quickly generate the result without blocking. Then, Play will serve this 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.

How to create a `Promise[Result]`

To create a `Promise[Result]` we need another promise first: the promise that will give us the actual value we need to compute the result:

```
val promiseOfPIValue : Promise[Double] = computePIAsynchronously()
val promiseOfResult : Promise[Result] = promiseOfPIValue.map { pi =>
    Ok("PI value computed: " + pi)
}
```

All of Play 2.0's asynchronous API calls give you a `Promise`. This is the case whether you are calling an external web service using the `play.api.libs.WS` API, or using Akka to schedule asynchronous tasks or to communicate with actors using `play.api.libs.Akka`.

A simple way to execute a block of code asynchronously and to get a `Promise` is to use the `play.api.libs.concurrent.Akka` helpers:

```
val promiseOfInt : Promise[Int] = Akka.future {
    intensiveComputation()
}
```

Note: Here, the intensive computation will just be run on another thread. It is also possible to run it remotely on a cluster of backend servers using Akka remote.

AsyncResult

While we were using `SimpleResult` until now, to send an asynchronous result, we need an `AsyncResult` to wrap the actual `SimpleResult`:

```
def index = Action {  
    val promiseOfInt = Akka.future { intensiveComputation() }  
    Async {  
        promiseOfInt.map(i => Ok("Got result: " + i))  
    }  
}
```

Note: `Async { }` is an helper method that builds an `AsyncResult` from a `Promise[Result]`.

Handling time-outs

It is often useful to handle time-outs properly, to avoid having the web browser block and wait if something goes wrong. You can easily compose a promise with a promise timeout to handle these cases:

```
def index = Action {  
    val promiseOfInt = Akka.future { intensiveComputation() }  
    Async {  
        promiseOfInt.orTimeout("Oops", 1000).map { eitherIntOrTimeout =>  
            eitherIntOrTimeout.fold(  
                timeout => InternalServerError(timeout),  
                i => Ok("Got result: " + i)  
            )  
        }  
    }  
}
```

Streaming HTTP responses

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, you are not specifying a `Content - Length` header when you send back a simple result, such as:

```
def index = Action {  
    Ok("Hello World")  
}
```

Of course, because the content you are sending is well-known, Play is able to compute the content size for you and to generate the appropriate header.

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

Actually, we previously saw that the response body is specified using a `play.api.libs.iteratee.Enumerator`:

```
def index = Action {  
    SimpleResult(  
        header = ResponseHeader(200),  
        body = Enumerator("Hello World")  
    )  
}
```

This means that to compute the `Content - Length` header properly, Play must consume the whole enumerator and load its content into memory.

Sending large amounts of data

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

Let's first see how to create an `Enumerator [Array[Byte]]` enumerating the file content:

```
val file = new java.io.File("/tmp/fileToServe.pdf")  
val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)
```

Now it looks simple right? Let's just use this enumerator to specify the response body:

```
def index = Action {  
  
    val file = new java.io.File("/tmp/fileToServe.pdf")  
    val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)  
  
    SimpleResult(  
        header = ResponseHeader(200),  
        body = fileContent  
    )  
}
```

Actually we have a problem here. As we don't specify the `Content-Length` header, Play will have to compute it itself, and the only way to do this is to consume the whole enumerator content and load it into memory, and then compute the response size.

That's a problem for large files that we don't want to load completely into memory. So to avoid that, we just have to specify the `Content-Length` header ourself.

```
def index = Action {  
  
    val file = new java.io.File("/tmp/fileToServe.pdf")  
    val fileContent: Enumerator[Array[Byte]] = Enumerator.fromFile(file)  
  
    SimpleResult(  
        header = ResponseHeader(200, Map(CONTENT_LENGTH -> file.length.toString)),  
        body = fileContent  
    )  
}
```

This way Play will consume the body enumerator in a lazy way, copying each chunk of data to the HTTP response as soon as it is available.

Serving files

Of course, Play provides easy-to-use helpers for common task of serving a local file:

```
def index = Action {  
    Ok.sendFile(new java.io.File("/tmp/fileToServe.pdf"))  
}
```

This helper will also compute the `Content-Type` header from the file name, and add the `Content-Disposition` header to specify how the web browser should handle this response. The default is to ask the web browser to download this file by adding the header `Content-Disposition: attachment; filename=fileToServe.pdf` to the HTTP response.

You also provide your own file name:

```
def index = Action {  
    Ok.sendFile(  
        content = new java.io.File("/tmp/fileToServe.pdf"),  
        fileName = _ => "termsOfService.pdf"  
    )  
}
```

If you want to serve this file `inline`:

```
def index = Action {  
    Ok.sendFile(  
        content = new java.io.File("/tmp/fileToServe.pdf"),  
        inline = true  
    )  
}
```

Now you don't have to specify a file name since the web browser will not try to download it, but will just display the file content in the web browser window. This is useful for content types supported natively by the web browser, such as text, HTML or images.

Chunked responses

For now, it works well with streaming file content since we are able to compute the content length before streaming it. But what about dynamically computed content, with no content size available?

For this kind of response we have to use **Chunked transfer encoding**.

Chunked transfer encoding is a data transfer mechanism in version 1.1 of the Hypertext Transfer Protocol (HTTP) in which a web server serves content in a series of chunks. It uses the `Transfer-Encoding` HTTP response header instead of the `Content-Length` header, which the protocol would otherwise require. Because the `Content-Length` header is not used, the server does not need to know the length of the content before it starts transmitting a response to the client (usually a web browser). Web servers can begin transmitting responses with dynamically-generated content before knowing the total size of that content.

The size of each chunk is sent right before the chunk itself, so that a client can tell when it has finished receiving data for that chunk. Data transfer is terminated by a final chunk of length zero.

http://en.wikipedia.org/wiki/Chunked_transfer_encoding

The advantage is that we can serve the data **live**, meaning that we send chunks of data as soon as they are available. The drawback is that since the web browser doesn't know the content size, it is not able to display a proper download progress bar.

Let's say that we have a service somewhere that provides a dynamic `InputStream` computing some data. First we have to create an `Enumerator` for this stream:

```
val data = getDataStream
val dataContent: Enumerator[Array[Byte]] = Enumerator.fromStream(data)
```

We can now stream these data using a `ChunkedResult`:

```
def index = Action {

    val data = getDataStream
    val dataContent: Enumerator[Array[Byte]] = Enumerator.fromStream(data)

    ChunkedResult(
        header = ResponseHeader(200),
        chunks = dataContent
    )
}
```

As always, there are helpers available to do this:

```
def index = Action {

    val data = getDataStream
    val dataContent: Enumerator[Array[Byte]] = Enumerator.fromStream(data)

    Ok.stream(dataContent)
}
```

Of course, we can use any `Enumerator` to specify the chunked data:

```
def index = Action {
    Ok.stream(
        Enumerator("kiki", "foo", "bar").andThen(Enumerator.eof)
    )
}
```

Tip: `Enumerator.callbackEnumerator` and `Enumerator.pushEnumerator` are two convenient way of creating reactive non-blocking enumerators in an imperative style.

We can inspect the HTTP response sent by the server:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

4
kiki
3
foo
3
bar
0
```

We get three chunks followed by one final empty chunk that closes the response.

Comet sockets

Using chunked responses to create Comet sockets

A good use for **Chunked responses** is to create Comet sockets. A Comet socket is just a chunked `text/html` response containing only `<script>` elements. At each chunk we write a `<script>` tag that is immediately executed by the web browser. This way we can send events live to the web browser from the server: for each message, wrap it into a `<script>` tag that calls a JavaScript callback function, and writes it to the chunked response.

Let's write a first proof-of-concept: an enumerator that generates `<script>` tags that each call the browser `console.log` JavaScript function:

```
def comet = Action {
    val events = Enumerator(
        """<script>console.log('kiki')</script>""",
        """<script>console.log('foo')</script>""",
        """<script>console.log('bar')</script>"""
    )
    Ok.stream(events >>> Enumerator.eof).as(HTML)
}
```

If you run this action from a web browser, you will see the three events logged in the browser console.

Tip: Writing `events >>> Enumerator.eof` is just another way of writing `events.andThen(Enumerator.eof)`

We can write this in a better way by using `play.api.libs.iteratee.Enumeratee` that is just an adapter to transform an `Enumerator[A]` into another `Enumerator[B]`. Let's use it to wrap standard messages into the `<script>` tags:

```
import play.api.templates.Html

// Transform a String message into an Html script tag
val toCometMessage = Enumeratee.map[String] { data =>
    Html("""<script>console.log(''" + data + "'')</script>""")
}

def comet = Action {
    val events = Enumerator("kiki", "foo", "bar")
    Ok.stream(events >>> Enumerator.eof &> toCometMessage)
}
```

Tip: Writing `events >>> Enumerator.eof &> toCometMessage` is just another way of writing `events.andThen(Enumerator.eof).through(toCometMessage)`

Using the `play.api.libs.Comet` helper

We provide a Comet helper to handle these Comet chunked streams that do almost the same stuff that we just wrote.

Note: Actually it does more, like pushing an initial blank buffer data for browser compatibility, and it supports both String and JSON messages. It can also be extended via type classes to support more message types.

Let's just rewrite the previous example to use it:

```
def comet = Action {  
    val events = Enumerator("kiki", "foo", "bar")  
    Ok.stream(events &> Comet(callback = "console.log"))  
}
```

Tip: `Enumerator.callbackEnumerator` and `Enumerator.pushEnumerator` are two convenient ways to create reactive non-blocking enumerators in an imperative style.

The forever iframe technique

The standard technique to write a Comet socket is to load an infinite chunked comet response in an HTML `iframe` and to specify a callback calling the parent frame:

```
def comet = Action {  
    val events = Enumerator("kiki", "foo", "bar")  
    Ok.stream(events &> Comet(callback = "parent.cometMessage"))  
}
```

With an HTML page like:

```
<script type="text/javascript">  
var cometMessage = function(event) {  
    console.log('Received event: ' + event)  
}  
</script>  
  
<iframe src="/comet"></iframe>
```

WebSockets

Using WebSockets instead of Comet sockets

A Comet socket is a kind of hack for sending live events to the web browser. Also, it only supports one-way communication from the server to the client. To push events to the server, the web browser has to send Ajax requests.

Note: It is also possible to achieve the same kind of live communication the other way around by using an infinite HTTP request handled by a custom `BodyParser` that receives chunks of input data, but that is far more complicated.

Modern web browsers natively support two-way live communication via WebSockets.

WebSocket is a web technology that provides bi-directional, full-duplex communication channels, over a single Transmission Control Protocol (TCP) socket. The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455.

WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. Because ordinary TCP connections to port numbers other than 80 are frequently blocked by administrators outside of home environments, it can be used as a way to circumvent these restrictions and provide similar functionality with some additional protocol overhead while multiplexing several WebSocket services over a single TCP port.

WebSocket is also useful for web applications that require real-time bi-directional communication. Before the implementation of WebSocket, such bi-directional communication was only possible using Comet channels; however, Comet is not trivial to implement reliably, and due to the TCP handshaking and HTTP header overhead, it may be inefficient for small messages. The WebSocket protocol aims to solve these problems without compromising the web's security assumptions.

<http://en.wikipedia.org/wiki/WebSocket>

Handling WebSockets

Until now, we were using `Action` instances to handle standard HTTP requests and send back standard HTTP responses. WebSockets are a totally different beast and can't be handled via standard `Action`.

To handle a WebSocket request, use a `WebSocket` instead of an `Action`:

```
def index = WebSocket.using[String] { request =>

    // Log events to the console
    val in = Iteratee.foreach[String](println).mapDone { _ =>
        println("Disconnected")
    }

    // Send a single 'Hello!' message
    val out = Enumerator("Hello!")

    (in, out)
}
```

A `WebSocket` has access to the request headers (from the HTTP request that initiates the `WebSocket` connection), allowing you to retrieve standard headers and session data. However, it doesn't have access to a request body, nor to the HTTP response.

When constructing a `WebSocket` this way, we must return both `in` and `out` channels.

- The `in` channel is an `Iteratee[A,Unit]` (where `A` is the message type - here we are using `String`) that will be notified for each message, and will receive `EOF` when the socket is closed on the client side.
- The `out` channel is an `Enumerator[A]` that will generate the messages to be sent to the Web client. It can close the connection on the server side by sending `EOF`.

In this example we are creating a simple iteratee that prints each message to console. To send messages, we create a simple dummy enumerator that will send a single **Hello!** message.

Tip: You can test WebSockets on <http://websocket.org/echo.html>. Just set the location to `ws://localhost:9000`.

Let's write another example that discards the input data and closes the socket just after sending the **Hello!** message:

```
def index = WebSocket.using[String] { request =>

    // Just consume and ignore the input
    val in = Iteratee.consume[String]()

    // Send a single 'Hello!' message and close
    val out = Enumerator("Hello!") >>> Enumerator.eof

    (in, out)
}
```

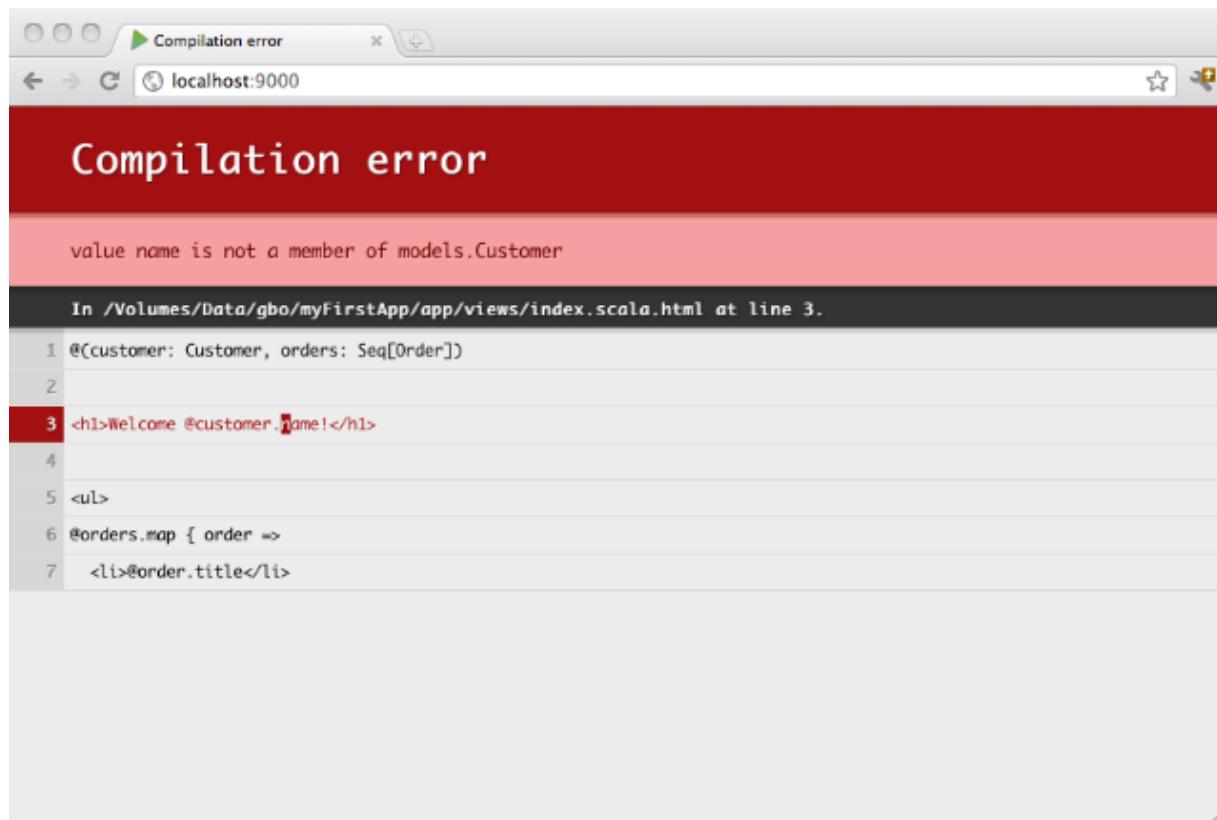
The template engine

A type safe template engine based on Scala

Play 2.0 comes with a new and really powerful Scala-based template engine. This new template engine's design was inspired by ASP.NET Razor. Specifically it is:

- **compact, expressive, and fluid**: it minimizes the number of characters and keystrokes required in a file, and enables a fast, fluid coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote server blocks within your HTML. The parser is smart enough to infer this from your code. This enables a really compact and expressive syntax which is clean, fast and fun to type.
- **easy to learn**: it enables you to quickly be productive with a minimum of concepts. You use all your existing Scala language and HTML skills.
- **not a new language**: we consciously chose not to create a new language. Instead we wanted to enable developers to use their existing Scala language skills, and deliver a template markup syntax that enables an awesome HTML construction workflow with your language of choice.
- **editable in any text editor**: it doesn't require a specific tool and enables you to be productive in any plain old text editor.

Templates are compiled, so you will see any errors right in your browser:



The screenshot shows a web browser window with the title "Compilation error". The address bar indicates the URL is "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below this, the error message "value name is not a member of models.Customer" is displayed. A dark grey bar below the message shows the full path: "In /Volumes/Data/gbo/myFirstApp/app/views/index.scala.html at line 3.". The code editor shows the following Scala code:

```
1 @(customer: Customer, orders: Seq[Order])
2
3 <h1>Welcome @customer.name!</h1>
4
5 <ul>
6   @orders.map { order =>
7     <li>@order.title</li>
```

Overview

A Play Scala template is a simple text file, that contains small blocks of Scala code. They can generate any text-based format, such as HTML, XML or CSV.

The template system has been designed to feel comfortable to those used to dealing with HTML, allowing web designers to easily work with the templates.

Templates are compiled as standard Scala functions, following a simple naming convention: If you create a `views/Application/index.scala.html` template file, it will generate a `views.html.Application.index` function.

For example, here is a simple template:

```
@(customer: Customer, orders: Seq[Order])  
  
<h1>Welcome @customer.name!</h1>  
  
<ul>  
@orders.map { order =>  
  <li>@order.title</li>  
}  
</ul>
```

You can then call this from any Scala code as you would call a function:

```
val html = views.html.Application.index(customer, orders)
```

Syntax: the magic '@' character

The Scala template uses `@` as the single special character. Every time this character is encountered, it indicates the begining of a Scala statement. It does not require you to explicitly close the code-block - this will be inferred from your code:

```
Hello @customer.name!  
^^^^^^^^^^^^^  
Scala code
```

Because the template engine automatically detects the end of your code block by analysing your code, this syntax only supports simple statements. If you want to insert a multi-token statement, explicitly mark it using brackets:

```
Hello @(customer.firstName + customer.lastName)!  
^^^^^^^^^^^^^  
Scala Code
```

You can also use curly brackets, as in plain Scala code, to write a multi-statements block:

```
Hello @{val name = customer.firstName + customer.lastName; name}!
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Scala Code
```

Because `@` is a special character, you'll sometimes need to escape it. Do this by using `@@`:

```
My email is bob@@example.com
```

Template parameters

A template is simply a function, so it needs parameters, which must be declared on the first line of the template file:

```
@(customer: models.Customer, orders: Seq[models.Order])
```

You can also use default values for parameters:

```
@(title: String = "Home")
```

Or even several parameter groups:

```
@(title:String)(body: => Html)
```

And even implicit parameters:

```
@(title: String)(body: => Html)(implicit request: play.api.mvc.Request)
```

Iterating

You can use the Scala for-comprehension, in a pretty standard way. But note that the template compiler will add a `yield` keyword before your block:

```
<ul>
@for(p <- products) {
  <li>@p.name (@$p.price)</li>
}
</ul>
```

As you probably know, this for-comprehension is just syntactic sugar for a classic map:

```
<ul>
@products.map { p =>
  <li>@p.name (@p.price)</li>
}
</ul>
```

If-blocks

If-blocks are nothing special. Simply use Scala's standard `if` statement:

```
@if(items.isEmpty) {
  <h1>Nothing to display</h1>
} else {
  <h1>@items.size items!</h1>
}
```

Pattern matching

You can also use pattern matching in your templates:

```
@connected match {

  case models.Admin(name) => {
    <span class="admin">Connected as admin (@name)</span>
  }

  case models.User(name) => {
    <span>Connected as @name</span>
  }

}
```

Declaring reusable blocks

You can create reusable code blocks:

```
@display(product: models.Product) = {  
    @product.name ($@product.price)  
}  
  
<ul>  
@products.map { p =>  
    @display(product = p)  
}  
</ul>
```

Note that you can also declare reusable pure Scala blocks:

```
@title(text: String) = @{  
    text.split(' ').map(_.capitalize).mkString(" ")  
}  
  
<h1>@title("hello world")</h1>
```

Note: Declaring Scala block this way in a template can be sometimes useful but keep in mind that a template is not the best place to write complex logic. It is often better to externalize these kind of code in a pure scala source file (that you can store under the `views/` package as well if you want).

By convention, a reusable block defined with a name starting with `implicit` will be marked as `implicit`:

```
@implicitFieldConstructor = @{ MyFieldConstructor () }
```

Declaring reusable values

You can define scoped values using the `defining` helper:

```
@defining(user.firstName + " " + user.lastName) { fullName =>  
    <div>Hello @fullName</div>  
}
```

Import statements

You can import whatever you want at the beginning of your template (or sub-template):

```
@(customer: models.Customer, orders: Seq[models.Order])  
  
@import utils._  
  
...
```

Comments

You can write server side block comments in templates using `[@* *@]`:

```
@*****  
* This is a comment *  
*****@
```

You can put a comment on the first line to document your template into the Scala API doc:

```
@*****  
* Home page.  
*  
* @param msg The message to display  
*****@  
@(msg: String)  
  
<h1>@msg</h1>
```

Escaping

By default, the dynamic content parts are escaped according the template type (e.g. HTML or XML) rules. If you want to output a raw content fragment, wrap it in the template content type.

For example to output raw HTML:

```
<p>  
  @Html(article.content)  
</p>
```

Scala templates common use cases

Templates , being simple functions, can be composed in any way you want. Below are examples of some common scenarios.

Layout

Let's declare a `views/main.scala.html` template that will act as a main layout template:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
  </body>
</html>
```

As you can see, this template takes two parameters: a title and an HTML content block. Now we can use it from another `views/Application/index.scala.html` template:

```
@main(title = "Home") {

  <h1>Home page</h1>

}
```

Note: We sometimes use named parameters (like `@main(title = "Home")`), sometimes not like `@main("Home")`. It is as you want, choose whatever is clearer in a specific context.

Sometimes you need a second page-specific content block for a sidebar or breadcrumb trail, for example. You can do this with an additional parameter:

```
@(title: String)(sidebar: Html)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="sidebar">@sidebar</section>
    <section class="content">@content</section>
  </body>
</html>
```

Using this from our ‘index’ template, we have:

```
@main("Home") {
  <h1>Sidebar</h1>

} {
  <h1>Home page</h1>

}
```

Alternatively, we can declare the sidebar block separately:

```
@sidebar = {
  <h1>Sidebar</h1>
}

@main("Home")(sidebar) {
  <h1>Home page</h1>

}
```

Tags (they are just functions, right?)

Let’s write a simple `views/tags/notice.scala.html` tag that displays an HTML notice:

```
@(level: String = "error")(body: (String) => Html)

@level match {

  case "success" => {
    <p class="success">
      @body("green")
    </p>
  }

  case "warning" => {
    <p class="warning">
      @body("orange")
    </p>
  }

  case "error" => {
    <p class="error">
      @body("red")
    </p>
  }

}
```

And now let's use it from another template:

```
@import tags._

@notice("error") { color =>
  Oops, something is <span style="color:@color">wrong</span>
}
```

Includes

Again, there's nothing special here. You can just call any other template you like (and in fact any other function coming from anywhere at all):

```
<h1>Home</h1>

<div id="side">
  @common.sideBar()
</div>
```

Handling form submission

Defining a form

The `play.api.data` package contains several helpers to handle HTTP form data submission and validation. The easiest way to handle a form submission is to define a `play.api.data.Form` structure:

```
import play.api.data._  
import play.api.data.Forms._  
  
val loginForm = Form(  
    tuple(  
        "email" -> text,  
        "password" -> text  
    )  
)
```

This form can generate a `(String, String)` result value from `Map[String, String]` data:

```
val anyData = Map("email" -> "bob@gmail.com", "password" -> "secret")  
val (user, password) = loginForm.bind(anyData).get
```

If you have a request available in the scope, you can bind directly to it from the request content:

```
val (user, password) = loginForm.bindFromRequest.get
```

Constructing complex objects

A form can use functions to construct and deconstruct the value. So you can, for example, define a form that wraps an existing case class:

```
import play.api.data._  
import play.api.data.Forms._  
  
case class User(name: String, age: Int)  
  
val userForm = Form(  
  mapping(  
    "name" -> text,  
    "age" -> number  
  )(User.apply)(User.unapply)  
)  
  
val anyData = Map("email" -> "bob@gmail.com", "age" -> "18")  
val user: User = userForm.bind(anyData).get
```

Note: The difference between using `tuple` and `mapping` is that when you are using `tuple` the construction and deconstruction functions don't need to be specified (we know how to construct and deconstruct a tuple, right?).

The `mapping` method just let you define your custom functions. When you want to construct and deconstruct a case class, you can just use its default `apply` and `unapply` functions, as they do exactly that!

Of course often the `Form` signature doesn't match the case class exactly. Let's use the example a form that contains an additional checkbox field, used to accept terms of service. We don't need to add this data to our `User` value. It's just a dummy field that is used for form validation but which doesn't carry any useful information once validated.

As we can define our own construction and deconstruction functions, it is easy to handle it:

```
val userForm = Form(  
  mapping(  
    "name" -> text,  
    "age" -> number,  
    "accept" -> checked("Please accept the terms and conditions")  
  )(  
    (name, age, _) => User(name, age),  
    (user: User) => Some(user.name, user.age, false)  
  )  
)
```

Note: The deconstruction function is used when we fill a form with an existing `User` value. This is useful if we want the load a user from the database and prepare a form to update it.

Defining constraints

For each mapping, you can also define additional validation constraints that will be checked during the binding phase:

```
import play.api.data._  
import play.api.data.Forms._  
import play.api.data.validation.Constraints._  
  
case class User(name: String, age: Int)  
  
val userForm = Form(  
  mapping(  
    "name" -> text verifying(required),  
    "age" -> number verifying(min(0), max(100))  
  )(User.apply)(User.unapply)  
)
```

Note: That can be also written:

```
mapping(  
  "name" -> nonEmptyText,  
  "age" -> number(min=0, max=100)  
)
```

This constructs the same mappings, with additional constraints.

You can also define ad-hoc constraints on the fields:

```
val loginForm = Form(  
  tuple(  
    "email" -> nonEmptyText,  
    "password" -> text  
  ) verifying("Invalid user name or password", {  
    case (e, p) => User.authenticate(e,p).isDefined  
  })  
)
```

Handling binding failure

If you can define constraints, then you need to be able to handle the binding errors. You can use the `fold` operation for this:

```
loginForm.bindFromRequest.fold(  
    formWithErrors => // binding failure, you retrieve the form containing  
    errors,  
    value => // binding success, you get the actual value  
)
```

Fill a form with initial default values

Sometimes you'll want to populate a form with existing values, typically for editing data:

```
val filledForm = userForm.fill(User("Bob", 18))
```

Nested values

A form mapping can define nested values:

```
case class User(name: String, address: Address)  
case class Address(street: String, city: String)  
  
val userForm = Form(  
    mapping(  
        "name" -> text,  
        "address" -> mapping(  
            "street" -> text,  
            "city" -> text  
        )(Address.apply)(Address.unapply)  
    )(User.apply, User.unapply)  
)
```

When you are using nested data this way, the form values sent by the browser must be named like `address.street`, `address.city`, etc.

Repeated values

A form mapping can also define repeated values:

```
case class User(name: String, emails: List[String])

val userForm = Form(
  mapping(
    "name" -> text,
    "emails" -> list(text)
  )(User.apply, User.unapply)
)
```

When you are using repeated data like this, the form values sent by the browser must be named `emails[0]`, `emails[1]`, `emails[2]`, etc.

Optional values

A form mapping can also define optional values:

```
case class User(name: String, email: Option[String])

val userForm = Form(
  mapping(
    "name" -> text,
    "email" -> optional(text)
  )(User.apply, User.unapply)
)
```

Note: The `email` field will be ignored and set to `None` if the field `email` is missing in the request payload or if it contains a blank value.

Now you can mix optional, nested and repeated mappings any way you want to create complex forms.

Using the form template helpers

Play provides several helpers for rendering form fields in HTML templates.

Create a `<form>` tag

The first thing is to be able to create the `<form>` tag. It is a pretty simple helper that has no more value than automatically setting the `action` and `method` tag parameters according to the reverse route you pass in:

```
@helper.form(action = routes.Application.submit) {  
}
```

You can also pass an extra set of parameters that will be added to the generated Html:

```
@helper.form(action = routes.Application.submit, 'id -> "myForm") {  
}
```

Rendering an `<input>` element

You can find several input helpers in the `views.html.helper` package. You feed them with a form field, and they display the corresponding HTML input, setting the value, constraints and errors:

```
@(myForm: Form[User])  
  
@helper.form(action = routes.Application.submit) {  
  
    @helper.inputText(myForm("username"))  
  
    @helper.inputPassword(myForm("password"))  
  
}
```

As for the `form` helper, you can specify an extra set of parameters that will be added to the generated Html:

```
@helper.inputText(myForm("username"), 'id -> "username", 'size -> 30)
```

Note: All extra parameters will be added to the generated Html, unless they start with the `_` character. Arguments starting with `_` are reserved for field constructor arguments (we will see that shortly).

Handling HTML input creation yourself

There is also a more generic `input` helper that lets you code the desired HTML result:

```
@helper.input(myForm("username")) { (id, name, value, args) =>
  <input type="date" name="@name" id="@id" @toHtmlArgs(args)>
}
```

Field constructors

A field rendering is not only composed of the `<input>` tag, but it also needs a `<label>` and possibly other tags used by your CSS framework to decorate the field.

All input helpers take an implicit `FieldConstructor` that handles this part. The default one (used if there are no other field constructors available in the scope), generates HTML like:

```
<dl class="error" id="username_field">
  <dt><label for="username"><label>Username :</label></dt>
  <dd><input type="text" name="username" id="username" value=""></dd>
  <dd class="error">This field is required!</dd>
  <dd class="error">Another error</dd>
  <dd class="info">Required</dd>
  <dd class="info">Another constraint</dd>
</dl>
```

This default field constructor supports additional options you can pass in the input helper arguments:

```
'_label -> "Custom label"
'_id -> "idForTheTopDlElement"
'_help -> "Custom help"
'_showConstraints -> false
'_error -> "Force an error"
'_showErrors -> false
```

Twitter bootstrap field constructor

There is also another built-in field constructor that can be used with TwitterBootstrap .

To use it, just import it in the current scope:

```
@import helper.twitterBootstrap._
```

It generates Html like:

```
<div class="clearfix error" id="username_field">
  <label for="username">Username:</label>
  <div class="input">
    <input type="text" name="username" id="username" value="">
    <span class="help-inline">This field is required!, Another error</span>
    <span class="help-block">Required, Another constraint</span>
  </div>
</div>
```

It supports the same set of options as the default field constructor (see below).

Writing your own field constructor

Often you will need to write your own field constructor. Start by writing a template like:

```
@(elements: helper.FieldElements)

<div class="@if(elements.hasErrors) {error}">
  <label for="@elements.id">@elements.label</label>
  <div class="input">
    @elements.input
    <span class="errors">@elements.errors.mkString(", ")</span>
    <span class="help">@elements.infos.mkString(", ")</span>
  </div>
</div>
```

Note: This is just a sample. You can make it as complicated as you need. You also have access to the original field using `@elements.field`.

Now create a `FieldConstructor` using this template function:

```
object MyHelpers {

  implicit val myFields = FieldConstructor(myFieldConstructorTemplate.f)

}
```

And to make the form helpers use it, just import it in your templates:

```
@import MyHelpers._

@inputText(myForm("username"))
```

It will then use your field constructor to render the input text.

Note: You can also set an implicit value for your `FieldConstructor` inline in your template this way:

```
@implicitField = @{ FieldConstructor(myFieldConstructorTemplate.f) }

@inputText(myForm("username"))
```

Handling repeated values

The last helper makes it easier to generate inputs for repeated values. Let's say you have this kind of form definition:

```
val myForm = Form(
  tuple(
    "name" -> text,
    "emails" -> list(email)
  )
)
```

Now you have to generate as many inputs for the `emails` field as the form contains. Just use the `repeat` helper for that:

```
@inputText(myForm("name"))

@repeat(myForm("emails"), min = 1) { emailField =>

  @inputText(emailField)

}
```

The `min` parameter allows you to display a minimum number of fields even if the corresponding form data are empty.

The Play JSON library

Overview

The recommend way of dealing with JSON is using Play's typeclass based JSON library, located at `play.api.libs.json`.

This library is built on top of Jerkson, which is a Scala wrapper around the super-fast Java based JSON library, Jackson.

The benefit of this approach is that both the Java and the Scala side of Play can share the same underlying library (Jackson), while Scala users can enjoy the extra type safety that Play's JSON support brings to the table.

`play.api.libs.json` package contains seven JSON data types:

- `JsObject`
- `JsNull`
- `JsUndefined`
- `JsBoolean`
- `JsNumber`
- `JsArray`
- `JsString`

All of them inherit from the generic JSON value, `JsValue`.

Parsing a Json String

You can easily parse any JSON string as a `JsValue`:

```
val json: JsValue = Json.parse(jsonString)
```

Navigating into a Json tree

As soon as you have a `JsValue` you can navigate into the tree. The API looks like the one provided to navigate into XML document by Scala using `NodeSeq`:

```
val json = Json.parse(jsonString)

val maybeName = (json \ "user" \ name).asOpt[String]
val emails = (json \ "user" \\ "emails").map(_.as[String])
```

Note that navigating using `and\`` never fails. You must handle the error case at the end using `asOpt[T]` that will return `None` if the value is missing. Otherwise you can use `as[T]` that will fail with an exception if the value was missing.

Converting a Scala value to Json

As soon as you have a type class able to transform the Scala type to Json, it is pretty easy to generate any Scala value to Json. For example let's create a simple Json object:

```
val jsonNumber = Json.toJson(4)
```

Or create a json array:

```
val jsonArray = Json.toJson(Seq(1, 2, 3, 4))
```

Here we have no problem to convert a `Seq[Int]` into a Json array. However it is more complicated if the `Seq` contains heterogeneous values:

```
val jsonArray = Json.toJson(Seq(1, "Bob", 3, 4))
```

Because there is no way to convert a `Seq[Any]` to Json (`Any` could be anything including something not supported by Json right?)

A simple solution is to handle it as a `Seq[JsValue]`:

```
val jsonArray = Json.toJson(Seq(
  toJson(1), toJson("Bob"), toJson(3), toJson(4)
))
```

Now let's see a last example of creating a more complex Json object:

```
val jsonObject = Json.toJson(  
  Map(  
    "users" -> Seq(  
      toJson(  
        Map(  
          "name" -> toJson("Bob"),  
          "age" -> toJson(31),  
          "email" -> toJson("bob@gmail.com")  
        )  
      ),  
      toJson(  
        Map(  
          "name" -> toJson("Kiki"),  
          "age" -> toJson(25),  
          "email" -> JsNull  
        )  
      )  
    )  
  )
```

That will generate this Json result:

```
{  
  "users": [  
    {  
      "name": "Bob",  
      "age": 31.0,  
      "email": "bob@gmail.com"  
    },  
    {  
      "name": "Kiki",  
      "age": 25.0,  
      "email": null  
    }  
  ]  
}
```

Serializing Json

Serializing a `JsValue` to its json String representation is easy:

```
val jsonString: String = Json.stringify(jsValue)
```

Other options

While the typeclass based solution described above is the one that's recommended, nothing is stopping users from using any other JSON libraries if needed.

For example, here is a small snippet which demonstrates how to marshal plain Scala objects into JSON and send it over the wire using the bundled, reflection based Jerkson library:

```
import com.codahale.jerkson.Json._

val json = generate(
  Map(
    "url" -> "http://nytimes.com",
    "attributes" -> Map(
      "name" -> "nytimes",
      "country" -> "US",
      "id" -> 25
    ),
    "links" -> List(
      "http://link1",
      "http://link2"
    )
  )
)
```

Handling and serving JSON requests

Handling a JSON request

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

By default an `Action` uses an **any content** body parser, which lets you retrieve the body as JSON (actually as a `JsValue`):

```
def sayHello = Action { request =>
    request.body.asJson.map { json =>
        (json \ "name").asOpt[String].map { name =>
            Ok("Hello " + name)
        }.getOrElse {
            BadRequest("Missing parameter [name]")
        }
    }.getOrElse {
        BadRequest("Expecting Json data")
    }
}
```

It's better (and simpler) to specify our own `BodyParser` to ask Play to parse the content body directly as JSON:

```
def sayHello = Action(parse.json) { request =>
    (request.body \ "name").asOpt[String].map { name =>
        Ok("Hello " + name)
    }.getOrElse {
        BadRequest("Missing parameter [name]")
    }
}
```

Note: When using a JSON body parser, the `request.body` value is directly a valid `JsValue`.

You can test it with **cURL** from the command line:

```
curl
--header "Content-type: application/json"
--request POST
--data '{"name": "Guillaume"}'
http://localhost:9000/sayHello
```

It replies with:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 15

Hello Guillaume
```

Serving a JSON response

In our previous example we handle a JSON request, but we reply with a `text/plain` response. Let's change that to send back a valid JSON HTTP response:

```
def sayHello = Action(parse.json) { request =>
  (request.body \ "name").asOpt[String].map { name =>
    Ok(toJson(
      Map("status" -> "OK", "message" -> ("Hello " + name)))
  ))
  .getOrElse {
    BadRequest(toJson(
      Map("status" -> "KO", "message" -> "Missing parameter [name]")))
  }
}
```

Now it replies with:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 43

{"status":"OK","message":"Hello Guillaume"}
```

Handling and serving XML requests

Handling an XML request

An XML request is an HTTP request using a valid XML payload as the request body. It must specify the `text/xml` MIME type in its `Content-Type` header.

By default an `Action` uses a **any content** body parser, which lets you retrieve the body as XML (actually as a `NodeSeq`):

```
def sayHello = Action { request =>
    request.body.asXml.map { xml =>
        (xml \\ "name" headOption).map(_.text).map { name =>
            Ok("Hello " + name)
        }.getOrElse {
            BadRequest("Missing parameter [name]")
        }
    }.getOrElse {
        BadRequest("Expecting Xml data")
    }
}
```

It's way better (and simpler) to specify our own `BodyParser` to ask Play to parse the content body directly as XML:

```
def sayHello = Action(parse.xml) { request =>
    (request.body \\ "name" headOption).map(_.text).map { name =>
        Ok("Hello " + name)
    }.getOrElse {
        BadRequest("Missing parameter [name]")
    }
}
```

Note: When using an XML body parser, the `request.body` value is directly a valid `NodeSeq`.

You can test it with **cURL** from a command line:

```
curl
--header "Content-type: text/xml"
--request POST
--data '<name>Guillaume</name>'
http://localhost:9000/sayHello
```

It replies with:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 15

Hello Guillaume
```

Serving an XML response

In our previous example we handle an XML request, but we reply with a `text/plain` response. Let's change that to send back a valid XML HTTP response:

```
def sayHello = Action(parse.xml) { request =>
  (request.body \\"name" headOption).map(_.text).map { name =>
    Ok(<message status="OK">Hello {name}</message>)
  }.getOrElse {
    BadRequest(<message status="K0">Missing parameter [name]</message>)
  }
}
```

Now it replies with:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 46

<message status="OK">Hello Guillaume</message>
```

Handling file upload

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 lets you mix standard form data with file attachment data.

Start by writing an HTML form:

```
@form(action = routes.Application.upload, 'enctype -> "multipart/form-data") {  
  
    <input type="file" name="picture">  
  
    <p>  
        <input type="submit">  
    </p>  
  
}
```

Now define the `upload` action using a `multipartFormData` body parser:

```
def upload = Action(parse.multipartFormData) { request =>  
    request.body.file("picture").map { picture =>  
        import java.io.File  
        val filename = picture.filename  
        val contentType = picture.contentType  
        picture.ref.moveTo(new File("/tmp/picture"))  
        Ok("File uploaded")  
    }.getOrElse {  
        Redirect(routes.Application.index).flashing(  
            "error" -> "Missing file"  
        )  
    }  
}
```

The `ref` attribute give you a reference to a `TemporaryFile`. This is the default way the `multipartFormData` parser handles file upload.

Note: As always, you can also use the `anyContent` body parser and retrieve it as `request.asMultipartFormData`.

Direct file upload

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

In this case we can just use a body parser to store the request body content in a file. For this example, let's use the `temporaryFile` body parser:

```
def upload = Action(parse.temporaryFile) { request =>
    request.body.moveTo(new File("/tmp/picture"))
    Ok("File uploaded")
}
```

Writing your own body parser

If you want to handle the file upload directly without buffering it in a temporary file, you can just write your own `BodyParser`. In this case, you will receive chunks of data that you are free to push anywhere you want.

If you want to use `multipart/form-data` encoding, you can still use the default `multipartFormData` parser by providing your own `PartHandler[FilePart[A]]`. You receive the part headers, and you have to provide an `Iteratee[Array[Byte], FilePart[A]]` that will produce the right `FilePart`.

Accessing an SQL database

Configuring JDBC connection pools

Play 2.0 provides a plug-in for managing JDBC connection pools. You can configure as many databases you need.

To enable the database plug-in, configure a connection pool in the `conf/application.conf` file. By convention, the default JDBC data source must be called `default`:

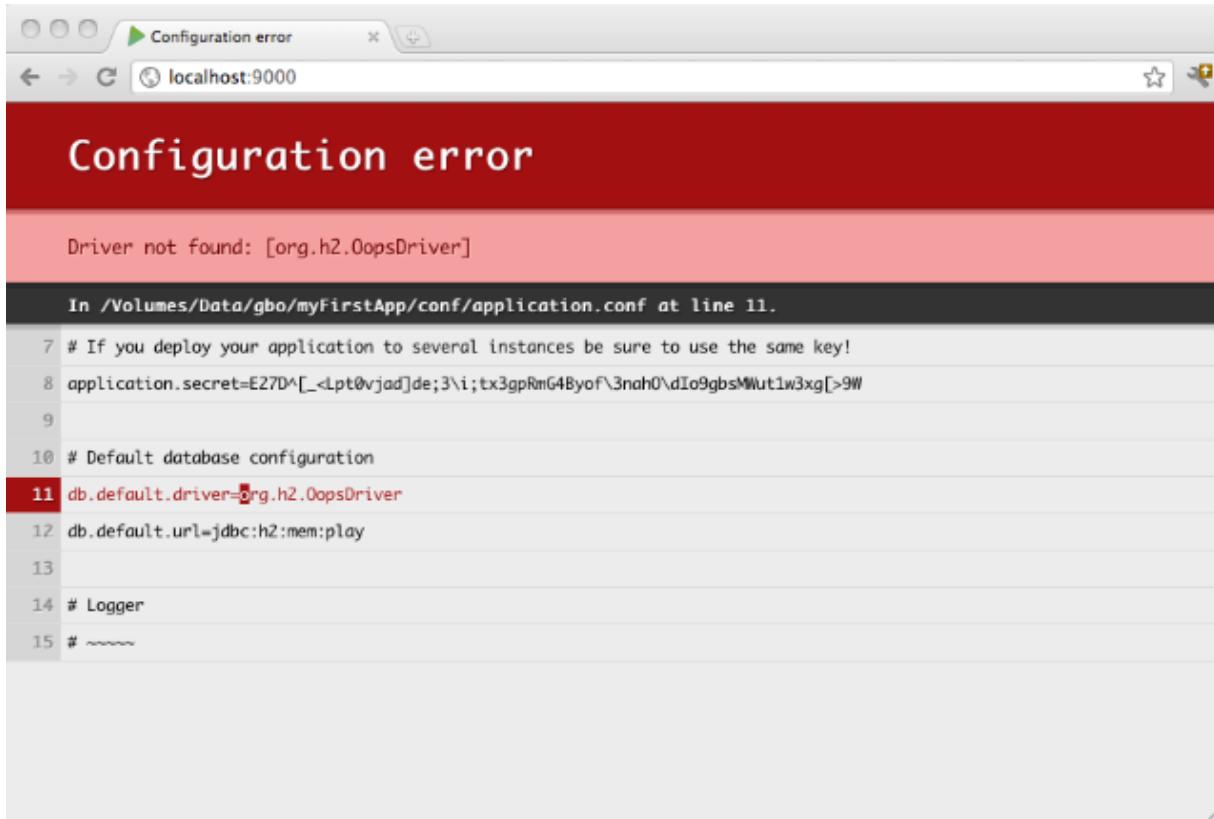
```
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url=jdbc:h2:mem:play
```

To configure several data sources:

```
# Orders database
db.orders.driver=org.h2.Driver
db.orders.url=jdbc:h2:mem:orders

# Customers database
db.customers.driver=org.h2.Driver
db.customers.url=jdbc:h2:mem:customers
```

If something isn't properly configured you will be notified directly in your browser:



Configuring the JDBC Driver

Other than h2 for in-memory database, useful mostly in development mode, Play 2.0 does not provide any database driver. Consequently, to deploy in production you will need to add your database driver as a dependency.

For example, if you use MySQL5, you need to add a dependency for the connector:

```
val appDependencies = Seq(
  "mysql" % "mysql-connector-java" % "5.1.18"
)
```

Accessing the JDBC datasource

The `play.api.db` package provides access to the configured data sources:

```
import play.api.db._

val ds = DB.getDatasource()
```

Obtaining a JDBC connection

There is several ways to retrieve a JDBC connection. The first is the most simple:

```
val connection = DB.getConnection()
```

But of course you need to call `close()` at some point on the opened connection to return it to the connection pool. Another way is to let Play manage closing the connection for you:

```
DB.withConnection { conn =>
    // do whatever you need with the connection
}
```

The connection will be automatically closed at the end of the block.

Tip: Each `Statement` and `ResultSet` created with this connection will be closed as well.

A variant is to set the connection auto-commit to `false` automatically and to manage a transaction for the block:

```
DB.withTransaction { conn =>
    // do whatever you need with the connection
}
```

Anorm, simple SQL data access

Play includes a simple data access layer called Anorm that uses plain SQL to interact with the database and provides an API to parse and transform the resulting datasets.

Anorm is Not an Object Relational Mapper

In the following documentation, we will use the MySQL world sample database .

If you want to enable it for your application, follow the MySQL website instructions, and enable it for your application by adding the following configuration line in your `conf/application.conf` file:

```
db.default.driver= com.mysql.jdbc.Driver  
db.default.url="jdbc:mysql://localhost/world"  
db.default.user=root  
db.default.password=secret
```

Overview

It can feel strange to return to plain old SQL to access an SQL database these days, especially for Java developers accustomed to using a high-level Object Relational Mapper like Hibernate to completely hide this aspect.

Although we agree that these tools are almost required in Java, we think that they are not needed at all when you have the power of a higher-level programming language like Scala. On the contrary, they will quickly become counter-productive .

Using JDBC is a pain, but we provide a better API

We agree that using the JDBC API directly is tedious, particularly in Java. You have to deal with checked exceptions everywhere and iterate over and over around the ResultSet to transform this raw dataset into your own data structure .

We provide a simpler API for JDBC; using Scala you don't need to bother with exceptions, and transforming data is really easy with a functional language. In fact, the goal of the Play Scala SQL access layer is to provide several APIs to effectively transform JDBC data into other Scala structures .

You don't need another DSL to access relational databases

SQL is already the best DSL for accessing relational databases. We don't need to invent something new. Moreover the SQL syntax and features can differ from one database vendor to another.

If you try to abstract this point with another proprietary SQL like DSL you will have to deal with several 'dialects' dedicated for each vendor (like Hibernate ones), and limit yourself by not using a particular database's interesting features .

Play will sometimes provide you with pre-filled SQL statements, but the idea is not to hide the fact that we use SQL under the hood. Play just saves typing a bunch of characters for trivial queries, and you can always fall back to plain old SQL.

A type safe DSL to generate SQL is a mistake

Some argue that a type safe DSL is better since all your queries are checked by the compiler. Unfortunately the compiler checks your queries based on a meta-model definition that you often write yourself by ‘mapping’ your data structure to the database schema.

There are no guarantees that this meta-model is correct. Even if the compiler says that your code and your queries are correctly typed, it can still miserably fail at runtime because of a mismatch in your actual database definition.

Take Control of your SQL code

Object Relational Mapping works well for trivial cases, but when you have to deal with complex schemas or existing databases, you will spend most of your time fighting with your ORM to make it generate the SQL queries you want.

Writing SQL queries yourself can be tedious for a simple ‘Hello World’ application, but for any real-life application, you will eventually save time and simplify your code by taking full control of your SQL code.

Executing SQL queries

To start you need to learn how to execute SQL queries.

First, import `anorm._`, and then simply use the `SQL` object to create queries. You need a `Connection` to run a query, and you can retrieve one from the `play.api.db.DB` helper:

```
import anorm._

DB.withConnection { implicit c =>
  val result: Boolean = SQL("Select 1").execute()
}
```

The `execute()` method returns a Boolean value indicating whether the execution was successful.

To execute an update, you can use `executeUpdate()`, which returns the number of rows updated.

```
val result: Int = SQL("delete from City where id = 99").executeUpdate()
```

Since Scala supports multi-line strings, feel free to use them for complex SQL statements:

```
val sqlQuery = SQL(
  """
    select * from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = 'FRA';
  """
)
```

If your SQL query needs dynamic parameters, you can declare placeholders like `{name}` in the query string, and later assign them a value:

```
SQL(
  """
    select * from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = {countryCode};
  """
).on("countryCode" -> "FRA")
```

Retrieving data using the Stream API

The first way to access the results of a select query is to use the Stream API.

When you call `apply()` on any SQL statement, you will receive a lazy `Stream` of `Row` instances, where each row can be seen as a dictionary:

```
// Create an SQL query
val selectCountries = SQL("Select * from Country")

// Transform the resulting Stream[Row] as a List[(String, String)]
val countries = selectCountries().map(row =>
  row[String]("code") -> row[String]("name")
).toList
```

In the following example we will count the number of `Country` entries in the database, so result set will be a single row with a single column:

```
// First retrieve the first row
val firstRow = SQL("Select count(*) as c from Country").apply().head

// Next get the content of the 'c' column as Long
val countryCount = firstRow[Long]("c")
```

Using Pattern Matching

You can also use Pattern Matching to match and extract the `Row` content. In this case the column name doesn't matter. Only the order and the type of the parameters is used to match.

The following example transform each row to the correct Scala type:

```
case class SmallCountry(name:String)
case class BigCountry(name:String)
case class France

val countries = SQL("Select name,population from Country")().collect {
  case Row("France", _) => France()
  case Row(name:String, pop:Int) if(pop > 1000000) => BigCountry(name)
  case Row(name:String, _) => SmallCountry(name)
}
```

Note that since `collect(...)` ignores the cases where the partial function isn't defined, it allows your code to safely ignore rows that you don't expect.

Dealing with Nullable columns

If a column can contain `Null` values in the database schema, you need to manipulate it as an `Option` type.

For example, the `indepYear` of the `Country` table is nullable, so you need to match it as `Option[Int]`:

```
SQL("Select name,indepYear from Country")().collect {
  case Row(name:String, Some(year:Int)) => name -> year
}
```

If you try to match this column as `Int` it won't be able to parse `Null` cases. Suppose you try to retrieve the column content as `Int` directly from the dictionary:

```
SQL("Select name,indepYear from Country")().map { row =>
  row[String]("name") -> row[Int]("indepYear")
}
```

This will produce an `UnexpectedNullableFound (COUNTRY.INDEPYEAR)` exception if it encounters a null value, so you need to map it properly to an `Option[Int]`, as:

```
SQL("Select name,indepYear from Country")().map { row =>
  row[String]("name") -> row[Option[Int]]("indepYear")
}
```

This is also true for the parser API, as we will see next.

Using the Parser API

You can use the parser API to create generic and reusable parsers that can parse the result of any select query.

Note: This is really useful, since most queries in a web application will return similar data sets.

For example, if you have defined a parser able to parse a `Country` from a result set, and another `Language` parser, you can then easily compose them to parse both `Country` and `Language` from a join query.

First you need to `import anorm.SqlParser._`

First you need a `RowParser`, i.e. a parser able to parse one row to a Scala value. For example we can define a parser to transform a single column result set row, to a Scala `Long`:

```
val rowParser = scalar[Long]
```

Then we have to transform it into a `ResultSetParser`. Here we will create a parser that parse a single row:

```
val rsParser = scalar[Long].single
```

So this parser will parse a result set to return a `Long`. It is useful to parse to result produced by a simple SQL `select count` query:

```
val count: Long = SQL("select count(*) from Country").as(scalar[Long].single)
```

Let's write a more complicated parser:

`str("name") ~ int("population")`, will create a `RowParser` able to parse a row containing a String `name` column and an Integer `population` column. Then we can create a `ResultSetParser` that will parse as many rows of this kind as it can, using `*`:

```
val populations :List[String~Int] = {
  SQL("select * from Country").as( str("name") ~ int("population") * )
}
```

As you see, this query's result type is `List[String~Int]` - a list of country name and population items.

You can also rewrite the same code as:

```
val result:List[String~Int] = {
  SQL("select * from Country").as(get[String]("name")~get[Int]("population")
*)
}
```

Now what about the `String~Int` type? This is an **Anorm** type that is not really convenient to use outside of your database access code. You would want have a simple tuple `(String, Int)` instead. You can use the `map` function on a `RowParser` to transform its result to a more convenient type:

```
str("name") ~ int("population") map { case n~p => (n,p) }
```

Note: We created a tuple `(String,Int)` here, but there is nothing stopping you from transforming the `RowParser` result to any other type, such as a custom case class.

Now, because transforming `A~B~C` types to `(A,B,C)` is a common task, we provide a `flatten` function that does exactly that. So you finally write:

```
val result:List[(String,Int)] = {
  SQL("select * from Country").as(
    str("name") ~ int("population") map(flatten) *
  )
}
```

Now let's try with a more complicated example. How to parse the result of the following query to retrieve the country name and all spoken languages for a country code?

```
select c.name, l.language from Country c
join CountryLanguage l on l.CountryCode = c.Code
where c.code = 'FRA'
```

Let's start by parsing all rows as a `List[(String, String)]` (a list of name,language tuple):

```
var p: ResultSetParser[List[(String, String)]] = {
  str("name") ~ str("language") map(flatten) *
}
```

Now we get this kind of result:

```
List(  
  ("France", "Arabic"),  
  ("France", "French"),  
  ("France", "Italian"),  
  ("France", "Portuguese"),  
  ("France", "Spanish"),  
  ("France", "Turkish")  
)
```

We can then use the Scala collection API, to transform it to the expected result:

```
case class SpokenLanguages (country:String, languages:Seq[String])  
  
languages.headOption.map { f =>  
  SpokenLanguages (f._1, languages.map(_.._2))  
}
```

Finally, we get this convenient function:

```
case class SpokenLanguages (country:String, languages:Seq[String])  
  
def spokenLanguages (countryCode: String): Option[SpokenLanguages] = {  
  val languages: List[(String, String)] = SQL(  
    """  
      select c.name, l.language from Country c  
      join CountryLanguage l on l.CountryCode = c.Code  
      where c.code = {code};  
    """  
  )  
  .on("code" -> countryCode)  
  .as(str("name") ~ str("language") map(flatten) *)  
  
  languages.headOption.map { f =>  
    SpokenLanguages (f._1, languages.map(_.._2))  
  }  
}
```

To continue, let's complicate our example to separate the official language from the others:

```
case class SpokenLanguages (
    country:String,
    officialLanguage : Option[String],
    otherLanguages :Seq[String]
)

def spokenLanguages (countryCode: String): Option[SpokenLanguages] = {
    val languages: List[(String, String, Boolean)] = SQL(
        """
            select * from Country c
            join CountryLanguage l on l.CountryCode = c.Code
            where c.code = {code};
        """
    )
    .on("code" -> countryCode)
    .as {
        str("name") ~ str("language") ~ str("isOfficial") map {
            case n~l~"T" => (n,l,true)
            case n~l~"F" => (n,l,false)
        } *
    }
}

languages.headOption.map { f =>
    SpokenLanguages (
        f._1,
        languages.find(_.._3).map(_.._2),
        languages.filterNot(_.._3).map(_.._2)
    )
}
}
```

If you try this on the world sample database, you will get:

```
$ spokenLanguages ("FRA")
> Some(
  SpokenLanguages (France,Some(French),List(
    Arabic, Italian, Portuguese, Spanish, Turkish
  )))
)
```

Integrating with other database libraries

You can use any **SQL** database access library you like with Play, as can easily retrieve either a `Connection` or a `Datasource` from the `play.api.db.DB` helper.

Integrating with ScalaQuery

From here you can integrate any JDBC access layer that needs a JDBC data source. For example, to integrate with ScalaQuery :

```
import play.api.db._
import play.api.Play.current

import org.scalaquery.ql._
import org.scalaquery.ql.TypeMapper._
import org.scalaquery.ql.extended.{ExtendedTable => Table}

import org.scalaquery.ql.extended.H2Driver.Implicit._

import org.scalaquery.session._

object Task extends Table[(Long, String, Date, Boolean)]("tasks") {

    lazy val database = Database.forDataSource(DB.getDataSource())

    def id = column[Long]("id", 0 PrimaryKey, 0 AutoInc)
    def name = column[String]("name", 0 NotNull)
    def dueDate = column[Date]("due_date")
    def done = column[Boolean]("done")
    def * = id ~ name ~ dueDate ~ done

    def findAll = database.withSession { implicit db:Session =>
        (for(t <- this) yield t.id ~ t.name).list
    }
}
```

Exposing the datasource through JNDI

Some libraries expect to retrieve the `Datasource` reference from JNDI. You can expose any Play managed datasource via JNDI by adding this configuration in `conf/application.conf` :

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.jndiName=DefaultDS
```

The Play cache API

The default implementation of the Cache API uses EHCache. You can also provide your own implementation via a plug-in.

Accessing the Cache API

The cache API is provided by the `play.api.cache.Cache` object. It requires a registered cache plug-in.

Note: The API is intentionally minimal to allow several implementation to be plugged. If you need a more specific API, use the one provided by your Cache plugin.

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

```
Cache.set("item.key", connectedUser)
```

And then retrieve it later:

```
val maybeUser: Option[User] = Cache.getAs[User]("item.key")
```

There is also a convenient helper to retrieve from cache or set the value in cache if it was missing:

```
val user: User = Cache.getOrElseAs[User]("item.key") {  
    User.findById(connectedUser)  
}
```

Caching HTTP responses

You can easily create smart cached actions using standard Action composition.

Note: Play HTTP `Result` instances are safe to cache and reuse later.

Play provides a default built-in helper for standard cases:

```
def index = Cached("homePage") {  
    Action {  
        Ok("Hello world")  
    }  
}
```

Or even:

```
def userProfile = Authenticated { user =>
    Cached(req => "profile." + user) {
        Action {
            Ok(views.html.profile(User.find(user)))
        }
    }
}
```

The Play WS API

Sometimes we would like to call other HTTP services from within a Play application. Play supports this via its `play.api.libs.ws.WS` library, which provides a way to make asynchronous HTTP calls.

Any calls made by `play.api.libs.ws.WS` should return a `Promise[play.api.libs.ws.Response]` which we can later handle with Play's asynchronous mechanisms.

Making an HTTP call

To send an HTTP request you start with `WS.url()` to specify the URL. Then you get a builder that you can use to specify various HTTP options, such as setting headers. You end by calling a final method corresponding to the HTTP method you want to use. For example:

```
val homePage: Promise[ws.Response] = WS.url("http://mysite.com").get()
```

Or:

```
val result: Promise[ws.Response] = {
    WS.url("http://localhost:9001/post").post("content")
}
```

Retrieving the HTTP response result

The call is asynchronous and you need to manipulate it as a `Promise[ws.Response]` to get the actual content. You can compose several promises and end with a `Promise[Result]` that can be handled directly by the Play server:

```
def feedTitle(feedUrl: String) = Action {
    Async {
        WS.url(feedUrl).get().map { response =>
            Ok("Feed title: " + (response.json \ "title").as[String])
        }
    }
}
```

OpenID Support in Play

OpenID is a protocol for users to access several services with a single account. As a web developer, you can use OpenID to offer users a way to log in using an account they already have, such as their Google account . In the enterprise , you may be able to use OpenID to connect to a company's SSO server.

The OpenID flow in a nutshell

1. The user gives you his OpenID (a URL).
2. Your server inspects the content behind the URL to produce a URL where you need redirect the user.
3. The user confirms the authorization on his OpenID provider, and gets redirected back your server.
4. Your server receives information from that redirect, and checks with the provider that the information is correct.

Step 1 may be omitted if all your users are using the same OpenID provider (for example if you decide to rely completely on Google accounts).

OpenID in Play

The OpenID API has two important functions:

- `OpenID.redirectURL` calculates the URL where you should redirect the user. It involves fetching the user's OpenID page, this is why it returns a `Promise[String]` rather than a `String`. If the OpenID is invalid, the returned `Promise` will be a `Thrown`.
- `OpenID.verifiedId` needs an implicit `Request`, and inspects it to establish the user information, including his verified OpenID. It will do a call to the OpenID server to check the authenticity of the information, this is why it returns a `Promise[UserInfo]` rather than just `UserInfo`. If the information is not correct or if the server check is false (for example if the redirect URL has been forged), the returned `Promise` will be a `Thrown`.

In any case, when the `Promise` you get is a `Thrown`, you should look at the `Throwable` and redirect back the user to the login page with relevant information.

Here is an example of usage (from a controller):

```
def login = Action {
    Ok(views.html.login())
}

def loginPost = Action { implicit request =>
    Form(single(
        "openid" -> nonEmptyText
    )).bindFromRequest.fold(
        error => {
            Logger.info("bad request " + error.toString)
            BadRequest(error.toString)
        },
        {
            case (openid) => AsyncResult(OpenID.redirectURL(openid,
                routes.Application.openIDCallback.absoluteURL()))
                .extend(_.value match {
                    case Redeemed(url) => Redirect(url)
                    case Thrown(t) => Redirect(routes.Application.login)
                }))
        }
    )
}

def openIDCallback = Action { implicit request =>
    AsyncResult(
        OpenID.verifiedId.extend(_.value match {
            case Redeemed(info) => Ok(info.id + "\n" + info.attributes)
            case Thrown(t) => {
                // Here you should look at the error, and give feedback to the user
                Redirect(routes.Application.login)
            }
        })
    )
}
```

Extended Attributes

The OpenID of a user gives you his identity. The protocol also supports getting extended attributes such as the e-mail address, the first name, or the last name.

You may request *optional* attributes and/or *required* attributes from the OpenID server. Asking for required attributes means the user cannot login to your service if he doesn't provides them.

Extended attributes are requested in the redirect URL:

```
OpenID.redirectURL(  
    openid,  
    routes.Application.openIDCallback.absoluteURL(),  
    Seq("email" -> "http://schema.openid.net/contact/email")  
)
```

Attributes will then be available in the `UserInfo` provided by the OpenID server.

OAuth

OAuth is a simple way to publish and interact with protected data. It's also a safer and more secure way for people to give you access. For example, it can be used to access your users' data on Twitter.

There are 2 very different versions of OAuth: OAuth 1.0 and OAuth 2.0. Version 2 is simple enough to be implemented easily without library or helpers, so Play only provides support for OAuth 1.0.

Required Information

OAuth requires you to register your application to the service provider. Make sure to check the callback URL that you provide, because the service provider may reject your calls if they don't match. When working locally, you can use `/etc/hosts` to fake a domain on your local machine.

The service provider will give you:

- Application ID
- Secret key
- Request Token URL
- Access Token URL
- Authorize URL

Authentication Flow

Most of the flow will be done by the Play library.

1. Get a request token from the server (in a server-to-server call)
2. Redirect the user to the service provider, where he will grant your application rights to use his data
3. The service provider will redirect the user back, giving you a /verifier/
4. With that verifier, exchange the /request token/ for an /access token/ (server-to-server call)

Now the /access token/ can be passed to any call to access protected data.

Example

```
object Twitter extends Controller {

    val KEY = ConsumerKey("xxxxx", "xxxxx")

    val TWITTER = OAuth(ServiceInfo(
        "https://api.twitter.com/oauth/request_token",
        "https://api.twitter.com/oauth/access_token",
        "https://api.twitter.com/oauth/authorize", KEY),
        false)

    def authenticate = Action { request =>
        request.queryString.get("oauth_verifier").flatMap(_.headOption).map
        { verifier =>
            val tokenPair = sessionTokenPair(request).get
            // We got the verifier; now get the access token, store it and back
            to index
            TWITTER.retrieveAccessToken(tokenPair, verifier) match {
                case Right(t) => {
                    // We received the authorized tokens in the OAuth object - store
                    it before we proceed
                    Redirect(routes.Application.index).withSession("token" ->
t.token, "secret" -> t.secret)
                }
                case Left(e) => throw e
            }
        }.getOrElse(
            TWITTER.retrieveRequestToken("http://localhost:9000/auth") match {
                case Right(t) => {
                    // We received the unauthorized tokens in the OAuth object -
                    store it before we proceed
                    Redirect(TWITTER.redirectUrl(t.token)).withSession("token" ->
t.token, "secret" -> t.secret)
                }
                case Left(e) => throw e
            })
    }

    def sessionTokenPair(implicit request: RequestHeader): Option
    [RequestToken] = {
        for {
            token <- request.session.get("token")
            secret <- request.session.get("secret")
        } yield {
            RequestToken(token, secret)
        }
    }
}
```

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. For fault-tolerance it adopts the ‘Let it crash’ model, which has been used with great success in the telecoms industry to build applications that self-heal - systems that never stop. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

The application actor system

Akka 2.0 can work with several containers called `ActorSystems`. An actor system manages the resources it is configured to use in order to run the actors which it contains.

A Play application defines a special actor system to be used by the application. This actor system follows the application life-cycle and restarts automatically when the application restarts.

Note: Nothing prevents you from using another actor system from within a Play application. The provided default is convenient if you only need to start a few actors without bothering to set-up your own actor system.

You can access the default application actor system using the `play.api.libs.concurrent.Akka` helper:

```
val myActor = Akka.system.actorOf(Props[MyActor], name = "myactor")
```

Configuration

The default actor system configuration is read from the Play application configuration file. For example, to configure the default dispatcher of the application actor system, add these lines to the `conf/application.conf` file:

```
akka.default-dispatcher.core-pool-size-max = 64  
akka.debug.receive = on
```

Note: You can also configure any other actor system from the same file; just provide a top configuration key.

Converting Akka `Future` to Play `Promise`

When you interact asynchronously with an Akka actor we will get `Future` object. You can easily convert it to a Play `Promise` using the implicit conversion provided in `play.libs.Akka._`:

```
def index = Action {  
    Async {  
        (myActor ? "hello").mapTo[String].asPromise.map { response =>  
            Ok(response)  
        }  
    }  
}
```

Executing a block of code asynchronously

A common use case within Akka is to have some computation performed concurrently, without needing the extra utility of an Actor. If you find yourself creating a pool of Actors for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
def index = Action {  
    Async {  
        Akka.future { longComputation() }.map { result =>  
            Ok("Got " + result)  
        }  
    }  
}
```

Scheduling asynchronous tasks

You can schedule sending messages to actors and executing tasks (functions or `Runnable`). 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:

```
Akka.system.scheduler.schedule(0 seconds, 30 minutes, testActor, "tick")
```

Note: This example uses implicit conversions defined in `akka.util.duration` to convert numbers to `Duration` objects with various time units.

Similarly, to run a block of code ten seconds from now:

```
Akka.system.scheduler.scheduleOnce(10 seconds) {  
    file.delete()  
}
```

Messages and internationalization

Specifying languages supported by your application

A valid language code is specified by a valid **ISO 639-2 language code**, optionally followed by a valid **ISO 3166-1 alpha-2 country code**, such as `fr` or `en-US`.

To start you need to specify the languages supported by your application in the `conf/application.conf` file:

```
application.langs=en, en-US, fr
```

Externalizing messages

You can externalize messages in the `conf/messages.xxx` files.

The default `conf/messages` file matches all languages. Additionally you can specify language-specific message files such as `conf/messages.fr` or `conf/messages.en-US`.

You can then retrieve messages using the `play.api.i18n.Messages` object:

```
val title = Messages("home.title")
```

All internationalization API calls take an implicit `play.api.i18.Lang` argument retrieved from the current scope. You can also specify it explicitly:

```
val title = Messages("home.title")(Lang("fr"))
```

Note: If you have an implicit `Request` in the scope, it will provide an implicit `Lang` value corresponding to the preferred language extracted from the `Accept-Language` header and matching one the application supported languages.

Messages format

Messages can be formatted using the `java.text.MessageFormat` library. For example, assuming you have message defined like:

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

You can then specify parameters as:

```
Messages("files.summary", d.files.length, d.name)
```

Retrieving supported language from an HTTP request

You can retrieve the languages supported by a specific HTTP request:

```
def index = Action { request =>
  Ok("Languages: " + request.acceptLanguages.map(_.code).mkString(", "))
}
```

Application global settings

The Global object

Defining a `Global` object in your project allows you to handle global settings for your application. This object must be defined in the default (empty) package.

```
import play.api._  
  
object Global extends GlobalSettings {  
  
}
```

Tip: You can also specify a custom `GlobalSettings` implementation class name using the `application.global` configuration key.

Hooking into application start and stop events

You can override the `onStart` and `onStop` methods to be notified of the events in the application life-cycle:

```
import play.api._  
  
object Global extends GlobalSettings {  
  
    override def onStart(app: Application) {  
        Logger.info("Application has started")  
    }  
  
    override def onStop(app: Application) {  
        Logger.info("Application shutdown...")  
    }  
  
}
```

Providing an application error page

When an exception occurs in your application, the `onError` operation will be called. The default is to use the internal framework error page:

```
import play.api._  
import play.api.mvc._  
import play.api.mvc.Results._  
  
object Global extends GlobalSettings {  
  
    override def onError(request: RequestHeader, ex: Throwable) = {  
        InternalServerError(  
            views.html.errorPage(ex)  
        )  
    }  
  
}
```

Handling missing actions and binding errors

If the framework doesn't find an `Action` for a request, the `onActionNotFound` operation will be called:

```
import play.api._  
import play.api.mvc._  
import play.api.mvc.Results._  
  
object Global extends GlobalSettings {  
  
    override def onActionNotFound(request: RequestHeader) = {  
        NotFound(  
            views.html.notFoundPage(request.path)  
        )  
    }  
  
}
```

The `onBadRequest` operation will be called if a route was found, but it was not possible to bind the request parameters:

```
import play.api._  
import play.api.mvc._  
import play.api.mvc.Results._  
  
object Global extends GlobalSettings {  
  
    override def onBadRequest(request: RequestHeader, error: String) = {  
        BadRequest("Bad Request: " + error)  
    }  
  
}
```

Intercepting requests

Overriding onRouteRequest

One another important aspect of the `Global` object is that it provides a way to intercept requests and execute business logic before a request is dispatched to an Action.

Tip This hook can be also used for hijacking requests, allowing developers to plug-in their own request routing mechanism.

Let's see how this works in practice:

```
import play.api._

// Note: this is in the default package.
object Global extends GlobalSettings {

    def onRouteRequest(request: RequestHeader): Option[Handler] = {
        println("executed before every request:" + request.toString)
        super.onRouteRequest(request)
    }

}
```

It's also possible to intercept a specific Action method, using Action composition .

Testing your application

Test source files must be placed in your application's `test` folder. You can run them from the Play console using the `test` and `test-only` tasks.

Using specs2

The default way to test a Play 2 application is by using specs2.

Unit specifications extend the `org.specs2.mutable.Specification` trait and are using the `should/in` format:

```
import org.specs2.mutable._

import play.api.test._
import play.api.test.Helpers._

class HelloWorldSpec extends Specification {

    "The 'Hello world' string" should {
        "contain 11 characters" in {
            "Hello world" must have size(11)
        }
        "start with 'Hello'" in {
            "Hello world" must startWith("Hello")
        }
        "end with 'world'" in {
            "Hello world" must endWith("world")
        }
    }
}
```

Running in a fake application

If the code you want to test depends of a running application, you can easily create a `FakeApplication` on the fly:

```
"Computer model" should {  
    "be retrieved by id" in {  
        running(FakeApplication()) {  
  
            val Some(macintosh) = Computer.findById(21)  
  
            macintosh.name must equalTo("Macintosh")  
            macintosh.introduced must beSome.which(dateIs(_, "1984-01-24"))  
  
        }  
    }  
}
```

You can also pass (or override) additional configuration to the fake application, or mock any plug-in. For example to create a `FakeApplication` using a `default` in memory database:

```
FakeApplication(additionalConfiguration = inMemoryDatabase())
```

Writing functional tests

Testing a template

Since a template is a standard Scala function, you can execute it from your test, and check the result:

```
"render index template" in {
    val html = views.html.index("Coco")

    contentType(html) must equalTo("text/html")
    contentAsString(html) must contain("Hello Coco")
}
```

Testing your controllers

You can call any `Action` code by providing a `FakeRequest`:

```
"respond to the index Action" in {
    val result = controllers.Application.index("Bob")(FakeRequest())

    status(result) must equalTo(OK)
    contentType(result) must beSome("text/html")
    charset(result) must beSome("utf-8")
    contentAsString(result) must contain("Hello Bob")
}
```

Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
"respond to the index Action" in {
    val Some(result) = routeAndCall(FakeRequest(GET, "/Bob"))

    status(result) must equalTo(OK)
    contentType(result) must beSome("text/html")
    charset(result) must beSome("utf-8")
    contentAsString(result) must contain("Hello Bob")
}
```

Starting a real HTTP server

Sometimes you want to test the real HTTP stack from with your test, in which case you can start a test server:

```
"run in a server" in {
    running(TestServer(3333)) {

        await(WS.url("http://localhost:3333").get).status must equalTo(OK)

    }
}
```

Testing from within a Web browser.

If you want to test your application using a browser, you can use Selenium WebDriver. Play will start the WebDriver for you, and wrap it in the convenient API provided by FluentLenium.

```
"run in a browser" in {
    running(TestServer(3333), HTMLUNIT) { browser =>

        browser.goTo("http://localhost:3333")
        browser.$("#title").getTexts().get(0) must equalTo("Hello Guest")

        browser.$("a").click()

        browser.url must equalTo("http://localhost:3333/Coco")
        browser.$("#title").getTexts().get(0) must equalTo("Hello Coco")

    }
}
```

Handling data streams reactively

Progressive Stream Processing and manipulation is an important task in modern Web Programming, starting from chunked upload/download to Live Data Streams consumption, creation, composition and publishing through different technologies including Comet and WebSockets.

Iteratees provide a paradigm and an api allowing this manipulation while focusing on several important aspects:

- Allowing user to create, consume and transform streams of data.
- Treating different data sources in the same manner (Files on disk, Websockets, Chunked Http, Data Upload, ...).
- Composable: use a rich set of adapters and transformers to change the shape of the source or the consumer; construct your own or start with primitives.
- Having control over when it is enough data sent and be informed when source is done sending data.
- Non blocking, reactive and allowing control over resource consumption (Thread, Memory)

Iteratee

An Iteratee is a consumer, it describes the way input will be consumed to produce some value. Iteratee is a consumer that returns a value it computes after being fed enough input.

```
// an iteratee that consumes chunkes of String and produces an Int
Iteratee[String,Int]
```

The Iteratee interface `Iteratee[E, A]` takes two type parameters, `E` representing the type of the Input it accepts and `A` the type of the calculated result.

An iteratee has one of three states, `Cont` meaning accepting more input, `Error` to indicate an error state and `Done` which carries the calculated result. These three states are defined by the `fold` method of an `Iteratee[E, A]` interface:

```
def fold[B] (
  done: (A, Input[E]) => Promise[B],
  cont: (Input[E] => Iteratee[E, A]) => Promise[B],
  error: (String, Input[E]) => Promise[B]
): Promise[B]
```

The fold method defines an iteratee as one of the three mentioned states. It accepts three callback functions and will call the appropriate one depending on its state to eventually extract a required value. When calling `fold` on an iteratee you are basically saying:

- If the iteratee is the state `Done`, then I'd take the calculated result of type `A` and what is left from the last consumed chunk of input `Input[E]` and eventually produce a `B`
- If the iteratee is the state `Cont`, then I'd take the provided continuation (which is accepting an input) `Input[E] => Iteratee[E, A]` and eventually produce a `B`. Note that this state provides the only way to push input into the iteratee, and get a new iteratee state, using the provided continuation function.
- If the iteratee is the state `Error`, then I'd take the error message of type `String` and the input that caused it and eventually produce a `B`.

Obviously, depending on the state of the iteratee, `fold` will produce the appropriate `B` using the corresponding passed function.

To sum up, iteratee consists of 3 states, and `fold` provides the means to do something useful with the state of the iteratee.

Some important types in the `Iteratee` definition:

Before providing some concrete examples of iteratees, let's clarify two important types we mentioned above:

- `Input[E]` represents a chunk of input that can be either an `E1[E]` containing some actual input, an `Empty` chunk or an `EOF` representing the end of the stream.
For example, `Input[String]` can be `E1("Hello!")`, `Empty`, or `EOF`
- `Promise[A]` represents, as its name tells, a promise of value of type `A`. This means that it will eventually be redeemed with a value of type `A` and you can schedule a callback, among other things you can do, if you are interested in that value. A promise is a very nice primitive for synchronization and composing async calls, and is explained further at the `PromiseScala` section.

Some primitive iteratees:

By implementing the iteratee, and more specifically its `fold` method, we can now create some primitive iteratees that we can use later on.

- An iteratee in the `Done` state producing an `1:Int` and returning `Empty` as left from last `Input[String]`

```
val doneIteratee = new Iteratee[String,Int] {
  def fold[B](
    done: (A, Input[E]) => Promise[B],
    cont: (Input[E] => Iteratee[E, A]) => Promise[B],
    error: (String, Input[E]) => Promise[B]): Promise[B] = done
  (1,Input.Empty)
}
```

As shown above, this is easily done by calling the appropriate callback function, in our case `done`, with the necessary information.

To use this iteratee we will make use of the `Promise.pure` that is a promise already in the Redeemed state.

```
val eventuallyMaybeResult : Promise[Option[Int]] = {
    doneIteratee.fold(
        // if done return the computed result
        (a, in) => Promise.pure(Some(a)),
        // if continue return None
        k => Promise.pure(None),
        // on error return None
        (msg, in) => Promise.pure(None)
    )
}
```

of course to see what is inside the `Promise` when it is redeemed we use `onRedeem`

```
// will eventually print 1
eventuallyMaybeResult.onRedeem(i => println(i))
```

There is already a built-in way allowing us to create an iteratee in the `Done` state by providing a result and input, generalizing what is implemented above:

```
val doneIteratee = Done[Int, String](1, Input.Empty)
```

Creating a `Done` iteratee is simple, and sometimes useful, but it obviously does not consume any input. Let's create an iteratee that consumes one chunk and eventually returns it as the computed result:

```
val consumeOneInputAndEventuallyReturnIt = new Iteratee[String, Int] {

    def fold[B](
        done: (Int, Input[String]) => Promise[B],
        cont: (Input[String] => Iteratee[String, Int]) => Promise[B],
        error: (String, Input[String]) => Promise[B]
    ): Promise[B] = {
        cont(in => Done(in, Input.Empty))
    }
}
```

As for `Done` there is a built-in way to define an iteratee in the `Cont` state by providing a function that takes `Input[E]` and returns a state of `Iteratee[E, A]`:

```
val consumeOneInputAndEventuallyReturnIt = {
  Cont[String, Int](in => Done(in, Input.Empty))
}
```

In the same manner there is a built-in way to create an iteratee in the `Error` state by providing an error message and an `Input[E]`

Back to the `consumeOneInputAndEventuallyReturnIt`, it is possible to create a two step simple iteratee manually but it becomes harder and cumbersome to create any real world iteratee capable of consuming a lot of chunks before, possibly conditionally, it eventually returns a result. Luckily there are some built-in methods to create common iteratee shapes in the `Iteratee` object.

Folding input:

One common task when using iteratees is maintaining some state and altering it each time input is pushed. This type of iteratee can be easily created using the `Iteratee.fold` which has the signature:

```
def fold[E, A](state: A)(f: (A, E) => A): Iteratee[E, A]
```

Reading the signature one can realize that this fold takes an initial state `A`, a function that takes the state and an input chunk `(A, E) => A` and returning an `Iteratee[E, A]` capable of consuming `E`s and eventually returning an `A`. The created iteratee will return `Done` with the computed `A` when an input `EOF` is pushed.

One example can be creating an iteratee that counts the number of bytes pushed in:

```
val inputLength: Iteratee[Array[Byte], Int] = {
  Iteratee.fold[Array[Byte], Int](0) { (length, bytes) => length +
    bytes.size }
}
```

Another could be consuming all input and eventually returning it:

```
val consume: Iteratee[String, String] = {
  Iteratee.fold[String, String]("") { (result, chunk) => result ++ chunk }
}
```

There is actually already a method in `Iteratee` object that does exactly this for any scala `TraversableLike` called `consume`, so our example becomes:

```
val consume = Iteratee.consume[String]()
```

One common case is to create an iteratee that does some imperative operation for each chunk of input:

```
val printlnIteratee = Iteratee.foreach[String](s => println(s))
```

More interesting methods exist like `repeat`, `ignore` and `fold1` which is different from the preceding `fold` in offering the opportunity to be asynchronous in treating input chunks.

Of course one should be worried now about how hard would it be to manually push input into an iteratee by folding over iteratee states over and over again. Indeed each time one has to push input into an iteratee, one has to use the `fold` function to check on its state, if it is a `Cont` then push the input and get the new state or otherwise return the computed result. That's when `Enumerator`'s come in handy.

Handling data streams reactively

The realm of Enumeratee

'Enumeratee' is a very important component in the iteratees API. It provides a way to adapt and transform streams of data. An `Enumeratee` that might sound familiar is the `Enumeratee.map`.

Starting with a simple problem, consider the following `Iteratee`:

```
val sum: Iteratee[Int, Int] = Iteratee.fold[Int, Int](0){ (s, e) => s + e }
```

This `Iteratee` takes `Int` objects as input and computes their sum. Now if we have an `Enumerator` like the following:

```
val strings: Enumerator[String] = Enumerator("1", "2", "3", "4")
```

Then obviously we can not apply the `strings: Enumerator[String]` to an `Iteratee[Int, Int]`. What we need is transform each `String` to the corresponding `Int` so that the source and the consumer can be fit together. This means we either have to adapt the `Iteratee[Int, Int]` to be `Iteratee[String, Int]`, or adapt the `Enumerator[String]` to be rather an `Enumerator[Int]`.

An `Enumeratee` is the right tool for doing that. We can create an `Enumeratee[String, Int]` and adapt our `Iteratee[Int, Int]` using it:

```
//create an Enumerator using the map method on Enumerator
val toInt: Enumerator[String, Int] = Enumerator.map[String]{ s => s.toInt }

val adaptedIteratee: Iteratee[String, Int] = toInt.transform(sum)

//this works!
strings >>| adaptedIteratee
```

There is a symbolic alternative to the `transform` method, `&>>` which we can use in our previous example:

```
strings >>> toInt &>> sum
```

The `map` method will create an 'Enumeratee' that uses a provided `From => To` function to map the input from the `From` type to the `To` type. We can also adapt the `Enumerator`:

```
val adaptedEnumerator : Enumerator[Int] = strings.through(toInt)

//this works!
adaptedEnumerator >>> sum
```

Here too, we can use a symbolic version of the `through` method:

```
strings &> toInt >>> sum
```

Let's have a look at the `transform` signature defined in the `Enumeratee` trait:

```
trait Enumeratee[From, To] {
  def transform[A](inner: Iteratee[To, A]): Iteratee[From, A] = ...
}
```

This is a fairly simple signature, and is the same for `through` defined on an `Enumerator`:

```
trait Enumerator[E] {
  def through[To](enumeratee: Enumeratee[E, To]): Enumerator[To]
}
```

The `transform` and `through` methods on an `Enumeratee` and `Enumerator`, respectively, both use the `apply` method on `Enumeratee`, which has a slightly more sophisticated signature:

```
trait Enumeratee[From, To] {
  def apply[A](inner: Iteratee[To, A]): Iteratee[From, Iteratee[To, A]] =
  ...
}
```

Indeed, an `Enumeratee` is more powerful than just transforming an `Iteratee` type. It really acts like an adapter in that you can get back your original `Iteratee` after pushing some different input through an `Enumeratee`. So in the previous example, we can get back the original `Iteratee[Int, Int]` to continue pushing some `Int` objects in:

```

val sum:Iteratee[Int,Int] = Iteratee.fold[Int,Int](0){ (s,e) => s + e }

//create an Enumeratee using the map method on Enumeratee
val toInt: Enumeratee[String,Int] = Enumeratee.map[String]{ s => s.toInt }

val adaptedIteratee: Iteratee[String,Iteratee[Int,Int]] = toInt(sum)

// pushing some strings
val afterPushingStrings: Iteratee[String,Iteratee[Int,Int]] = {
  Enumerator("1","2","3","4") >>> adaptedIteratee
}

val originalIteratee: Iteratee[Int,Int] = flatten(afterPushingString.run)

val moreInts: Iteratee[Int,Int] = Enumerator(5,6,7) >>> originalIteratee

moreInts.run.onRedeem(sum => println(sum) ) // eventually prints 28

```

That's why we call the adapted (original) `Iteratee` 'inner' and the resulting `Iteratee` 'outer'.

Now that the `Enumeratee` picture is clear, it is important to know that `transform` drops the left input of the inner `Iteratee` when it is `Done`. This means that if we use `Enumeratee.map` to transform input, if the inner `Iteratee` is `Done` with some left transformed input, the `transform` method will just ignore it.

That might have seemed like a bit too much detail, but it is useful for grasping the model.

Back to our example on `Enumeratee.map`, there is a more general method `Enumeratee.mapInput` which, for example, gives the opportunity to return an `EOF` on some signal:

```

val toIntOrEnd: Enumeratee[String,Int] = Enumeratee.mapInput[String] {
  case Input.El("end") => Input.EOF
  case other => other.map(e => e.toInt)
}

```

`Enumeratee.map` and `Enumeratee.mapInput` are pretty straight forward, they operate on a per chunk basis and they convert them. Another useful `Enumeratee` is the `Enumeratee.filter`:

```
def filter[E](predicate: E => Boolean): Enumeratee[E, E]
```

The signature is pretty obvious, `Enumeratee.filter` creates an `Enumeratee[E,E]` and it will test each chunk of input using the provided `predicate: E => Boolean` and it passes it along to the inner (adapted) iteratee if it satisfies the predicate:

```
val numbers = Enumerator(1,2,3,4,5,6,7,8,9,10)

val onlyOdds = Enumeratee.filter[Int](i => i % 2 != 0)

numbers.through(onlyOdds) >>> sum
```

There are methods, such as `Enumeratee.collect`, `Enumeratee.drop`, `Enumeratee.dropWhile`, `Enumeratee.take`, `Enumeratee.takeWhile`, which work on the same principle.

Let try to use the `Enumeratee.take` on an Input of chunks of bytes:

```
// computes the size in bytes
val fillInMemory: Iteratee[Array[Byte], Int] = {
  Iteratee.consume[Array[Byte]]()
}

val limitTo100: Enumeratee[Array[Byte], [Array[Byte]]] = {
  Enumeratee.take[Array[Byte]](100)
}

val limitedFillInMemory: Iteratee[Array[Byte], Int] = {
  limit &>> fillInMemory
}
```

It looks good, but how many bytes are we taking? What would ideally limit the size, in bytes, of loaded input. What we do above is to limit the number of chunks instead, whatever the size of each chunk is. It seems that the `Enumeratee.take` is not enough here since it has no information about the type of input (in our case an `Array[Byte]`) and this is why it can't count what's inside.

Luckily there is a `Traversable` object that offers a set of methods for creating `Enumeratee` instances for Input types that are `TraversableLike`. An `Array[Byte]` is `TraversableLike` and so we can use `Traversable.take`:

```
// computes the size in bytes
val fillInMemory: Iteratee[Array[Byte], Int] = {
  Iteratee.consume[Array[Byte]]()
}

val limitTo100: Enumeratee[Array[Byte], [Array[Byte]]] = {
  Traversable.take[Array[Byte]](100)
}

// We are sure not to get more than 100 bytes loaded into memory
val limitedFillInMemory: Iteratee[Array[Byte], Int] = {
  limit &>> fillInMemory
}
```

Other `Traversable` methods exist including `Traversable.takeUpTo`, `Traversable.drop`.

Finally, you can compose different `Enumeratee` instances using the `compose` method, which has the symbolic equivalent `><>`. Note that any left input on the `Done` of the composed `Enumeratee` instances will be dropped. However, if you use `composeConcat` aliased `>+>`, any left input will be concatenated.

Your first Play application

Let's write a simple to do list application with Play 2.0 and deploy it to the cloud.

Prerequisites

First of all, make sure that you have a working Play installation. You only need Java (version 6 minimum), and to unzip the Play binary package to start; everything is included.

As we will use the command line a lot, it's better to use a Unix-like OS. If you run a Windows system, it will also work fine; you'll just have to type a few commands in the command prompt.

You will of course need a text editor. You can also use a Scala IDE such as Eclipse or IntelliJ if you like. However, with Play you can have fun working with a simple text editor like Textmate , Emacs or vi. This is because the framework manages the compilation and the deployment process itself.

Project creation

Now that Play is correctly installed, it's time to create the new application. Creating a Play application is pretty easy and fully managed by the Play command line utility. That allows for standard project layouts between all Play applications.

On the command line and type:

```
$ play new todolist
```

It will prompt you for a few question. Select the *Create a simple Scala application* project template.

```
$ play new todolist
[info] Loading project definition from /private/tmp/todolist/project
[info] Set current project to todolist (todolist:0.1-SNAPSHOT)
play! 2.0-RC1-SNAPSHOT, http://www.playframework.org

The new application will be created in /private/tmp/todolist

What is the application name?
> todolist

Which template do you want to use for this new application?

1 - Create a simple Scala application
2 - Create a simple Java application
3 - Create an empty project

> 1

OK, application todolist is created.

Have fun!
```

The `play new` command creates a new directory `todolist/` and populates it with a series of files and directories, the most important being:

- `app/` contains the application's core, split between models, controllers and views directories. This is the directory where `.scala` source files live.
- `conf/` contains all the application's configuration files, especially the main `application.conf` file, the `routes` definition files and the `messages` files used for internationalization.
- `project/` contains the build scripts. The build system is based on sbt. But a new play application comes with a default build script that will just work for our application.
- `public/` contains all the publicly available resources, which includes JavaScript, stylesheets and images directories.
- `test/` contains all the application tests. Tests are written either as Specs2 specifications.

Because Play uses UTF-8 as single encoding, it's very important that all text files hosted in these directories are encoded using this charset. Make sure to configure your text editor accordingly.

Using the Play console

Once you have an application created, you can run the Play console. Go to the new `todolist/` directory and run:

```
$ play
```

This launches the Play console. There are several things you can do from the Play console, but let's start by running the application. From the console prompt, type `run`:

```
[todolist] $ run
```

2. java

```
$ play
[info] Loading project definition from /private/tmp/todolist/project
[info] Set current project to todolist (in build file:/private/tmp/todolist/)

play! 2.0-RC1-SNAPSHOT, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[todolist] $ run

[info] Updating {file:/private/tmp/todolist/}todolist...
[info] Done updating.
--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)
```

The application is now running in development mode. Open a browser at <http://localhost:9000/>:

Welcome to Play 2.0

localhost:9000

Play framework 2.0-RC1-SNAPSHOT

Your new application is ready.

Welcome to Play 2.0

Congratulation, you've just created a new Play application. This page will help you in the few next steps.

You are using Play 2.0-RC1-SNAPSHOT

Why do you see this page?

The `conf/routes` file defines a route that tell Play to invoke the `Application.index` action when a browser requests the `/` URI using the GET method:

```
# Home page
GET      /           controllers.Application.index
```

So Play has invoked the `controllers.Application.index` method to obtain the Action to execute:

```
def index = Action {
  Ok("Your new application is ready!")}
```

Browse

- [Local documentation](#)
- [Browse the Scala API](#)

Start here

- [Using the Play console](#)
- [Setting up your preferred IDE](#)
- [Your first application](#)

Search

Get help with google

Google Custom Search

Note: Read more about [The Play Console](#).

Overview

Let's see how the new application can display this page.

The main entry point of your application is the `conf/routes` file. This file defines all of the application's accessible URLs. If you open the generated routes file you will see this first *route*:

```
GET      /      controllers.Application.index
```

That simply tells Play that when the web server receives a GET request for the / path, it must retrieve the `Action` to execute from the `controllers.Application.index` method.

Let's see how the `controllers.Application.index` method looks like. Open the `todolist/app/controllers/Application.scala` source file:

```
package controllers

import play.api._
import play.api.mvc._

object Application extends Controller {

    def index = Action {
        Ok(views.html.index("Your new application is ready."))
    }
}
```

You see that `controllers.Application.index` returns an `Action` that will handle the request. An `Action` must return a `Result` that represents the HTTP response to send back to the web browser.

Note: Read more about [Actions](#).

Here the action returns a **200 OK** response filled with HTML content. The HTML content is provided by a template. Play templates are compiled to standard Scala functions, here as `views.html.index(message: String)`.

This template is defined in the `app/views/index.scala.html` source file:

```
@(message: String)

@main("Welcome to Play 2.0") {

    @play20.welcome(message)

}
```

The first line defines the function signature. Here it takes a single `String` parameter. Then the template content mix HTML (or any text based language) with Scala statements. The Scala statements starts with the special `@` character.

Development workflow

Now let's make some modifications to the new application. In the `Application.scala` change the content of the response:

```
def index = Action {
    Ok("Hello world")
}
```

With this change the `index` action will now respond with a simple `text/plain` **Hello world** response. To test this change, just refresh the home page in your browser:

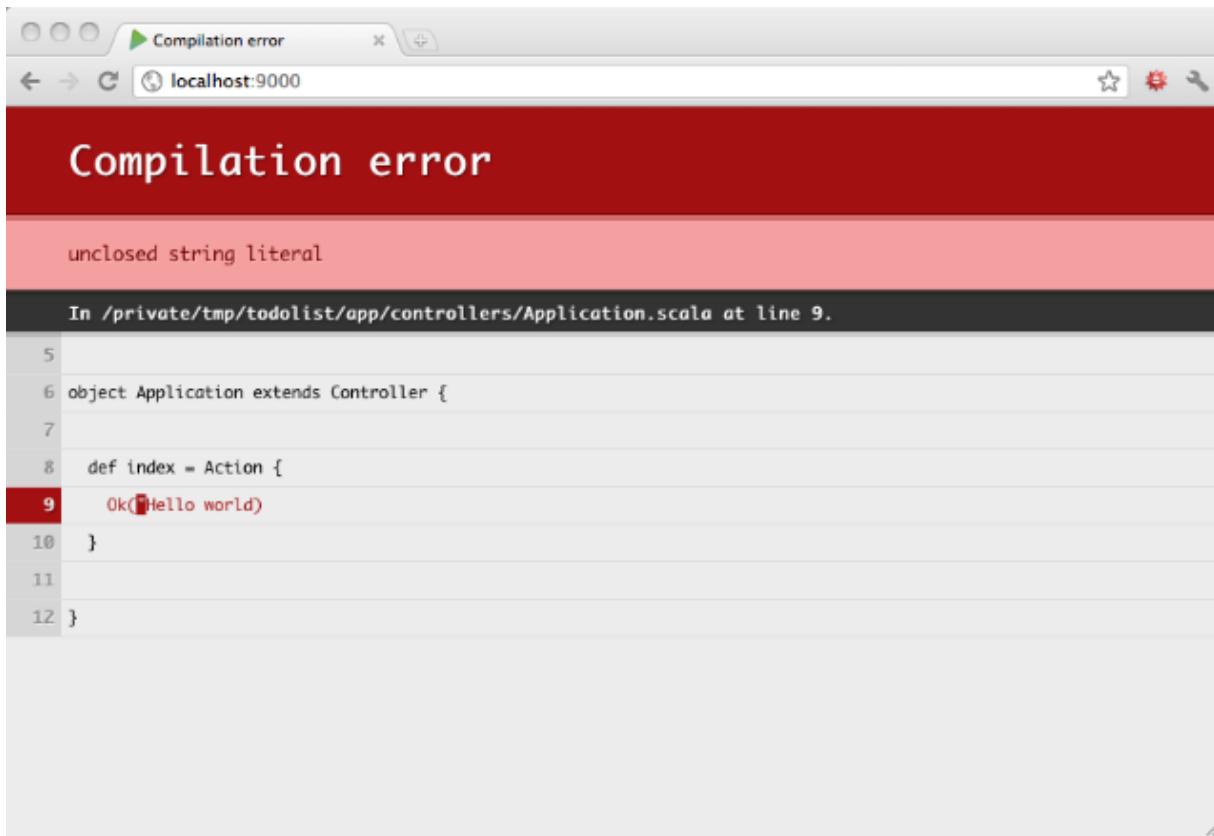


There is no need to compile the code yourself or restart the server to see the modification. It is automatically reloaded when a change is detected. But what happens when you make a mistake in your code?

Let's try:

```
def index = Action {
    Ok("Hello world")
}
```

Now reload the home page in your browser:



As you see errors are beautifully displayed directly in your browser.

Preparing the application

For our to do list application, we need a few actions and the corresponding urls. Let's start by defining the **routes**.

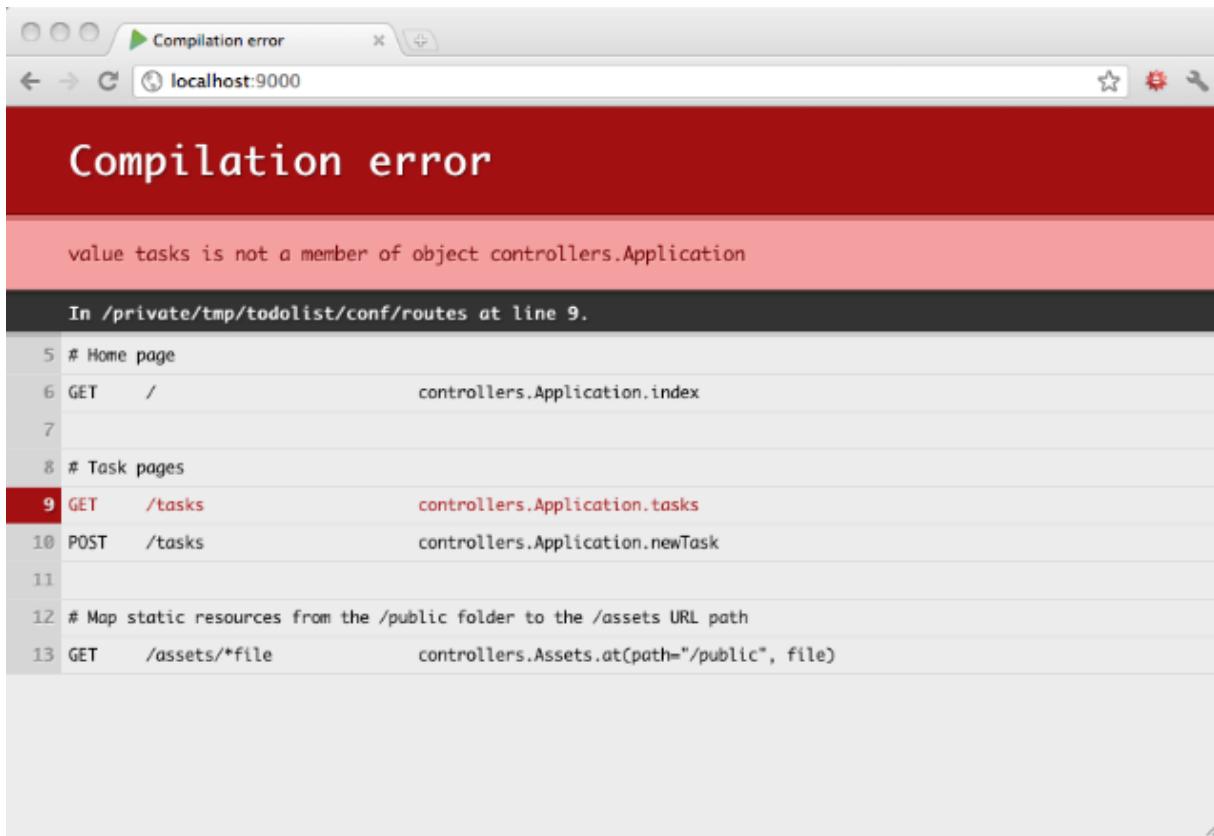
Edit the `conf/routes` file:

```
# Home page
GET      /           controllers.Application.index

# Tasks
GET      /tasks      controllers.Application.tasks
POST     /tasks      controllers.Application.newTask
POST     /tasks/:id/delete controllers.Application.deleteTask(id: Long)
```

We create a route to list all tasks, and a couple of others to handle task creation and deletion. The route to handle task deletion defines a variable argument `:id` in the URL path. This value is then passed to the `deleteTask` method that will create the `Action`.

Now if you reload in your browser, you will see that Play cannot compile your `routes` file:



This is because the new routes reference non-existent action methods. So let's add them to the `Application.scala` file:

```
object Application extends Controller {

    def index = Action {
        Ok("Hello world")
    }

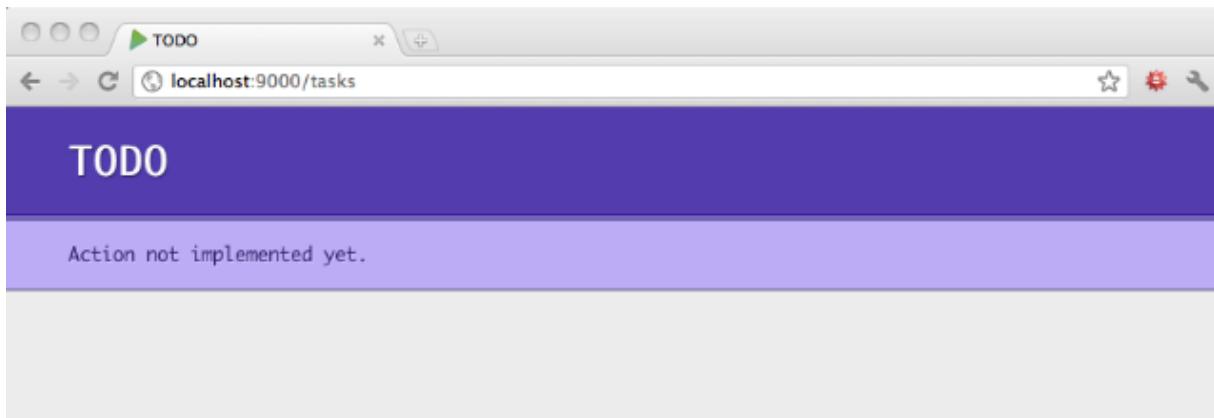
    def tasks = TODO

    def newTask = TODO

    def deleteTask(id: Long) = TODO
}
```

As you see we use `TODO` to define our action implementations. Because we don't want to write the action implementations yet, we can use the built-in `TODO` action that will return a `503 Not Implemented` HTTP response.

You can try to access the `http://localhost:9000/tasks` to see that:



Now the last thing we need to fix before starting the action implementation is the `index` action. We want it to automatically redirect to the tasks list page:

```
def index = Action {  
    Redirect(routes.Application.tasks)  
}
```

As you can see, we use `Redirect` instead of `Ok` to specify a `303 See Other` HTTP response type. We also use the reverse router to get the URL needed to fetch the `tasks` actions.

Note: Read more about the Router and reverse router.

Prepare the `Task` model

Before continuing the implementation we need to define what a `Task` looks like in our application. Create a `case class` for it in the `app/models/Task.scala` file:

```
package models  
  
case class Task(id: Long, label: String)  
  
object Task {  
  
    def all(): List[Task] = Nil  
  
    def create(label: String) {}  
  
    def delete(id: Long) {}  
  
}
```

We have also created a companion object to manage `Task` operations. For now we wrote a dummy implementation for each operation, but later in this tutorial we will write implementations that store the tasks in a relational database.

The application template

Our simple application will use a single web page that shows both the tasks list and the task creation form. Let's modify the `index.scala.html` template for that:

```
@(tasks: List[Task], taskForm: Form[String])

@import helper._

@main("Todo list") {

    <h1>@tasks.size task(s)</h1>

    <ul>
        @tasks.map { task =>
            <li>
                @task.label

                @form(routes.Application.deleteTask(task.id)) {
                    <input type="submit" value="Delete">
                }
            </li>
        }
    </ul>

    <h2>Add a new task</h2>

    @form(routes.Application.newTask) {

        @inputText(taskForm("label"))

        <input type="submit" value="Create">

    }
}
}
```

We changed the template signature to take two parameters :

- a list of tasks to display
- a task form.

We also imported `helper._` that gives us the form creation helpers, typically the `form` function, which creates an HTML `<form>` with filled `action` and `method` attributes, and the `inputText` function that creates an HTML input for a form field.

Note: Read more about the [Templating system](#) and [Forms helper](#).

The task form

A `Form` object encapsulates an HTML form definition, including validation constraints. Let's create a very simple form in the `Application` controller: we only need a form with a single `label` field. The form will also check that the label provided by the user is not empty:

```
import play.api.data._  
import play.api.data.Forms._  
  
val taskForm = Form(  
    "label" -> nonEmptyText  
)
```

The type of `taskForm` is then `Form[String]` since it is a form generating a simple `String`. You also need to import some `play.api.data` classes.

Note: Read more about the `Form` definitions.

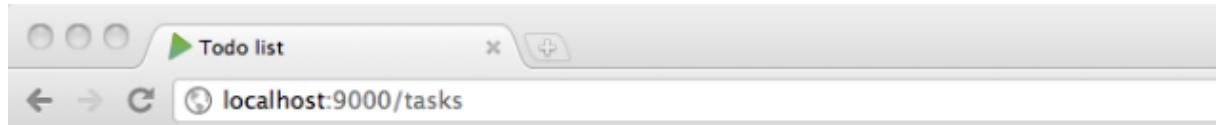
Rendering the first page

Now we have all elements needed to display the application page. Let's write the `tasks` action:

```
def tasks = Action {  
    Ok(views.html.index(Task.all(), taskForm))  
}
```

This renders a **200 OK** result filled with the HTML rendered by the `index.scala.html` template called with the tasks list and the task form.

You can now try to access `http://localhost:9000/tasks` in your browser:



0 task(s)

Add a new task

label

Required

[Create](#)

Handling the form submission

For now, if we submit the task creation form, we still get the TODO page. Let's write the implementation of the `newTask` action:

```
def newTask = Action { implicit request =>
    taskForm.bindFromRequest.fold(
        errors => BadRequest(views.html.index(Task.all(), errors)),
        label => {
            Task.create(label)
            Redirect(routes.Application.tasks)
        }
    )
}
```

To fill the form we need to have the `request` in the scope, used by `bindFromRequest` to create a new form filled with the request data. If there are any errors in the form, we redisplay it (here we use **400 Bad Request** instead of **200 OK**). If there are no errors, we create the task and then redirect to the task list.

Note: Read more about the Form submissions.

Persist the tasks in a database

It's now time to persist the tasks in a database to make the application useful. Let's start by enabling a database in our application. In the `conf/application.conf` file, add:

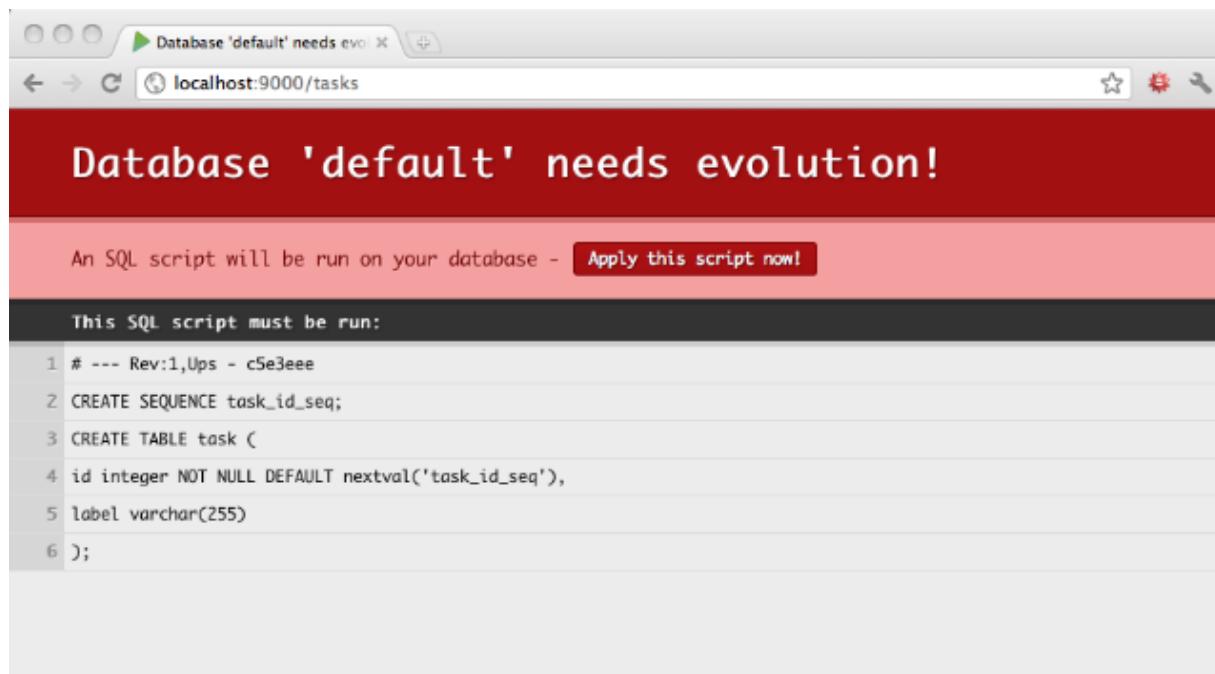
```
db.default.driver=org.h2.Driver  
db.default.url="jdbc:h2:mem:play"
```

For now we will use a simple in memory database using **H2**. No need to restart the server, refreshing the browser is enough to set up the database.

We will use **Anorm** in this tutorial to query the database. First we need to define the database schema. Let's use Play evolutions for that, so create a first evolution script in `conf/evolutions/default/1.sql`:

```
# Tasks schema  
  
# --- !Ups  
  
CREATE SEQUENCE task_id_seq;  
CREATE TABLE task (  
    id integer NOT NULL DEFAULT nextval('task_id_seq'),  
    label varchar(255)  
);  
  
# --- !Downs  
  
DROP TABLE task;  
DROP SEQUENCE task_id_seq;
```

Now if you refresh your browser, Play will warn you that your database needs evolution:



Just click the **Apply script** button to run the script. Your database schema is now ready!

Note: Read more about [Evolutions](#).

It's now time to implement the SQL queries in the `Task` companion object, starting with the `all()` operation. Using **Anorm** we can define a parser that will transform a JDBC `ResultSet` row to a `Task` value:

```
import anorm._  
import anorm.SqlParser._  
  
val task = {  
    get[Long]("id") ~  
    get[String]("label") map {  
        case id~label => Task(id, label)  
    }  
}
```

Here, `task` is a parser that, given a JDBC `ResultSet` row with at least an `id` and a `label` column, is able to create a `Task` value.

We can now use this parser to write the `all()` method implementation:

```
import play.api.db._  
import play.api.Play.current  
  
def all(): List[Task] = DB.withConnection { implicit c =>  
    SQL("select * from task").as(task *)  
}
```

We use the Play `DB.withConnection` helper to create and release automatically a JDBC connection.

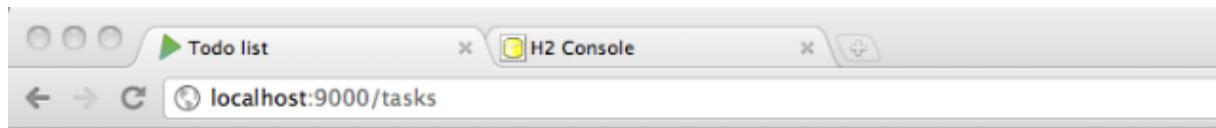
Then we use the **Anorm** `SQL` method to create the query. The `as` method allows to parse the `ResultSet` using the `task *` parser: it will parse as many task rows as possible and then return a `List[Task]` (since our `task` parser returns a `Task`).

It's time to complete the implementation:

```
def create(label: String) {
    DB.withConnection { implicit c =>
        SQL("insert into task (label) values ({label})").on(
            'label -> label
        ).executeUpdate()
    }
}

def delete(id: Long) {
    DB.withConnection { implicit c =>
        SQL("delete from task where id = {id}").on(
            'id -> id
        ).executeUpdate()
    }
}
```

Now you can play with the application; creating new tasks should work.



2 task(s)

- Complete the tutorial
[Delete](#)
- Buy milk
[Delete](#)

Add a new task

label

Required

[Create](#)

Note: Read more about Anorm.

Deleting tasks

Now that we can create tasks, we need to be able to delete them. Very simple: we just need to finish the implementation of the `deleteTask` action:

```
def deleteTask(id: Long) = Action {
    Task.delete(id)
    Redirect(routes.Application.tasks)
}
```

Deploying to Heroku

All features are complete, so it's time to deploy our application in production. Let's deploy it to Heroku. First, you need to create a `Procfile` for Heroku. Create the `Procfile` in the root application directory:

```
web: target/start -Dhttp.port=${PORT} -DapplyEvolutions.default=true -
Ddb.default.url=${DATABASE_URL} -Ddb.default.driver=org.postgresql.Driver
```

Note: Read more about [Deploying to Heroku](#).

We use system properties to override the application configuration, when running on Heroku. Since Heroku provides an PostgreSQL database, we need to add the required driver to our application dependencies.

Specify it into the `project/Build.scala` file:

```
val appDependencies = Seq(
    "postgresql" % "postgresql" % "8.4-702.jdbc4"
)
```

Note: Read more about [Dependencies management](#).

Heroku uses **git** to deploy your application. Let's initialize the git repository:

```
$ git init
$ git add .
$ git commit -m "init"
```

Now we can create the application on Heroku:

```
$ heroku create --stack cedar

Creating warm-frost-1289... done, stack is cedar
http://warm-1289.herokuapp.com/ | git@heroku.com:warm-1289.git
Git remote heroku added
```

And then deploy it using simple `git push heroku master`:

```
$ git push heroku master

Counting objects: 34, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (34/34), 35.45 KiB, done.
Total 34 (delta 0), reused 0 (delta 0)

-----> Heroku receiving push
-----> Scala app detected
-----> Building app with sbt v0.11.0
-----> Running: sbt clean compile stage
...
-----> Discovering process types
      Procfile declares types -> web
-----> Compiled slug size is 46.3MB
-----> Launching... done, v5
      http://8044.herokuapp.com deployed to Heroku

To git@heroku.com:floating-lightning-8044.git
 * [new branch]      master -> master
```

Heroku will build your application and deploy it to a node somewhere on the cloud. You can check the state of the application's processes:

```
$ heroku ps

Process     State            Command
----- -----
web.1       up for 10s      target/start
```

It's started, you can now open it in your browser.

Your first application is now up and running in production!

Play 2.0 for Java developers

The Java API for the Play 2.0 application developers is available in the `play` package.

The API available in the `play.api` package (such as `play.api.mvc`) is reserved for Scala developers. As a Java developer, look at `play.mvc`.

Actions, Controllers and Results

What is an Action?

Most of the requests received by a Play application are handled by an `Action`.

An action is basically a Java method that processes the request parameters, and produces a result to be sent to the client.

```
public static Result index() {
    return ok("Got request " + request() + "!");
}
```

An action returns a `play.mvc.Result` value, representing the HTTP response to send to the web client. In this example `ok` constructs a **200 OK** response containing a `text/plain` response body.

Controllers

A controller is nothing more than a class extending `play.mvc.Controller` that groups several action methods.

The simplest syntax for defining an action is a static method with no parameters that returns a `Result` value:

```
public static Result index() {
    return ok("It works!");
}
```

An action method can also have parameters:

```
public static Result index(String name) {
    return ok("Hello " + name);
}
```

These parameters will be resolved by the `Router` and will be filled with values from the request URL. The parameter values can be extracted from either the URL path or the URL query string.

Results

Let's start with simple results: an HTTP result with a status code, a set of HTTP headers and a body to be sent to the web client.

These results are defined by `play.mvc.Result`, and the `play.mvc.Results` class provides several helpers to produce standard HTTP results, such as the `ok` method we used in the previous section:

```
public static Result index() {  
    return ok("Hello world!");  
}
```

Here are several examples that create various results:

```
Result ok = ok("Hello world!");  
Result notFound = notFound();  
Result pageNotFound = notFound("<h1>Page not found</h1>").as("text/html");  
Result badRequest = badRequest(views.html.form.render(formWithErrors));  
Result oops = internalServerError ("Oops");  
Result anyStatus = status(488, "Strange response type");
```

All of these helpers can be found in the `play.mvc.Results` class.

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.

There are several helpers available to create redirect results:

```
public static Result index() {  
    return redirect("/user/home");  
}
```

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

```
public static Result index() {  
    return temporaryRedirect ("/user/home");  
}
```

HTTP routing

The built-in HTTP router

The router is the component that translates each incoming HTTP request to an action call (a static, public method in a controller class).

An HTTP request is seen as an event by the MVC framework. This event contains two major pieces of information:

- the request path (such as `/clients/1542`, `/photos/list`), including the query string.
- the HTTP method (GET, POST, ...).

Routes are defined in the `conf/routes` file, which is compiled. This means that you'll see route errors directly in your browser:

```
too many arguments for method index: ()play.api.mvc.Action[play.api.mvc.AnyContent]

In /Volumes/Data/gbo/myFirstApp/conf/routes at line 6.

2 # This file defines all application routes (Higher priority routes first)
3 #
4
5 # Home page
6 GET    /           controllers.Application.index(page, repeat: Int ?= 10)
7
8 # Map static resources from the /public folder to the /assets URL path
9 GET    /assets/*file   controllers.Assets.at(path="/public", file)
```

The routes file syntax

`conf/routes` is the configuration file used by the router. This file lists all of the routes needed by the application. Each route consists of an HTTP method and URI pattern associated with a call to an action method.

Let's see what a route definition looks like:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Note that in the action call, the parameter type comes after the parameter name, like in Scala.

Each route starts with the HTTP method, followed by the URI pattern. The last element of a route is the call definition.

You can also add comments to the route file, with the `#` character:

```
# Display a client.  
GET /clients/:id controllers.Clients.show(id: Long)
```

The HTTP method

The HTTP method can be any of the valid methods supported by HTTP (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`).

The URI pattern

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

Static path

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

```
GET /clients controllers.Clients.list()
```

Dynamic parts

If you want to define a route that, say, retrieves a client by id, you need to add a dynamic part:

```
GET /clients/:id controllers.Clients.show(id: Long)
```

Note that a URI pattern may have more than one dynamic part.

The default matching strategy for a dynamic part is defined by the regular expression `[^/]+`, meaning that any dynamic part defined as `:id` will match exactly one URI path segment.

Dynamic parts spanning several /

If you want a dynamic part to capture more than one URI path segment, separated by forward slashes, you can define a dynamic part using the `*id` syntax, which uses the `.*` regular expression:

```
GET  /files/*name      controllers.Application.download(name)
```

Here, for a request like `GET /files/images/logo.png`, the `name` dynamic part will capture the `images/logo.png` value.

Dynamic parts with custom regular expressions

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

```
GET  /clients/$id<[0-9]+>  controllers.Clients.show(id: Long)
```

Call to action generator method

The last part of a route definition is the call. This part must define a valid call to an action method.

If the method does not define any parameters, just give the fully-qualified method name:

```
GET  /      controllers.Application.homePage()
```

If the action method defines parameters, the corresponding parameter values will be searched for in the request URI, either extracted from the URI path itself, or from the query string.

```
# Extract the page parameter from the path.  
# i.e. http://myserver.com/index  
GET  /:page      controllers.Application.show(page)
```

Or:

```
# Extract the page parameter from the query string.  
# i.e. http://myserver.com/?page=index  
GET  /      controllers.Application.show(page)
```

Here is the corresponding `show` method definition in the `controllers.Application` controller:

```
public static Result show(String page) {  
    String content = Page.getContentOf(page);  
    response().setContentType("text/html");  
    return ok(content);  
}
```

Parameter types

For parameters of type `String`, the parameter type is optional. If you want Play to transform the incoming parameter into a specific Scala type, you can add an explicit type:

```
GET /client/:id controllers.Clients.show(id: Long)
```

Then use the same type for the corresponding action method parameter in the controller:

```
public static Result show(Long id) {  
    Client client = Client.findById(id);  
    return ok(views.html.Client.show(client));  
}
```

Note: The parameter types are specified using a suffix syntax. Also The generic types are specified using the `[]` symbols instead of `<>`, as in Java. For example, `List[String]` is the same type as the Java `List<String>`.

Parameters with fixed values

Sometimes you'll want to use a fixed value for a parameter:

```
# Extract the page parameter from the path, or fix the value for /  
GET / controllers.Application.show(page = "home")  
GET /:page controllers.Application.show(page)
```

Parameters with default values

You can also provide a default value that will be used if no value is found in the incoming request:

```
# Pagination links, like /clients?page=3  
GET /clients controllers.Clients.list(page: Integer ?= 1)
```

Routing priority

Many routes can match the same request. If there is a conflict, the first route (in declaration order) is used.

Reverse routing

The router can be used to generate a URL from within a Java call. This makes it possible to centralize all your URI patterns in a single configuration file, so you can be more confident when refactoring your application.

For each controller used in the routes file, the router will generate a ‘reverse controller’ in the routes package, having the same action methods, with the same signature, but returning a play.mvc.Call instead of a play.mvc.Result.

The play.mvc.Call defines an HTTP call, and provides both the HTTP method and the URI.

For example, if you create a controller like:

```
package controllers;

import play.*;
import play.mvc.*;

public class Application extends Controller {

    public static Result hello(String name) {
        return ok("Hello " + name + "!");
    }

}
```

And if you map it in the conf/routes file:

```
# Hello action
GET /hello/:name controllers.Application.hello(name)
```

You can then reverse the URL to the hello action method, by using the controllers.routes.Application reverse controller:

```
// Redirect to /hello/Bob
public static Result index() {
    return redirect(controllers.routes.Application.hello("Bob"));
}
```

Manipulating the response

Changing the default Content-Type

The result content type is automatically inferred from the Java value you specify as body.

For example:

```
Result textResult = ok("Hello World!");
```

Will automatically set the `Content-Type` header to `text/plain`, while:

```
Result jsonResult = ok(jerksonObject);
```

will set 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 similiar result with a different `Content-Type` header:

```
Result htmlResult = ok("<h1>Hello World!</h1>").as("text/html");
```

You can also set the content type on the HTTP response:

```
public static Result index() {
    response().setContentType("text/html");
    return ok("<h1>Hello World!</h1>");
}
```

Setting HTTP response headers

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

```
public static Result index() {
    response().setContentType("text/html");
    response().setHeader(CACHE_CONTROL, "max-age=3600");
    response().setHeader(ETAG, "xxx");
    return ok("<h1>Hello World!</h1>");
}
```

Note that setting an HTTP header will automatically discard any previous value.

Setting and discarding cookies

Cookies are just a special form of HTTP headers, but Play provides a set of helpers to make it easier.

You can easily add a Cookie to the HTTP response:

```
response().setCookie("theme", "blue");
```

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

```
response().discardCookies("theme");
```

Specifying the character encoding for text results

For a text-based HTTP response it is very important to handle the character encoding correctly. Play 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=xxxx` extension to the `Content-Type` header.

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

```
public static Result index() {  
    response().setContentType("text/html; charset=iso-8859-1");  
    return ok("<h1>Hello World!</h1>", "iso-8859-1");  
}
```

Session and Flash scopes

How it is different in Play

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 are available during the whole user session, and data stored in the flash scope are 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 Play 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 Play built-in cache mechanism and use store a unique ID in the user session 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.).

Reading a Session value

You can retrieve the incoming Session from the HTTP request:

```
public static Result index() {  
    String user = session("connected");  
    if(user != null) {  
        return ok("Hello " + user);  
    } else {  
        return unauthorized("Oops, you are not connected");  
    }  
}
```

Storing data into the Session

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

```
public static Result index() {  
    session("connected", "user@gmail.com");  
    return ok("Welcome!");  
}
```

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

```
public static Result index() {  
    session.remove("connected");  
    return ok("Bye");  
}
```

Discarding the whole session

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

```
public static Result index() {  
    session().clear();  
    return ok("Bye");  
}
```

Flash scope

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

- data are kept for only one request
- 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 are 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 subject to race conditions.

Here are a few examples using the Flash scope:

```
public static Result index() {
    String message = flash("success");
    if(message == null) {
        message = "Welcome!";
    }
    return ok(message);
}

public static Result save() {
    flash("success", "The item has been created");
    return redirect("/home");
}
```

Body parsers

What is a body parser?

An HTTP request (at least for those using the POST and PUT operations) contains a body. This body can be formatted with any format specified in the Content-Type header. A **body parser** transforms this request body into a Java value.

Note: You can't write `BodyParser` implementation directly using Java. Because a Play `BodyParser` must handle the body content incrementally using an `Iteratee[Array[Byte], A]` it must be implemented in Scala.

However Play provides default `BodyParser`s that should fit most use cases (parsing Json, Xml, Text, uploading files). And you can reuse these default parsers to create your own directly in Java; for example you can provide an RDF parsers based on the Text one.

The `BodyParser` Java API

In the Java API, all body parsers must generate a `play.mvc.Http.RequestBody` value. This value computed by the body parser can then be retrieved via `request().body()`:

```
public static Result index() {
    RequestBody body = request().body();
    ok("Got body: " + body);
}
```

You can specify the `BodyParser` to use for a particular action using the `@BodyParser.Of` annotation:

```
@BodyParser.Of(BodyParser.Json.class)
public static Result index() {
    RequestBody body = request().body();
    ok("Got json: " + body.asJson());
}
```

The `Http.RequestBody` API

As we just said all body parsers in the Java API will give you a `play.mvc.Http.RequestBody` value. From this body object you can retrieve the request body content in the most appropriate Java type.

Note: The `RequestBody` methods like `asText()` or `asJson()` will return null if the parser used to compute this request body doesn't support this content type. For example in an action method annotated with `@BodyParser.Of(BodyParser.Json.class)`, calling `asXml()` on the generated body will return null.

Some parsers can provide a most specific type than `Http.RequestBody` (ie. a subclass of `Http.RequestBody`). You can automatically cast the request body into another type using the `as(...)` helper method:

```
@BodyParser.Of(BodyLengthParser.class)
public static Result index() {
    BodyLength body = request().body().as(BodyLength.class);
    ok("Request body length: " + body.getLength());
}
```

Default body parser: AnyContent

If you don't specify your own body parser, Play will use the default one guessing the most appropriate content type from the `Content-Type` header:

- **text/plain**: `String`, accessible via `asText()`
- **application/json**: `JsonNode`, accessible via `asJson()`
- **text/xml**: `org.w3c.Document`, accessible via `asXml()`
- **application/form-urlencoded**: `Map<String, String[]>`, accessible via `asFormUrlEncoded()`
- **multipart/form-data**: `Http.MultipartFormData`, accessible via `asMultipartFormData()`
- Any other content type: `Http.RawBuffer`, accessible via `asRaw()`

Example:

```
public static Result save() {
    RequestBody body = request().body();
    String textBody = body.asText();

    if(textBody != null) {
        ok("Got: " + text);
    } else {
        badRequest("Expecting text/plain request body");
    }
}
```

Max content length

Text based body parsers (such as `text`, `json`, `xml` or `formUrlEncoded`) use a max content length because they have to load all the content into memory.

There is a default content length (the default is 100KB).

Tip: The default content size can be defined in `application.conf`:

```
parsers.text.maxLength=128K
```

You can also specify a maximum content length via the `@BodyParser.Of` annotation:

```
// Accept only 10KB of data.
@BodyParser.Of(value = BodyParser.Text.class, maxLength = 10 * 1024)
public static Result index() {
    if(request().body().isMaxSizeExceeded()) {
        return badRequest("Too much data!");
    } else {
        ok("Got body: " + request().body().asText());
    }
}
```

Action composition

This chapter introduces several ways to define generic action functionality.

Reminder about actions

Previously, we said that an action is a Java method that returns a `play.mvc.Result` value. Actually, Play manages internally actions as functions. Because Java doesn't support first class functions, an action provided by the Java API is an instance of `play.mvc.Action`:

```
public abstract class Action {  
  
    public abstract Result call(HttpContext ctx);  
  
}
```

Play builds a root action for you that just calls the proper action method. This allows for more complicated action composition.

Composing actions

You can compose the code provided by the action method with another `play.mvc.Action`, using the `@With` annotation:

```
@With(VerboseAction.class)  
public static Result index() {  
    return ok("It works!");  
}
```

Here is the definition of the `VerboseAction`:

```
public class VerboseAction extends Action.Simple {  
  
    public Result call(HttpContext ctx) throws Throwable {  
        Logger.info("Calling action for " + ctx);  
        return delegate.call(ctx);  
    }  
}
```

At one point you need to delegate to the wrapped action using `delegate.call(...)`.

You also mix with several actions:

```
@With(Authenticated.class, Cached.class)
public static Result index() {
    return ok("It works!");
}
```

Note: `play.mvc.Security.Authenticated` and `play.cache.Cached` annotations and the corresponding predefined Actions are shipped with Play. See the relevant API documentation for more information.

Defining custom action annotations

You can also mark action composition with your own annotation, which must itself be annotated using `@With`:

```
@With(VerboseAction.class)
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Verbose {
    boolean value() default true;
}
```

You can then use your new annotation with an action method:

```
@Verbose(false)
public static Result index() {
    return ok("It works!");
}
```

Your `Action` definition retrieves the annotation as configuration:

```
public class VerboseAction extends Action<Verbose> {

    public Result call(HttpContext ctx) {
        if(configuration.value) {
            Logger.info("Calling action for " + ctx);
        }
        return delegate.call(ctx);
    }
}
```

Annotating controllers

You can also put any action composition annotation directly on the `Controller` class. In this case it will be applied to all action methods defined by this controller.

```
@Authenticated  
public Admin extends Controller {  
  
    ...  
}
```

Handling asynchronous results

Why asynchronous results?

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

Because of the way Play 2.0 works, action code must be as fast as possible (i.e. non blocking). So what should we return as result if we are not yet able to compute it? The response should be a promise of a result!

A `Promise<Result>` will eventually be redeemed with a value of type `Result`. By giving a `Promise<Result>` instead of a normal `Result`, we are able to compute the result quickly without blocking anything. Play will then serve this 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.

How to create a `Promise<Result>`

To create a `Promise<Result>` we need another promise first: the promise that will give us the actual value we need to compute the result:

```
Promise<Double> promiseOfPIValue = computePIAsynchronously();
Promise<Result> promiseOfResult = promiseOfPIValue.map(
    new Function<Double,Result>() {
        public Result apply(Double pi) {
            return ok("PI value computed: " + pi);
        }
    }
);
```

Note: Writing functional composition in Java is really verbose for at the moment, but it should be better when Java supports lambda notation.

Play 2.0 asynchronous API methods give you a `Promise`. This is the case when you are calling an external web service using the `play.libs.WS` API, or if you are using Akka to schedule asynchronous tasks or to communicate with Actors using `play.libs.Akka`.

A simple way to execute a block of code asynchronously and to get a `Promise` is to use the `play.libs.Akka` helpers:

```
Promise<Integer> promiseOfInt = Akka.future(
    new Callable<Integer>() {
        public Integer call() {
            intensiveComputation();
        }
    }
);
```

Note: Here, the intensive computation will just be run on another thread. It is also possible to run it remotely on a cluster of backend servers using Akka remote.

AsyncResult

While we were using `Results.Status` until now, to send an asynchronous result we need an `Results.AsyncResult` that wraps the actual result:

```
public static Result index() {
    Promise<Integer> promiseOfInt = Akka.future(
        new Callable<Integer>() {
            public Integer call() {
                intensiveComputation();
            }
        }
    );
    async(
        promiseOfInt.map(
            new Function<Integer,Result>() {
                public Result apply(Integer i) {
                    return ok("Got result: " + i);
                }
            }
        );
    );
}
```

Note: `async()` is an helper method building an `AsyncResult` from a `Promise<Result>`.

Next: Streaming HTTP responses

Streaming HTTP responses

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 static Result index() {  
    return ok("Hello World")  
}
```

You are not specifying a `Content-Length` header. Of course, because the content you are sending is well known, Play 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, Play must consume the whole response data and load its content into memory.

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.

Play provides easy to use helpers to this common task of serving a local file:

```
public static Result index() {  
    return ok(new java.io.File("/tmp/fileToServe.pdf"));  
}
```

Additionally this helper will also compute the `Content-Type` header from the file name. And it will also add the `Content-Disposition` header to specify how the web browser should handle this response. The default is to ask the web browser to download this file by using `Content-Disposition: attachment; filename=fileToServe.pdf`.

Chunked responses

For now, this works well with streaming file content, since we are able to compute the content length before streaming it. But what about dynamically-computed content with no content size available?

For this kind of response we have to use **Chunked transfer encoding**.

Chunked transfer encoding is a data transfer mechanism in version HTTP 1.1 in which a web server serves content in a series of chunks. This uses the `Transfer-Encoding` HTTP response header instead of the `Content-Length` header, which the protocol would otherwise require. Because the `Content-Length` header is not used, the server does not need to know the length of the content before it starts transmitting a response to the client (usually a web browser). Web servers can begin transmitting responses with dynamically-generated content before knowing the total size of that content.

The size of each chunk is sent right before the chunk itself so that a client can tell when it has finished receiving data for that chunk. The data transfer is terminated by a final chunk of length zero.

http://en.wikipedia.org/wiki/Chunked_transfer_encoding

The advantage is that we can serve data **live**, meaning that we send chunks of data as soon as they are available. The drawback is that since the web browser doesn't know the content size, it is not able to display a proper download progress bar.

Let's say that we have a service somewhere that provides a dynamic `InputStream` that computes some data. We can ask Play to stream this content directly using a chunked response:

```
public static Result index() {
    InputStream is = getDynamicStreamSomewhere();
    return ok(is);
}
```

You can also set up your own chunked response builder. The Play Java API supports both text and binary chunked streams (via `String` and `byte[]`):

```
public static index() {
    // Prepare a chunked text stream
    Chunks<String> chunks = new StringChunks() {

        // Called when the stream is ready
        public void onReady(Chunks.Out<String> out) {
            registerOutChannelSomewhere(out);
        }

    }

    // Serves this stream with 200 OK
    ok(chunks);
}
```

The `onReady` method is called when it is safe to write to this stream. It gives you a `Chunks.Out` channel you can write to.

Let's say we have an asynchronous process (like an `Actor`) somewhere pushing to this stream:

```
public void registerOutChannelSomewhere(Chunks.Out<String> out) {
    out.write("kiki");
    out.write("foo");
    out.write("bar");
    out.close();
}
```

We can inspect the HTTP response sent by the server:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

4
kiki
3
foo
3
bar
0
```

We get three chunks and one final empty chunk that closes the response.

Comet sockets

Using chunked responses to create Comet sockets

An useful usage of **Chunked responses** is to create Comet sockets. A Comet socket is just a chunked `text/html` response containing only `<script>` elements. For each chunk, we write a `<script>` tag containing JavaScript that is immediately executed by the web browser. This way we can send events live to the web browser from the server: for each message, wrap it into a `<script>` tag that calls a JavaScript callback function, and write it to the chunked response.

Let's write a first proof-of-concept: create an enumerator generating `<script>` tags calling the browser `console.log` function:

```
public static Result index() {
    // Prepare a chunked text stream
    Chunks<String> chunks = new StringChunks() {

        // Called when the stream is ready
        public void onReady(Chunks.Out<String> out) {
            out.write("<script>console.log('kiki')</script>");
            out.write("<script>console.log('foo')</script>");
            out.write("<script>console.log('bar')</script>");
            out.close();
        }

    }

    response().setContentType("text/html");

    ok(chunks);
}
```

If you run this action from a web browser, you will see the three events logged in the browser console.

Using the `play.libs.Comet` helper

We provide a Comet helper to handle these comet chunked streams that does almost the same as what we just wrote.

Note: Actually it does more, such as pushing an initial blank buffer data for browser compatibility, and supporting both String and JSON messages.

Let's just rewrite the previous example to use it:

```
public static Result index() {
    Comet comet = new Comet("console.log") {
        public void onConnected() {
            sendMessage("kiki");
            sendMessage("foo");
            sendMessage("bar");
            close();
        }
    };
    ok(comet);
}
```

The forever iframe technique

The standard technique to write a Comet socket is to load an infinite chunked comet response in an iframe and to specify a callback calling the parent frame:

```
public static Result index() {
    Comet comet = new Comet("parent.cometMessage") {
        public void onConnected() {
            sendMessage("kiki");
            sendMessage("foo");
            sendMessage("bar");
            close();
        }
    };
    ok(comet);
}
```

With an HTML page like:

```
<script type="text/javascript">
    var cometMessage = function(event) {
        console.log('Received event: ' + event)
    }
</script>

<iframe src="/comet"></iframe>
```

WebSockets

Using WebSockets instead of Comet sockets

A Comet socket is a kind of hack for sending live events to the web browser. Also, Comet only supports one-way communication from the server to the client. To push events to the server, the web browser can make Ajax requests.

Modern web browsers natively support two-way live communication via WebSockets.

WebSocket is a web technology providing for bi-directional, full-duplex communications channels, over a single Transmission Control Protocol (TCP) socket. The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455.

WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. Because ordinary TCP connections to port numbers other than 80 are frequently blocked by administrators outside of home environments, it can be used as a way to circumvent these restrictions and provide similar functionality with some additional protocol overhead while multiplexing several WebSocket services over a single TCP port. Additionally, it serves a purpose for web applications that require real-time bi-directional communication. Before the implementation of WebSocket, such bi-directional communication was only possible using comet channels; however, a comet is not trivial to implement reliably, and due to the TCP Handshake and HTTP header overhead, it may be inefficient for small messages. The WebSocket protocol aims to solve these problems without compromising security assumptions of the web.

<http://en.wikipedia.org/wiki/WebSocket>

Handling WebSockets

Until now we were using a simple action method to handle standard HTTP requests and send back standard HTTP results. WebSockets are a totally different beast, and can't be handled via standard actions.

To handle a WebSocket your method must return a `WebSocket` instead of a `Result`:

```
public static WebSocket<String> index() {
    return new WebSocket<String>() {

        // Called when the Websocket Handshake is done.
        public void onReady(WebSocket.In<String> in, WebSocket.Out<String> out)
        {

            // For each event received on the socket,
            in.onMessage(new Callback<String>() {
                public void invoke(String event) {

                    // Log events to the console
                    println(event);

                }
            });

            // When the socket is closed.
            in.onClose(new Callback0() {
                public void invoke() {

                    println("Disconnected")

                }
            });

            // Send a single 'Hello!' message
            out.write("Hello!");

        }

    }
}
```

A `WebSocket` has access to the request headers (from the HTTP request that initiates the `WebSocket` connection) allowing you to retrieve standard headers and session data. But it doesn't have access to any request body, nor to the HTTP response.

When the `WebSocket` is ready, you get both `in` and `out` channels.

In this example, we print each message to console and we send a single **Hello!** message.

Tip: You can test your `WebSocket` controller on <http://websocket.org/echo.html>. Just set the location to `ws://localhost:9000`.

Let's write another example that totally discards the input data and closes the socket just after sending the **Hello!** message:

```
public static WebSocket<String> index() {
    return new WebSocket<String>() {

        public void onReady(WebSocket.In<String> in, WebSocket.Out<String> out)
        {
            out.write("Hello!");
            out.close()
        }

    }
}
```

The template engine

A type safe template engine based on Scala

Play 2.0 comes with a new and really powerful Scala-based template engine, whose design was inspired by ASP.NET Razor. Specifically it is:

- **compact, expressive, and fluid**: it minimizes the number of characters and keystrokes required in a file, and enables a fast, fluid coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote server blocks within your HTML. The parser is smart enough to infer this from your code. This enables a really compact and expressive syntax which is clean, fast and fun to type.
- **easy to learn**: it allows you to quickly become productive, with a minimum of concepts. You use simple Scala constructs and all your existing HTML skills.
- **not a new language**: we consciously chose not to create a new language. Instead we wanted to enable Scala developers to use their existing Scala language skills, and deliver a template markup syntax that enables an awesome HTML construction workflow.
- **editable in any text editor**: it doesn't require a specific tool and enables you to be productive in any plain old text editor.

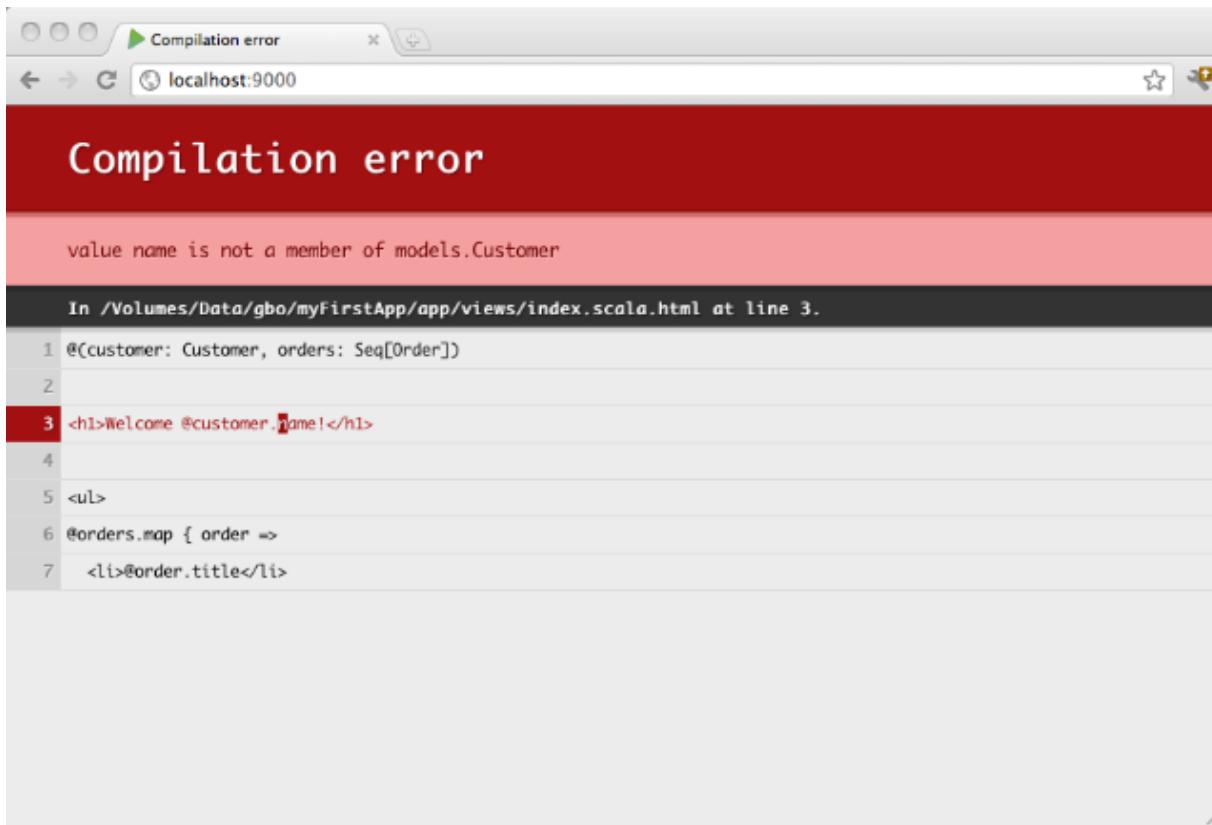
Note: Even though the template engine uses Scala as expression language, this is not a problem for Java developers. You can almost use it as if the language were Java.

Remember that a template is not a place to write complex logic. You don't have to write complicated Scala code here. Most of the time you will just access data from your model objects, as follows:

```
myUser .getProfile () .getUsername ()
```

Parameter types are specified using a suffix syntax. Generic types are specified using the `[]` symbols instead of the usual `<>` Java syntax. For example, you write `List[String]`, which is the same as `List<String>` in Java.

Templates are compiled, so you will see any errors in your browser:



Overview

A Play Scala template is a simple text file that contains small blocks of Scala code. Templates can generate any text-based format, such as HTML, XML or CSV.

The template system has been designed to feel comfortable to those used to working with HTML, allowing front-end developers to easily work with the templates.

Templates are compiled as standard Scala functions, following a simple naming convention. If you create a `views/Application/index.scala.html` template file, it will generate a `views.html.Application.index` class that has a `render()` method.

For example, here is a simple template:

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.getTitle()</li>
}
</ul>
```

You can then call this from any Java code as you would normally call a method on a class:

```
Content html = views.html.Application.index.render(customer, orders);
```

Syntax: the magic '@' character

The Scala template uses `@` as the single special character. Every time this character is encountered, it indicates the beginning of a dynamic statement. You are not required to explicitly close the code block - the end of the dynamic statement will be inferred from your code:

```
Hello @customer.getName()!  
^^^^^^^^^^^^^^^^^^^^  
Dynamic code
```

Because the template engine automatically detects the end of your code block by analysing your code, this syntax only supports simple statements. If you want to insert a multi-token statement, explicitly mark it using brackets:

```
Hello @(customer.getFirstName() + customer.getLastName())!  
^^^^^^^^^^^^^^^^^^^^^  
Dynamic Code
```

You can also use curly brackets, to write a multi-statement block:

```
Hello @{val name = customer.getFirstName() + customer.getLastName(); name}!  
^^^^^^^^^  
Dynamic Code
```

Because `@` is a special character, you'll sometimes need to escape it. Do this by using `@@`:

```
My email is bob@@example.com
```

Template parameters

A template is like a function, so it needs parameters, which must be declared at the top of the template file:

```
@(customer: models.Customer, orders: List[models.Order])
```

You can also use default values for parameters:

```
@(title: String = "Home")
```

Or even several parameter groups:

```
@(title:String)(body: Html)
```

Iterating

You can use the `for` keyword, in a pretty standard way:

```
<ul>
@for(p <- products) {
<li>@p.getName() (@$p.getPrice())</li>
}
</ul>
```

If-blocks

If-blocks are nothing special. Simply use Scala's standard `if` statement:

```
@if(items.isEmpty) {
<h1>Nothing to display</h1>
} else {
<h1>@items.size() items!</h1>
}
```

Declaring reusable blocks

You can create reusable code blocks:

```
@display(product: models.Product) = {
@product.getName() (@$product.getPrice())
}

<ul>
@for(product <- products) {
@display(product)
}
</ul>
```

Note that you can also declare reusable pure code blocks:

```
@title(text: String) = @{
    text.split(' ').map(_.capitalize).mkString(" ")
}

<h1>@title("hello world")</h1>
```

Note: Declaring code block this way in a template can be sometime useful but keep in mind that a template is not the best place to write complex logic. It is often better to externalize these kind of code in a Java class (that you can store under the `views/` package as well if your want).

By convention a reusable block defined with a name starting with `implicit` will be marked as `implicit`:

```
@implicitFieldConstructor = @{} MyFieldConstructor () }
```

Declaring reusable values

You can define scoped values using the `defining` helper:

```
@defining(user.getFirstName() + " " + user.getLastName()) { fullName =>
    <div>Hello @fullName</div>
}
```

Import statements

You can import whatever you want at the beginning of your template (or sub-template):

```
@(customer: models.Customer, orders: List[models.Order])

@import utils._

...
```

Comments

You can write server side block comments in templates using `@* *@`:

```
*****
* This is a comment *
*****@
```

You can put a comment on the first line to document your template into the Scala API doc:

```
@*****
 * Home page.
 *
 * @param msg The message to display
 ****@

@(msg: String)

<h1>@msg</h1>
```

Escaping

By default, dynamic content parts are escaped according to the template type's (e.g. HTML or XML) rules. If you want to output a raw content fragment, wrap it in the template content type.

For example to output raw HTML:

```
<p>
  @Html(article.content)
</p>
```

Common template use cases

Templates, being simple functions, can be composed in any way you want. Below are a few examples of some common scenarios.

Layout

Let's declare a `views/main.scala.html` template that will act as a main layout template:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
  </body>
</html>
```

As you can see, this template takes two parameters: a title and an HTML content block. Now we can use it from another `views/Application/index.scala.html` template:

```
@main(title = "Home") {

  <h1>Home page</h1>

}
```

Note: You can use both named parameters (like `@main(title = "Home")`) and positional parameters, like `@main("Home")`. Choose whichever is clearer in a specific context.

Sometimes you need a second page-specific content block for a sidebar or breadcrumb trail, for example. You can do this with an additional parameter:

```
@(title: String)(sidebar: Html)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
    <section class="sidebar">@sidebar</section>
  </body>
</html>
```

Using this from our ‘index’ template, we have:

```
@main("Home") {
  <h1>Sidebar</h1>

} {
  <h1>Home page</h1>

}
```

Alternatively, we can declare the sidebar block separately:

```
@sidebar = {
  <h1>Sidebar</h1>
}

@main("Home")(sidebar) {
  <h1>Home page</h1>

}
```

Tags (they are just functions right?)

Let’s write a simple `views/tags/notice.scala.html` tag that displays an HTML notice:

```
@(level: String = "error")(body: (String) => Html)

@level match {

  case "success" => {
    <p class="success">
      @body("green")
    </p>
  }

  case "warning" => {
    <p class="warning">
      @body("orange")
    </p>
  }

  case "error" => {
    <p class="error">
      @body("red")
    </p>
  }

}
```

And now let's use it from another template:

```
@import tags._

@notice("error") { color =>
  Oops, something is <span style="color:@color">wrong</span>
}
```

Includes

Again, there's nothing special here. You can just call any other template you like (or in fact any other function, wherever it is defined):

```
<h1>Home</h1>

<div id="side">
  @common.sideBar()
</div>
```

Handling form submission

Defining a form

The `play.data` package contains several helpers to handle HTTP form data submission and validation. The easiest way to handle a form submission is to define a `play.data.Form` that wraps an existing class:

```
public class User {  
    public String email;  
    public String password;  
}
```

```
Form<User> userForm = form(User.class);
```

Note: The underlying binding is done using Spring data binder.

This form can generate a `User` result value from `HashMap<String, String>` data:

```
Map<String, String> anyData = new HashMap();  
anyData.put("email", "bob@gmail.com");  
anyData.put("password", "secret");  
  
User user = userForm.bind(anyData).get();
```

If you have a request available in the scope, you can bind directly from the request content:

```
User user = userForm.bindFromRequest().get();
```

Defining constraints

You can define additional constraints that will be checked during the binding phase using JSR-303 (Bean Validation) annotations:

```
public class User {  
  
    @Required  
    public String email;  
    public String password;  
}
```

Tip: The `play.data.validation.Constraints` class contains several built-in validation annotations.

You can also define an ad-hoc validation by adding a `validate` method to your top object:

```
public class User {  
  
    @Required  
    public String email;  
    public String password;  
  
    public String validate() {  
        if(authenticate(email,password) == null) {  
            return "Invalid email or password";  
        }  
        return null;  
    }  
}
```

Handling binding failure

Of course if you can define constraints, then you need to be able to handle the binding errors.

```
if(userForm.hasErrors()) {  
    return badRequest(form.render(userForm));  
} else {  
    User user = userForm.get();  
    return ok("Got user " + user);  
}
```

Filling a form with initial default values

Sometimes you'll want to fill a form with existing values, typically for editing:

```
userForm.fill(new User("bob@gmail.com", "secret"))
```

Register a custom DataBinder

In case you want to define a mapping from a custom object to a form field string and vice versa you need to register a new Formatter for this object.

For an object like JodaTime's `LocalTime` it could look like this:

```
Formatters.register(LocalTime.class, new SimpleFormatter<LocalTime>() {  
  
    private Pattern timePattern = Pattern.compile(  
        "([012]?\\\\d)(?:[\\\\\\s:\\\\\\_.\\\\\\-]+([0-5]\\\\\\d))?"  
    );  
  
    @Override  
    public LocalTime parse(String input, Locale l) throws ParseException {  
        Matcher m = timePattern.matcher(input);  
        if (!m.find()) throw new ParseException("No valid Input",0);  
        int hour = Integer.valueOf(m.group(1));  
        int min = m.group(2) == null ? 0 : Integer.valueOf(m.group(2));  
        return new LocalTime(hour, min);  
    }  
  
    @Override  
    public String print(LocalTime localTime, Locale l) {  
        return localTime.toString("HH:mm");  
    }  
});
```

Form template helpers

Play provides several helpers to help you render form fields in HTML templates.

Creating a `<form>` tag

The first helper creates the `<form>` tag. It is a pretty simple helper that automatically sets the `action` and `method` tag parameters according to the reverse route you pass in:

```
@helper.form(action = routes.Application.submit()) {  
}
```

You can also pass an extra set of parameters that will be added to the generated HTML:

```
@helper.form(action = routes.Application.submit(), 'id -> "myForm") {  
}
```

Rendering an `<input>` element

There are several input helpers in the `views.html.helper` package. You feed them with a form field, and they display the corresponding HTML form control, with a populated value, constraints and errors:

```
@(myForm: Form[User])  
  
@helper.form(action = routes.Application.submit()) {  
  
    @helper.inputText(myForm("username"))  
  
    @helper.inputPassword(myForm("password"))  
  
}
```

As for the `form` helper, you can specify an extra set of parameters that will be added to the generated HTML:

```
@helper.inputText(myForm("username"), 'id -> "username", 'size -> 30)
```

Note: All extra parameters will be added to the generated HTML, except for ones whose name starts with the `_` character. Arguments starting with an underscore are reserved for field constructor argument (which we will see later).

Handling HTML input creation yourself

There is also a more generic `input` helper that let you code the desired HTML result:

```
@helper.input(myForm("username")) { (id, name, value, args) =>
  <input type="date" name="@name" id="@id" @toHtmlArgs(args)>
}
```

Field constructors

A rendered field does not only consist of an `<input>` tag, but may also need a `<label>` and a bunch of other tags used by your CSS framework to decorate the field.

All input helpers take an implicit `FieldConstructor` that handles this part. The default one (used if there are no other field constructors available in the scope), generates HTML like:

```
<dl class="error" id="username_field">
  <dt><label for="username"></label>Username:</dt>
  <dd><input type="text" name="username" id="username" value=""></dd>
  <dd class="error">This field is required!</dd>
  <dd class="error">Another error</dd>
  <dd class="info">Required</dd>
  <dd class="info">Another constraint</dd>
</dl>
```

This default field constructor supports additional options you can pass in the input helper arguments:

```
'_label -> "Custom label"
'_id -> "idForTheTopDlElement"
'_help -> "Custom help"
'_showConstraints -> false
'_error -> "Force an error"
'_showErrors -> false
```

Twitter bootstrap field constructor

There is another built-in field constructor that can be used with Twitter Bootstrap .

To use it, just import it in the current scope:

```
@import helper.twitterBootstrap._
```

This field constructor generates HTML like the following:

```
<div class="clearfix error" id="username_field">
  <label for="username">Username:</label>
  <div class="input">
    <input type="text" name="username" id="username" value="">
    <span class="help-inline">This field is required!, Another error</span>
    <span class="help-block">Required, Another constraint</d></span>
  </div>
</div>
```

It supports the same set of options as the default field constructor (see above).

Writing your own field constructor

Often you will need to write your own field constructor. Start by writing a template like:

```
@(elements: helper.FieldElements)

<div class="@if(elements.hasErrors) {error}">
  <label for="@elements.id">@elements.label</label>
  <div class="input">
    @elements.input
    <span class="errors">@elements.errors.mkString(", ")</span>
    <span class="help">@elements.infos.mkString(", ")</span>
  </div>
</div>
```

Note: This is just a sample. You can make it as complicated as you need. You have also access to the original field using `@elements.field`.

Now create a `FieldConstructor` somewhere, using:

```
@implicitField = @{ FieldConstructor(myFieldConstructorTemplate.f) }

@inputText(myForm("username"))
```

Handling repeated values

The last helper makes it easier to generate inputs for repeated values. Suppose you have this kind of form definition:

```
val myForm = Form(  
    tuple(  
        "name" -> text,  
        "emails" -> list(email)  
    )  
)
```

Now you have to generate as many inputs for the `emails` field as the form contains. Just use the `repeat` helper for that:

```
@inputText(myForm("name"))  
  
@repeat(myForm("emails"), min = 1) { emailField =>  
  
    @inputText(emailField)  
  
}
```

Use the `min` parameter to display a minimum number of fields, even if the corresponding form data are empty.

Handling and serving JSON requests

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.

By default an action uses an **any content** body parser, which you can use to retrieve the body as JSON (actually as a Jerkson `JsonNode`):

```
public static index sayHello() {
    JsonNode json = request().body().asJson();
    if(json == null) {
        return badRequest("Expecting Json data");
    } else {
        String name = json.findPath("name").getTextView();
        if(name == null) {
            return badRequest("Missing parameter [name]");
        } else {
            return ok("Hello " + name);
        }
    }
}
```

Of course it's way better (and simpler) to specify our own `BodyParser` to ask Play to parse the content body directly as JSON:

```
@BodyParser.Of(Json.class)
public static index sayHello() {
    String name = json.findPath("name").getTextView();
    if(name == null) {
        return badRequest("Missing parameter [name]");
    } else {
        return ok("Hello " + name);
    }
}
```

Note: This way, a 400 HTTP response will be automatically returned for non JSON requests.

You can test it with **cURL** from a command line:

```
curl  
--header "Content-type: application/json"  
--request POST  
--data '{"name": "Guillaume"}'  
http://localhost:9000/sayHello
```

It replies with:

```
HTTP/1.1 200 OK  
Content-Type: text/plain; charset=utf-8  
Content-Length: 15  
  
Hello Guillaume
```

Serving a JSON response

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

```
@BodyParser.Of(Json.class)  
public static index sayHello() {  
    ObjectNode result = Json.newObject();  
    String name = json.findPath("name").getTextView();  
    if(name == null) {  
        result.put("status", "KO");  
        result.put("message", "Missing parameter [name]");  
        return badRequest(result);  
    } else {  
        result.put("status", "OK");  
        result.put("message", "Hello " + name);  
        return ok(result);  
    }  
}
```

Now it replies with:

```
HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8  
Content-Length: 43  
  
{"status": "OK", "message": "Hello Guillaume"}
```

Handling and serving XML requests

Handling an XML request

An XML request is an HTTP request using a valid XML payload as request body. It must specify the `text/xml` MIME type in its `Content-Type` header.

By default, an action uses an **any content** body parser, which you can use to retrieve the body as XML (actually as a `org.w3c.Document`):

```
public static index sayHello() {
    Document dom = request().body().asXml();
    if(dom == null) {
        return badRequest("Expecting Xml data");
    } else {
        String name = XPath.selectText("//name", dom);
        if(name == null) {
            return badRequest("Missing parameter [name]");
        } else {
            return ok("Hello " + name);
        }
    }
}
```

Of course it's way better (and simpler) to specify our own `BodyParser` to ask Play to parse the content body directly as XML:

```
@BodyParser.Of(Xml.class)
public static index sayHello() {
    String name = XPath.selectText("//name", dom);
    if(name == null) {
        return badRequest("Missing parameter [name]");
    } else {
        return ok("Hello " + name);
    }
}
```

Note: This way, a 400 HTTP response will be automatically returned for non-XML requests.

You can test it with **cURL** on the command line:

```
curl  
--header "Content-type: text/xml"  
--request POST  
--data '<name>Guillaume</name>'  
http://localhost:9000/sayHello
```

It replies with:

```
HTTP/1.1 200 OK  
Content-Type: text/plain; charset=utf-8  
Content-Length: 15  
  
Hello Guillaume
```

Serving an XML response

In our previous example, we handled an XML request, but replied with a `text/plain` response. Let's change it to send back a valid XML HTTP response:

```
@BodyParser.Of(Xml.class)  
public static index sayHello() {  
    String name = XPath.selectText("//name", dom);  
    if(name == null) {  
        return badRequest("<message status=\"KO\">Missing parameter [name]</message>");  
    } else {  
        return ok("<message status=\"OK\">Hello " + name + "</message>");  
    }  
}
```

Now it replies with:

```
HTTP/1.1 200 OK  
Content-Type: text/xml; charset=utf-8  
Content-Length: 46  
  
<message status="OK">Hello Guillaume</message>
```

Handling file upload

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:

```
@form(action = routes.Application.upload, 'enctype -> "multipart/form-data") {  
  
    <input type="file" name="picture">  
  
    <p>  
        <input type="submit">  
    </p>  
  
}
```

Now let's define the `upload` action:

```
public static Result upload() {  
    MultipartFormData body = request().body().asMultipartFormData();  
    FilePart picture = body.getFile("picture");  
    if (picture != null) {  
        String fileName = picture.getFilename();  
        String contentType = picture.getContentType();  
        File file = picture.getFile();  
        return ok("File uploaded");  
    } else {  
        flash("error", "Missing file");  
        return redirect(routes.Application.index());  
    }  
}
```

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.

```
public static Result upload() {  
    File file = request().body().asRaw().asFile();  
    return ok("File uploaded");  
}
```

Accessing an SQL database

Configuring JDBC connection pools

Play 2.0 provides a plugin for managing JDBC connection pools. You can configure as many databases you need.

To enable the database plugin, configure a connection pool in the `conf/application.conf` file. By convention the default JDBC data source must be called `default`:

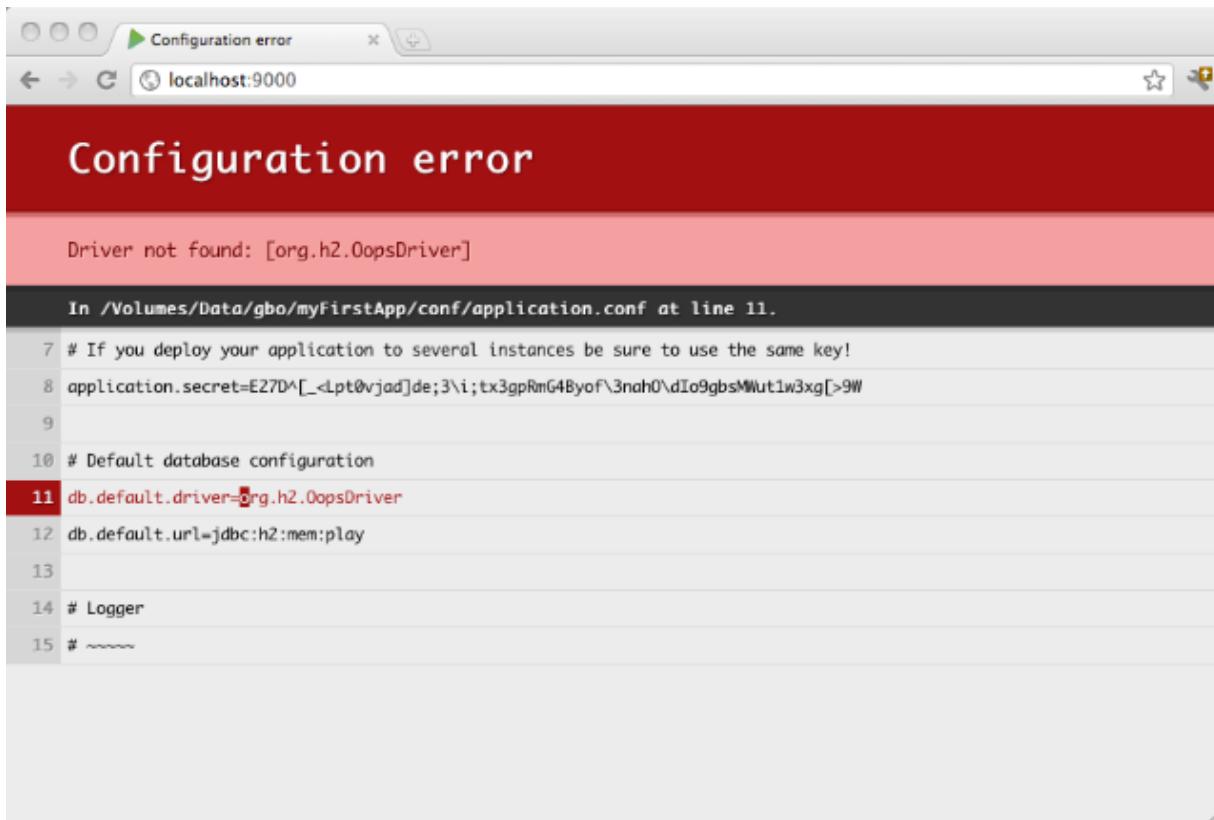
```
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

To configure several data sources:

```
# Orders database
db.orders.driver=org.h2.Driver
db.orders.url="jdbc:h2:mem:orders"

# Customers database
db.customers.driver=org.h2.Driver
db.customers.url="jdbc:h2:mem:customers"
```

If something isn't properly configured, you will be notified directly in your browser:



Accessing the JDBC datasource

The `play.db` package provides access to the configured data sources:

```
import play.db.*;
DataSource ds = DB.getDatasource();
```

Obtaining a JDBC connection

You can retrieve a JDBC connection the same way:

```
Connection connection = DB.getConnection();
```

Exposing the datasource through JNDI

Some libraries expect to retrieve the `Datasource` reference from JNDI. You can expose any Play managed datasource via JNDI by adding this configuration in `conf/application.conf`:

```
db.default.driver=org.h2.Driver  
db.default.url="jdbc:h2:mem:play"  
db.default.jndiName=DefaultDS
```

Importing a Database Driver

Other than for the h2 in-memory database, useful mostly in development mode, Play 2.0 does not provide any database drivers. Consequently, to deploy in production you will have to add your database driver as an application dependency.

For example, if you use MySQL5, you need to add a dependency for the connector:

```
val appDependencies = Seq(  
    // Add your project dependencies here,  
    ...  
    "mysql" % "mysql-connector-java" % "5.1.18"  
    ...  
)
```

Using the Ebean ORM

Configuring Ebean

Play 2.0 comes with the Ebean ORM. To enable it, add the following line to `conf/application.conf`:

```
ebean.default="models.*"
```

This defines a `default` Ebean server, using the `default` data source, which must be properly configured. You can actually create as many Ebean servers you need, and explicitly define the mapped class for each server.

```
ebean.orders="models.Order,models.OrderItem"  
ebean.customers="models.Customer,models.Address"
```

In this example, we have access to two Ebean servers - each using its own database.

For more information about Ebean, see the [Ebean documentation](#).

Using the `play.db.ebean.Model` superclass

Play 2.0 defines a convenient superclass for your Ebean model classes. Here is a typical Ebean class, mapped in Play 2.0:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.ebean.*;
import play.data.format.*;
import play.data.validation.*;

@Entity
public class Task extends Model {

    @Id
    @Constraints.Min(10)
    public Long id;

    @Constraints.Required
    public String name;

    public boolean done;

    @Formats.DateTime(pattern="dd/MM/yyyy")
    public Date dueDate = new Date();

    public static Finder<Long,Task> find = new Finder<Long,Task>(
        Long.class, Task.class
    );

}
```

As you can see, we've added a `find` static field, defining a `Finder` for an entity of type `Task` with a `Long` identifier. This helper field is then used to simplify querying our model:

```
// Find all tasks
List<Task> tasks = Task.find.all();

// Find a task by ID
Task anyTask = Task.find.byId(34L);

// Delete a task by ID
Task.find.ref(34L).delete();

// More complex task query
List<Task> tasks = find.where()
    .ilike("name", "%coco%")
    .orderBy("dueDate asc")
    .findPagingList(25)
    .getPage(1);
```

Transactional actions

By default Ebean will not use transactions. However, you can use any transaction helper provided by Ebean to create a transaction. For example:

```
// run in Transactional scope...
Ebean.execute(new TxRunnable() {
    public void run() {

        // code running in "REQUIRED" transactional scope
        // ... as "REQUIRED" is the default TxType
        System.out.println(Ebean.currentTransaction());

        // find stuff...
        User user = Ebean.find(User.class, 1);
        ...

        // save and delete stuff...
        Ebean.save(user);
        Ebean.delete(order);
        ...

    }
});
```

You can also annotate your action method with `@play.db.ebean.Transactional` to compose your action method with an `Action` that will automatically manage a transaction:

```
@Transactional
public static Result save() {
    ...
}
```

Integrating with JPA

Exposing the datasource through JNDI

JPA requires the datasource to be accessible via JNDI. You can expose any Play-managed datasource via JNDI by adding this configuration in `conf/application.conf`:

```
db.default.driver=org.h2.Driver  
db.default.url="jdbc:h2:mem:play"  
db.default.jndiName=DefaultDS
```

Adding a JPA implementation to your project

There is no built-in JPA implementation in Play 2.0; you can choose any available implementation. For example, to use Hibernate, just add the dependency to your project:

```
val appDependencies = Seq(  
    "org.hibernate" % "hibernate-entitymanager" % "3.6.9.Final"  
)
```

Creating a persistence unit

Next you have to create a proper `persistence.xml` JPA configuration file. Put it into the `conf/META-INF` directory, so it will be properly added to your classpath.

Here is a sample configuration file to use with Hibernate:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="defaultPersistenceUnit" transaction-
type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>DefaultDS</non-jta-data-source>
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
        </properties>
    </persistence-unit>

</persistence>
```

Annotating JPA actions with `@Transactional`

Every JPA call must be done in a transaction so, to enable JPA for a particular action, annotate it with `@play.db.jpa.Transactional`. This will compose your action method with a JPA `Action` that manages the transaction for you:

```
@Transactional
public static Result index() {
    ...
}
```

If your action performs only queries, you can set the `readOnly` attribute to `true`:

```
@Transactional(readOnly=true)
public static Result index() {
    ...
}
```

Using the `play.db.jpa.JPA` helper

At any time you can retrieve the current entity manager from the `play.db.jpa.JPA` helper class:

```
public static Company findById(Long id) {
    return JPA.em().find(Company.class, id);
}
```

The Play cache API

Caching data is a typical optimization in modern applications, and so Play 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 in place in case the data goes missing. This philosophy is one of the fundamentals behind Play, and is different from Java EE, where the session is expected to retain values throughout its lifetime.

The default implementation of the cache API uses EHCache. You can also provide your own implementation via a plugin.

Accessing the Cache API

The cache API is provided by the `play.cache.Cache` object. This requires a cache plugin to be registered.

Note: The API is intentionally minimal to allow various implementations to be plugged in. If you need a more specific API, use the one provided by your Cache plugin.

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

```
Cache.set("item.key", frontPageNews);
```

You can retrieve the data later:

```
News news = Cache.get("item.key");
```

Caching HTTP responses

You can easily create a smart cached action using standard `Action` composition.

Note: Play HTTP `Result` instances are safe to cache and reuse later.

Play provides a default built-in helper for the standard case:

```
@Cached("homePage")
public static Result index() {
    return ok("Hello world");
}
```

Caching in templates

You may also access the cache from a view template.

```
@cache.Cache.getOrElse("cached-content", 3600) {  
    <div>I'm cached for an hour</div>  
}
```

Session cache

Play provides a global cache, whose data are visible to anybody. How would one restrict visibility to a given user? For instance you may want to cache metrics that only apply to a given user.

```
// Generate a unique ID  
String uuid=session("uuid");  
if(uuid==null) {  
    uuid=java.util.UUID.randomUUID().toString();  
    session("uuid", uuid);  
}  
  
// Access the cache  
News userNews = Cache.get(uuid+"item.key");  
if(userNews==null) {  
    userNews = generateNews(uuid);  
    Cache.set(uuid+"item.key", userNews );  
}
```

The Play WS API

Sometimes you want to call other HTTP services from within a Play application. Play supports this via its `play.libs.WS` library, which provides a way to make asynchronous HTTP calls.

A call made by `play.libs.WS` should return a `Promise<Ws.Response>`, which you can handle later with Play's asynchronous mechanisms.

Making HTTP calls

To make an HTTP request, you start with `WS.url()` to specify the URL. Then you get a builder that you can use to specify HTTP options, such as setting headers. You end by calling a method corresponding to the HTTP method you want to use:

```
Promise<WS.Response> HomePage = WS.url("http://mysite.com").get();
```

Alternatively:

```
Promise<WS.Response> result = WS.url("http://localhost:9001").post  
("content");
```

Retrieving the HTTP response result

The call is made asynchronously and you need to manipulate it as a `Promise<WS.Response>` to get the actual content. You can compose several promises and end up with a `Promise<Result>` that can be handled directly by the Play server:

```
import play.libs.F.Function;
import play.libs.WS;
import play.mvc.*;

public class Controller extends Controller {

    public static Result feedTitle(String feedUrl) {
        return async(
            WS.url(feedUrl).get().map(
                new Function<WS.Response, Result>() {
                    public Result apply(WS.Response response) {
                        return ok("Feed title:" + response.asJson().findPath("title"));
                    }
                }
            );
    }

}
```

OpenID Support in Play

OpenID is a protocol for users to access several services with a single account. As a web developer, you can use OpenID to offer users a way to login with an account they already have (their Google account for example). In the enterprise, you can use OpenID to connect to a company's SSO server if it supports it.

The OpenID flow in a nutshell

1. The user gives you his OpenID (a URL)
2. Your server inspect the content behind the URL to produce a URL where you need to redirect the user
3. The user validates the authorization on his OpenID provider, and gets redirected back to your server
4. Your server receives information from that redirect, and check with the provider that the information is correct

The step 1. may be omitted if all your users are using the same OpenID provider (for example if you decide to rely completely on Google accounts).

OpenID in Play Framework

The OpenID API has two important functions:

- `OpenID.redirectURL` calculates the URL where you should redirect the user. It involves fetching the user's OpenID page, this is why it returns a `Promise<String>` rather than a `String`. If the OpenID is invalid, an exception will be thrown.
- `OpenID.verifiedId` inspects the current request to establish the user information, including his verified OpenID. It will do a call to the OpenID server to check the authenticity of the information, this is why it returns a `Promise<UserInfo>` rather than just `UserInfo`. If the information is not correct or if the server check is false (for example if the redirect URL has been forged), the returned `Promise` will be a `Thrown`.

In any case, you should catch exceptions and if one is thrown redirect back the user to the login page with relevant information.

Extended Attributes

The OpenID of a user gives you his identity. The protocol also support getting extended attributes such as the email address, the first name, the last name...

You may request from the OpenID server *optional* attributes and/or *required* attributes. Asking for required attributes means the user can not login to your service if he doesn't provides them.

Extended attributes are requested in the redirect URL:

```
Map<String, String> attributes = new HashMap<String, String>();
attributes.put("email", "http://schema.openid.net/contact/email");
OpenID.redirectURL(
    openid,
    routes.Application.openIDCallback.absoluteURL(),
    attributes
);
```

Attributes will then be available in the `UserInfo` provided by the OpenID server.

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. For fault-tolerance it adopts the ‘Let it crash’ model, which has been used with great success in the telecoms industry to build applications that self-heal - systems that never stop. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

The application actor system

Akka 2.0 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.

A Play application defines a special actor system to be used by the application. This actor system follows the application life-cycle and restarts automatically when the application restarts.

Note: Nothing prevents you from using another actor system from within a Play application. The provided default actor system is just a convenient way to start a few actors without having to set-up your own.

You can access the default application actor system using the `play.libs.Akka` helper:

```
ActorRef myActor = Akka.system().actorOf(new Props(MyActor.class));
```

Configuration

The default actor system configuration is read from the Play application configuration file. For example to configure the default dispatcher of the application actor system, add these lines to the `conf/application.conf` file:

```
akka.default-dispatcher.core-pool-size-max = 64
akka.debug.receive = on
```

Note: You can also configure any other actor system from the same file, just provide a top configuration key.

Converting Akka `Future` to Play `Promise`

When you interact asynchronously with an Akka actor we will get `Future` object. You can easily convert them to play `Promise` using the conversion method provided in `play.libs.Akka.asPromise()`:

```
import static akka.pattern.Patterns.ask;
import play.libs.Akka;
import play.mvc.Result;
import static play.mvc.Results.async;
import play.libs.F.Function;

public static Result index() {
    return async(
        Akka.asPromise(ask(myActor, "hello", 1000)).map(
            new Function<Object,Result>() {
                public Result apply(Object response) {
                    return ok(response.toString());
                }
            }
        )
    );
}
```

Executing a block of code asynchronously

A common use case within Akka is to have some computation performed concurrently without needing the extra utility of an Actor. If you find yourself creating a pool of Actors for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import static play.libs.Akka.future;
import play.libs.F.*;
import java.util.concurrent.Callable;

public static Result index() {
    return async(
        future(new Callable<Integer>() {
            public Integer call() {
                return longComputation();
            }
        }).map(new Function<Integer,Result>() {
            public Result apply(Integer i) {
                return ok("Got " + i);
            }
        })
    );
}
```

Scheduling asynchronous tasks

You can schedule sending messages 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:

```
Akka.system().scheduler().schedule(
    Duration.create(0, TimeUnit.MILLISECONDS),
    Duration.create(30, TimeUnit.MINUTES)
    testActor,
    "tick"
)
```

Alternatively, to run a block of code ten seconds from now:

```
Akka.system().scheduler().scheduleOnce(
    Duration.create(10, TimeUnit.SECONDS),
    new Runnable() {
        public void run() {
            file.delete()
        }
    }
);
```

Externalising messages and internationalization

Specifying languages supported by your application

To specify your application's languages, you need a valid language code, specified by a valid **ISO Language Code**, optionally followed by a valid **ISO Country Code**. For example, `fr` or `en-US`.

To start, you need to specify the languages that your application supports in its `conf/application.conf` file:

```
application.langs=en,en-US,fr
```

Externalizing messages

You can externalize messages in the `conf/messages.xxx` files.

The default `conf/messages` file matches all languages. You can specify additional language messages files, such as `conf/messages.fr` or `conf/messages.en-US`.

You can retrieve messages for the current language using the `play.api.i18n.Messages` object:

```
String title = Messages.get("home.title")
```

You can also specify the language explicitly:

```
String title = Messages.get(new Lang("fr"), "home.title")
```

Note: If you have a `Request` in the scope, it will provide a default `Lang` value corresponding to the preferred language extracted from the `Accept-Language` header and matching one of the application's supported languages.

Formatting messages

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

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

You can then specify parameters as:

```
Messages.get("files.summary", d.files.length, d.name)
```

Retrieving supported languages from an HTTP request

You can retrieve a specific HTTP request's supported languages:

```
public static Result index() {  
    return ok(request().acceptLanguages());  
}
```

Application global settings

The Global object

Defining a `Global` object in your project allows you to handle global settings for your application. This object must be defined in the root package.

```
import play.*;  
  
public class Global extends GlobalSettings {  
  
}
```

Intercepting application start-up and shutdown

You can override the `onStart` and `onStop` operation to be notified of the corresponding application lifecycle events:

```
import play.*;  
  
public class Global extends GlobalSettings {  
  
    @Override  
    public void onStart(Application app) {  
        Logger.info("Application has started");  
    }  
  
    @Override  
    public void onStop(Application app) {  
        Logger.info("Application shutdown...");  
    }  
  
}
```

Providing an application error page

When an exception occurs in your application, the `onError` operation will be called. The default is to use the internal framework error page. You can override this:

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

    @Override
    public Result onError(Throwable t) {
        return internalServerError(
            views.html.errorPage(t)
        );
    }

}
```

Handling action not found

If the framework doesn't find an action method for a request, the `onActionNotFound` operation will be called:

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

    @Override
    public Result onActionNotFound(String uri) {
        return notFound(
            views.html.pageNotFound(uri)
        );
    }

}
```

The `onBadRequest` operation will be called if a route was found, but it was not possible to bind the request parameters :

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

    @Override
    public Result onBadRequest(String uri, String error) {
        return badRequest("Don't try to hack the URI!");
    }

}
```

Intercepting requests

Overriding onRequest

One important aspect of the `GlobalSettings` class is that it provides a way to intercept requests and execute business logic before a request is dispatched to an action.

For example:

```
import play.*;

public class Global extends GlobalSettings {

    @Override
    public Action onRequest(Request request, Method actionMethod) {
        System.out.println("before each request..." + request.toString());
        return super.onRequest(request, actionMethod);
    }
}
```

It's also possible to intercept a specific action method. This can be achieved via Action composition.

Testing your application

Test source files must be placed in your application's `test` folder. You can run tests from the Play console using the `test` and `test-only` tasks.

Using JUnit

The default way to test a Play 2 application is with JUnit.

```
package test;

import org.junit.*;
import play.mvc.*;
import play.test.*;
import play.libs.F.*;

import static play.test.Helpers.*;
import static org.fest.assertions.Assertions.*;

public class SimpleTest {

    @Test
    public void simpleCheck() {
        int a = 1 + 1;
        assertThat(a).isEqualTo(2);
    }
}
```

Running in a fake application

If the code you want to test depends on a running application, you can easily create a `FakeApplication` on the fly:

```
@Test
public void findById() {
    running(fakeApplication(), new Runnable() {
        public void run() {
            Computer macintosh = Computer.find.byId(211);
            assertThat(macintosh.name).isEqualTo("Macintosh");
            assertThat(formatted(macintosh.introduced)).isEqualTo("1984-01-24");
        }
    });
}
```

You can also pass (or override) additional application configuration, or mock any plugin. For example to create a `FakeApplication` using a `default` in-memory database:

```
fakeApplication(inMemoryDatabase())
```

Writing functional tests

Testing a template

As a template is a standard Scala function, you can execute it from a test and check the result:

```
@Test
public void renderTemplate() {
    Content html = views.html.index.render("Coco");
    assertThat(contentType(html)).isEqualTo("text/html");
    assertThat(contentAsString(html)).contains("Coco");
}
```

Testing your controllers

You can also retrieve an action reference from the reverse router, such as `controllers.routes.ref.Application.index`. You can then invoke it:

```
@Test
public void callIndex() {
    Result result = callAction(
        controllers.routes.ref.Application.index("Kiki")
    );
    assertThat(status(result)).isEqualTo(OK);
    assertThat(contentType(result)).isEqualTo("text/html");
    assertThat(charset(result)).isEqualTo("utf-8");
    assertThat(contentAsString(result)).contains("Hello Kiki");
}
```

Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
@Test
public void badRoute() {
    Result result = routeAndCall(fakeRequest(GET, "/xx/Kiki"));
    assertThat(result).isNull();
}
```

Starting a real HTTP server

Sometimes you want to test the real HTTP stack from with your test. You can do this by starting a test server:

```
@Test
public void testInServer() {
    running(testServer(3333), new Callback0() {
        public void invoke() {
            assertThat(
                WS.url("http://localhost:3333").get().get().status
            ).isEqualTo(OK);
        }
    });
}
```

Testing from within a web browser

If you want to test your application from with a Web browser, you can use Selenium WebDriver. Play will start the WebDriver for you, and wrap it in the convenient API provided by FluentLenium.

```
@Test
public void runInBrowser() {
    running(testServer(3333), HTMLUNIT, new Callback<TestBrowser>() {
        public void invoke(TestBrowser browser) {
            browser.goTo("http://localhost:3333");
            assertThat(browser.$("#title").getTexts().get(0)).isEqualTo
("Hello Guest");
            browser.$("a").click();
            assertThat(browser.url()).isEqualTo("http://localhost:3333/Coco");
;
            assertThat(browser.$("#title", 0).getText()).isEqualTo("Hello
Coco");
        }
    });
}
```

Your first Play application

Let's write a simple task list application with Play 2.0 and deploy it to the cloud.

Prerequisites

First of all, make sure that you have a working Play installation. You only need Java (version 6 minimum), and to unzip the Play binary package to start; everything is included.

As we will use the command line a lot, it's better to use a Unix-like OS. If you run a Windows system, it will also work fine; you'll just have to type a few commands in the command prompt.

You will of course need a text editor. If you are used-to a fully-featured Java IDE, such as Eclipse or IntelliJ, you can of course use it. However, with Play you can have fun working with a simple text editor like TextMate, Emacs or vi. This is because the framework manages compilation and the deployment process itself.

Project creation

Now that Play is correctly installed, it's time to create the new application. Creating a Play application is pretty easy and fully managed by the Play command line utility. This encourages a standard project layout across all Play applications.

Open a new command line and enter:

```
$ play new todolist
```

The Play tool will ask you a few questions. Choose to create a **simple Java application** project template.

```
$ play new todolist
[Play Logo]
play! 2.0-RC1-SNAPSHOT, http://www.playframework.org

The new application will be created in /private/tmp/todolist

What is the application name?
> todolist

Which template do you want to use for this new application?

1 - Create a simple Scala application
2 - Create a simple Java application
3 - Create an empty project

> 1

OK, application todolist is created.

Have fun!
```

The `play new` command creates a new directory `todolist/` and populates it with a series of files and directories. The most important are as follows.

- `app/` contains the application's core, split between models, controllers and views directories. This is the directory where `.java` source files live.
- `conf/` contains all the application's configuration files, especially the main `application.conf` file, the `routes` definition files and the `messages` files used for internationalization.
- `project/` contains the build scripts. The build system is based on sbt. But a new play application comes with a default build script that will just work for our application.
- `public/` contains all the publicly available resources, which includes JavaScript, stylesheets and images directories.
- `test/` contains all the application tests. Tests can be written as JUnit tests.

Because Play uses UTF-8 as the single encoding, it's very important that all text files hosted in these directories use this encoding. Make sure to configure your text editor accordingly.

Using the Play console

Once you have an application created, you can run the Play console. Go to the new `todolist/` directory and run:

```
$ play
```

This launches the Play console. There are several things you can do from the Play console, but let's start by running the application. From the console prompt, type `run`:

```
[todolist] $ run
```

2. java

```
$ play
[info] Loading project definition from /private/tmp/todolist/project
[info] Set current project to todolist (in build file:/private/tmp/todolist/)

play! 2.0-RC1-SNAPSHOT, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[todolist] $ run

[info] Updating {file:/private/tmp/todolist/}todolist...
[info] Done updating.
--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)
```

Now the application is running in development mode. Open a browser at <http://localhost:9000/>:

The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL localhost:9000.
- Page Title:** Welcome to Play 2.0
- Page Content:**
 - A green header bar says "Your new application is ready."
 - The main content area says "Welcome to Play 2.0".
 - A yellow box at the bottom left says "You are using Play 2.0-RC1-SNAPSHOT".
 - A horizontal line separates the content from the sidebar.
 - A sidebar on the right contains links for "Browse" (Local documentation, Browse the Scala API), "Start here" (Using the Play console, Setting up your preferred IDE, Your first application), and a search bar.
 - At the bottom, it says "So Play has invoked the controllers.Application.index method to obtain the Action to execute:" followed by the code:

```
def index = Action {
  Ok(views.html.index("Your new application is ready!"))
```

Note: Read more about [The Play Console](#).

Overview

Let's see how the new application can display this page.

The main entry point of your application is the `conf/routes` file. This file defines all of the application's accessible URLs. If you open the generated routes file you will see this first *route*:

```
GET      /      controllers.Application.index()
```

That simply tells Play that when the web server receives a GET request for the `/` path, it must call the `controllers.Application.index()` method.

Let's see what the `controllers.Application.index()` method looks like. Open the `todolist/app/controllers/Application.java` source file:

```
package controllers;

import play.*;
import play.mvc.*;

import views.html.*;

public class Application extends Controller {

    public static Result index() {
        return ok(index.render("Your new application is ready."));
    }

}
```

You see that `controllers.Application.index()` returns a `Result`. All action methods must return a `Result`, which represents the HTTP response to send back to the web browser.

Note: Read more about [Actions](#).

Here, the action returns a **200 OK** response with an HTML response body. The HTML content is provided by a template. Play templates are compiled to standard Java methods, here as `views.html.index.render(String message)`.

This template is defined in the `app/views/index.scala.html` source file:

```
@(message: String)

@main("Welcome to Play 2.0") {

    @play20.welcome(message)

}
```

The first line defines the function signature. Here it takes a single `String` parameter. Then the template content mixes HTML (or any text-based language) with Scala statements. The Scala statements start with the special `@` character.

Note: Don't worry about the template engine using Scala as its expression language. This is not a problem for a Java developer, and you can almost use it as the language was Java.

Development work-flow

Now let's make some modifications to the new application. In the `Application.java` change the content of the response:

```
public static Result index() {
    return ok("Hello world");
}
```

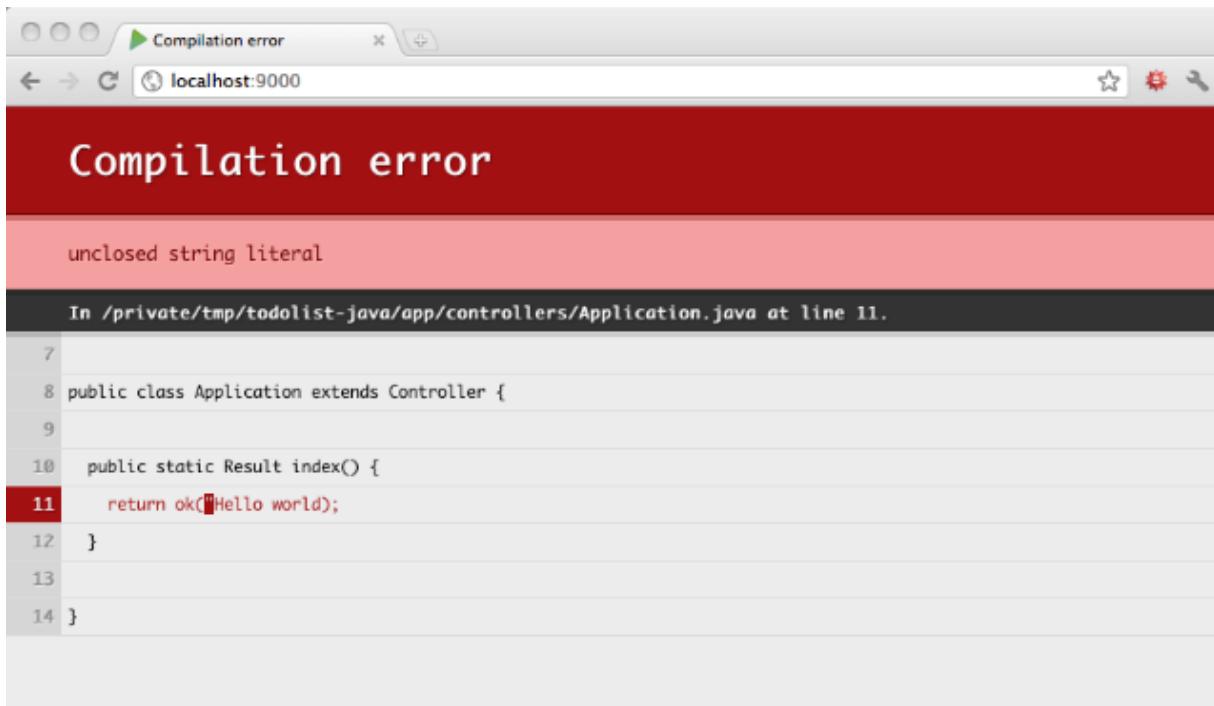
With this change, the `index` action will now respond with a simple `text/plain` **Hello world** response. To see this change, just refresh the home page in your browser:



There is no need to compile the code yourself or restart the server to see the modification. It is automatically reloaded when a change is detected. But what happens when you make a mistake in your code? Let's try:

```
public static Result index() {
    return ok("Hello world");
}
```

Now reload the home page in your browser:



```
7  
8 public class Application extends Controller {  
9  
10    public static Result index() {  
11        return ok("Hello world");  
12    }  
13  
14 }
```

As you can see, errors are beautifully displayed directly in your browser.

Preparing the application

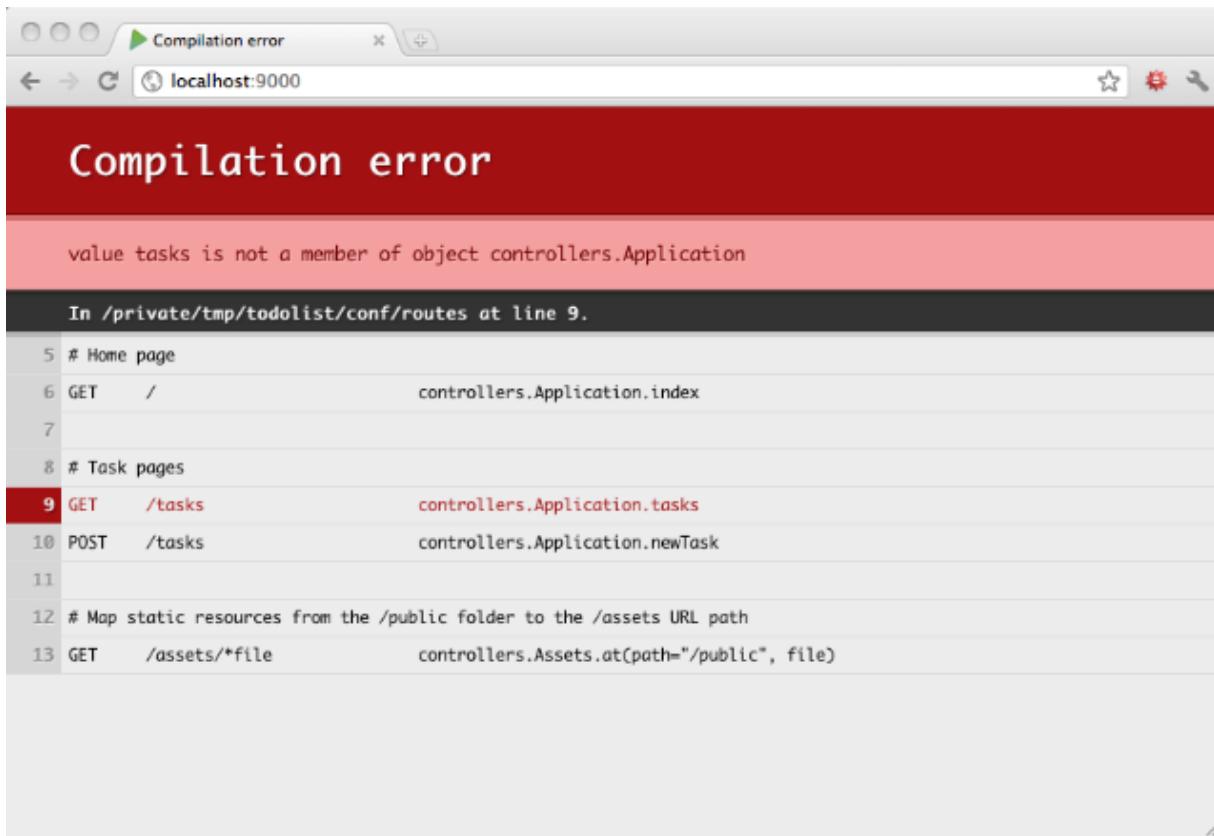
For our todo list application, we need a few actions and the corresponding URLs. Let's start by defining the **routes**.

Edit the `conf/routes` file:

```
# Home page  
GET      /           controllers.Application.index()  
  
# Tasks  
GET      /tasks      controllers.Application.tasks()  
POST     /tasks      controllers.Application.newTask()  
POST     /tasks/:id/delete controllers.Application.deleteTask(id: Long)
```

We create a route to list all tasks, and a couple of others to handle task creation and deletion. The route to handle task deletion defines a variable argument `id` in the URL path. This value is then passed to the `deleteTask` action method.

Now if you reload in your browser, you will see that Play cannot compile your routes files:



This is because the routes reference non-existent action methods. So let's add them to the `Application.java` file:

```
public class Application extends Controller {

    public static Result index() {
        return ok(index.render("Your new application is ready."));
    }

    public static Result tasks() {
        return TODO;
    }

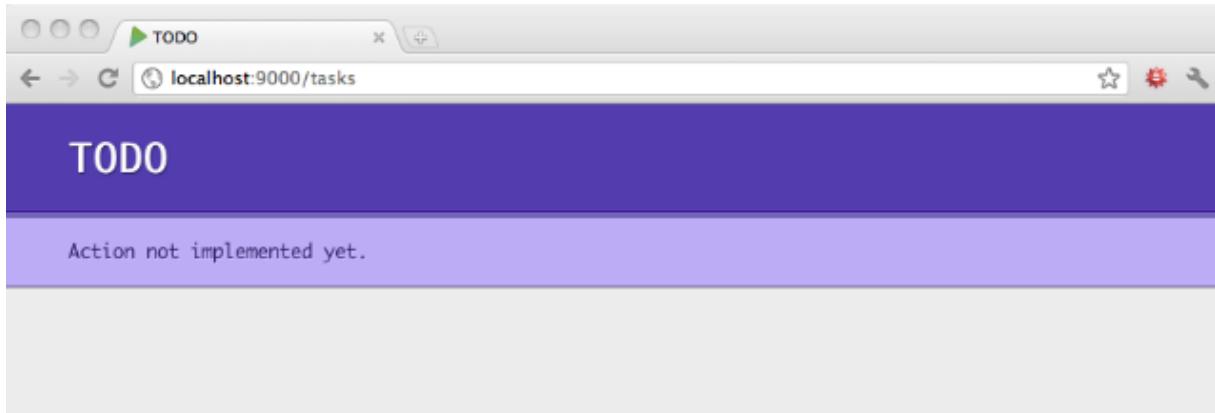
    public static Result newTask() {
        return TODO;
    }

    public static Result deleteTask(Long id) {
        return TODO;
    }

}
```

As you see we use `TODO` as result in our actions implementation. Because we don't want to write the actions implementation yet, we can use the built-in `TODO` result that will return a `503 Not Implemented` response.

You can try to access the `http://localhost:9000/tasks` to see that:



Now the last thing we need to fix before starting the action implementation is the `index` action. We want it to redirect automatically to the tasks list page:

```
public static Result index() {  
    return redirect(routes.Application.tasks());  
}
```

As you see we use `redirect` instead of `ok` to specify a `303 See Other` response type. We also use the reverse router to get the URL needed to fetch the `tasks` actions.

Note: Read more about the Router and reverse router.

Prepare the `Task` model

Before continuing the implementation we need to define what a `Task` looks like in our application.

Create a `class` for it in the `app/models/Task.java` file:

```
package models;

import java.util.*;

public class Task {

    public Long id;
    public String label;

    public static List<Task> all() {
        return new ArrayList<Task>();
    }

    public static void create(Task task) {
    }

    public static void delete(Long id) {
    }

}
```

We have also created a bunch of static methods to manage `Task` operations. For now we wrote dummy implementation for each operation, but later in this tutorial we will write implementations that will store the tasks into a relational database.

The application template

Our simple application will use a single Web page containing both the tasks list and the task creation form. Let's modify the `index.scala.html` template for that:

```
@(tasks: List[Task], taskForm: Form[Task])

@import helper._

@main("Todo list") {

    <h1>@tasks.size() task(s)</h1>

    <ul>
        @for(task <- tasks) {
            <li>
                @task.label

                @form(routes.Application.deleteTask(task.id)) {
                    <input type="submit" value="Delete">
                }
            </li>
        }
    </ul>

    <h2>Add a new task</h2>

    @form(routes.Application.newTask()) {

        @inputText(taskForm("label"))

        <input type="submit" value="Create">

    }
}
```

We changed the template signature to take 2 parameters :

- A list of tasks to display
- A task form

We also imported `helper._` that give us the form creation helpers, typically the `form` function that creates the HTML `<form>` with filled `action` and `method` attributes, and the `inputText` function that creates the HTML input given a form field.

Note: Read more about the [Templating system](#) and [Forms helper](#).

The task form

A `Form` object encapsulates an HTML form definition, including validation constraints. Let's create a form for our `Task` class. Add this to your `Application` controller:

```
static Form<Task> taskForm = form(Task.class);
```

The type of `taskForm` is then `Form<Task>` since it is a form generating a simple `Task`. You also need to import `play.data.*`.

We can add a constraint to the `Task` type using **JSR-303** annotations. Let's make the `label` field required:

```
package models;

import java.util.*;

import play.data.validation.Constraints.*;

public class Task {

    public Long id;

    @Required
    public String label;

    ...
}
```

Note: Read more about the `Form` definitions.

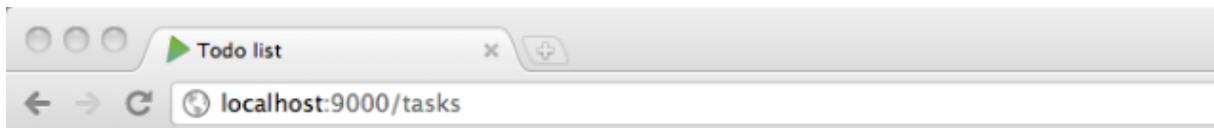
Rendering the first page

Now we have all elements needed to display the application page. Let's write the `tasks` action:

```
public static Result tasks() {
    return ok(
        views.html.index.render(Task.all(), taskForm)
    );
}
```

It renders a **200 OK** result filled with the HTML rendered by the `index.scala.html` template called with the tasks list and the task form.

You can now try to access `http://localhost:9000/tasks` in your browser:



0 task(s)

Add a new task

label

Required

[Create](#)

Handling the form submission

For now if we submit the task creation form, we still get the TODO page. Let's write the implementation of the `newTask` action:

```
public static Result newTask() {
    Form<Task> filledForm = taskForm.bindFromRequest();
    if(filledForm.hasErrors()) {
        return badRequest(
            views.html.index.render(Task.all(), filledForm)
        );
    } else {
        Task.create(filledForm.get());
        return redirect(routes.Application.tasks());
    }
}
```

We use `bindFromRequest` to create a new form filled with the request data. If there are any errors in the form, we redisplay it (here we use **400 Bad Request** instead of **200 OK**). If there are no errors, we create the task and then redirect to the tasks list.

Note: Read more about the [Form submissions](#).

Persist the tasks in a database

It's now time to persist the tasks in a database to make the application useful. Let's start by enabling a database in our application. In the `conf/application.conf` file, add:

```
db.default.driver=org.h2.Driver  
db.default.url="jdbc:h2:mem:play"
```

For now we will use a simple in memory database using **H2**. No need to restart the server, refreshing the browser is enough to set up the database.

We will use **EBean** in this tutorial to query the database. So you'll have to enable it in the `application.conf` file as well:

```
ebean.default="models.*"
```

By doing this we create an Ebean server connected to the `default` datasource, managing all entities found in the `models` package. Now it's time to transform our `Task` class to a valid EBean entity:

```
package models;  
  
import java.util.*;  
  
import play.db.ebean.*;  
import play.data.validation.Constraints.*;  
  
import javax.persistence.*;  
  
@Entity  
public class Task extends Model {  
  
    @Id  
    public Long id;  
  
    @Required  
    public String label;  
  
    public static Finder<Long,Task> find = new Finder(  
        Long.class, Task.class  
    );  
  
    ...
```

We made the `Task` class extend the `play.db.ebean.Model` super class to have access to Play built-in Ebean helper. We also added proper persistence annotations, and created a `find` helper to initiate queries.

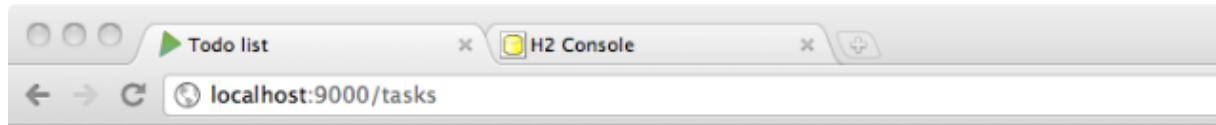
Let's implement the CRUD operations:

```
public static List<Task> all() {
    return find.all();
}

public static void create(Task task) {
    task.save();
}

public static void delete(Long id) {
    find.ref(id).delete();
}
```

Now you can play again with the application, creating new tasks should work.



2 task(s)

- Complete the tutorial
[Delete](#)
- Buy milk
[Delete](#)

Add a new task

label
Required

[Create](#)

Note: Read more about Ebean.

Deleting tasks

Now that we can create tasks, we need to be able to delete them. Very simple, we just need to finish the implementation of the `deleteTask` action:

```
public static Result deleteTask(Long id) {
    Task.delete(id);
    return redirect(routes.Application.tasks());
}
```

Deploying to Heroku

All features are completed. It's time to deploy our application in production. Let's deploy it to heroku. First you have to create a `Procfile` for Heroku in the root application directory:

```
web: target/start -Dhttp.port=${PORT} -DapplyEvolutions.default=true -  
Ddb.default.url=${DATABASE_URL} -Ddb.default.driver=org.postgresql.Driver
```

Note: Read more about [Deploying to Heroku](#).

Using system properties we override the application configuration when running on Heroku. But since heroku provides a PostgreSQL database we'll have to add the required driver to our application dependencies.

Specify it into the `project/Build.scala` file:

```
val appDependencies = Seq(  
  "postgresql" % "postgresql" % "8.4-702.jdbc4"  
)
```

Note: Read more about [Dependencies management](#).

Heroku uses **git** to deploy your application. Let's initialize the git repository:

```
$ git init  
$ git add .  
$ git commit -m "init"
```

Now we can create the application on Heroku:

```
$ heroku create --stack cedar  
  
Creating warm-frost-1289... done, stack is cedar  
http://warm-1289.herokuapp.com/ | git@heroku.com:warm-1289.git  
Git remote heroku added
```

And then deploy it using a simple `git push heroku master`:

```
$ git push heroku master

Counting objects: 34, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (34/34), 35.45 KiB, done.
Total 34 (delta 0), reused 0 (delta 0)

-----> Heroku receiving push
-----> Scala app detected
-----> Building app with sbt v0.11.0
-----> Running: sbt clean compile stage
...
-----> Discovering process types
      Procfile declares types -> web
-----> Compiled slug size is 46.3MB
-----> Launching... done, v5
      http://8044.herokuapp.com deployed to Heroku

To git@heroku.com:floating-lightning-8044.git
 * [new branch]      master -> master
```

Heroku will build your application and deploy it to a node somewhere on the cloud. You can check the state of the application's processes:

```
$ heroku ps

Process     State          Command
----- -----
web.1       up for 10s    target/start
```

It's started, you can now open it in your browser.

Your first application is now up and running in production!

Detailed topics

1. The Build system
 1. About sbt settings
 2. Manage application dependencies
 3. Working with sub-projects
2. Working with public assets
 1. Using CoffeeScript
 2. Using LESS CSS
 3. Using Google Closure Compiler
3. Managing database evolutions
4. Configuration file syntax and features
 1. Configuring the JDBC connection pool
 2. Configuring the internal Akka system
 3. Configuring logging
5. Deploying your application
 1. Creating a standalone version of your application
 2. Additional configuration
 3. Deploying to Heroku
 4. Set-up a front-end HTTP server
6. Hacking Play 2.0
 1. Building Play 2.0 from source
 2. CI server at Cloudbees
 3. Repositories
 4. Issue tracker
 5. Pull requests
 6. Contributor guidelines

The Build System

The Play 2.0 build system is based on sbt, a minimally non-intrusive build tool for Scala and Java projects.

The `/project` directory

All the build configuration is stored in the `project` directory. This folder contains 3 main files:

- `build.properties`: This is a marker file that describes the sbt version used.
- `Build.scala`: This is the application project build description.
- `plugins.sbt`: SBT plugins used by the project build.

Default build for a Play 2.0 application

The default build generated by the `play new` command looks like this:

```
import sbt._  
import Keys._  
import PlayProject._  
  
object ApplicationBuild extends Build {  
  
    val appName          = "Your application"  
    val appVersion       = "1.0"  
  
    val appDependencies = Seq(  
        // Add your project dependencies here,  
    )  
  
    val main = PlayProject(  
        appName, appVersion, appDependencies, mainLang = SCALA  
    ).settings(  
        // Add your own project settings here  
    )  
  
}
```

It is written this way to make it easy to define standard options like application name, version and dependencies.

Note that every sbt feature is available in a Play 2.0 project.

Play plugin for sbt

The Play console and all development features like live reloading are implemented via a sbt plugin. It is registered in the `plugins.sbt` file:

```
addSbtPlugin("play" % "sbt-plugin" % "2.0")
```

About SBT Settings

About sbt settings

The sbt build script defines settings for your project. You can also define your own custom settings for your project, as described in the sbt documentation .

To set a basic setting, use the `:=` operator:

```
val main = PlayProject(appName, appVersion, appDependencies).settings(  
  confDirectory := "myConfFolder"  
)
```

Default settings for Java applications

Play 2.0 defines a default set of settings suitable for Java-based applications. To enable them add the `defaultJavaSettings` set of settings to your application definition:

```
val main = PlayProject(appName, appVersion, appDependencies, mainLang =  
  JAVA)
```

These default settings mostly define the default imports for generated templates. For example, it imports `java.lang.*`, so types like `Long` are the Java ones by default instead of the Scala ones. It also imports `java.util.*` so the default collection library will be the Java one.

Default settings for Scala applications

Play 2.0 defines a default set of settings suitable for Scala-based applications. To enable them add the `defaultScalaSettings` set of settings to your application definition:

```
val main = PlayProject(appName, appVersion, appDependencies, mainLang =  
  SCALA)
```

These default settings define the default imports for generated templates (such as internationalized messages, and core APIs).

Play project settings with their default value

When you define your sbt project using `PlayProject` instead of `Project`, you will get a default set of settings. Here is the default configuration:

```
resolvers ++= Seq(
```

```
target <= baseDirectory / "target",

sourceDirectory in Compile <= baseDirectory / "app",

confDirectory <= baseDirectory / "conf",

scalaSource in Compile <= baseDirectory / "app",

javaSource in Compile <= baseDirectory / "app",

distDirectory <= baseDirectory / "dist",

libraryDependencies += "play" %% "play" % play.core.PlayVersion.current,

sourceGenerators in Compile <+= (confDirectory, sourceManaged in Compile)
map RouteFiles,

sourceGenerators in Compile <+= (sourceDirectory in Compile, sourceManaged in Compile, templatesTypes, templatesImport) map ScalaTemplates,

commands ++= Seq(
  playCommand, playRunCommand, playStartCommand, playHelpCommand,
  h2Command, classpathCommand, licenseCommand, computeDependenciesCommand
),

shellPrompt := playPrompt,

copyResources in Compile <= (copyResources in Compile, playCopyResources)
map { (r, pr) => r ++ pr },

mainClass in (Compile, run) := Some(classOf[play.core.server.NettyServer]
  .getName),

compile in (Compile) <= PostCompile,

dist <= distTask,

computeDependencies <= computeDependenciesTask,

playCopyResources <= playCopyResourcesTask,

playCompileEverything <= playCompileEverythingTask,

playPackageEverything <= playPackageEverythingTask,

playReload <= playReloadTask,

playStage <= playStageTask,

cleanFiles <+= distDirectory.identity,

resourceGenerators in Compile <+= LessCompiler,
```

```
playResourceDirectories += Seq.empty ++ Seq[  
  playResourceDirectories <+= baseDirectory / "conf",  
  playResourceDirectories <+= baseDirectory / "public",  
  templatesImport := Seq("play.api.templates._",  
    "play.api.templates.PlayMagic._"),  
  templatesTypes := {  
    case "html" => ("play.api.templates.Html",  
      "play.api.templates.HtmlFormat")  
    case "txt" => ("play.api.templates.Txt", "play.api.templates.TxtFormat")  
    case "xml" => ("play.api.templates.Xml", "play.api.templates.XmlFormat")  
  }  
}
```

Managing library dependencies

Unmanaged dependencies

Most people end up using managed dependencies - which allows for fine-grained control, but unmanaged dependencies can be simpler when starting out.

Unmanaged dependencies work like this: create a `lib/` directory in the root of your project and then add jar files to that directory. They will automatically be added to the application classpath. There's not much else to it!

There's nothing to add to `project/Build.scala` to use unmanaged dependencies, though you could change a configuration key if you'd like to use a directory different to `lib`.

Managed dependencies

Play 2.0 uses Apache Ivy (via sbt) to implement managed dependencies, so if you're familiar with Maven or Ivy, you won't have much trouble.

Most of the time, you can simply list your dependencies in the `project/Build.scala` file. It's also possible to write a Maven POM file or Ivy configuration file to externally configure your dependencies, and have sbt use those external configuration files.

Declaring a dependency looks like this (defining `group`, `artifact` and `revision`):

```
val appDependencies = Seq(  
    "org.apache.derby" % "derby" % "10.4.1.3"  
)
```

or like this, with an optional `configuration`:

```
val appDependencies = Seq(  
    "org.apache.derby" % "derby" % "10.4.1.3" % "test"  
)
```

Of course, sbt (via Ivy) has to know where to download the module. If your module is in one of the default repositories sbt comes with, this will just work.

Getting the right Scala version with `%%`

If you use `groupId %% artifactID % revision` instead of `groupId % artifactID % revision` (the difference is the double `%%` after the `groupId`), sbt will add your project's Scala version to the artifact name. This is just a shortcut.

You could write this without the `%%`:

```
val appDependencies = Seq(  
    "org.scala-tools" % "scala-stm_2.9.1" % "0.3"  
)
```

Assuming the `scalaVersion` for your build is `2.9.1`, the following is identical:

```
val appDependencies = Seq(  
    "org.scala-tools" %% "scala-stm" % "0.3"  
)
```

Resolvers

Not all packages live on the same server; sbt uses the standard Maven2 repository and the Scala Tools Releases (<http://scala-tools.org/repo-releases>) repositories by default. If your dependency isn't on one of the default repositories, you'll have to add a resolver to help Ivy find it.

Use the `resolvers` setting key to add your own resolver.

```
resolvers += name at location
```

For example:

```
resolvers += "Scala-Tools Maven2 Snapshots Repository" at "http://scala-  
tools.org/repo-snapshots"
```

sbt can search your local Maven repository if you add it as a repository:

```
resolvers += (  
    "Local Maven Repository" at "file://" + Path.userHome.getAbsolutePath + "/.m2/  
repository"  
)
```

Final example

Here is a final example, for a project defining several managed dependencies, with custom resolvers:

```
import sbt._  
import Keys._  
import PlayProject._  
  
object ApplicationBuild extends Build {  
  
    val appName          = "My first application"  
    val appVersion       = "1.0"  
  
    val appDependencies = Seq(  
  
        "org.scala-tools" %% "scala-stm" % "0.3",  
        "org.apache.derby" % "derby" % "10.4.1.3" % "test"  
  
    )  
  
    val main = PlayProject(appName, appVersion, appDependencies).settings  
(defaultScalaSettings :_*)  
    .settings(  
  
        resolvers += "JBoss repository" at "https://repository.jboss.org/  
nexus/content/repositories/",  
        resolvers += "Scala-Tools Maven2 Snapshots Repository" at "http://  
scala-tools.org/repo-snapshots"  
  
    )  
  
}
```

Working with sub-projects

A complex project is not necessarily composed of a single Play application. You may want to split a large project into several smaller applications, or even extract some logic into a standard Java or Scala library that has nothing to do with a Play application.

It will be helpful to read the SBT documentation on multi-project builds. Sub-projects do not have their own build file, but share the parent project's build file.

Adding a simple library sub-project

You can make your application depend on a simple library project. Just add another sbt project definition in your `project/Build.scala` build file:

```
import sbt._  
import Keys._  
import PlayProject._  
  
object ApplicationBuild extends Build {  
  
    val appName          = "my-first-application"  
    val appVersion       = "1.0"  
  
    val appDependencies = Seq()  
  
    val mySubProject = Project("my-library", file("modules/myLibrary"))  
  
    val main = PlayProject(  
        appName, appVersion, appDependencies  
    ).dependsOn(mySubProject)  
}
```

Here we have defined a sub-project in the application's `modules/myLibrary` folder. This sub-project is a standard sbt project, using the default layout:

```
modules  
└ myLibrary  
  └ src  
    └ main  
      └ java  
      └ scala
```

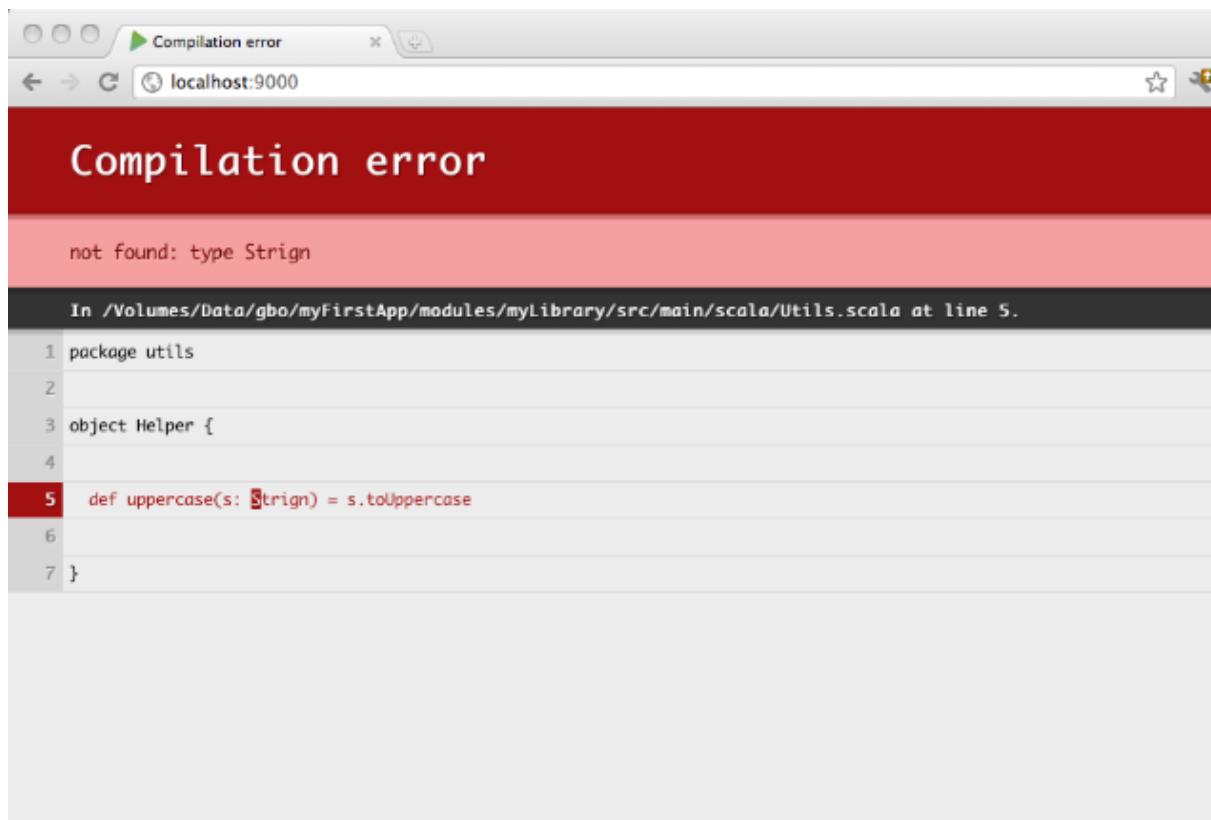
When you have a sub-project enabled in your build, you can focus on this project and compile, test or run it individually. Just use the `projects` command in the Play console prompt to display all projects:

```
[my-first-application] $ projects
[info] In file:/Volumes/Data/gbo/myFirstApp/
[info]      * my-first-application
[info]          my-library
```

To change the current project use the `project` command:

```
[my-first-application] $ project my-library
[info] Set current project to my-library
>
```

When you run your Play application in dev mode, the dependent projects are automatically recompiled, and if something cannot compile you will see the result in your browser:



Splitting your web application into several parts

As a Play application is just a standard sbt project with a default configuration, it can depend on another Play application.

The configuration is very close to the previous one. Simply configure your sub-project as a `PlayProject`:

```
import sbt._  
import Keys._  
import PlayProject._  
  
object ApplicationBuild extends Build {  
  
    val appName = "zenexity.com"  
    val appVersion = "1.2"  
  
    val common = PlayProject(  
        appName + "-common", appVersion, path = file("modules/common"))  
    )  
  
    val website = PlayProject(  
        appName + "-website", appVersion, path = file("modules/website"))  
.dependsOn(common)  
  
    val adminArea = PlayProject(  
        appName + "-admin", appVersion, path = file("modules/admin"))  
.dependsOn(common)  
  
    val main = PlayProject(  
        appName, appVersion  
    ).dependsOn(  
        website, adminArea  
    )  
}
```

Here we define a complete project split in two main parts: the website and the admin area. Moreover these two parts depend themselves on a common module.

Working with public assets

This section covers serving your application's static resources such as JavaScript, CSS and images.

Serving a public resource in Play 2.0 is the same as serving any other HTTP request. It uses the same routing as regular resources: using the controller/action path to distribute CSS, JavaScript or image files to the client.

The public/ folder

By convention, public assets are stored in the `public` folder of your application. This folder is organized as follows:

```
public
  └─ javascripts
  └─ stylesheets
  └─ images
```

If you follow this structure it will be simpler to get started, but nothing stops you to modifying it once you understand how it works.

How are public assets packaged ?

During the build process, the contents of the `public` folder are processed and added to the application classpath. When you package your application, these files are packaged into the application JAR file (under the `public/` path).

The Assets controller

Play 2.0 comes with a built-in controller to serve public assets. By default, this controller provides caching, ETag, gzip compression and JavaScript minification support.

The controller is available in the default Play JAR as `controllers.Assets`, and defines a single `at` action with two parameters :

```
Assets.at(folder: String, file: String)
```

The `folder` parameter must be fixed and defines the directory managed by the action. The `file` parameter is usually dynamically extracted from the request path.

Here is the typical mapping of the `Assets` controller in your `conf/routes` file:

```
GET /assets/*file          Assets.at("public", file)
```

Note that we define the `*file` dynamic part that will match the `.*` regular expression. So for example, if you send this request to the server:

```
GET /assets/javascripts/jquery.js
```

The router will invoke the `Assets.at` action with the following parameters:

```
controllers.Assets.at("public", "javascripts/jquery.js")
```

This action will look-up the file and serve it, if it exists.

Reverse routing for public assets

As for any controller mapped in the routes file, a reverse controller is created in `controllers.routes.Assets`. You use this to reverse the URL needed to fetch a public resource. For example, from a template:

```
<script src="@routes.Assets.at("javascripts/jquery.js")"></script>
```

This will produce the following result:

```
<script src="/assets/javascripts/jquery.js"></script>
```

Note that we don't specify the first `folder` parameter when we reverse the route. This is because our routes file defines a single mapping for the `Assets.at` action, where the `folder` parameter is fixed. So it doesn't need to be specified explicitly.

However, if you define two mappings for the `Assets.at` action, like this:

```
GET /javascripts/*file      Assets.at("public/javascripts", file)
GET /images/*file           Assets.at("public/images", file)
```

Then you will need to specify both parameters when using the reverse router:

```
<script src="@routes.Assets.at("public/javascripts", "jquery.js")"></script>
<image src="@routes.Assets.at("public/images", "logo.png")">
```

Etag support

The `Assets` controller automatically manages **ETag** HTTP Headers. The ETag value is generated from the resource name and 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**.

Gzip support

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
```

Additional `Cache-Control` directive

Usually, using Etag is enough to have proper caching. However if you want to specify a custom `Cache-Control` header for a particular resource, you can specify it in your `application.conf` file. For example:

```
# Assets configuration
# ~~~~~
"assets.cache ./public/stylesheets/bootstrap.min.css"="max-age=3600"
```

Using CoffeeScript

CoffeeScript is a small and elegant language that compiles into JavaScript. It provides a nicer syntax for writing JavaScript code.

Compiled assets in Play 2.0 must be defined in the `app/assets` directory. They are handled by the build process, and CoffeeScript sources are compiled into standard JavaScript files. The generated JavaScript files are distributed as standard resources into the same `public/` folder as other unmanaged assets, meaning that there is no difference in the way you use them once compiled.

Note that managed resources are not copied directly into your application's `public` folder, but maintained in a separate folder in `target/scala-2.9.1/resources_managed`.

For example a CoffeeScript source file `app/assets/javascripts/main.coffee` will be available as a standard JavaScript resource, at `public/javascripts/main.js`.

CoffeeScript sources are compiled automatically during a `compile` command, or when you refresh any page in your browser while you are running in development mode. Any compilation errors will be displayed in your browser:

The screenshot shows a web browser window with the title "Compilation error". The address bar indicates the URL is "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below this, the error message "unclosed (on line 9" is displayed in pink. A black bar below the error message shows the full path: "In /Volumes/Data/gbo/myFirstApp/app/assets/javascripts/main.coffee at line 9.". The code editor shows lines 5 through 13 of a CoffeeScript file. Line 9 is highlighted in red and contains the error: "square = ((x) -> x * x".

```
unclosed ( on line 9
In /Volumes/Data/gbo/myFirstApp/app/assets/javascripts/main.coffee at line 9.
5 # Conditions:
6 number = -42 if opposite
7
8 # Functions:
9 square = ((x) -> x * x
10
11 # Arrays:
12 list = [1, 2, 3, 4, 5]
13
```

Layout

Here is an example layout for using CoffeeScript in your projects:

```
app
  assets
    javascripts
      main.coffee
```

Two JavaScript files will be compiled: `public/javascripts/main.js` and `public/javascripts/main.min.js`. The first one is a readable file useful in development, and the second one a minified file that you can use in production. You can use either one in your template:

```
<script src="@routes.Assets.at("javascripts/main.js")">
```

```
<script src="@routes.Assets.at("javascripts/main.min.js")">
```

Options

CoffeeScript compilation can be configured in your project's `Build.scala` file. The only option currently supported is `bare` mode.

```
coffeeScriptOptions := Seq("bare")
```

By default, the JavaScript code is generated inside a top-level function safety wrapper, preventing it from polluting the global scope. The `bare` option removes this function wrapper.

Using LESS CSS

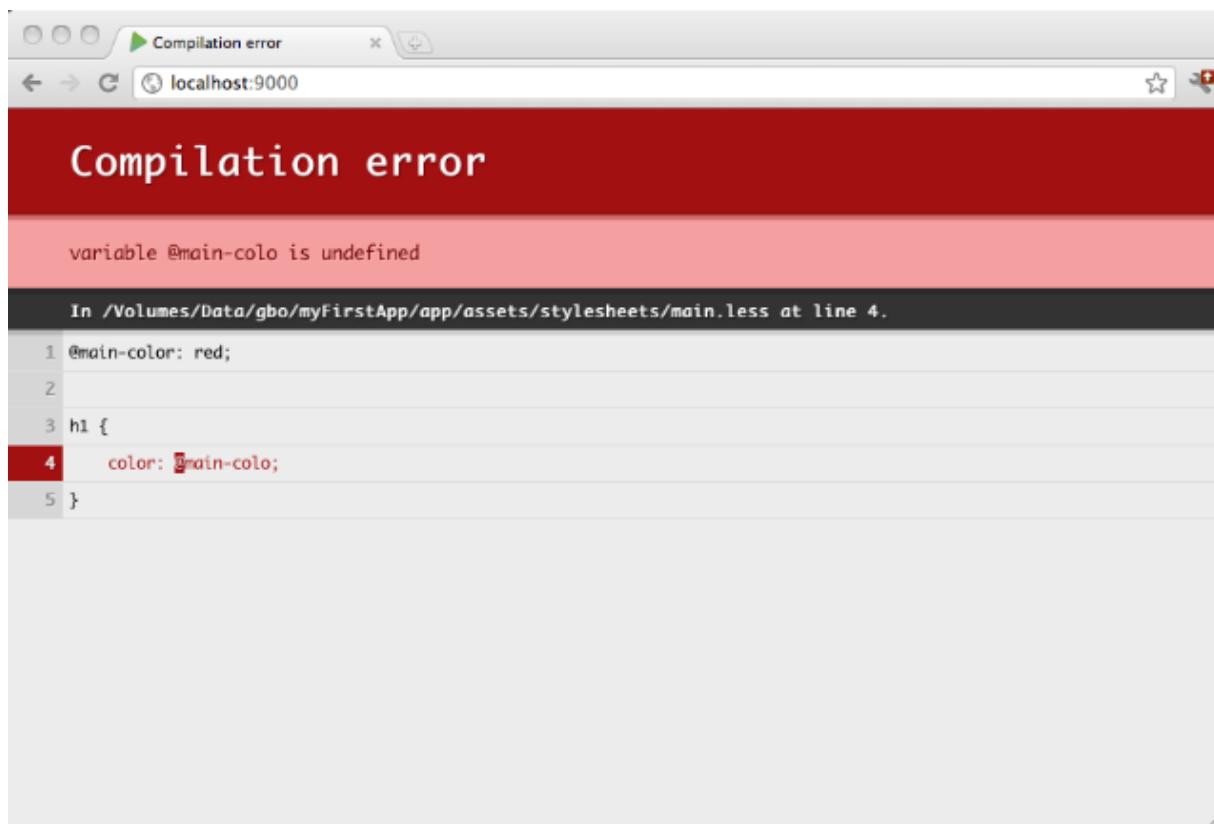
LESS CSS is a dynamic stylesheet language. It allows greater flexibility in the way you write CSS files: including support for variables, mixins and more.

Compilable assets in Play 2.0 must be defined in the `app/assets` directory. They are handled by the build process, and LESS sources are compiled into standard CSS files. The generated CSS files are distributed as standard resources into the same `public/` folder as the unmanaged assets, meaning that there is no difference in the way you use them once compiled.

Note that managed resources are not copied directly into your application `public` folder, but maintained in a separate folder in `target/scala-2.9.1/resources_managed`.

For example a LESS source file at `app/assets/stylesheets/main.less` will be available as a standard resource at `public/stylesheets/main.css`.

LESS sources are compiled automatically during a `compile` command, or when you refresh any page in your browser while you are running in development mode. Any compilation errors will be displayed in your browser:



The screenshot shows a web browser window with the title "Compilation error". The address bar says "localhost:9000". The main content area has a red header bar with the text "Compilation error". Below it, the error message "variable @main-colo is undefined" is displayed. Underneath that, it says "In /Volumes/Data/gbo/myFirstApp/app/assets/stylesheets/main.less at line 4.". The code editor shows the following LESS code:

```
1 @main-color: red;
2
3 h1 {
4   color: @main-col;
5 }
```

Working with partial LESS source files

You can split your LESS source into several libraries, and use the LESS `import` feature.

To prevent library files from being compiled individually (or imported) we need them to be skipped by the compiler. To do this, partial source files must be prefixed with the underscore (`_`) character, for example: `_myLibrary.less`. To configure this behavior, see the *Configuration* section at the end of this page.

Layout

Here is an example layout for using LESS in your project:

```
app
  assets
    stylesheets
      main.less
      utils
        _reset.less
        _layout.less
```

With the following `main.less` source:

```
@import "utils/_reset.less";
@import "utils/_layout.less";

h1 {
  color: red;
}
```

The resulting CSS file will be compiled as `public/stylesheets/main.css`, and you can use this in your template as any regular public asset. A minified version will also be generated.

```
<link rel="stylesheet" href="@routes.Assets.at("stylesheets/main.css")">
```

```
<link rel="stylesheet" href="@routes.Assets.at("stylesheets/main.min.css")">
```

Configuration

The default behavior of compiling every file that is not prepended by an underscore may not fit every project; for example if you include a library that has not been designed that way.

This can be configured in `project/Build.scala` by overriding the `lessEntryPoints` key. This key holds a `PathFinder`.

For example, to compile `app/assets/stylesheets/main.less` and nothing else:

```
val main = PlayProject(appName, appVersion, mainLang = SCALA).settings(  
  lessEntryPoints <= baseDirectory(_ / "app" / "assets" / "stylesheets"  
  ** "main.less")  
)
```

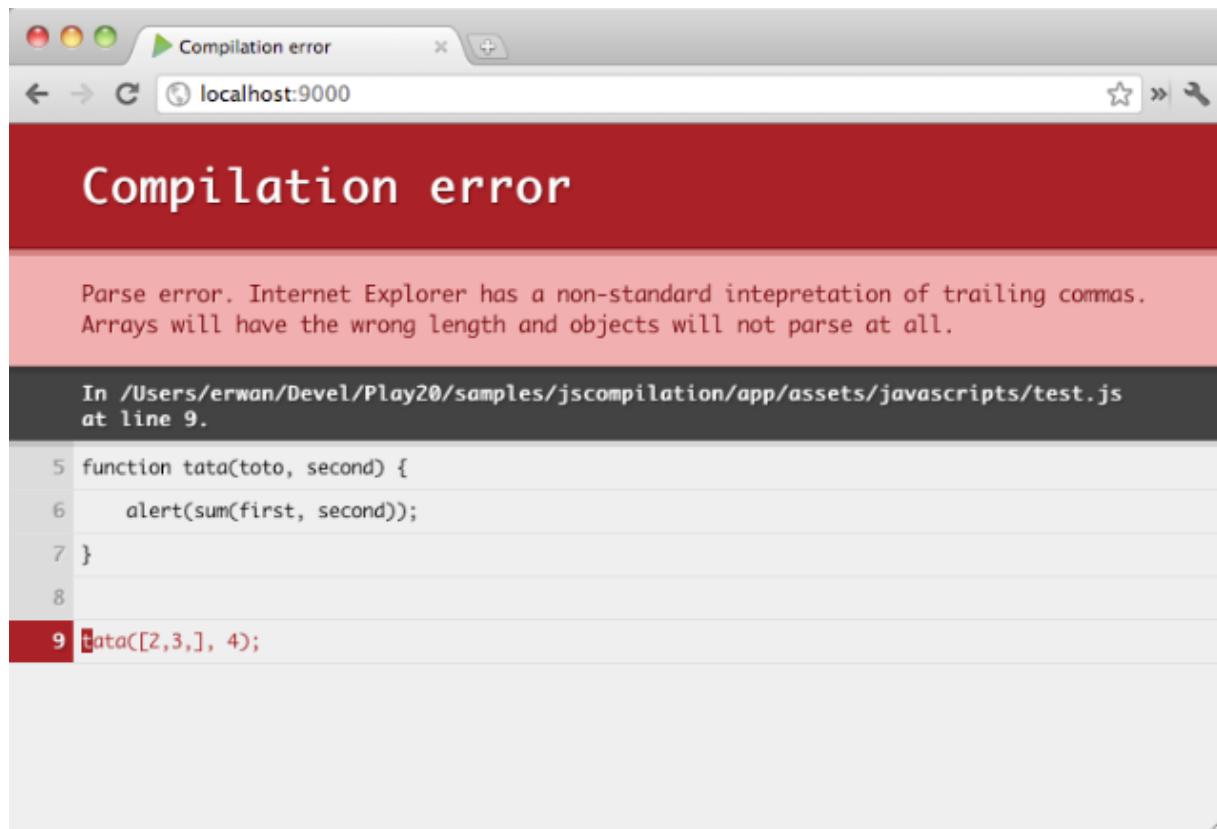
Using Google Closure Compiler

The Closure Compiler is a tool for making JavaScript download and run faster. It is a true compiler for JavaScript - though instead of compiling from a source language to machine code, it compiles JavaScript to better JavaScript. It parses your JavaScript, analyzes it, removes dead code and rewrites and minimizes what's left.

Any JavaScript file present in `app/assets` will be parsed by Google Closure compiler, checked for errors and dependencies and minified if activated in the build configuration.

Check JavaScript sanity

JavaScript code is compiled during the `compile` command, as well as automatically when modified. Errors are shown in the browser just like any other compilation error.



CommonJS-style dependencies

Play's Closure Compiler integration can also resolve dependencies that you declare with a CommonJS style, similarly to RequireJS.

Consider the `lib.js` file:

```
// The lib

function sum(a, b) {
    return a + b;
}
exports.sum = sum;
```

And the `test.js` file:

```
// The test

require("lib");

function showSum(first, second) {
    alert(require("lib").sum(first, second));
}

showSum([2,3], 4);
```

After compilation, the file served for `test.js` will be the following:

```
require.register("lib", function(module, exports, require){
// The lib

function sum(a, b) {
    return a + b;
}
exports.sum = sum;

}

require("lib")

// The test

function showSum(first, second) {
    alert(require("lib").sum(first, second));
}

showSum([2,3], 4);
```

Minification

A minified file is also generated, where `.js` is replaced by `.min.js`. In our example, it would be `test.min.js`. If you want to use the minified file instead of the regular file, you need to change the script source attribute in your HTML.

Entry Points

By default, any JavaScript file not prepended by an underscore will be compiled. This behavior can be changed in `project/Build.scala` by overriding the `javascriptEntryPoints` key. This key holds a `PathFinder`.

For example, to compile only `.js` file from the `app/assets/javascripts/main` directory:

```
val main = PlayProject(appName, appVersion, mainLang = SCALA).settings(  
    javascriptEntryPoints <=> baseDirectory(base =>  
        base / "app" / "assets" / "javascripts" / "main" ** "*.*js"  
    )  
)
```

The default definition is:

```
javascriptEntryPoints <=> (sourceDirectory in Compile)(base =>  
    ((base / "assets" ** "*.*js") --- (base / "assets" ** "_*")).get  
)
```

Managing database evolutions

When you use a relational database, you need a way to track and organize your database schema evolutions. Typically there are several situation where you need a more sophisticated way to track your database schema changes:

- When you work within a team of developers, each person needs to know about any schema change.
- When you deploy on a production server, you need to have a robust way to upgrade your database schema.
- If you work on several machines, you need to keep all database schemas synchronized.

Evolutions scripts

Play tracks your database evolutions using several evolutions script. These scripts are written in plain old SQL and should be located in the `[conf/evolutions/{database name}]` directory of your application. If the evolutions apply to your default database, this path is `[conf/evolutions/default]`.

The first script is named `[1.sql]`, the second script `[2.sql]`, and so on...

Each script contains two parts:

- The **Ups** part the describe the required transformations .
- The **Downs** part that describe how to revert them.

For example, take a look at this first evolution script that bootstrap a basic application:

```
# Users schema

# --- !Ups

CREATE TABLE User (
    id bigint(20) NOT NULL AUTO_INCREMENT ,
    email varchar(255) NOT NULL ,
    password varchar(255) NOT NULL ,
    fullname varchar(255) NOT NULL ,
    isAdmin boolean NOT NULL ,
    PRIMARY KEY (id)
);

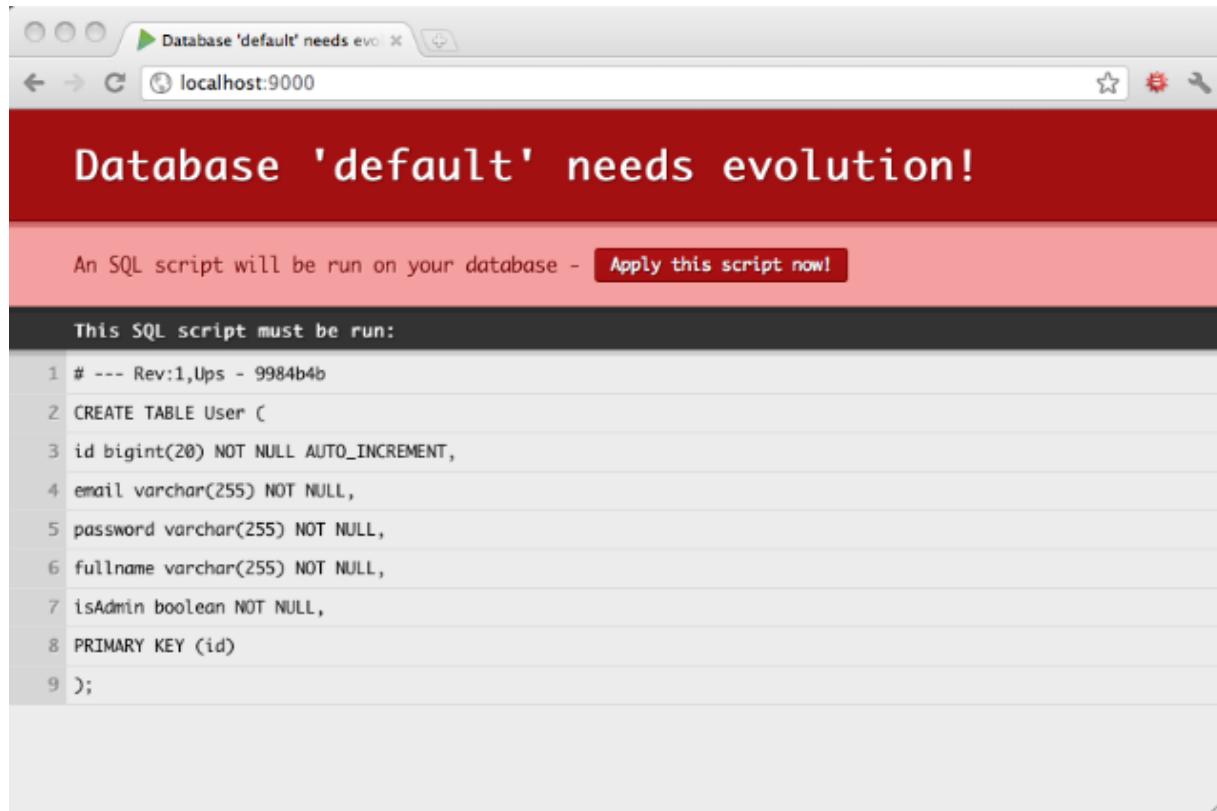
# --- !Downs

DROP TABLE User;
```

As you see you have to delimitate the both Ups and Downs section by using comments in your SQL script.

Evolutions are automatically activated if a database is configured in `application.conf` and evolution scripts are present. You can disable them by setting `evolutionplugin=disabled`. For example when tests set up their own database you can disable evolutions for the test environment.

When evolutions are activated, Play will check your database schema state before each request in DEV mode, or before starting the application in PROD mode. In DEV mode, if your database schema is not up to date, an error page will suggest that you synchronise your database schema by running the appropriate SQL script.



If you agree with the SQL script, you can apply it directly by clicking on the 'Apply evolutions' button.

Synchronizing concurrent changes

Now let's imagine that we have two developers working on this project. Developer A will work on a feature that requires a new database table. So he will create the following `2.sql` evolution script:

```
# Add Post

# --- !Ups
CREATE TABLE Post (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    title varchar(255) NOT NULL,
    content text NOT NULL,
    postedAt date NOT NULL,
    author_id bigint(20) NOT NULL,
    FOREIGN KEY (author_id) REFERENCES User(id),
    PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Post;
```

Play will apply this evolution script to Developer A's database.

On the other hand, developer B will work on a feature that requires altering the User table. So he will also create the following `2.sql` evolution script:

```
# Update User

# --- !Ups
ALTER TABLE User ADD age INT;

# --- !Downs
ALTER TABLE User DROP age;
```

Developer B finishes his feature and commits (let's say they are using Git). Now developer A has to merge the his colleague's work before continuing, so he runs `git pull`, and the merge has a conflict, like:

```
Auto-merging db/evolutions/2.sql
CONFLICT (add/add): Merge conflict in db/evolutions/2.sql
Automatic merge failed; fix conflicts and then commit the result.
```

Each developer has created a `2.sql` evolution script. So developer A needs to merge the contents of this file:

```
<<<<< HEAD
# Add Post

# --- !Ups
CREATE TABLE Post (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    title varchar(255) NOT NULL,
    content text NOT NULL,
    postedAt date NOT NULL,
    author_id bigint(20) NOT NULL,
    FOREIGN KEY (author_id) REFERENCES User(id),
    PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Post;
=====

# Update User

# --- !Ups
ALTER TABLE User ADD age INT;

# --- !Downs
ALTER TABLE User DROP age;
>>>>> devB
```

The merge is really easy to do:

```
# Add Post and update User

# --- !Ups
ALTER TABLE User ADD age INT;

CREATE TABLE Post (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    title varchar(255) NOT NULL,
    content text NOT NULL,
    postedAt date NOT NULL,
    author_id bigint(20) NOT NULL,
    FOREIGN KEY (author_id) REFERENCES User(id),
    PRIMARY KEY (id)
);

# --- !Downs
ALTER TABLE User DROP age;

DROP TABLE Post;
```

This evolution script represents the new revision 2 of the database, that is different of the previous revision 2 that developer A has already applied.

So Play will detect it and ask developer A to synchronize his database by first reverting the old revision 2 already applied, and by applying the new revision 2 script:

Inconsistent states

Sometimes you will make a mistake in your evolution scripts, and they will fail. In this case, Play will mark your database schema as being in an inconsistent state and will ask you to manually resolve the problem before continuing.

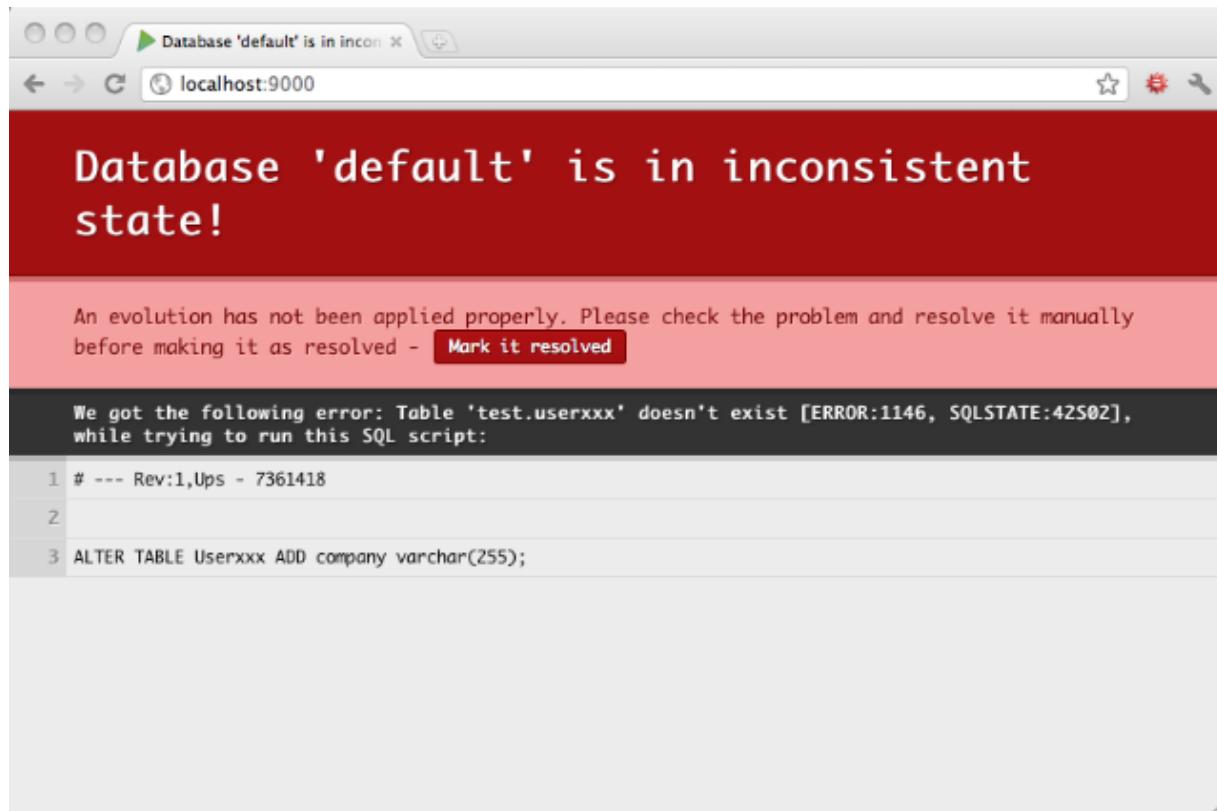
For example, the Ups script of this evolution has an error:

```
# Add another column to User

# --- !Ups
ALTER TABLE Userxxx ADD company varchar(255);

# --- !Downs
ALTER TABLE User DROP company;
```

So trying to apply this evolution will fail, and Play will mark your database schema as inconsistent:



Now before continuing you have to fix this inconsistency. So you run the fixed SQL command:

```
ALTER TABLE User ADD company varchar(255);
```

... and then mark this problem as manually resolved by clicking on the button.

But because your evolution script has errors, you probably want to fix it. So you modify the `3.sql` script:

```
# Add another column to User

# --- !Ups
ALTER TABLE User ADD company varchar(255);

# --- !Downs
ALTER TABLE User DROP company;
```

Play detects this new evolution that replaces the previous 3 one, and will run the appropriate script. Now everything is fixed, and you can continue to work.

In developement mode however it is often simpler to simply trash your developement database and reapply all evolutions from the beginning.

Configuration file syntax and features

The configuration file used by Play is based on the Typesafe config library .

The default configuration file of a Play 2.0 application must be defined in `conf/application.conf`. It uses the HOCON format ("Human-Optimized Config Object Notation").

Specifying an alternative configuration file

System properties can be used to force a different config source:

- `config.resource` specifies a resource name - not a basename, i.e. `application.conf` not `application`
- `config.file` specifies a filesystem path, again it should include the extension, not be a basename
- `config.url` specifies a URL

These system properties specify a replacement for `application.conf`, not an addition. In the replacement config file, you can use `include "application"` to include the original default config file; after the `include` statement you could go on to override certain settings.

Using with Akka

Akka 2.0 will use the same configuration file as the one defined for your Play 2.0 application. Meaning that you can configure anything in Akka in the `application.conf` directory.

HOCON Syntax

Much of this is defined with reference to JSON; you can find the JSON spec at <http://json.org/> of course.

Unchanged from JSON

- files must be valid UTF-8
- quoted strings are in the same format as JSON strings
- values have possible types: string, number, object, array, boolean, null
- allowed number formats matches JSON; as in JSON, some possible floating-point values are not represented, such as `NaN`

Comments

Anything between `//` or `#` and the next newline is considered a comment and ignored, unless the `//` or `#` is inside a quoted string.

Omit root braces

JSON documents must have an array or object at the root. Empty files are invalid documents, as are files containing only a non-array non-object value such as a string.

In HOCON, if the file does not begin with a square bracket or curly brace, it is parsed as if it were enclosed with `{}` curly braces.

A HOCON file is invalid if it omits the opening `{` but still has a closing `}`; the curly braces must be balanced.

Key-value separator

The `=` character can be used anywhere JSON allows `:`, i.e. to separate keys from values.

If a key is followed by `{`, the `:` or `=` may be omitted. So `"foo" {}` means `"foo" : {}"`

Commas

Values in arrays, and fields in objects, need not have a comma between them as long as they have at least one ASCII newline (`\n`, decimal value 10) between them.

The last element in an array or last field in an object may be followed by a single comma. This extra comma is ignored.

- `[1, 2, 3,]` and `[1, 2, 3]` are the same array.
- `[1\n2\n3]` and `[1, 2, 3]` are the same array.
- `[1, 2, 3, ,]` is invalid because it has two trailing commas.
- `[, 1, 2, 3]` is invalid because it has an initial comma.
- `[1, , 2, 3]` is invalid because it has two commas in a row.
- these same comma rules apply to fields in objects.

Duplicate keys

The JSON spec does not clarify how duplicate keys in the same object should be handled. In HOCON, duplicate keys that appear later override those that appear earlier, unless both values are objects. If both values are objects, then the objects are merged.

Note: this would make HOCON a non-superset of JSON if you assume that JSON requires duplicate keys to have a behavior. The assumption here is that duplicate keys are invalid JSON.

To merge objects:

- add fields present in only one of the two objects to the merged object.
- for non-object-valued fields present in both objects, the field found in the second object must be used.
- for object-valued fields present in both objects, the object values should be recursively merged according to these same rules.

Object merge can be prevented by setting the key to another value first. This is because merging is always done two values at a time; if you set a key to an object, a non-object, then an object, first the non-object falls back to the object (non-object always wins), and then the object falls back to the non-object (no merging, object is the new value). So the two objects never see each other.

These two are equivalent:

```
{  
    "foo" : { "a" : 42 },  
    "foo" : { "b" : 43 }  
}  
  
{  
    "foo" : { "a" : 42, "b" : 43 }  
}
```

And these two are equivalent:

```
{  
    "foo" : { "a" : 42 },  
    "foo" : null,  
    "foo" : { "b" : 43 }  
}  
  
{  
    "foo" : { "b" : 43 }  
}
```

The intermediate setting of `"foo"` to `null` prevents the object merge.

Paths as keys

If a key is a path expression with multiple elements, it is expanded to create an object for each path element other than the last. The last path element, combined with the value, becomes a field in the most-nested object.

In other words:

```
foo.bar : 42
```

is equivalent to:

```
foo { bar : 42 }
```

and:

```
foo.bar.baz : 42
```

is equivalent to:

```
foo { bar { baz : 42 } }
```

and so on. These values are merged in the usual way; which implies that:

```
a.x : 42, a.y : 43
```

is equivalent to:

```
a { x : 42, y : 43 }
```

Because path expressions work like value concatenations, you can have whitespace in keys:

```
a b c : 42
```

is equivalent to:

```
"a b c" : 42
```

Because path expressions are always converted to strings, even single values that would normally have another type become strings.

- `true : 42` is `"true" : 42`
- `3.14 : 42` is `"3.14" : 42`

As a special rule, the unquoted string `include` may not begin a path expression in a key, because it has a special interpretation (see below).

Substitutions

Substitutions are a way of referring to other parts of the configuration tree.

The syntax is `${pathexpression}` or `${?pathexpression}` where the `pathexpression` is a path expression as described above. This path expression has the same syntax that you could use for an object key.

The `?` in `${?pathexpression}` must not have whitespace before it; the three characters `${?` must be exactly like that, grouped together.

For substitutions which are not found in the configuration tree, implementations may try to resolve them by looking at system environment variables or other external sources of configuration. (More detail on environment variables in a later section.)

Substitutions are not parsed inside quoted strings. To get a string containing a substitution, you must use value concatenation with the substitution in the unquoted portion:

```
key : ${animal.favorite} is my favorite animal
```

Or you could quote the non-substitution portion:

```
key : "${animal.favorite}" is my favorite animal"
```

Substitutions are resolved by looking up the path in the configuration. The path begins with the root configuration object, i.e. it is “absolute” rather than “relative.”

Substitution processing is performed as the last parsing step, so a substitution can look forward in the configuration. If a configuration consists of multiple files, it may even end up retrieving a value from another file. If a key has been specified more than once, the substitution will always evaluate to its latest-assigned value (the merged object or the last non-object value that was set).

If a configuration sets a value to `null` then it should not be looked up in the external source. Unfortunately there is no way to “undo” this in a later configuration file; if you have `{ "HOME" : null }` in a root object, then `${HOME}` will never look at the environment variable. There is no equivalent to JavaScript’s `delete` operation in other words.

If a substitution does not match any value present in the configuration and is not resolved by an external source, then it is `undefined`. An `undefined` substitution with the `${foo}` syntax is invalid and should generate an error.

If a substitution with the `${?foo}` syntax is `undefined`:

- if it is the value of an object field then the field should not be created. If the field would have overridden a previously-set value for the same field, then the previous value remains.
- if it is an array element then the element should not be added.
- if it is part of a value concatenation then it should become an empty string.
- `foo : ${?bar}` would avoid creating field `foo` if `bar` is `undefined`, but `foo : ${?bar} ${?baz}` would be a value concatenation so if `bar` or `baz` are not defined, the result is an empty string.

Substitutions are only allowed in object field values and array elements (value concatenations), they are not allowed in keys or nested inside other substitutions (path expressions).

A substitution is replaced with any value type (number, object, string, array, true, false, null). If the substitution is the only part of a value, then the type is preserved. Otherwise, it is value-concatenated to form a string.

Circular substitutions are invalid and should generate an error.

Implementations must take care, however, to allow objects to refer to paths within themselves. For example, this must work:

```
bar : { foo : 42,
        baz : ${bar.foo}
    }
```

Here, if an implementation resolved all substitutions in `bar` as part of resolving the substitution `${bar.foo}`, there would be a cycle. The implementation must only resolve the `foo` field in `bar`, rather than recursing the entire `bar` object.

Includes

Include syntax

An *include statement* consists of the unquoted string `include` and a single quoted string immediately following it. An include statement can appear in place of an object field.

If the unquoted string `include` appears at the start of a path expression where an object key would be expected, then it is not interpreted as a path expression or a key.

Instead, the next value must be a *quoted* string. The quoted string is interpreted as a filename or resource name to be included.

Together, the unquoted `include` and the quoted string substitute for an object field syntactically, and are separated from the following object fields or includes by the usual comma (and as usual the comma may be omitted if there's a newline).

If an unquoted `include` at the start of a key is followed by anything other than a single quoted string, it is invalid and an error should be generated.

There can be any amount of whitespace, including newlines, between the unquoted `include` and the quoted string.

Value concatenation is NOT performed on the “argument” to `include`. The argument must be a single quoted string. No substitutions are allowed, and the argument may not be an unquoted string or any other kind of value.

Unquoted `include` has no special meaning if it is not the start of a key’s path expression.

It may appear later in the key:

```
# this is valid
{ foo include : 42 }
# equivalent to
{ "foo include" : 42 }
```

It may appear as an object or array value:

```
{ foo : include } # value is the string "include"
[ include ]         # array of one string "include"
```

You can quote `"include"` if you want a key that starts with the word `"include"`, only unquoted `include` is special:

```
{ "include" : 42 }
```

Include semantics: merging

An *including file* contains the `include` statement and an *included file* is the one specified in the `include` statement. (They need not be regular files on a filesystem, but assume they are for the moment.)

An included file must contain an object, not an array. This is significant because both JSON and HOCON allow arrays as root values in a document.

If an included file contains an array as the root value, it is invalid and an error should be generated.

The included file should be parsed, producing a root object. The keys from the root object are conceptually substituted for the `include` statement in the including file.

- If a key in the included object occurred prior to the `include` statement in the including object, the included key's value overrides or merges with the earlier value, exactly as with duplicate keys found in a single file.
- If the including file repeats a key from an earlier-included object, the including file's value would override or merge with the one from the included file.

Include semantics: substitution

Substitutions in included files are looked up at two different paths; first, relative to the root of the included file; second, relative to the root of the including configuration.

Recall that substitution happens as a final step, *after* parsing. It should be done for the entire app's configuration, not for single files in isolation.

Therefore, if an included file contains substitutions, they must be “fixed up” to be relative to the app's configuration root.

Say for example that the root configuration is this:

```
{ a : { include "foo.conf" } }
```

And “foo.conf” might look like this:

```
{ x : 10, y : ${x} }
```

If you parsed “foo.conf” in isolation, then `${x}` would evaluate to 10, the value at the path `x`. If you include “foo.conf” in an object at key `a`, however, then it must be fixed up to be `${a.x}` rather than `${x}`.

Say that the root configuration redefines `a.x`, like this:

```
{
  a : { include "foo.conf" }
  a : { x : 42 }
}
```

Then the `${x}` in “foo.conf”, which has been fixed up to `${a.x}`, would evaluate to 42 rather than to 10. Substitution happens *after* parsing the whole configuration.

However, there are plenty of cases where the included file might intend to refer to the application’s root config. For example, to get a value from a system property or from the reference configuration. So it’s not enough to only look up the “fixed up” path, it’s necessary to look up the original path as well.

Include semantics: missing files

If an included file does not exist, the include statement should be silently ignored (as if the included file contained only an empty object).

Include semantics: locating resources

Conceptually speaking, the quoted string in an include statement identifies a file or other resource “adjacent to” the one being parsed and of the same type as the one being parsed. The meaning of “adjacent to”, and the string itself, has to be specified separately for each kind of resource.

Implementations may vary in the kinds of resources they support including.

On the Java Virtual Machine, if an include statement does not identify anything “adjacent to” the including resource, implementations may wish to fall back to a classpath resource. This allows configurations found in files or URLs to access classpath resources.

For resources located on the Java classpath:

- included resources are looked up by calling `getResource()` on the same class loader used to look up the including resource.
- if the included resource name is absolute (starts with '/') then it should be passed to `getResource()` with the '/' removed.
- if the included resource name does not start with '/' then it should have the "directory" of the including resource. prepended to it, before passing it to `getResource()`. If the including resource is not absolute (no '/') and has no "parent directory" (is just a single path element), then the included relative resource name should be left as-is.
- it would be wrong to use `getResource()` to get a URL and then locate the included name relative to that URL, because a class loader is not required to have a one-to-one mapping between paths in its URLs and the paths it handles in `getResource()`. In other words, the "adjacent to" computation should be done on the resource name not on the resource's URL.

For plain files on the filesystem:

- if the included file is an absolute path then it should be kept absolute and loaded as such.
- if the included file is a relative path, then it should be located relative to the directory containing the including file. The current working directory of the process parsing a file must NOT be used when interpreting included paths.
- if the file is not found, fall back to the classpath resource. The classpath resource should not have any package name added in front, it should be relative to the "root"; which means any leading "/" should just be removed (absolute is the same as relative since it's root-relative). The "/" is handled for consistency with including resources from inside other classpath resources, where the resource name may not be root-relative and "/" allows specifying relative to root.

URLs:

- for both filesystem files and Java resources, if the included name is a URL (begins with a protocol), it would be reasonable behavior to try to load the URL rather than treating the name as a filename or resource name.
 - for files loaded from a URL, “adjacent to” should be based on parsing the URL’s path component, replacing the last path element with the included name.
 - file: URLs should behave in exactly the same way as a plain filename
-

Duration format

The supported unit strings for duration are case sensitive and must be lowercase. Exactly these strings are supported:

- `ns`, `nanosecond`, `nanoseconds`
 - `us`, `microsecond`, `microseconds`
 - `ms`, `millisecond`, `milliseconds`
 - `s`, `second`, `seconds`
 - `m`, `minute`, `minutes`
 - `h`, `hour`, `hours`
 - `d`, `day`, `days`
-

Size in bytes format

For single bytes, exactly these strings are supported:

- `B`, `b`, `byte`, `bytes`

For powers of ten, exactly these strings are supported:

- `kB`, `kilobyte`, `kilobytes`
- `MB`, `megabyte`, `megabytes`
- `GB`, `gigabyte`, `gigabytes`
- `TB`, `terabyte`, `terabytes`
- `PB`, `petabyte`, `petabytes`
- `EB`, `exabyte`, `exabytes`
- `ZB`, `zettabyte`, `zettabytes`
- `YB`, `yottabyte`, `yottabytes`

For powers of two, exactly these strings are supported:

- `K`, `k`, `Ki`, `KiB`, `kibibyte`, `kibibytes`
- `M`, `m`, `Mi`, `MiB`, `mebibyte`, `mebibytes`
- `G`, `g`, `Gi`, `GiB`, `gibibyte`, `gibibytes`
- `T`, `t`, `Ti`, `TiB`, `tebibyte`, `tebibytes`
- `P`, `p`, `Pi`, `PiB`, `pebibyte`, `pebibytes`
- `E`, `e`, `Ei`, `EiB`, `exbibyte`, `exbibytes`
- `Z`, `z`, `Zi`, `ZiB`, `zebibyte`, `zebibytes`
- `Y`, `y`, `Yi`, `YiB`, `yobibyte`, `yobibytes`

Conventional override by system properties

For an application's config, Java system properties *override* settings found in the configuration file. This supports specifying config options on the command line.

Configuring the JDBC pool.

The Play 2.0 JDBC datasource is managed by BoneCP .

Special URLs

Play supports special url format for both **MySQL** and **PostgreSQL**:

```
# To configure MySQL  
db.default.url=mysql://localhost:root@secret/myDatabase  
  
# To configure PostgreSQL  
db.default.url=postgres://localhost:root@secret/myDatabase
```

Reference

In addition to the classical `driver`, `url`, `user`, `password` configuration properties, it also supports additional tuning parameters if you need them:

```
# The JDBC driver to use  
db.default.driver=org.h2.Driver  
  
# The JDBC url  
db.default.url="jdbc:h2:mem:play"  
  
# User name  
db.default.user=sa  
  
# Password  
db.default.password=secret  
  
# Set a connection's default autocommit setting  
db.default.autocommit=true  
  
# Set a connection's default isolation level  
db.default.isolation=READ_COMMITTED  
  
# In order to reduce lock contention and thus improve performance,  
# each incoming connection request picks off a connection from a  
# pool that has thread-affinity.  
# The higher this number, the better your performance will be for the  
# case when you have plenty of short-lived threads.  
# Beyond a certain threshold, maintenance of these pools will start  
# to have a negative effect on performance (and only for the case  
# when connections on a partition start running out).  
db.default.partitionCount=2
```

Configuring the internal Akka system

```
# The number of initial connections, per partition.
```

Play 2.0 uses an internal Akka Actor system to handle request processing. You can configure it in your application `application.conf` configuration file.

```
# When the available connections are about to run out, BoneCP will
```

```
# dynamically create new ones in batches. This property controls
```

```
# how many new connections to create in one go (up to a maximum of
```

Action invoker actors

```
db.default.acquireIncrement=1
```

The action invoker Actors are used to execute the `Action` code. To be able to execute several Action concurrently we are using several of these Actors managed by a Round Robin router. These actors are stateless.

```
db.default.acquireRetryAttempts=10
```

These action invoker Actors are also used to retrieve the **body parser** needed to parse the request body. Because this part waits for a reply (the `BodyParser` object to use), it will fail after a configurable timeout.

```
# How long to wait before attempting to obtain a connection again after a failure.
```

```
db.default.acquireRetryDelay=5 seconds
```

Action invoker actors are run by the `actions-dispatcher` dispatcher.

```
# The maximum time to wait before a call
```

```
# to getConnection is timed out.
```

```
db.default.connectionTimeout=1 second
```

Promise invoker actors

```
# Idle max age
```

The promise invoker Actors are used to execute all asynchronous callback needed by `Promise`.

```
db.default.idleMaxAge=10 minute
```

Several Actors must be available to execute several Promise callbacks concurrently. These actors

are stateless.

```
# This sets the time for a connection to remain idle before sending a test
```

```
query to the DB.
```

Promise invoker actors are run by the `promises-dispatcher` dispatcher.

```
# This is useful to prevent a DB from timing out connections on its end.
```

```
db.default.idleConnectionTestPeriod=5 minutes
```

WebSockets agent actors

```
# An initial SQL statement that is run only when
```

```
# a connection is first created.
```

Each WebSocket connection state is managed by an Agent actor. A new actor is created for each WebSocket, and is killed when the socket is closed. These actors are statefull.

```
# If enabled, log SQL statements being executed.
```

WebSockets agent actors are run by the `websockets-dispatcher` dispatcher.

```
# The maximum connection age.
```

```
db.default.maxConnectionAge=1 hour
```

Default configuration

Here is the reference configuration used by Play 2.0 if you don't override it. Adapt it according your application needs.

```
play {
    akka {
        event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]
        loglevel = WARNING

        actor {
```

```
        nr-of-instances = 24
    }

    /promises {
        router = round-robin
        nr-of-instances = 24
    }

}

retrieveBodyParserTimeout = 1 second

actions-dispatcher = {
    fork-join-executor {
        parallelism-factor = 1.0
        parallelism-max = 24
    }
}

promises-dispatcher = {
    fork-join-executor {
        parallelism-factor = 1.0
        parallelism-max = 24
    }
}

websockets-dispatcher = {
    fork-join-executor {
        parallelism-factor = 1.0
        parallelism-max = 24
    }
}

default-dispatcher = {
    fork-join-executor {
        parallelism-factor = 1.0
        parallelism-max = 24
    }
}

}
```

Configuring logging

Play 2.0 uses logback as its logging engine.

Configuration logging level in application.conf

The easiest way to configure the logging level is to use the `logger` key in your `conf/application.conf` file.

Play defines a default `application` logger for your application, which is automatically used when you use the default `Logger` operations.

```
# Root logger:  
logger=ERROR  
  
# Logger used by the framework:  
logger.play=INFO  
  
# Logger provided to your application:  
logger.application=DEBUG
```

The root logger configuration affects all log calls, rather than requiring custom logging levels. Additionally, if you want to enable the logging level for a specific library, you can specify it here. For example to enable `TRACE` log level for Spring, you could add:

```
logger.org.springframework=TRACE
```

Configuring logback

The default is to define two appenders, one dispatched to the standard out stream, and the other to the `logs/application.log` file.

If you want to fully customize logback, just define a `conf/logger.xml` configuration file. Here is the default configuration file used by Play:

```
<configuration>

    <conversionRule conversionWord="coloredLevel"
converterClass="play.api.Logger$ColoredLevel" />

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${application.home}/logs/application.log</file>
        <encoder>
            <pattern>%date - [%level] - from %logger in %thread %n%message%n%xException%n</pattern>
        </encoder>
    </appender>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%coloredLevel %logger{15} - %message%n%xException{5}</pattern>
        </encoder>
    </appender>

    <logger name="play" level="INFO" />
    <logger name="application" level="INFO" />

    <root level="ERROR">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>

</configuration>
```

Changing the logback configuration file

You can also specify another logback configuration file via a System property. It is particularly useful when running in production.

Using `-Dlogger.resource`

Specify another logback configuration file to be loaded from the classpath:

```
$ start -Dlogger.resource=conf/prod-logger.xml
```

Using `-Dlogger.file`

Specify another logback configuration file to be loaded from the file system:

```
$ start -Dlogger.file=/opt/prod/logger.xml
```

Using `-Dlogger.url`

Specify another loback configuration file to be loaded from an URL:

```
$ start -Dlogger.url=http://conf.mycompany.com/logger.xml
```

Starting your application in production mode

There are several ways to deploy a Play application in production mode. Let's start by using the simplest way, using a local Play installation.

Using the start command

The easiest way to start an application in production mode is to use the `start` command from the Play console. This requires a Play 2.0 installation on the server.

```
[My first application] $ start
```

Note that the `run` command is only for development mode and should never be used to run an application in production. For each request a complete check is handled by sbt.

```
$ play
[info] Loading project definition from /Volumes/Data/gbo/myFirstApp/project
[info] Set current project to My first application (in build file:/Volumes/Data/gbo/myFirstApp/)

[My first application] $ start

(Starting server. Type Ctrl+D to exit logs, the server will remain in background)

Process ID is 69916
[info] play - Application is started
[info] play - Listening for HTTP on port 9000...

```

When you run the `start` command, Play forks a new JVM and runs the default Netty HTTP server. The standard output stream is redirected to the Play console, so you can monitor its status.

The server's process id is displayed at bootstrap and written to the `RUNNING_PID` file. To kill a running Play server, it is enough to send a `SIGTERM` to the process to properly shutdown the application.

If you type `Ctrl+D`, the Play console will quit, but the created server process will continue running in background. The forked JVM's standard output stream is then closed, and logging can be read from the `logs/application.log` file.

If you type `Ctrl+C`, you will kill both JVMs: the Play console and the forked Play server.

Alternatively you can directly use `play start` at your OS command prompt, which does the same thing:

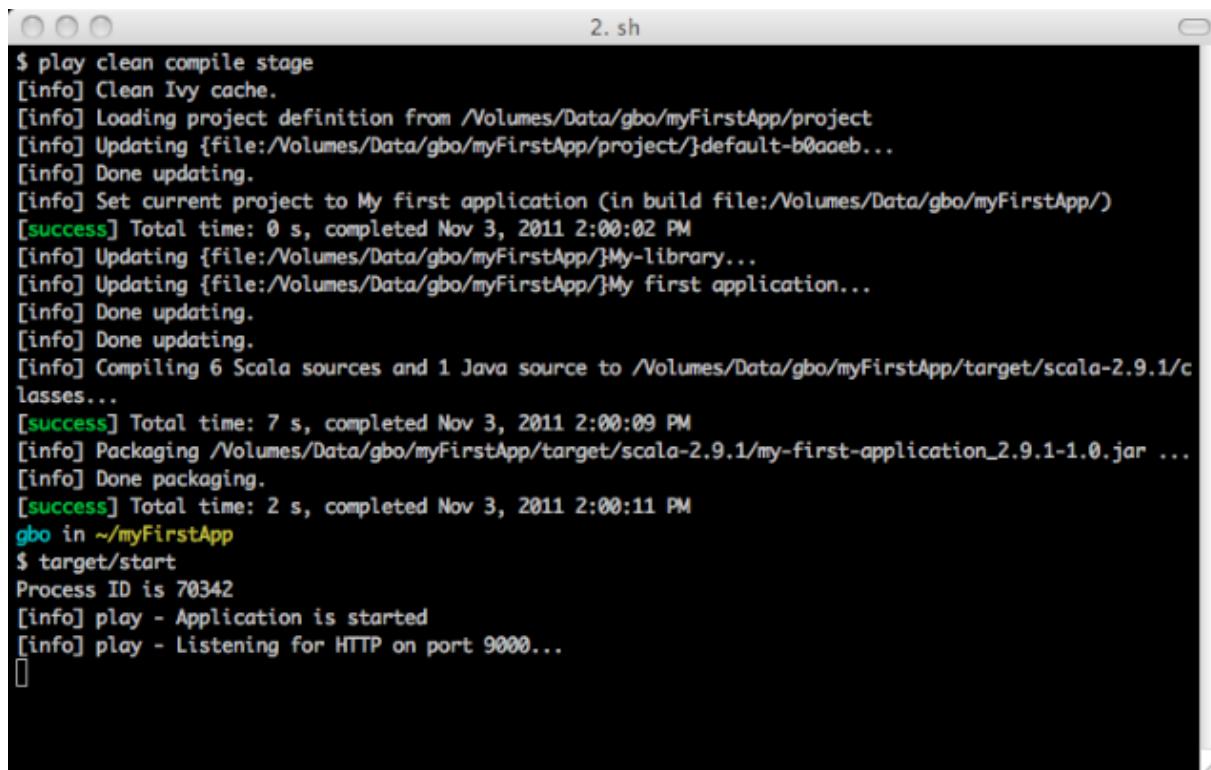
```
$ play start
```

Using the stage task

The problem with the `start` command is that it starts the application interactively, which means that human interaction is needed, and `Ctrl+D` is required to detach the process. This solution is not really convenient for automated deployment.

You can use the `stage` task to prepare your application to be run in place. The typical command for preparing a project to be run in place is:

```
$ play clean compile stage
```



```
$ play clean compile stage
[info] Clean Ivy cache.
[info] Loading project definition from /Volumes/Data/gbo/myFirstApp/project
[info] Updating {file:/Volumes/Data/gbo/myFirstApp/project/}default-b0aaeb...
[info] Done updating.
[info] Set current project to My first application (in build file:/Volumes/Data/gbo/myFirstApp/)
[success] Total time: 0 s, completed Nov 3, 2011 2:00:02 PM
[info] Updating {file:/Volumes/Data/gbo/myFirstApp/}My-library...
[info] Updating {file:/Volumes/Data/gbo/myFirstApp/}My first application...
[info] Done updating.
[info] Done updating.
[info] Compiling 6 Scala sources and 1 Java source to /Volumes/Data/gbo/myFirstApp/target/scala-2.9.1/c
lasses...
[success] Total time: 7 s, completed Nov 3, 2011 2:00:09 PM
[info] Packaging /Volumes/Data/gbo/myFirstApp/target/scala-2.9.1/my-first-application_2.9.1-1.0.jar ...
[info] Done packaging.
[success] Total time: 2 s, completed Nov 3, 2011 2:00:11 PM
gbo in ~/myFirstApp
$ target/start
Process ID is 70342
[info] play - Application is started
[info] play - Listening for HTTP on port 9000...
```

This cleans and compiles your application, retrieves the required dependencies and copies them to the `target/staged` directory. It also creates a `target/start` script that runs the Play server.

You can start your application using:

```
$ target/start
```

The generated `start` script is very simple - in fact, you could even execute the `java` command directly.

If you don't have Play installed on the server, you can use sbt to do the same thing:

```
$ sbt clean compile stage
```

Creating a standalone version of your application

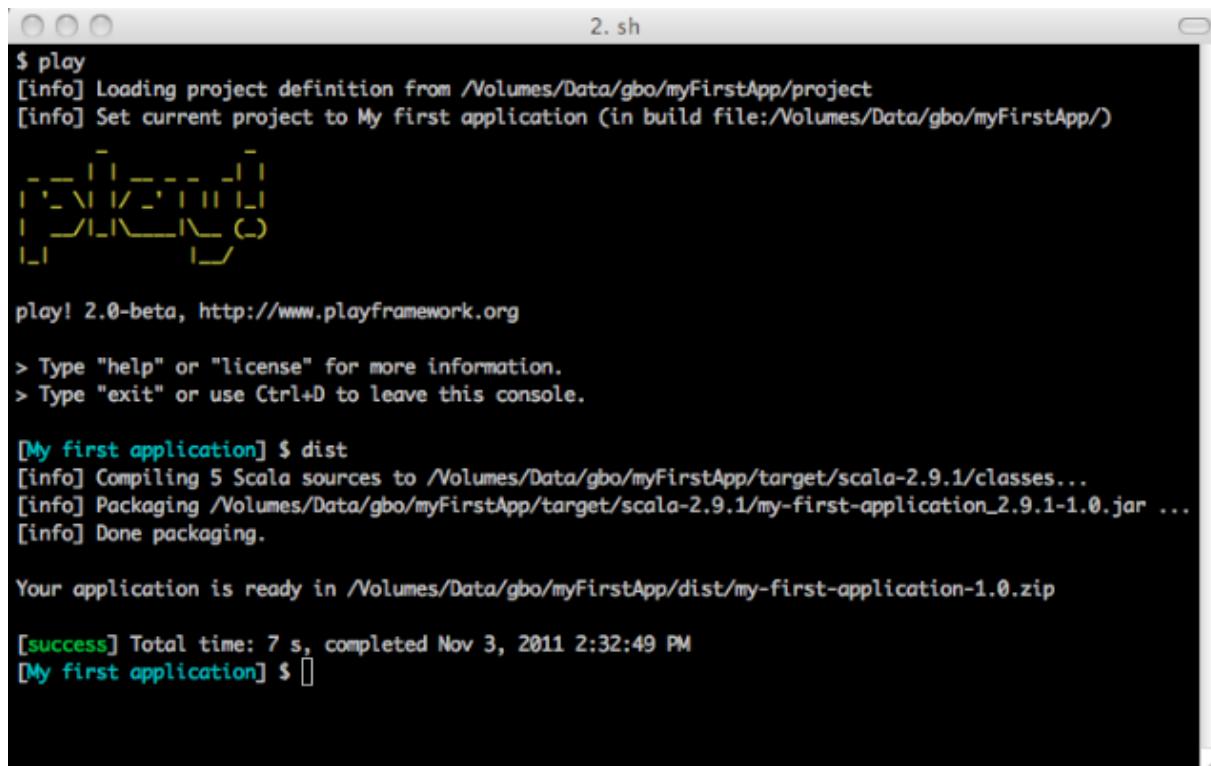
Using the dist task

The simplest way to deploy a Play 2.0 application is to retrieve the source (typically via a git workflow) on the server and to use either `play start` or `play stage` to start it in place.

However, you sometimes need to build a binary version of your application and deploy it to the server without any dependencies on Play itself. You can do this with the `dist` task.

In the Play console, simply type `dist`:

```
[My first application] $ dist
```



```
$ play
[info] Loading project definition from /Volumes/Data/gbo/myFirstApp/project
[info] Set current project to My first application (in build file:/Volumes/Data/gbo/myFirstApp/)

play! 2.0-beta, http://www.playframework.org

> Type "help" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[My first application] $ dist
[info] Compiling 5 Scala sources to /Volumes/Data/gbo/myFirstApp/target/scala-2.9.1/classes...
[info] Packaging /Volumes/Data/gbo/myFirstApp/target/scala-2.9.1/my-first-application_2.9.1-1.0.jar ...
[info] Done packaging.

Your application is ready in /Volumes/Data/gbo/myFirstApp/dist/my-first-application-1.0.zip

[success] Total time: 7 s, completed Nov 3, 2011 2:32:49 PM
[My first application] $
```

one can easily use an external `application.conf` by using a special system property called `conf.file`, so assuming your production `application.conf` is stored under your home directory, the following command should create a play distribution using the custom `application.conf`:

```
bash $ play -Dconfig.file=/home/peter/prod/application.conf dist
```

This produces a ZIP file containing all JAR files needed to run your application in the `target` folder of your application, the ZIP file's contents are organized as:

```
my-first-application-1.0
└ lib
  └ *.jar
└ start
```

You can use the generated `start` script to run your application.

Alternatively you can run `play dist` directly from your OS shell prompt, which does the same thing:

```
$ play dist
```

Publishing to a Maven (or Ivy) repository

You can also publish your application to a Maven repository. This publishes both the JAR file containing your application and the corresponding POM file.

You have to configure the repository you want to publish to, in the `project/Build.scala` file:

```
val main = PlayProject(appName, appVersion, appDependencies).settings(
  publishTo := Some(
    "My resolver" at "http://mycompany.com/repo"
  )

  credentials += Credentials(
    "Repo", "http://mycompany.com/repo", "admin", "admin123"
  )
)
```

Then in the Play console, use the `publish` task:

```
[My first application] $ publish
```

Check the sbt documentation to get more information about the resolvers and credentials definition.

Additional configuration

When running an application in production mode you can override any configuration. This section covers the more common use cases.

All these additional configurations are specified using Java System properties and can be used directly if you are using one of the `start` script generated by Play.

Specifying the HTTP server address and port

You can provide both HTTP port and address. The default is to listen on port `9000` at the `0.0.0.0` address (all addresses).

```
$ start -Dhttp.port=1234 -Dhttp.address=127.0.0.1
```

Note that these configuration are only provided for the default embeded Netty server.

Specifying additional JVM arguments

You can specify any JVM arguments to the `start` script. Otherwise the default JVM settings will be used:

```
$ start -Xms128M -Xmx512m -server
```

Specifying alternative configuration file

The default is to load the `application.conf` file from the classpath. You can specify an alternative configuration file if needed:

Using `-Dconfig.resource`

It will search for an alternative configuration file in the application classpath (you usually provide these alternative configuration files into your application `conf/` directory before packaging).

```
$ start -Dconfig.resource=prod.conf
```

Using `-Dconfig.file`

You can also specify another local configuration file not packaged into the application artifacts:

```
$ start -Dconfig.file=/opt/conf/prod.conf
```

Using `-Dconfig.url`

You can also specify a configuration file to be loaded from any URL:

```
$ start -Dconfig.url=http://conf.mycompany.com/conf/prod.conf
```

Note that you can always reference the original configuration file in a new `prod.conf` file using the `include` directive, such as:

```
include "application.conf"

key.to.override=blah
```

Overriding specific configuration keys

Sometimes you don't want to specify another complete configuration file, but just override a bunch of specific keys. You can do that by specifying them as Java System properties:

```
$ start -Dapplication.secret=verysecretkey -Ddb.default.password=toto
```

Using environment variables

You can also reference environment variables from your `application.conf` file:

```
my.key = defaultvalue
my.key = ${?MY_KEY_ENV}
```

Here, the override field `my.key = ${?MY_KEY_ENV}` simply vanishes if there's no value for `MY_KEY_ENV`, but if you set an environment variable `MY_KEY_ENV` for example, it would be used.

Changing the logback configuration file

You can also specify another logback configuration file via a System property.

Using `-Dlogger.resource`

Specify another loback configuration file to be loaded from the classpath:

```
$ start -Dlogger.resource=conf/prod-logger.xml
```

Using `-Dlogger.file`

Specify another loback configuration file to be loaded from the file system:

```
$ start -Dlogger.file=/opt/prod/logger.xml
```

Using `-Dlogger.url`

Specify another loback configuration file to be loaded from an URL:

```
$ start -Dlogger.url=http://conf.mycompany.com/logger.xml
```

Deploying to Heroku

Heroku is a cloud application platform – a new way of building and deploying web apps.

Add the Procfile

Heroku requires a special file in the application root called `Procfile`. Create a simple text file with the following content:

```
web: target/start -Dhttp.port=${PORT} ${JAVA_OPTS}
```

Store your application in git

Just create a git repository for your application:

```
$ git init  
$ git add .  
$ git commit -m "init"
```

Create a new application on Heroku

Note that you need an Heroku account, and to install the heroku gem.

```
$ heroku create --stack cedar  
Creating warm-frost-1289... done, stack is cedar  
http://warm-1289.herokuapp.com/ | git@heroku.com:warm-1289.git  
Git remote heroku added
```

Deploy your application

To deploy your application on Heroku, just use git to push it into the `heroku` remote repository:

```
$ git push heroku master
Counting objects: 34, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (34/34), 35.45 KiB, done.
Total 34 (delta 0), reused 0 (delta 0)

-----> Heroku receiving push
-----> Scala app detected
-----> Building app with sbt v0.11.0
-----> Running: sbt clean compile stage
...
-----> Discovering process types
      Procfile declares types -> web
-----> Compiled slug size is 46.3MB
-----> Launching... done, v5
      http://8044.herokuapp.com deployed to Heroku

To git@heroku.com:floating-lightning-8044.git
 * [new branch]      master -> master
```

Heroku will run `sbt clean compile stage` to prepare your application. On the first deployment, all dependencies will be downloaded, which takes a while to complete (but will be cached for future deployments).

Check that your application has been deployed

Now, let's check the state of the application's processes:

```
$ heroku ps
Process     State            Command
-----       -----
web.1        up for 10s      target/start
```

The web process is up. Review the logs for more information:

```
$ heroku logs
2011-08-18T00:13:41+00:00 heroku[web.1]: Starting process with command
`target/start'
2011-08-18T00:14:18+00:00 app[web.1]: Starting on port:28328
2011-08-18T00:14:18+00:00 app[web.1]: Started.
2011-08-18T00:14:19+00:00 heroku[web.1]: State changed from starting to up
...
```

Looks good. We can now visit the app with `heroku open`.

Running play commands remotely

Cedar allows you to launch a REPL process attached to your local terminal for experimenting in your application's environment:

Set-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:

```
$ start -Dhttp.port=80
```

Note that you probably need root permissions to bind a process on this port.

But 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 never give you better performance than using Play server directly.

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 )
    )
}
```

Set-up with Apache

The example below shows a simple set-up with Apache httpd server running in front of a standard Play 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>
```

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 Play app and the proxy running on the same machine, the Play 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 you enable the forward support in the XForwardedSupport configuration, Play will change the request.remoteAddress from the proxy's IP to the client's IP. You have to list the IP addresses of your proxy servers for this to work.

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 these two techniques, your app will appear to be directly exposed.

If you don't want this play app to occupy the whole root, add an exclusion directive to the proxy config:

```
ProxyPass /excluded !
```

Apache as a front proxy to allow transparent upgrade of your application

The basic idea is to run two Play instances of your web application and let the front-end proxy load-balance them. In case one is not available, it will forward all the requests to the available one.

Let's start the same Play application two times: one on port 9999 and one on port 9998.

```
$ start -Dhttp.port=9998
$ start -Dhttp.port=9999
```

Now, let's configure our Apache web server to have a load balancer.

In Apache, I have the following configuration:

```
<VirtualHost mysuperwebapp.com:80>
    ServerName mysuperwebapp.com
    <Location /balancer-manager>
        SetHandler balancer-manager
        Order Deny,Allow
        Deny from all
        Allow from .mysuperwebapp.com
    </Location>
    <Proxy balancer://mycluster>
        BalancerMember http://localhost:9999
        BalancerMember http://localhost:9998 status=+H
    </Proxy>
    <Proxy *>
        Order Allow,Deny
        Allow From All
    </Proxy>
    ProxyPreserveHost On
    ProxyPass /balancer-manager !
    ProxyPass / balancer://mycluster/
    ProxyPassReverse / http://localhost:9999/
    ProxyPassReverse / http://localhost:9998/
</VirtualHost>
```

The important part is `balancer://mycluster`. This declares a load balancer. The `+H` option means that the second Play application is on stand-by. But you can also instruct it to load-balance.

Apache also provides a way to view the status of your cluster. Simply point your browser to `/balancer-manager` to view the current status of your clusters.

Because Play is completely stateless you don't have to manage sessions between the 2 clusters. You can actually easily scale to more than 2 Play instances.

Building Play 2.0 from sources

To benefit from the latest improvements and bug fixes after the initial beta release, you may want to compile Play 2.0 from sources. You'll need a Git client to fetch the sources.

From the shell, first checkout the Play 2.0 sources:

```
$ git clone git://github.com/playframework/Play20.git
```

Then go to the `Play20/framework` directory and launch the `build` script to enter the sbt build console:

```
$ cd Play20/framework  
$ ./build  
> build-repository
```

Once in the sbt console, run `build-repository` to compile and build everything. This will also create the local Ivy repository containing all of the required dependencies.

Note that you don't need to install sbt yourself: Play 2.0 embeds its own version (currently sbt 0.11.2).

If you want to make changes to the code you can use `compile` and `publish-local` to rebuild the framework.

Running tests

You can run basic tests from the sbt console using the `test` task:

```
> test
```

We are also using several Play applications to test the framework. To run this complete test suite, use the `runtests` script:

```
$ ./runtests
```

Creating projects

Creating projects using the Play version you have built from source works much the same as a regular Play application.

```
export PATH=$PATH:/Play20
```

If you have an existing Play 2.0 application that you are upgrading from Play 2.0 Beta to edge, please add

```
resolvers ++= Seq(  
  ...  
  Resolver.file("Local Repository", file("<projdir>/Play20/repository/  
local"))(Resolver.ivyStylePatterns),  
  ...  
)  
  
addSbtPlugin("play" % "sbt-plugin" % "2.0-RC1-SNAPSHOT")
```

to project/plugins.sbt.

Continuous integration server

Our continuous integration runs on Cloudbees.



<https://playframework2.ci.cloudbees.com/>

Artifact repositories

Typesafe repository

All Play artifacts are published to the Typesafe repository at <http://repo.typesafe.com/typesafe/releases/>.

Note: it's a Maven2 compatible repository.

To enable it in your sbt build, you must add a proper resolver (typically in `plugins.sbt`):

```
// The Typesafe repository
resolvers += "Typesafe Releases" at "http://repo.typesafe.com/typesafe/
releases/"
```

Accessing snapshots

Snapshots are published daily from our Continuous Server to the Typesafe snapshots repository at <http://repo.typesafe.com/typesafe/snapshots/>.

```
// The Typesafe snapshots repository
resolvers += "Typesafe Snapshots" at "http://repo.typesafe.com/typesafe/
snapshots/"
```

Issues tracker

We use Lighthouse as issue tracker at:

<https://play.lighthouseapp.com/projects/82401-play-20/overview>.

Reporting bugs

Bug reports are incredibly helpful, so take time to report bugs and request features in our ticket tracker. We're always grateful for patches to Play's code. Indeed, bug reports with attached patches will get fixed far quickly than those without any.

Please include as much relevant information as possible including the exact framework version you're using and a code snippet that reproduces the problem.

Don't have too much expectations. Unless the bug is really a serious "everything is broken" thing, you're creating a ticket to start a discussion. Having a patch (or a branch on Github we can pull from) is better, but then again we'll only pull high quality branches that make sense to be in the core of Play.

Pull requests, and Play!

Here are some guidelines for submitting pull requests to the Play project (and why your request might not be accepted).

The reason we've written it is that we've had to reject a bunch of requests recently - not because they're not good - just because they don't match our current project priorities. Each request needs to be assessed and managed appropriately , and if you look at the commit logs you'll see we're stupidly busy at the moment and need to make some tough decisions on where our time goes.

A moving target

The Play2.0 framework is under heavy active development by the core team, and the codebase is changing rapidly. Small fixes and changes for bugs without tickets are a very low priority at the moment (though this will obviously change once the framework become more mature). Pull requests for minor issues - especially for things like cleaning whitespace, indenting, or fixing minor typographical issues will most likely be rejected.

Features are forever

Unfortunately , pull requests that add new features to Play are also likely to be rejected. A framework needs to fit the majority of cases, and can never cater for every situation: features that are absolutely essential to one team, will be redundant bloat to another. Additionally, any code that is merged into Play needs to be supported for a long long time. Adding support for something means maintaining, testing, and updating the related code throughout the framework's life - which requires resources from other parts of the project.

Any decision to add or remove even a minor feature has serious consequences and has to be considered carefully and thoroughly .

If you're keen to contribute

But good code is good code, and good ideas are good ideas. If you're serious about getting involved, you spot problems with code, or have a feature that you really think is missing (or shouldn't be present) then jump on the mailing lists and discuss it. We just don't want anyone to waste time creating some cool stuff that we can't use.

Contributor Guidelines

Implementation-wise, the following things should be avoided as much as possible:

- public mutable state
- global state
- implicit conversions
- threadLocal
- locks
- casting

Also, be careful with introducing new, heavy external dependencies.

source format

- run scalariform-format before commit

git commits

- prefer rebase
- bigger changesets

API design

- java only APIs should go to `framework/play/src/main/java`, package structure is `play.myapipackage .xxxx`
- java and scala APIs should be implemented the following way:
 - implement the core API in scala
 - if your component requires life cycle management or needs to be swappable, create a plugin, otherwise skip this step
 - wrap core API for scala users (example)
 - wrap scala API for java users (example)

Testing and documentation

- each and every public facing method and class need to have a corresponding scaladoc or javadoc with examples, description etc.
- each feature requires either a functional test (`framework/integrationtest`) or a spec (`/play/src/test`)
- run Play's integration test suite `framework/runtests` before pushing. If a test fails, fix it, do not ignore it.