# PLAY2 FRAMEWORK

For Scala Developer

DECEMBER 13, 2015

CONSOLIDIATE BY SCOTT HUANG
Personal Hobby

# Contents Index

# 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._

class 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.Result`:

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

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", 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
```

**Next:** HTTP Routing

# 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:

# Dependency Injection

Play supports generating two types of routers, one is a dependency injected router, the other is a static router. The default is the static router, but if you created a new Play application using the Play seed Activator templates, your project will include the following configuration in `build.sbt` telling it to use the injected router:

```
routesGenerator := InjectedRoutesGenerator
```

The code samples in Play's documentation assumes that you are using the injected routes generator. If you are not using this, you can trivially adapt the code samples for the static routes generator, either by prefixing the controller invocation part of the route with an `@` symbol, or by declaring each of your controllers as an `object` rather than a `class`.

# 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   /items/$id<[0-9]+>    controllers.Items.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   /clients/: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)
```

## Optional parameters

You can also specify an optional parameter that does not need to be present in all requests:

```
# The version parameter is optional. E.g. /api/list-all?version=3.0
GET   /api/list-all         controllers.Api.list(version: Option[String])
```

# 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._

class 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"))
}
```

**Next:** Manipulating results

# 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 `application/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 htmlResult2 = 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:

```
val 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:

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

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

```
val result2 = result.discardingCookies(DiscardingCookie("theme"))
```

You can also set and remove cookies as part of the same response:

```
val result3 = result.withCookies(Cookie("theme", "blue")).discardingCookies(DiscardingCookie("skin"))
```

# 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 (see why to use utf-8).

The charset is used to both convert the text response to the corresponding bytes to send over the network socket, and to update the `Content-Type` header with the proper `;charset=xxx` 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:

```
class 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.

**Next:** Session and Flash scopes

# 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. The default name for the cookie is `PLAY_SESSION`. This can be changed by configuring the key `session.cookieName` in application.conf.

If the name of the cookie is changed, the earlier cookie can be discarded using the same methods mentioned in Setting and discarding cookies.

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 store a unique ID in the user Session to keep them related to a specific user.

By default, 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.). You can also set the maximum age of the session cookie by configuring the key `session.maxAge` (in milliseconds) in `application.conf`.

# 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(
  request.session + ("saidHello" -> "yes"))
```

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

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

# 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")
  }
}
```

# 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 {
    request.flash.get("success").getOrElse("Welcome!")
  }
}

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

To retrieve the Flash scope value in your view, add an implicit Flash parameter:

```
@()(implicit flash: Flash)
...
@flash.get("success").getOrElse("Welcome!")
...
```

And in your Action, specify an `implicit request =>` as shown below:

```
def index = Action { implicit request =>
  Ok(views.html.index())
}
```

An implicit Flash will be provided to the view based on the implicit request.

If the error '*could not find implicit value for parameter flash: play.api.mvc.Flash*' is raised then this is because your Action didn't have an implicit request in scope.

**Next:** Body parsers

# 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 that 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`
- **application/xml**, **text/xml** or **application/XXX+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 extracts 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 {
    sys.error("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 max content length because they have to load all the content into memory. By default, the maximum content length that they will parse is 100KB. It can be overridden by specifying the `play.http.parser.maxMemoryBuffer` property in `application.conf`:

```
play.http.parser.maxMemoryBuffer=128K
```

For parsers that buffer content on disk, such as the raw parser or `multipart/form-data`, the maximum content length is specified using the`play.http.parser.maxDiskBuffer` property, it defaults to 10MB.

The `multipart/form-data` parser also enforces the text max length property for the aggregate of the data fields.

You can also override the default maximum length for a given action:

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

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

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

**Next:** Actions composition

# Action composition

This chapter introduces several ways of defining generic action functionality.

# Custom action builders

We saw previously that there are multiple ways to declare an action - with a request parameter, without a request parameter, with a body parser etc. In fact there are more than this, as we'll see in the chapter on asynchronous programming.
These methods for building actions are actually all defined by a trait called `ActionBuilder` and the `Action` object that we use to declare our actions is just an instance of this trait. By implementing your own `ActionBuilder`, you can declare reusable action stacks, that can then be used to build actions.

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

The first way is to implement this functionality in the `invokeBlock` method, which is called for every action built by the `ActionBuilder`:

```
import play.api.mvc._

object LoggingAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    Logger.info("Calling action")
    block(request)
  }
```

```
}
```

Now we can use it the same way we use `Action`:

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

Since `ActionBuilder` provides all the different methods of building actions, this also works with, for example, declaring a custom body parser:

```
def submit = LoggingAction(parse.text) { request =>
  Ok("Got a body " + request.body.length + " bytes long")
}
```

## Composing actions

In most applications, we will want to have multiple action builders, some that do different types of authentication, some that provide different types of generic functionality, etc. In which case, we won't want to rewrite our logging action code for each type of action builder, we will want to define it in a reuseable way.

Reusable action code can be implemented by wrapping actions:

```
import play.api.mvc._

case class Logging[A](action: Action[A]) extends Action[A] {

  def apply(request: Request[A]): Future[Result] = {
    Logger.info("Calling action")
    action(request)
  }

  lazy val parser = action.parser
}
```

We can also use the `Action` action builder to build actions without defining our own action class:

```
import play.api.mvc._

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

Actions can be mixed in to action builders using the `composeAction` method:

```
object LoggingAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    block(request)
  }
  override def composeAction[A](action: Action[A]) = new Logging(action)
}
```

Now the builder can be used in the same way as before:

```
def index = LoggingAction {
```

```
  Ok("Hello World")
}
```

We can also mix in wrapping actions without the action builder:

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

## More complicated actions

So far we've only shown actions that don't impact the request at all. Of course, we can also read and modify the incoming request object:

```
import play.api.mvc._

def xForwardedFor[A](action: Action[A]) = Action.async(action.parser) { request =>
  val newRequest = request.headers.get("X-Forwarded-For").map { xff =>
    new WrappedRequest[A](request) {
      override def remoteAddress = xff
    }
  } getOrElse request
  action(newRequest)
}
```

**Note:** Play already has built in support for `X-Forwarded-For` headers.

We could block the request:

```
import play.api.mvc._

def onlyHttps[A](action: Action[A]) = Action.async(action.parser) { request =>
  request.headers.get("X-Forwarded-Proto").collect {
    case "https" => action(request)
  } getOrElse {
    Future.successful(Forbidden("Only HTTPS requests allowed"))
  }
}
```

And finally we can also modify the returned result:

```
import play.api.mvc._
import play.api.libs.concurrent.Execution.Implicits._

def addUaHeader[A](action: Action[A]) = Action.async(action.parser) { request =>
  action(request).map(_.withHeaders("X-UA-Compatible" -> "Chrome=1"))
}
```

# Different request types

While action composition allows you to perform additional processing at the HTTP request and response level, often you want to build pipelines of data transformations that add context to or perform validation on the request itself. `ActionFunction` can be thought of as a function on the request, parameterized over both the input request type and the output type passed on to the next layer. Each action function may represent modular processing such as authentication, database lookups for objects, permission checks, or other operations that you wish to compose and reuse across actions.

There are a few pre-defined traits implementing `ActionFunction` that are useful for different types of processing:

- `ActionTransformer` can change the request, for example by adding additional information.
- `ActionFilter` can selectively intercept requests, for example to produce errors, without changing the request value.
- `ActionRefiner` is the general case of both of the above.
- `ActionBuilder` is the special case of functions that take `Request` as input, and thus can build actions.

You can also define your own arbitrary `ActionFunction` by implementing the `invokeBlock` method. Often it is convenient to make the input and output types instances of `Request` (using `WrappedRequest`), but this is not strictly necessary.

## Authentication

One of the most common use cases for action functions is authentication. We can easily implement our own authentication action transformer that determines the user from the original request and adds it to a new `UserRequest`. Note that this is also an `ActionBuilder` because it takes a simple `Request` as input:

```
import play.api.mvc._

class UserRequest[A](val username: Option[String], request: Request[A]) extends
WrappedRequest[A](request)

object UserAction extends
   ActionBuilder[UserRequest] with ActionTransformer[Request, UserRequest] {
 def transform[A](request: Request[A]) = Future.successful {
   new UserRequest(request.session.get("username"), request)
  }
}
```

Play also provides a built in authentication action builder. Information on this and how to use it can be found here.

**Note:** The built in authentication action builder is just a convenience helper to minimise the code necessary to implement authentication for simple cases, its implementation is very similar to the example above.

If you have more complex requirements than can be met by the built in authentication action, then implementing your own is not only simple, it is recommended.

## Adding information to requests

Now let's consider a REST API that works with objects of type `Item`. There may be many routes under the `/item/:itemId` path, and each of these need to look up the item. In this case, it may be useful to put this logic into an action function.

First of all, we'll create a request object that adds an `Item` to our `UserRequest`:

```
import play.api.mvc._

class ItemRequest[A](val item: Item, request: UserRequest[A]) extends WrappedRequest[A](request) {
  def username = request.username
}
```

Now we'll create an action refiner that looks up that item and returns `Either` an error (`Left`) or a new `ItemRequest` (`Right`). Note that this action refiner is defined inside a method that takes the id of the item:

```
def ItemAction(itemId: String) = new ActionRefiner[UserRequest, ItemRequest] {
  def refine[A](input: UserRequest[A]) = Future.successful {
    ItemDao.findById(itemId)
      .map(new ItemRequest(_, input))
      .toRight(NotFound)
  }
}
```

## Validating requests

Finally, we may want an action function that validates whether a request should continue. For example, perhaps we want to check whether the user from `UserAction` has permission to access the item from `ItemAction`, and if not return an error:

```
object PermissionCheckAction extends ActionFilter[ItemRequest] {
  def filter[A](input: ItemRequest[A]) = Future.successful {
    if (!input.item.accessibleByUser(input.username))
      Some(Forbidden)
    else
      None
  }
}
```

## Putting it all together

Now we can chain these action functions together (starting with an `ActionBuilder`) using `andThen` to create an action:

```
def tagItem(itemId: String, tag: String) =
  (UserAction andThen ItemAction(itemId) andThen PermissionCheckAction) { request =>
    request.item.addTag(tag)
    Ok("User " + request.username + " tagged " + request.item.id)
  }
```

Play also provides a global filter API , which is useful for global cross cutting concerns.

**Next:** Content negotiation

# Content negotiation

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

## Language

You can get the list of acceptable languages for a request using the`play.api.mvc.RequestHeader#acceptLanguages` method that retrieves them from the `Accept-Language` header and sorts them according to their quality value. Play uses it in the `play.api.mvc.Controller#lang` method that provides an implicit`play.api.i18n.Lang` value to your actions, so they automatically use the best possible language (if supported by your application, otherwise your application's default language is used).

## Content

Similarly, the `play.api.mvc.RequestHeader#acceptedTypes` method gives the list of acceptable result's MIME types for a request. It retrieves them from the `Accept` request header and sorts them according to their quality factor.
Actually, the `Accept` header does not really contain MIME types but media ranges (*e.g.* a request accepting all text results may set the `text/*` range, and the `*/*` range means that all result types are acceptable). Controllers provide a higher-level `render` method to help you to handle media ranges. Consider for example the following action definition:

```
val list = Action { implicit request =>
  val items = Item.findAll
  render {
    case Accepts.Html() => Ok(views.html.list(items))
    case Accepts.Json() => Ok(Json.toJson(items))
  }
}
```

`Accepts.Html()` and `Accepts.Json()` are extractors testing if a given media range matches `text/html` and `application/json`, respectively. The `render` method takes a partial function from `play.api.http.MediaRange` to `play.api.mvc.Result` and tries to apply it to each media range found in the request `Accept` header, in order of preference. If none of the acceptable media ranges is supported by your function, the`NotAcceptable` result is returned.

For example, if a client makes a request with the following value for the `Accept` header:`*/*;q=0.5,application/json`, meaning that it accepts any result type but prefers JSON, the above code will return the JSON representation. If another client makes a request with the following value for the `Accept` header: `application/xml`, meaning that it only accepts XML, the above code will return `NotAcceptable`.

## Request extractors

See the API documentation of the `play.api.mvc.AcceptExtractors.Accepts` object for the list of the MIME types supported by Play out of the box in the `render` method. You can easily create your own extractor for a given MIME type using the`play.api.mvc.Accepting` case class, for example the following code creates an extractor checking that a media range matches the `audio/mp3` MIME type:

```
 val AcceptsMp3 = Accepting("audio/mp3")
 render {
  case AcceptsMp3() => ???
 }
}
```

**Next:** Handling errors

# Handling errors

There are two main types of errors that an HTTP application can return - client errors and server errors. Client errors indicate that the connecting client has done something wrong, server errors indicate that there is something wrong with the server.

Play will in many circumstances automatically detect client errors - these include errors such as malformed header values, unsupported content types, and requests for resources that can’t be found. Play will also in many circumstances automatically handle server errors - if your action code throws an exception, Play will catch this and generate a server error page to send to the client.

The interface through which Play handles these errors is `HttpErrorHandler`. It defines two methods, `onClientError`, and `onServerError`.

# Supplying a custom error handler

A custom error handler can be supplied by creating a class in the root package called `ErrorHandler` that implements `HttpErrorHandler`, for example:

```
import play.api.http.HttpErrorHandler
import play.api.mvc._
import play.api.mvc.Results._
import scala.concurrent._

class ErrorHandler extends HttpErrorHandler {

  def onClientError(request: RequestHeader, statusCode: Int, message: String) = {
    Future.successful(
      Status(statusCode)("A client error occurred: " + message)
    )
  }

  def onServerError(request: RequestHeader, exception: Throwable) = {
    Future.successful(
      InternalServerError("A server error occurred: " + exception.getMessage)
    )
  }
}
```

If you don't want to place your error handler in the root package, or if you want to be able to configure different error handlers for different environments, you can do this by configuring the `play.http.errorHandler` configuration property in `application.conf`:

```
play.http.errorHandler = "com.example.ErrorHandler"
```

# Extending the default error handler

Out of the box, Play's default error handler provides a lot of useful functionality. For example, in dev mode, when a server error occurs, Play will attempt to locate and render the piece of code in your application that caused that exception, so that you can quickly see and identify the problem. You may want to provide custom server errors in production, while still maintaining that functionality in development. To facilitate this, Play provides a `DefaultHttpErrorHandler` that has some convenience methods that you can override so that you can mix in your custom logic with Play's existing behavior.

For example, to just provide a custom server error message in production, leaving the development error message untouched, and you also wanted to provide a specific forbidden error page:

```
import javax.inject._

import play.api.http.DefaultHttpErrorHandler
import play.api._
import play.api.mvc._
```

```
import play.api.mvc.Results._
import play.api.routing.Router
import scala.concurrent._

class ErrorHandler @Inject() (
    env: Environment,
    config: Configuration,
    sourceMapper: OptionalSourceMapper,
    router: Provider[Router]
  ) extends DefaultHttpErrorHandler(env, config, sourceMapper, router) {

  override def onProdServerError(request: RequestHeader, exception: UsefulException) = {
    Future.successful(
      InternalServerError("A server error occurred: " + exception.getMessage)
    )
  }

  override def onForbidden(request: RequestHeader, message: String) = {
    Future.successful(
      Forbidden("You're not allowed to access this resource.")
    )
  }
}
```

Checkout the full API documentation for `DefaultHttpErrorHandler` to see what methods are available to override, and how you can take advantage of them.

**Next:** Asynchronous HTTP programming

# Handling asynchronous results

## Make controllers asynchronous

Internally, Play Framework is asynchronous from the bottom up. Play handles every request in an asynchronous, non-blocking way.

The default configuration is tuned for asynchronous controllers. In other words, the application code should avoid blocking in controllers, i.e., having the controller code wait for an operation. Common examples of such blocking operations are JDBC calls, streaming API, HTTP requests and long computations.

Although it's possible to increase the number of threads in the default execution context to allow more concurrent requests to be processed by blocking controllers, following the recommended approach of keeping the controllers asynchronous makes it easier to scale and to keep the system responsive under load.

# Creating non-blocking actions

Because of the way Play 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 generate it? The response is a *future* result!

A `Future[Result]` will eventually be redeemed with a value of type `Result`. By giving a`Future[Result]` instead of a normal `Result`, we are able to quickly generate the result without blocking. Play will then serve the result as soon as the promise is redeemed.

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

# How to create a `Future[Result]`

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

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext

val futurePIValue: Future[Double] = computePIAsynchronously()
val futureResult: Future[Result] = futurePIValue.map { pi =>
  Ok("PI value computed: " + pi)
}
```

All of Play's asynchronous API calls give you a `Future`. 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`. Here is a simple way to execute a block of code asynchronously and to get a `Future`:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext

val futureInt: Future[Int] = scala.concurrent.Future {
  intensiveComputation()
}
```

**Note:** It's important to understand which thread code runs on with futures. In the two code blocks above, there is an import on Plays default execution context. This is an implicit parameter that gets passed to all methods on the future API that accept callbacks. The execution context will often be equivalent to a thread pool, though not necessarily.

You can't magically turn synchronous IO into asynchronous by wrapping it in a `Future`. If you can't change the application's architecture to avoid blocking operations, at some point that operation

will have to be executed, and that thread is going to block. So in addition to enclosing the operation in a `Future`, it's necessary to configure it to run in a separate execution context that has been configured with enough threads to deal with the expected concurrency. See Understanding Play thread pools for more information.
It can also be helpful to use Actors for blocking operations. Actors provide a clean model for handling timeouts and failures, setting up blocking execution contexts, and managing any state that may be associated with the service. Also Actors provide patterns like`ScatterGatherFirstCompletedRouter` to address simultaneous cache and database requests and allow remote execution on a cluster of backend servers. But an Actor may be overkill depending on what you need.

# Returning futures

While we were using the `Action.apply` builder method to build actions until now, to send an asynchronous result we need to use the `Action.async` builder method:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext

def index = Action.async {
  val futureInt = scala.concurrent.Future { intensiveComputation() }
  futureInt.map(i => Ok("Got result: " + i))
}
```

# Actions are asynchronous by default

Play actions are asynchronous by default. For instance, in the controller code below, the`{ Ok(...) }` part of the code is not the method body of the controller. It is an anonymous function that is being passed to the `Action` object's `apply` method, which creates an object of type `Action`. Internally, the anonymous function that you wrote will be called and its result will be enclosed in a `Future`.

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

**Note:** Both `Action.apply` and `Action.async` create `Action` objects that are handled internally in the same way. There is a single kind of `Action`, which is asynchronous, and not two kinds (a synchronous one and an asynchronous one). The `.async` builder is just a facility to simplify creating actions based on APIs that return a `Future`, which makes it easier to write non-blocking code.

# 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:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.duration._

def index = Action.async {
  val futureInt = scala.concurrent.Future { intensiveComputation() }
  val timeoutFuture = play.api.libs.concurrent.Promise.timeout("Oops", 1.second)
  Future.firstCompletedOf(Seq(futureInt, timeoutFuture)).map {
    case i: Int => Ok("Got result: " + i)
    case t: String => InternalServerError(t)
  }
}
```

**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, 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 {
  Result(
    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)

  Result(
    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)

  Result(
    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 can 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.

https://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 `Ok.chunked`:

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

  Ok.chunked(dataContent)
}
```

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

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

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.

**Next:** Comet sockets

# 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.chunked(events).as(HTML)
}
```

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

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.twirl.api.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.chunked(events &> toCometMessage)
}
```

**Tip:** Writing `events &> toCometMessage` is just another way of writing`events.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.chunked(events &> Comet(callback = "console.log"))
}
```

# 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.chunked(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>
```

**Next:** WebSockets

# WebSockets

WebSockets are sockets that can be used from a web browser based on a protocol that allows two way full duplex communication. The client can send messages and the server can receive messages at any time, as long as there is an active WebSocket connection between the server and the client.

Modern HTML5 compliant web browsers natively support WebSockets via a JavaScript WebSocket API. However WebSockets are not limited in just being used by WebBrowsers, there are many WebSocket client libraries available, allowing for example servers to talk to each other, and also native mobile apps to use WebSockets. Using WebSockets in these contexts has the advantage of being able to reuse the existing TCP port that a Play server uses.

# 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`.

Play provides two different built in mechanisms for handling WebSockets. The first is using actors, the second is using iteratees. Both of these mechanisms can be accessed using the builders provided on WebSocket.

# Handling WebSockets with actors

To handle a WebSocket with an actor, we need to give Play a `akka.actor.Props` object that describes the actor that Play should create when it receives the WebSocket connection. Play will give us an `akka.actor.ActorRef` to send upstream messages to, so we can use that to help create the `Props` object:

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

def socket = WebSocket.acceptWithActor[String, String] { request => out =>
  MyWebSocketActor.props(out)
}
```

The actor that we're sending to here in this case looks like this:

```
import akka.actor._

object MyWebSocketActor {
  def props(out: ActorRef) = Props(new MyWebSocketActor(out))
}
```

```
class MyWebSocketActor(out: ActorRef) extends Actor {
  def receive = {
    case msg: String =>
      out ! ("I received your message: " + msg)
  }
}
```

Any messages received from the client will be sent to the actor, and any messages sent to the actor supplied by Play will be sent to the client. The actor above simply sends every message received from the client back with `I received your message:` prepended to it.

## Detecting when a WebSocket has closed

When the WebSocket has closed, Play will automatically stop the actor. This means you can handle this situation by implementing the actors `postStop` method, to clean up any resources the WebSocket might have consumed. For example:

```
override def postStop() = {
  someResource.close()
}
```

## Closing a WebSocket

Play will automatically close the WebSocket when your actor that handles the WebSocket terminates. So, to close the WebSocket, send a `PoisonPill` to your own actor:

```
import akka.actor.PoisonPill

self ! PoisonPill
```

## Rejecting a WebSocket

Sometimes you may wish to reject a WebSocket request, for example, if the user must be authenticated to connect to the WebSocket, or if the WebSocket is associated with some resource, whose id is passed in the path, but no resource with that id exists. Play provides `tryAcceptWithActor` to address this, allowing you to return either a result (such as forbidden, or not found), or the actor to handle the WebSocket with:

```
import scala.concurrent.Future
import play.api.mvc._
import play.api.Play.current

def socket = WebSocket.tryAcceptWithActor[String, String] { request =>
  Future.successful(request.session.get("user") match {
    case None => Left(Forbidden)
    case Some(_) => Right(MyWebSocketActor.props)
  })
}
```

## Handling different types of messages

So far we have only seen handling `String` frames. Play also has built in handlers for `Array[Byte]` frames, and `JsValue` messages parsed from `String` frames. You can pass these as the type parameters to the WebSocket creation method, for example:

```
import play.api.mvc._
import play.api.libs.json._
```

```
import play.api.Play.current

def socket = WebSocket.acceptWithActor[JsValue, JsValue] { request => out =>
  MyWebSocketActor.props(out)
}
```

You may have noticed that there are two type parameters, this allows us to handle differently typed messages coming in to messages going out. This is typically not useful with the lower level frame types, but can be useful if you parse the messages into a higher level type.

For example, let's say we want to receive JSON messages, and we want to parse incoming messages as `InEvent` and format outgoing messages as `OutEvent`. The first thing we want to do is create JSON formats for out `InEvent` and `OutEvent` types:

```
import play.api.libs.json._

implicit val inEventFormat = Json.format[InEvent]
implicit val outEventFormat = Json.format[OutEvent]
```

Now we can create WebSocket `FrameFormatter`'s for these types:

```
import play.api.mvc.WebSocket.FrameFormatter

implicit val inEventFrameFormatter = FrameFormatter.jsonFrame[InEvent]
implicit val outEventFrameFormatter = FrameFormatter.jsonFrame[OutEvent]
```

And finally, we can use these in our WebSocket:

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

def socket = WebSocket.acceptWithActor[InEvent, OutEvent] { request => out =>
  MyWebSocketActor.props(out)
}
```

Now in our actor, we will receive messages of type `InEvent`, and we can send messages of type `OutEvent`.

# Handling WebSockets with iteratees

While actors are a better abstraction for handling discrete messages, iteratees are often a better abstraction for handling streams.

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

```
import play.api.mvc._
import play.api.libs.iteratee._
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

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

  // Log events to the console
  val in = Iteratee.foreach[String](println).map { _ =>
    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`.

It 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 https://www.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:

```
import play.api.mvc._
import play.api.libs.iteratee._

def socket = WebSocket.using[String] { request =>

  // Just ignore the input
  val in = Iteratee.ignore[String]

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

  (in, out)
}
```

Here is another example in which the input data is logged to standard out and broadcast to the client utilizing `Concurrent.broadcast`.

```
import play.api.mvc._
import play.api.libs.iteratee._
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

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

 // Concurrent.broadcast returns (Enumerator, Concurrent.Channel)
 val (out, channel) = Concurrent.broadcast[String]

 // log the message to stdout and send response back to client
 val in = Iteratee.foreach[String] {
   msg =>
     println(msg)
     // the Enumerator returned by Concurrent.broadcast subscribes to the channel and will
     // receive the pushed messages
     channel push("I received your message: " + msg)
 }
 (in,out)
}
```

**Next:** The template engine

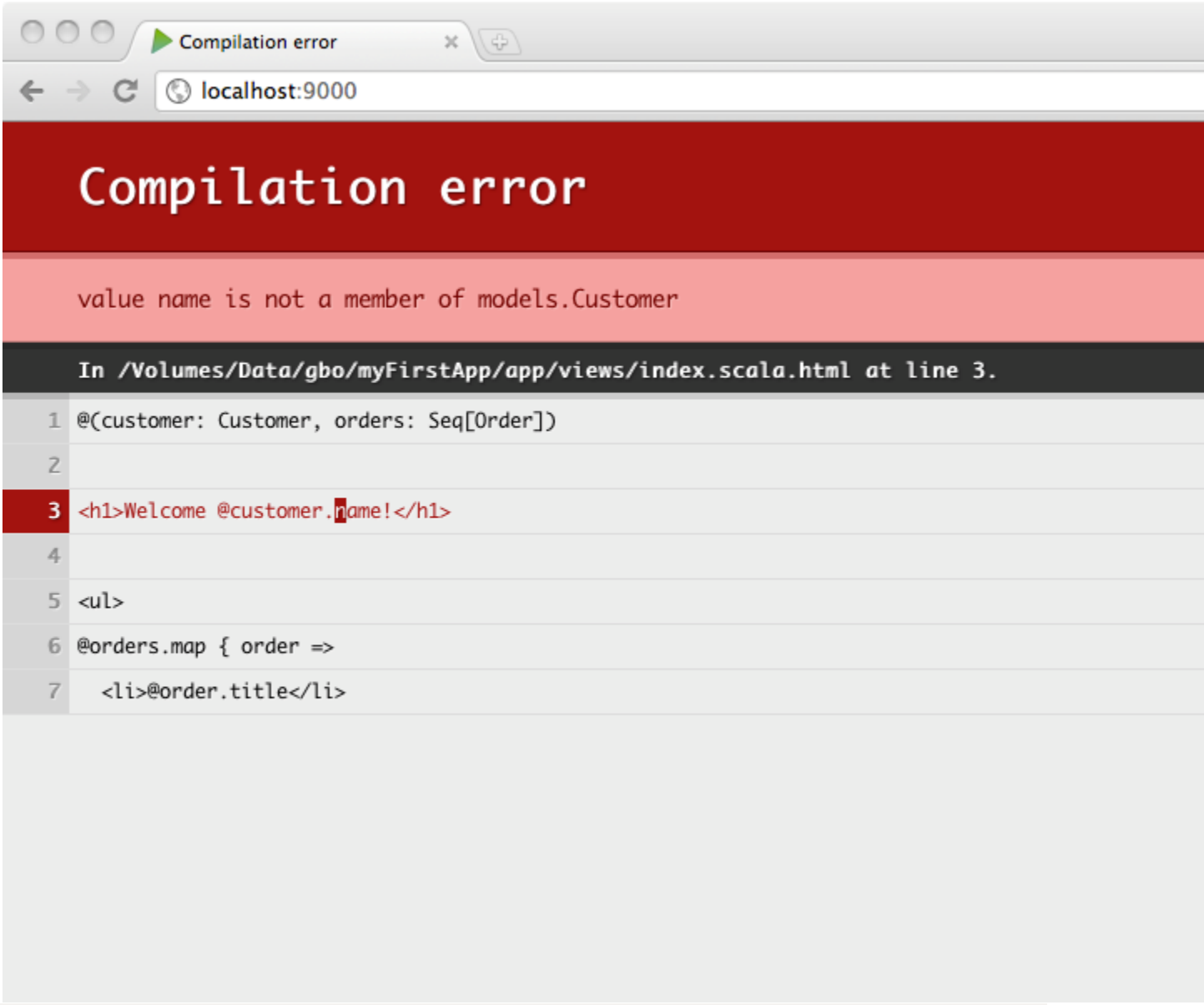
# The template engine

# A type safe template engine based on Scala

Play comes with Twirl, a 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.

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 an `apply()` method.

For example, here is a simple template:

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.title</li>
}
</ul>
```

You can then call this from any Scala code as you would normally call a method on a class:

```
val content = views.html.Application.index(c, o)
```

# 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.name!
      ^^^^^^^^^^^^^
      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.firstName + customer.lastName)!
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
            Dynamic Code
```

You can also use curly brackets, to write a multi-statement block:

```
Hello @{val name = customer.firstName + customer.lastName; 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: Customer, orders: List[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.name ($@p.price)</li>
}
</ul>
```

**Note:** Make sure that `{` is on the same line with `for` to indicate that the expression continues to next line.

# 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: Product) = {
  @product.name ($@product.price)
}

<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 Scala class (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: Customer, orders: List[Order])

@import utils._

...
```

To make an absolute resolution, use ***root*** prefix in the import statement.

```
@import _root_.company.product.core._
```

If you have common imports, which you need in all templates, you can declare in `build.sbt`

```
TwirlKeys.templateImports += "org.abc.backend._"
```

# 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>
```

# String interpolation

The template engine can be used as a string interpolator. You basically trade the "@" for a "$":

```
import play.twirl.api.StringInterpolation

val name = "Martin"
val p = html"<p>Hello $name</p>"
```

**Next:** Common use cases

# 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>
```

# moreScripts and moreStyles equivalents

To define old moreScripts or moreStyles variables equivalents (like on Play! 1.x) on a Scala template, you can define a variable in the main template like this :

```
@(title: String, scripts: Html = Html(""))(content: Html)

<!DOCTYPE html>

<html>
   <head>
      <title>@title</title>
      <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
      <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
      <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>
      @scripts
   </head>
   <body>
      <div class="navbar navbar-fixed-top">
         <div class="navbar-inner">
            <div class="container">
               <a class="brand" href="#">Movies</a>
            </div>
         </div>
      </div>
      <div class="container">
         @content
      </div>
   </body>
</html>
```

And on an extended template that need an extra script :

```
@scripts = {
   <script type="text/javascript">alert("hello !");</script>
}

@main("Title",scripts){

  Html content here ...

}
```

And on an extended template that not need an extra script, just like this :

```
@main("Title"){

  Html content here ...

}
```

**Next:** Custom format

# Adding support for a custom format to the template engine

The built-in template engine supports common template formats (HTML, XML, etc.) but you can easily add support for your own formats, if needed. This page summarizes the steps to follow to support a custom format.

---

# Overview of the templating process

The template engine builds its result by appending static and dynamic content parts of a template. Consider for instance the following template:

```
foo @bar baz
```

It consists in two static parts (`foo ` and ` baz`) around one dynamic part (`bar`). The template engine concatenates these parts together to build its result. Actually, in order to prevent cross-site scripting attacks, the value of `bar` can be escaped before being concatenated to the rest of the result. This escaping process is specific to each format: e.g. in the case of HTML you want to transform “<” into “<”.
How does the template engine know which format correspond to a template file? It looks at its extension: e.g. if it ends with `.scala.html` it associates the HTML format to the file.

Finally, you usually want your template files to be used as the body of your HTTP responses, so you have to define how to make a Play result from a template rendering result.

In summary, to support your own template format you need to perform the following steps:

- Implement the text integration process for the format ;
- Associate a file extension to the format ;
- Eventually tell Play how to send the result of a template rendering as an HTTP response body.

# Implement a format

Implement the `play.twirl.api.Format[A]` trait that has the methods `raw(text: String): A` and `escape(text: String): A` that will be used to integrate static and dynamic template parts, respectively.
The type parameter `A` of the format defines the result type of the template rendering, e.g.`Html` for a HTML template. This type must be a subtype of the`play.twirl.api.Appendable[A]` trait that defines how to concatenates parts together.
For convenience, Play provides a `play.twirl.api.BufferedContent[A]` abstract class that implements `play.twirl.api.Appendable[A]` using a `StringBuilder` to build its result and that implements the `play.twirl.api.Content` trait so Play knows how to serialize it as an HTTP response body (see the last section of this page for details).
In short, you need to write to classes: one defining the result (implementing`play.twirl.api.Appendable[A]`) and one defining the text integration process (implementing `play.twirl.api.Format[A]`). For instance, here is how the HTML format is defined:

```
// The `Html` result type. We extend `BufferedContent[Html]` rather than just `Appendable[Html]` so
// Play knows how to make an HTTP result from a `Html` value
class Html(buffer: StringBuilder) extends BufferedContent[Html](buffer) {
  val contentType = MimeTypes.HTML
}

object HtmlFormat extends Format[Html] {
  def raw(text: String): Html = …
  def escape(text: String): Html = …
}
```

# Associate a file extension to the format

The templates are compiled into a `.scala` files by the build process just before compiling the whole application sources. The `TwirlKeys.templateFormats` key is a sbt setting of type `Map[String, String]` defining the mapping between file extensions and template formats. For instance, if HTML was not supported out of the box by Play, you would have to write the following in your build file to associate the `.scala.html` files to the `play.twirl.api.HtmlFormat` format:

```
TwirlKeys.templateFormats += ("html" -> "my.HtmlFormat.instance")
```

Note that the right side of the arrow contains the fully qualified name of a value of type `play.twirl.api.Format[_]`.

# Tell Play how to make an HTTP result from a template result type

Play can write an HTTP response body for any value of type `A` for which it exists an implicit `play.api.http.Writeable[A]` value. So all you need is to define such a value for your template result type. For instance, here is how to define such a value for HTTP:

```
implicit def writableHttp(implicit codec: Codec): Writeable[Http] =
  Writeable[Http](result => codec.encode(result.body), Some(ContentTypes.HTTP))
```

**Note:** if your template result type extends `play.twirl.api.BufferedContent` you only need to define an implicit `play.api.http.ContentTypeOf` value:

```
scala implicit def contentTypeHttp(implicit codec: Codec):
ContentTypeOf[Http] = ContentTypeOf[Http](Some(ContentTypes.HTTP))
```

**Next:** Form submission and validation

# Handling form submission

# Overview

Form handling and submission is an important part of any web application. Play comes with features that make handling simple forms easy and complex forms possible.

Play's form handling approach is based around the concept of binding data. When data comes in from a POST request, Play will look for formatted values and bind them to a `Form` object. From there, Play can use the bound form to value a case class with data, call custom validations, and so on.

Typically forms are used directly from a `Controller` instance. However, `Form` definitions do not have to match up exactly with case classes or models: they are purely for handling input and it is reasonable to use a distinct `Form` for a distinct POST.

# Imports

To use forms, import the following packages into your class:

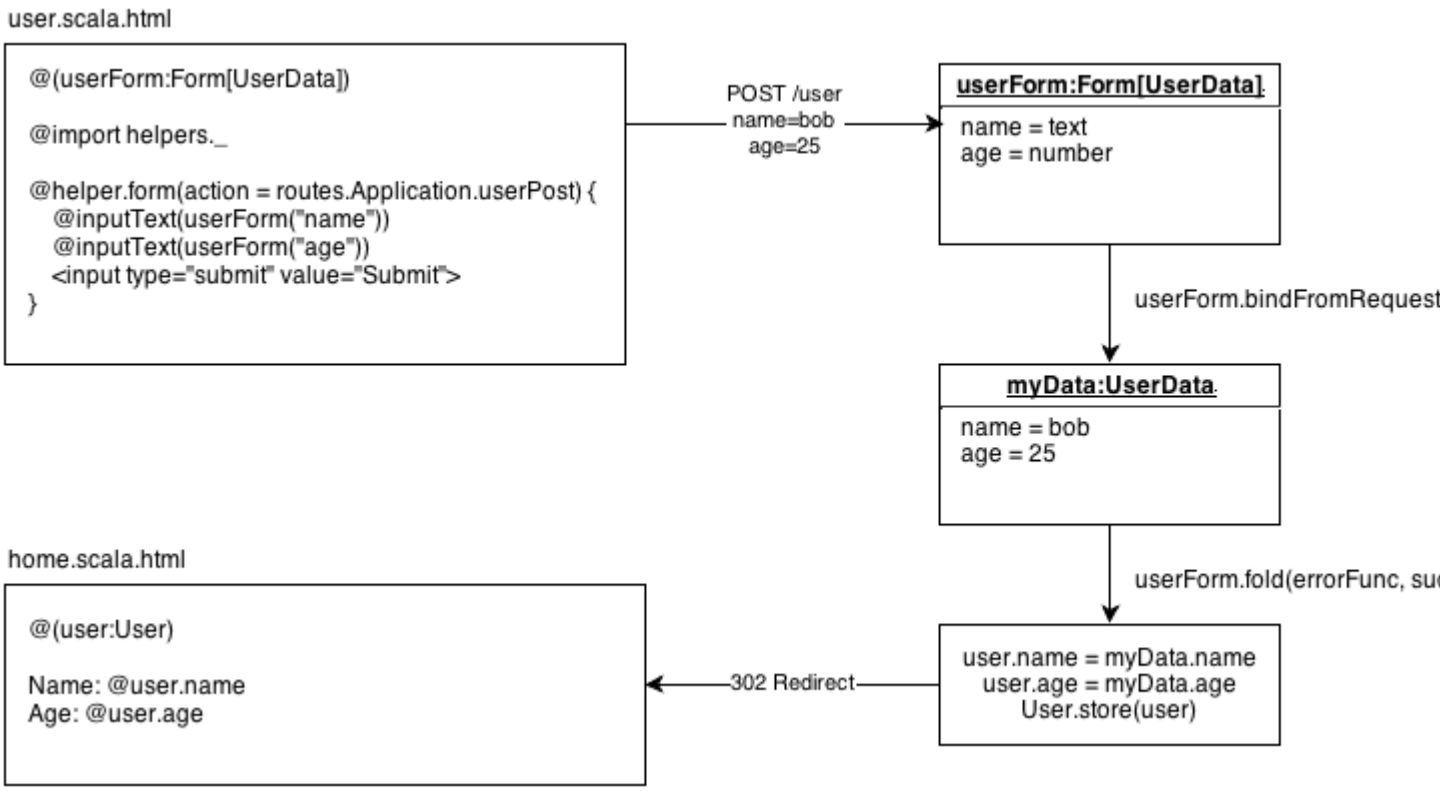
```
import play.api.data._
import play.api.data.Forms._
```

# Form Basics

We'll go through the basics of form handling:

- defining a form,
- defining constraints in the form,
- validating the form in an action,
- displaying the form in a view template,
- and finally, processing the result (or errors) of the form in a view template.

The end result will look something like this:




## Defining a form

First, define a case class which contains the elements you want in the form. Here we want to capture the name and age of a user, so we create a `UserData` object:

```
case class UserData(name: String, age: Int)
```

Now that we have a case class, the next step is to define a `Form` structure. The function of a `Form is to transform form data into a bound instance of a case class, and we define it like follows:

```
val userForm = Form(
  mapping(
    "name" -> text,
    "age" -> number
  )(UserData.apply)(UserData.unapply)
)
```

The Forms object defines the `mapping` method. This method takes the names and constraints of the form, and also takes two functions: an `apply` function and an `unapply`function. Because UserData is a case class, we can plug its `apply` and `unapply`methods directly into the mapping method.

**Note:** Maximum number of fields for a single tuple or mapping is 22 due to the way form handling is implemented. If you have more than 22 fields in your form, you should break down your forms using lists or nested values.

A form will create `UserData` instance with the bound values when given a Map:

```
val anyData = Map("name" -> "bob", "age" -> "21")
val userData = userForm.bind(anyData).get
```

But most of the time you'll use forms from within an Action, with data provided from the request. `Form` contains `bindFromRequest`, which will take a request as an implicit parameter. If you define an implicit request, then `bindFromRequest` will find it.

```
val userData = userForm.bindFromRequest.get
```

**Note:** There is a catch to using `get` here. If the form cannot bind to the data, then `get` will throw an exception. We'll show a safer way of dealing with input in the next few sections.

You are not limited to using case classes in your form mapping. As long as the apply and unapply methods are properly mapped, you can pass in anything you like, such as tuples using the `Forms.tuple` mapping or model case classes. However, there are several advantages to defining a case class specifically for a form:

- **Form specific case classes are convenient.** Case classes are designed to be simple containers of data, and provide out of the box features that are a natural match with`Form` functionality.
- **Form specific case classes are powerful.** Tuples are convenient to use, but do not allow for custom apply or unapply methods, and can only reference contained data by arity (`_1`, `_2`, etc.)
- **Form specific case classes are targeted specifically to the Form.** Reusing model case classes can be convenient, but often models will contain additional domain logic and even persistence details that can lead to tight coupling. In addition, if there is not a direct 1:1 mapping between the form and the model, then sensitive fields must be explicitly ignored to prevent a parameter tampering attack.

## Defining constraints on the form

The `text` constraint considers empty strings to be valid. This means that `name` could be empty here without an error, which is not what we want. A way to ensure that `name` has the appropriate value is to use the `nonEmptyText` constraint.

```
val userFormConstraints2 = Form(
  mapping(
    "name" -> nonEmptyText,
    "age" -> number(min = 0, max = 100)
```

```
  )(UserData.apply)(UserData.unapply)
)
```

Using this form will result in a form with errors if the input to the form does not match the constraints:

```
val boundForm = userFormConstraints2.bind(Map("bob" -> "", "age" -> "25"))
boundForm.hasErrors must beTrue
```

The out of the box constraints are defined on the Forms object:

- `text`: maps to `scala.String`, optionally takes `minLength` and `maxLength`.
- `nonEmptyText`: maps to `scala.String`, optionally takes `minLength` and`maxLength`.
- `number`: maps to `scala.Int`, optionally takes `min`, `max`, and `strict`.
- `longNumber`: maps to `scala.Long`, optionally takes `min`, `max`, and `strict`.
- `bigDecimal`: takes `precision` and `scale`.
- `date`, `sqlDate`, `jodaDate`: maps to `java.util.Date`, `java.sql.Date` and`org.joda.time.DateTime`, optionally takes `pattern` and `timeZone`.
- `jodaLocalDate`: maps to `org.joda.time.LocalDate`, optionally takes `pattern`.
- `email`: maps to `scala.String`, using an email regular expression.
- `boolean`: maps to `scala.Boolean`.
- `checked`: maps to `scala.Boolean`.
- `optional`: maps to `scala.Option`.

## Defining ad-hoc constraints

You can define your own ad-hoc constraints on the case classes using the validation package.

```
val userFormConstraints = Form(
  mapping(
    "name" -> text.verifying(nonEmpty),
    "age" -> number.verifying(min(0), max(100))
  )(UserData.apply)(UserData.unapply)
)
```

You can also define ad-hoc constraints on the case classes themselves:

```
def validate(name: String, age: Int) = {
  name match {
    case "bob" if age >= 18 =>
      Some(UserData(name, age))
    case "admin" =>
      Some(UserData(name, age))
    case _ =>
      None
  }
}

val userFormConstraintsAdHoc = Form(
  mapping(
    "name" -> text,
```

```
    "age" -> number
  )(UserData.apply)(UserData.unapply) verifying("Failed form constraints!", fields => fields match {
    case userData => validate(userData.name, userData.age).isDefined
  })
)
```

You also have the option of constructing your own custom validations. Please see the custom validations section for more details.

## Validating a form in an Action

Now that we have constraints, we can validate the form inside an action, and process the form with errors.

We do this using the `fold` method, which takes two functions: the first is called if the binding fails, and the second is called if the binding succeeds.

```
userForm.bindFromRequest.fold(
  formWithErrors => {
    // binding failure, you retrieve the form containing errors:
    BadRequest(views.html.user(formWithErrors))
  },
  userData => {
    /* binding success, you get the actual value. */
    val newUser = models.User(userData.name, userData.age)
    val id = models.User.create(newUser)
    Redirect(routes.Application.home(id))
  }
)
```

In the failure case, we render the page with BadRequest, and pass in the form *with errors* as a parameter to the page. If we use the view helpers (discussed below), then any errors that are bound to a field will be rendered in the page next to the field.

In the success case, we're sending a `Redirect` with a route to `routes.Application.home` here instead of rendering a view template. This pattern is called Redirect after POST, and is an excellent way to prevent duplicate form submissions.

**Note:** "Redirect after POST" is **required** when using `flashing` or other methods with flash scope, as new cookies will only be available after the redirected HTTP request.

Alternatively, you can use the `parse.form` body parser that binds the content of the request to your form.

```
val userPost = Action(parse.form(userForm)) { implicit request =>
  val userData = request.body
  val newUser = models.User(userData.name, userData.age)
  val id = models.User.create(newUser)
  Redirect(routes.Application.home(id))
}
```

In the failure case, the default behaviour is to return an empty BadRequest response. You can override this behaviour with your own logic. For instance, the following code is completely equivalent to the preceding one using `bindFromRequest` and `fold`.

```
val userPostWithErrors = Action(parse.form(userForm, onErrors = (formWithErrors: Form[UserData]) =>
BadRequest(views.html.user(formWithErrors)))) { implicit request =>
```

```
  val userData = request.body
  val newUser = models.User(userData.name, userData.age)
  val id = models.User.create(newUser)
  Redirect(routes.Application.home(id))
}
```

## Showing forms in a view template

Once you have a form, then you need to make it available to the template engine. You do this by including the form as a parameter to the view template. For `user.scala.html`, the header at the top of the page will look like this:

```
@(userForm: Form[UserData])(implicit messages: Messages)
```

Because `user.scala.html` needs a form passed in, you should pass the empty`userForm` initially when rendering `user.scala.html`:

```
def index = Action {
  Ok(views.html.user(userForm))
}
```

The first thing is to be able to create the form tag. It is a simple view helper that creates aform tag and sets the `action` and `method` tag parameters according to the reverse route you pass in:

```
@helper.form(action = routes.Application.userPost()) {
  @helper.inputText(userForm("name"))
  @helper.inputText(userForm("age"))
}
```

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 displaying errors when a form binding fails.

**Note:** You can use `@import helper._` in the template to avoid prefixing helpers with`@helper.`

There are several input helpers, but the most helpful are:

- `form`: renders a form element.
- `inputText`: renders a text input element.
- `inputPassword`: renders a password input element.
- `inputDate`: renders a date input element.
- `inputFile`: renders a file input element.
- `inputRadioGroup`: renders a radio input element.
- `select`: renders a select element.
- `textarea`: renders a textarea element.
- `checkbox`: renders a checkbox element.
- `input`: renders a generic input element (which requires explicit arguments).

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

```
@helper.inputText(userForm("name"), 'id -> "name", 'size -> 30)
```

The generic `input` helper mentioned above will let you code the desired HTML result:

```
@helper.input(userForm("name")) { (id, name, value, args) =>
```

```
    <input type="text" name="@name" id="@id" @toHtmlArgs(args)>
}
```

**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.

For complex form elements, you can also create your own custom view helpers (using scala classes in the `views` package) and custom field constructors.

## Displaying errors in a view template

The errors in a form take the form of `Map[String,FormError]` where `FormError` has:

- `key`: should be the same as the field.
- `message`: a message or a message key.
- `args`: a list of arguments to the message.

The form errors are accessed on the bound form instance as follows:

- `errors`: returns all errors as `Seq[FormError]`.
- `globalErrors`: returns errors without a key as `Seq[FormError]`.
- `error("name")`: returns the first error bound to key as `Option[FormError]`.
- `errors("name")`: returns all errors bound to key as `Seq[FormError]`.

Errors attached to a field will render automatically using the form helpers, so `@helper.inputText` with errors can display as follows:

```
<dl class="error" id="age_field">
    <dt><label for="age">Age:</label></dt>
    <dd><input type="text" name="age" id="age" 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>
```

Global errors that are not bound to a key do not have a helper and must be defined explicitly in the page:

```
@if(userForm.hasGlobalErrors) {
  <ul>
  @for(error <- userForm.globalErrors) {
    <li>@error.message</li>
  }
  </ul>
}
```

## Mapping with tuples

You can use tuples instead of case classes in your fields:

```
val userFormTuple = Form(
  tuple(
    "name" -> text,
    "age" -> number
```

```
  ) // tuples come with built-in apply/unapply
)
```

Using a tuple can be more convenient than defining a case class, especially for low arity tuples:

```
val anyData = Map("name" -> "bob", "age" -> "25")
val (name, age) = userFormTuple.bind(anyData).get
```

## Mapping with single

Tuples are only possible when there are multiple values. If there is only one field in the form, use `Forms.single` to map to a single value without the overhead of a case class or tuple:

```
val singleForm = Form(
  single(
    "email" -> email
  )
)

val emailValue = singleForm.bind(Map("email" -> "bob@example.com")).get
```

## Fill values

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

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

When you use this with a view helper, the value of the element will be filled with the value:

```
@helper.inputText(filledForm("name")) @* will render value="Bob" *@
```

Fill is especially helpful for helpers that need lists or maps of values, such as the `select` and `inputRadioGroup` helpers. Use `options` to value these helpers with lists, maps and pairs.

## Nested values

A form mapping can define nested values by using `Forms.mapping` inside an existing mapping:

```
case class AddressData(street: String, city: String)

case class UserAddressData(name: String, address: AddressData)
val userFormNested: Form[UserAddressData] = Form(
  mapping(
    "name" -> text,
    "address" -> mapping(
      "street" -> text,
      "city" -> text
    )(AddressData.apply)(AddressData.unapply)
  )(UserAddressData.apply)(UserAddressData.unapply)
)
```

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

```
@helper.inputText(userFormNested("name"))
@helper.inputText(userFormNested("address.street"))
@helper.inputText(userFormNested("address.city"))
```

## Repeated values

A form mapping can define repeated values using `Forms.list` or `Forms.seq`:

```
case class UserListData(name: String, emails: List[String])
val userFormRepeated = Form(
  mapping(
    "name" -> text,
    "emails" -> list(email)
  )(UserListData.apply)(UserListData.unapply)
)
```

When you are using repeated data like this, there are two alternatives for sending the form values in the HTTP request. First, you can suffix the parameter with an empty bracket pair, as in “emails[]”. This parameter can then be repeated in the standard way, as in `http://foo.com/request?emails[]=a@b.com&emails[]=c@d.com`.

Alternatively, the client can explicitly name the parameters uniquely with array subscripts, as in `emails[0]`, `emails[1]`, `emails[2]`, and so on. This approach also allows you to maintain the order of a sequence of inputs.

If you are using Play to generate your form HTML, you can generate as many inputs for the `emails` field as the form contains, using the `repeat` helper:

```
@helper.inputText(myForm("name"))
@helper.repeat(myForm("emails"), min = 1) { emailField =>
    @helper.inputText(emailField)
}
```

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

## Optional values

A form mapping can also define optional values using `Forms.optional`:

```
case class UserOptionalData(name: String, email: Option[String])
val userFormOptional = Form(
  mapping(
    "name" -> text,
    "email" -> optional(email)
  )(UserOptionalData.apply)(UserOptionalData.unapply)
)
```

This maps to an `Option[A]` in output, which is `None` if no form value is found.

## Default values

You can populate a form with initial values using `Form#fill`:

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

Or you can define a default mapping on the number using `Forms.default`:

```
Form(
  mapping(
    "name" -> default(text, "Bob")
    "age" -> default(number, 18)
  )(User.apply)(User.unapply)
)
```

## Ignored values

If you want a form to have a static value for a field, use `Forms.ignored`:

```
val userFormStatic = Form(
  mapping(
    "id" -> ignored(23L),
    "name" -> text,
    "email" -> optional(email)
  )(UserStaticData.apply)(UserStaticData.unapply)
)
```

# Putting it all together

Here's an example of what a model and controller would look like for managing an entity.

Given the case class `Contact`:

```
case class Contact(firstname: String,
                   lastname: String,
                   company: Option[String],
                   informations: Seq[ContactInformation])

object Contact {
  def save(contact: Contact): Int = 99
}

case class ContactInformation(label: String,
                              email: Option[String],
                              phones: List[String])
```

Note that `Contact` contains a `Seq` with `ContactInformation` elements and a `List` of`String`. In this case, we can combine the nested mapping with repeated mappings (defined with `Forms.seq` and `Forms.list`, respectively).

```
val contactForm: Form[Contact] = Form(

  // Defines a mapping that will handle Contact values
  mapping(
    "firstname" -> nonEmptyText,
    "lastname" -> nonEmptyText,
    "company" -> optional(text),

    // Defines a repeated mapping
    "informations" -> seq(
      mapping(
        "label" -> nonEmptyText,
        "email" -> optional(email),
        "phones" -> list(
          text verifying pattern("""[0-9.+]+""".r, error="A valid phone number is required")
        )
      )(ContactInformation.apply)(ContactInformation.unapply)
    )
  )(Contact.apply)(Contact.unapply)
```

```
)
```

And this code shows how an existing contact is displayed in the form using filled data:

```
def editContact = Action {
  val existingContact = Contact(
    "Fake", "Contact", Some("Fake company"), informations = List(
      ContactInformation(
        "Personal", Some("fakecontact@gmail.com"), List("01.23.45.67.89", "98.76.54.32.10")
      ),
      ContactInformation(
        "Professional", Some("fakecontact@company.com"), List("01.23.45.67.89")
      ),
      ContactInformation(
        "Previous", Some("fakecontact@oldcompany.com"), List()
      )
    )
  )
  Ok(views.html.contact.form(contactForm.fill(existingContact)))
}
```

Finally, this is what a form submission handler would look like:

```
def saveContact = Action { implicit request =>
  contactForm.bindFromRequest.fold(
    formWithErrors => {
      BadRequest(views.html.contact.form(formWithErrors))
    },
    contact => {
      val contactId = Contact.save(contact)
      Redirect(routes.Application.showContact(contactId)).flashing("success" -> "Contact saved!")
    }
  )
}
```

**Next:** Protecting against CSRF

# Protecting against Cross Site Request Forgery

Cross Site Request Forgery (CSRF) is a security exploit where an attacker tricks a victims browser into making a request using the victims session. Since the session token is sent with every request, if an attacker can coerce the victims browser to make a request on their behalf, the attacker can make requests on the users behalf.

It is recommended that you familiarise yourself with CSRF, what the attack vectors are, and what the attack vectors are not. We recommend starting with this information from OWASP.

Simply put, an attacker can coerce a victims browser to make the following types of requests:

- All `GET` requests
- `POST` requests with bodies of type `application/x-www-form-urlencoded`,`multipart/form-data` and `text/plain`

An attacker can not:

- Coerce the browser to use other request methods such as `PUT` and `DELETE`
- Coerce the browser to post other content types, such as `application/json`
- Coerce the browser to send new cookies, other than those that the server has already set
- Coerce the browser to set arbitrary headers, other than the normal headers the browser adds to requests

Since `GET` requests are not meant to be mutative, there is no danger to an application that follows this best practice. So the only requests that need CSRF protection are `POST`requests with the above mentioned content types.

## Play's CSRF protection

Play supports multiple methods for verifying that a request is not a CSRF request. The primary mechanism is a CSRF token. This token gets placed either in the query string or body of every form submitted, and also gets placed in the users session. Play then verifies that both tokens are present and match.

To allow simple protection for non browser requests, such as requests made through AJAX, Play also supports the following:

- If an `X-Requested-With` header is present, Play will consider the request safe. `X-Requested-With` is added to requests by many popular Javascript libraries, such as jQuery.
- If a `Csrf-Token` header with value `nocheck` is present, or with a valid CSRF token, Play will consider the request safe.

# Applying a global CSRF filter

Play provides a global CSRF filter that can be applied to all requests. This is the simplest way to add CSRF protection to an application. To enable the global filter, add the Play filters helpers dependency to your project in `build.sbt`:

```
libraryDependencies += filters
```

Now add them to your `Filters` class as described in HTTP filters:

```
import play.api.http.HttpFilters
import play.filters.csrf.CSRFFilter
```

```
import javax.inject.Inject

class Filters @Inject() (csrfFilter: CSRFFilter) extends HttpFilters {
  def filters = Seq(csrfFilter)
}
```

The `Filters` class can either be in the root package, or if it has another name or is in another package, needs to be configured using `play.http.filters` in `application.conf`:

```
play.http.filters = "filters.MyFilters"
```

## Getting the current token

The current CSRF token can be accessed using the `getToken` method. It takes an implicit `RequestHeader`, so ensure that one is in scope.

```
import play.filters.csrf.CSRF

val token = CSRF.getToken(request)
```

To help in adding CSRF tokens to forms, Play provides some template helpers. The first one adds it to the query string of the action URL:

```
@import helper._

@form(CSRF(routes.ItemsController.save())) {
   ...
}
```

This might render a form that looks like this:

```
<form method="POST" action="/items?csrfToken=1234567890abcdef">
  ...
</form>
```

If it is undesirable to have the token in the query string, Play also provides a helper for adding the CSRF token as hidden field in the form:

```
@form(routes.ItemsController.save()) {
   @CSRF.formField
   ...
}
```

This might render a form that looks like this:

```
<form method="POST" action="/items">
  <input type="hidden" name="csrfToken" value="1234567890abcdef"/>
  ...
</form>
```

The form helper methods all require an implicit token or request to be available in scope. This will typically be provided by adding an implicit `RequestHeader` parameter to your template, if it doesn't have one already.

## Adding a CSRF token to the session

To ensure that a CSRF token is available to be rendered in forms, and sent back to the client, the global filter will generate a new token for all GET requests that accept HTML, if a token isn't already available in the incoming request.

# Applying CSRF filtering on a per action basis

Sometimes global CSRF filtering may not be appropriate, for example in situations where an application might want to allow some cross origin form posts. Some non session based standards, such as OpenID 2.0, require the use of cross site form posting, or use form submission in server to server RPC communications.

In these cases, Play provides two actions that can be composed with your applications actions.

The first action is the `CSRFCheck` action, and it performs the check. It should be added to all actions that accept session authenticated POST form submissions:

```
import play.api.mvc._
import play.filters.csrf._

def save = CSRFCheck {
  Action { req =>
    // handle body
    Ok
  }
}
```

The second action is the `CSRFAddToken` action, it generates a CSRF token if not already present on the incoming request. It should be added to all actions that render forms:

```
import play.api.mvc._
import play.filters.csrf._

def form = CSRFAddToken {
  Action { implicit req =>
    Ok(views.html.itemsForm())
  }
}
```

A more convenient way to apply these actions is to use them in combination with Play's action composition:

```
import play.api.mvc._
import play.filters.csrf._
```

```
object PostAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) = CSRFCheck(action)
}

object GetAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    // authentication code here
    block(request)
  }
  override def composeAction[A](action: Action[A]) = CSRFAddToken(action)
}
```

Then you can minimise the boiler plate code necessary to write actions:

```
def save = PostAction {
  // handle body
  Ok
}

def form = GetAction { implicit req =>
  Ok(views.html.itemsForm())
}
```

# CSRF configuration options

The full range of CSRF configuration options can be found in the filters reference.conf. Some examples include:

- `play.filters.csrf.token.name` - The name of the token to use both in the session and in the request body/query string. Defaults to `csrfToken`.
- `play.filters.csrf.cookie.name` - If configured, Play will store the CSRF token in a cookie with the given name, instead of in the session.
- `play.filters.csrf.cookie.secure` - If `play.filters.csrf.cookie.name` is set, whether the CSRF cookie should have the secure flag set. Defaults to the same value as `play.http.session.secure`.
- `play.filters.csrf.body.bufferSize` - In order to read tokens out of the body, Play must first buffer the body and potentially parse it. This sets the maximum buffer size that will be used to buffer the body. Defaults to 100k.
- `play.filters.csrf.token.sign` - Whether Play should use signed CSRF tokens. Signed CSRF tokens ensure that the token value is randomised per request, thus defeating BREACH style attacks.

**Next:** Custom Validations

# Using Custom Validations

The validation package allows you to create ad-hoc constraints using the `verifying` method. However, Play gives you the option of creating your own custom constraints, using the `Constraint` case class.

Here, we'll implement a simple password strength constraint that uses regular expressions to check the password is not all letters or all numbers. A `Constraint` takes a function which returns a `ValidationResult`, and we use that function to return the results of the password check:

```
val allNumbers = """\d*""".r
val allLetters = """[A-Za-z]*""".r

val passwordCheckConstraint: Constraint[String] = Constraint("constraints.passwordcheck")({
  plainText =>
    val errors = plainText match {
      case allNumbers() => Seq(ValidationError("Password is all numbers"))
      case allLetters() => Seq(ValidationError("Password is all letters"))
      case _ => Nil
    }
    if (errors.isEmpty) {
      Valid
    } else {
      Invalid(errors)
    }
})
```

**Note:** This is an intentionally trivial example. Please consider using the OWASP guidefor proper password security.

We can then use this constraint together with `Constraints.min` to add additional checks on the password.

```
val passwordCheck: Mapping[String] = nonEmptyText(minLength = 10)
  .verifying(passwordCheckConstraint)
```

**Next:** Custom Field Constructors

# Custom 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">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
```

# 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 {
  import views.html.helper.FieldConstructor
  implicit val myFields = FieldConstructor(html.myFieldConstructorTemplate.f)
}
```

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

```
@import MyHelpers._
@helper.inputText(myForm("username"))
```

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

You can also set an implicit value for your `FieldConstructor` inline:

```
@implicitField = @{ helper.FieldConstructor(myFieldConstructorTemplate.f) }
@helper.inputText(myForm("username"))
```

**Next:** Working with Json

# JSON basics

Modern web applications often need to parse and generate data in the JSON (JavaScript Object Notation) format. Play supports this via its JSON library.

JSON is a lightweight data-interchange format and looks like this:

```
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
```

To learn more about JSON, see json.org.

# The Play JSON library

The `play.api.libs.json` package contains data structures for representing JSON data and utilities for converting between these data structures and other data representations. Types of interest are:

`JsValue`

This is a trait representing any JSON value. The JSON library has a case class extending `JsValue` to represent each valid JSON type:

- `JsString`
- `JsNumber`
- `JsBoolean`
- `JsObject`
- `JsArray`

- `JsNull`

  Using the various `JsValue` types, you can construct a representation of any JSON structure.

  ### `Json`

  The `Json` object provides utilities, primarily for conversion to and from `JsValue`structures.

  ### `JsPath`

  Represents a path into a `JsValue` structure, analogous to XPath for XML. This is used for traversing `JsValue` structures and in patterns for implicit converters.

# Converting to a `JsValue`

## Using string parsing

```
import play.api.libs.json._

val json: JsValue = Json.parse("""
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
```

## Using class construction

```
import play.api.libs.json._

val json: JsValue = JsObject(Seq(
  "name" -> JsString("Watership Down"),
  "location" -> JsObject(Seq("lat" -> JsNumber(51.235685), "long" -> JsNumber(-1.309197))),
  "residents" -> JsArray(Seq(
    JsObject(Seq(
      "name" -> JsString("Fiver"),
      "age" -> JsNumber(4),
      "role" -> JsNull
    )),
    JsObject(Seq(
      "name" -> JsString("Bigwig"),
      "age" -> JsNumber(6),
      "role" -> JsString("Owsla")
```

```
   ))
  ))
))
```

`Json.obj` and `Json.arr` can simplify construction a bit. Note that most values don't need to be explicitly wrapped by JsValue classes, the factory methods use implicit conversion (more on this below).

```
import play.api.libs.json.{JsNull,Json,JsString,JsValue}

val json: JsValue = Json.obj(
  "name" -> "Watership Down",
  "location" -> Json.obj("lat" -> 51.235685, "long" -> -1.309197),
  "residents" -> Json.arr(
    Json.obj(
      "name" -> "Fiver",
      "age" -> 4,
      "role" -> JsNull
    ),
    Json.obj(
      "name" -> "Bigwig",
      "age" -> 6,
      "role" -> "Owsla"
    )
  )
)
```

## Using Writes converters

Scala to `JsValue` conversion is performed by the utility method `Json.toJson[T](T)(implicit writes: Writes[T])`. This functionality depends on a converter of type`Writes[T]` which can convert a `T` to a `JsValue`.

The Play JSON API provides implicit `Writes` for most basic types, such as `Int`,`Double`, `String`, and `Boolean`. It also supports `Writes` for collections of any type `T`that a `Writes[T]` exists.

```
import play.api.libs.json._

// basic types
val jsonString = Json.toJson("Fiver")
val jsonNumber = Json.toJson(4)
val jsonBoolean = Json.toJson(false)

// collections of basic types
val jsonArrayOfInts = Json.toJson(Seq(1, 2, 3, 4))
val jsonArrayOfStrings = Json.toJson(List("Fiver", "Bigwig"))
```

To convert your own models to `JsValue`s, you must define implicit `Writes` converters and provide them in scope.

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
import play.api.libs.json._

implicit val locationWrites = new Writes[Location] {
```

```
  def writes(location: Location) = Json.obj(
    "lat" -> location.lat,
    "long" -> location.long
  )
}

implicit val residentWrites = new Writes[Resident] {
  def writes(resident: Resident) = Json.obj(
    "name" -> resident.name,
    "age" -> resident.age,
    "role" -> resident.role
  )
}

implicit val placeWrites = new Writes[Place] {
  def writes(place: Place) = Json.obj(
    "name" -> place.name,
    "location" -> place.location,
    "residents" -> place.residents)
}

val place = Place(
  "Watership Down",
  Location(51.235685, -1.309197),
  Seq(
    Resident("Fiver", 4, None),
    Resident("Bigwig", 6, Some("Owsla"))
  )
)

val json = Json.toJson(place)
```

Alternatively, you can define your `Writes` using the combinator pattern:

Note: The combinator pattern is covered in detail in JSON Reads/Writes/Formats Combinators.

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val residentWrites: Writes[Resident] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "age").write[Int] and
  (JsPath \ "role").writeNullable[String]
)(unlift(Resident.unapply))

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location] and
  (JsPath \ "residents").write[Seq[Resident]]
)(unlift(Place.unapply))
```

# Traversing a JsValue structure

You can traverse a `JsValue` structure and extract specific values. The syntax and functionality is similar to Scala XML processing.

Note: The following examples are applied to the JsValue structure created in previous examples.

## Simple path `\`

Applying the `\` operator to a `JsValue` will return the property corresponding to the field argument, supposing this is a `JsObject`.

```
val lat = (json \ "location" \ "lat").get
// returns JsNumber(51.235685)
```

## Recursive path `\\`

Applying the `\\` operator will do a lookup for the field in the current object and all descendants.

```
val names = json \\ "name"
// returns Seq(JsString("Watership Down"), JsString("Fiver"), JsString("Bigwig"))
```

## Index lookup (for JsArrays)

You can retrieve a value in a `JsArray` using an apply operator with the index number.

```
val bigwig = (json \ "residents")(1)
// returns {"name":"Bigwig","age":6,"role":"Owsla"}
```

# Converting from a JsValue

## Using String utilities

Minified:

```
val minifiedString: String = Json.stringify(json)
{"name":"Watership Down","location":{"lat":51.235685,"long":-
1.309197},"residents":[{"name":"Fiver","age":4,"role":null},{"name":"Bigwig","age":6,"role":"Owsla"}]}
```

Readable:

```
val readableString: String = Json.prettyPrint(json)
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
```

```
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
```

## Using JsValue.as/asOpt

The simplest way to convert a `JsValue` to another type is using `JsValue.as[T](implicit fjs: Reads[T]): T`. This requires an implicit converter of type `Reads[T]` to convert a `JsValue` to `T` (the inverse of `Writes[T]`). As with `Writes`, the JSON API provides `Reads` for basic types.

```
val name = (json \ "name").as[String]
// "Watership Down"

val names = (json \\ "name").map(_.as[String])
// Seq("Watership Down", "Fiver", "Bigwig")
```

The `as` method will throw a `JsResultException` if the path is not found or the conversion is not possible. A safer method is `JsValue.asOpt[T](implicit fjs: Reads[T]): Option[T]`.

```
val nameOption = (json \ "name").asOpt[String]
// Some("Watership Down")

val bogusOption = (json \ "bogus").asOpt[String]
// None
```

Although the `asOpt` method is safer, any error information is lost.

## Using validation

The preferred way to convert from a `JsValue` to another type is by using its `validate`method (which takes an argument of type `Reads`). This performs both validation and conversion, returning a type of `JsResult`. `JsResult` is implemented by two classes:

- `JsSuccess`: Represents a successful validation/conversion and wraps the result.
- `JsError`: Represents unsuccessful validation/conversion and contains a list of validation errors.

You can apply various patterns for handling a validation result:

```
val json = { ... }

val nameResult: JsResult[String] = (json \ "name").validate[String]

// Pattern matching
nameResult match {
  case s: JsSuccess[String] => println("Name: " + s.get)
  case e: JsError => println("Errors: " + JsError.toFlatJson(e).toString())
}

// Fallback value
val nameOrFallback = nameResult.getOrElse("Undefined")
```

```
// map
val nameUpperResult: JsResult[String] = nameResult.map(_.toUpperCase())

// fold
val nameOption: Option[String] = nameResult.fold(
  invalid = {
    fieldErrors => fieldErrors.foreach(x => {
      println("field: " + x._1 + ", errors: " + x._2)
    })
    None
  },
  valid = {
    name => Some(name)
  }
)
```

## JsValue to a model

To convert from JsValue to a model, you must define implicit `Reads[T]` where `T` is the type of your model.

Note: The pattern used to implement `Reads` and custom validation are covered in detail in JSON Reads/Writes/Formats Combinators.

```
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)

implicit val residentReads: Reads[Resident] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "age").read[Int] and
  (JsPath \ "role").readNullable[String]
)(Resident.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location] and
  (JsPath \ "residents").read[Seq[Resident]]
)(Place.apply _)


val json = { ... }

val placeResult: JsResult[Place] = json.validate[Place]
// JsSuccess(Place(...),)

val residentResult: JsResult[Resident] = (json \ "residents")(1).validate[Resident]
// JsSuccess(Resident(Bigwig,6,Some(Owsla)),)
```

**Next:** JSON with HTTP

# JSON with HTTP

Play supports HTTP requests and responses with a content type of JSON by using the HTTP API in combination with the JSON library.

See HTTP Programming for details on Controllers, Actions, and routing.

We'll demonstrate the necessary concepts by designing a simple RESTful web service to GET a list of entities and accept POSTs to create new entities. The service will use a content type of JSON for all data.

Here's the model we'll use for our service:

```
case class Location(lat: Double, long: Double)

case class Place(name: String, location: Location)

object Place {

  var list: List[Place] = {
    List(
      Place(
        "Sandleford",
        Location(51.377797, -1.318965)
      ),
      Place(
        "Watership Down",
        Location(51.235685, -1.309197)
      )
    )
  }

  def save(place: Place) = {
    list = list ::: List(place)
  }
}
```

# Serving a list of entities in JSON

We'll start by adding the necessary imports to our controller.

```
import play.api.mvc._
import play.api.libs.json._
```

```
import play.api.libs.functional.syntax._

object Application extends Controller {

}
```

Before we write our `Action`, we'll need the plumbing for doing conversion from our model to a `JsValue` representation. This is accomplished by defining an implicit`Writes[Place]`.

```
implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location]
)(unlift(Place.unapply))
```

Next we write our `Action`:

```
def listPlaces = Action {
  val json = Json.toJson(Place.list)
  Ok(json)
}
```

The `Action` retrieves a list of `Place` objects, converts them to a `JsValue` using`Json.toJson` with our implicit `Writes[Place]`, and returns this as the body of the result. Play will recognize the result as JSON and set the appropriate `Content-Type`header and body value for the response.

The last step is to add a route for our `Action` in `conf/routes`:

```
GET   /places             controllers.Application.listPlaces
```

We can test the action by making a request with a browser or HTTP tool. This example uses the unix command line tool cURL.

```
curl --include http://localhost:9000/places
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 141

[{"name":"Sandleford","location":{"lat":51.377797,"long":-1.318965}},{"name":"Watership
Down","location":{"lat":51.235685,"long":-1.309197}}]
```

# Creating a new entity instance in JSON

For this `Action` we'll need to define an implicit `Reads[Place]` to convert a `JsValue` to our model.

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location]
)(Place.apply _)
```

Next we'll define the `Action`.

```
def savePlace = Action(BodyParsers.parse.json) { request =>
  val placeResult = request.body.validate[Place]
  placeResult.fold(
    errors => {
      BadRequest(Json.obj("status" ->"KO", "message" -> JsError.toFlatJson(errors)))
    },
    place => {
      Place.save(place)
      Ok(Json.obj("status" ->"OK", "message" -> ("Place '"+place.name+"' saved.") ))
    }
  )
}
```

This `Action` is more complicated than our list case. Some things to note:

- This `Action` expects a request with a `Content-Type` header of `text/json` or `application/json` and a body containing a JSON representation of the entity to create.
- It uses a JSON specific `BodyParser` which will parse the request and provide `request.body` as a `JsValue`.
- We used the `validate` method for conversion which will rely on our implicit `Reads[Place]`.
- To process the validation result, we used a `fold` with error and success flows. This pattern may be familiar as it is also used for form submission.
- The `Action` also sends JSON responses.

Finally we'll add a route binding in `conf/routes`:

```
POST  /places             controllers.Application.savePlace
```

We'll test this action with valid and invalid requests to verify our success and error flows.

Testing the action with a valid data:

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"name":"Nuthanger Farm","location":{"lat" : 51.244031,"long" : -1.263224}}'
  http://localhost:9000/places
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 57
```

```
{"status":"OK","message":"Place 'Nuthanger Farm' saved."}
```

Testing the action with a invalid data, missing “name” field:

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"location":{"lat" : 51.244031,"long" : -1.263224}}'
  http://localhost:9000/places
```

Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 79

{"status":"KO","message":{"obj.name":[{"msg":"error.path.missing","args":[]}]}}
```

Testing the action with a invalid data, wrong data type for “lat”:

```
curl --include
  --request POST
  --header "Content-type: application/json"
  --data '{"name":"Nuthanger Farm","location":{"lat" : "xxx","long" : -1.263224}}'
  http://localhost:9000/places
```

Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 92

{"status":"KO","message":{"obj.location.lat":[{"msg":"error.expected.jsnumber","args":[]}]}}
```

# Summary

Play is designed to support REST with JSON and developing these services should hopefully be straightforward. The bulk of the work is in writing `Reads` and `Writes` for your model, which is covered in detail in the next section.

**Next:** JSON Reads/Writes/Format Combinators

# JSON Reads/Writes/Format Combinators

JSON basics introduced `Reads` and `Writes` converters which are used to convert between `JsValue` structures and other data types. This page covers in greater detail how to build these converters and how to use validation during conversion.
The examples on this page will use this `JsValue` structure and corresponding model:

```
import play.api.libs.json._

val json: JsValue = Json.parse("""
{
  "name" : "Watership Down",
  "location" : {
    "lat" : 51.235685,
    "long" : -1.309197
  },
  "residents" : [ {
    "name" : "Fiver",
    "age" : 4,
    "role" : null
  }, {
    "name" : "Bigwig",
    "age" : 6,
    "role" : "Owsla"
  } ]
}
""")
case class Location(lat: Double, long: Double)
case class Resident(name: String, age: Int, role: Option[String])
case class Place(name: String, location: Location, residents: Seq[Resident])
```

# JsPath

`JsPath` is a core building block for creating `Reads`/`Writes`. `JsPath` represents the location of data in a `JsValue` structure. You can use the `JsPath` object (root path) to define a `JsPath` child instance by using syntax similar to traversing `JsValue`:

```
import play.api.libs.json._

val json = { ... }

// Simple path
val latPath = JsPath \ "location" \ "lat"

// Recursive path
val namesPath = JsPath \\ "name"

// Indexed path
val firstResidentPath = (JsPath \ "residents")(0)
```

The `play.api.libs.json` package defines an alias for `JsPath`: `__` (double underscore). You can use this if you prefer:

```
val longPath = __ \ "location" \ "long"
```

# Reads

`Reads` converters are used to convert from a `JsValue` to another type. You can combine and nest `Reads` to create more complex `Reads`.

You will require these imports to create `Reads`:

```
import play.api.libs.json._ // JSON library
import play.api.libs.json.Reads._ // Custom validation helpers
import play.api.libs.functional.syntax._ // Combinator syntax
```

## Path Reads

`JsPath` has methods to create special `Reads` that apply another `Reads` to a `JsValue` at a specified path:

- `JsPath.read[T](implicit r: Reads[T]): Reads[T]` - Creates a `Reads[T]` that will apply the implicit argument `r` to the `JsValue` at this path.
- `JsPath.readNullable[T](implicit r: Reads[T]): Reads[Option[T]]readNullable` - Use for paths that may be missing or can contain a null value.

Note: The JSON library provides implicit `Reads` for basic types such as `String`, `Int`,`Double`, etc.

Defining an individual path `Reads` looks like this:

```
val nameReads: Reads[String] = (JsPath \ "name").read[String]
```

## Complex Reads

You can combine individual path `Reads` to form more complex `Reads` which can be used to convert to complex models.

For easier understanding, we'll break down the combine functionality into two statements. First combine `Reads` objects using the `and` combinator:

```
val locationReadsBuilder =
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
```

This will yield a type of `FunctionalBuilder[Reads]#CanBuild2[Double, Double]`. This is an intermediary object and you don't need to worry too much about it, just know that it's used to create a complex `Reads`.

Second call the `apply` method of `CanBuildX` with a function to translate individual values to your model, this will return your complex `Reads`. If you have a case class with a matching constructor signature, you can just use its `apply` method:

```
implicit val locationReads = locationReadsBuilder.apply(Location.apply _)
```

Here's the same code in a single statement:

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)
```

## Validation with Reads

The `JsValue.validate` method was introduced in JSON basics as the preferred way to perform validation and conversion from a `JsValue` to another type. Here's the basic pattern:

```
val json = { ... }

val nameReads: Reads[String] = (JsPath \ "name").read[String]

val nameResult: JsResult[String] = json.validate[String](nameReads)

nameResult match {
  case s: JsSuccess[String] => println("Name: " + s.get)
  case e: JsError => println("Errors: " + JsError.toFlatJson(e).toString())
}
```

Default validation for `Reads` is minimal, such as checking for type conversion errors. You can define custom validation rules by using `Reads` validation helpers. Here are some that are commonly used:

- `Reads.email` - Validates a String has email format.
- `Reads.minLength(nb)` - Validates the minimum length of a String.
- `Reads.min` - Validates a minimum numeric value.
- `Reads.max` - Validates a maximum numeric value.
- `Reads[A] keepAnd Reads[B] => Reads[A]` - Operator that tries `Reads[A]` and`Reads[B]` but only keeps the result of `Reads[A]` (For those who know Scala parser combinators `keepAnd == <~` ).
- `Reads[A] andKeep Reads[B] => Reads[B]` - Operator that tries `Reads[A]` and`Reads[B]` but only keeps the result of `Reads[B]` (For those who know Scala parser combinators `andKeep == ~>` ).
- `Reads[A] or Reads[B] => Reads` - Operator that performs a logical OR and keeps the result of the last `Reads` checked.

To add validation, apply helpers as arguments to the `JsPath.read` method:

```
val improvedNameReads =
  (JsPath \ "name").read[String](minLength[String](2))
```

## Putting it all together

By using complex `Reads` and custom validation we can define a set of effective `Reads`for our example model and apply them:

```
import play.api.libs.json._
import play.api.libs.json.Reads._
import play.api.libs.functional.syntax._

implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply _)

implicit val residentReads: Reads[Resident] = (
  (JsPath \ "name").read[String](minLength[String](2)) and
  (JsPath \ "age").read[Int](min(0) keepAnd max(150)) and
```

```
  (JsPath \ "role").readNullable[String]
)(Resident.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String](minLength[String](2)) and
  (JsPath \ "location").read[Location] and
  (JsPath \ "residents").read[Seq[Resident]]
)(Place.apply _)


val json = { ... }

json.validate[Place] match {
  case s: JsSuccess[Place] => {
    val place: Place = s.get
    // do something with place
  }
  case e: JsError => {
    // error handling flow
  }
}
```

Note that complex `Reads` can be nested. In this case, `placeReads` uses the previously defined implicit `locationReads` and `residentReads` at specific paths in the structure.

# Writes

`Writes` converters are used to convert from some type to a `JsValue`.
You can build complex `Writes` using `JsPath` and combinators very similar to `Reads`.
Here's the `Writes` for our example model:

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val residentWrites: Writes[Resident] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "age").write[Int] and
  (JsPath \ "role").writeNullable[String]
)(unlift(Resident.unapply))

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location] and
  (JsPath \ "residents").write[Seq[Resident]]
)(unlift(Place.unapply))


val place = Place(
```

```
  "Watership Down",
  Location(51.235685, -1.309197),
  Seq(
    Resident("Fiver", 4, None),
    Resident("Bigwig", 6, Some("Owsla"))
  )
)

val json = Json.toJson(place)
```

There are a few differences between complex `Writes` and `Reads`:

- The individual path `Writes` are created using the `JsPath.write` method.
- There is no validation on conversion to `JsValue` which makes the structure simpler and you won't need any validation helpers.
- The intermediary `FunctionalBuilder#CanBuildX` (created by `and` combinators) takes a function that translates a complex type `T` to a tuple matching the individual path`Writes`. Although this is symmetrical to the `Reads` case, the `unapply` method of a case class returns an `Option` of a tuple of properties and must be used with `unlift`to extract the tuple.

# Recursive Types

One special case that our example model doesn't demonstrate is how to handle `Reads`and `Writes` for recursive types. `JsPath` provides `lazyRead` and `lazyWrite` methods that take call-by-name parameters to handle this:

```
case class User(name: String, friends: Seq[User])

implicit lazy val userReads: Reads[User] = (
  (__ \ "name").read[String] and
  (__ \ "friends").lazyRead(Reads.seq[User](userReads))
)(User)

implicit lazy val userWrites: Writes[User] = (
  (__ \ "name").write[String] and
  (__ \ "friends").lazyWrite(Writes.seq[User](userWrites))
)(unlift(User.unapply))
```

# Format

`Format[T]` is just a mix of the `Reads` and `Writes` traits and can be used for implicit conversion in place of its components.

## Creating Format from Reads and Writes

You can define a `Format` by constructing it from `Reads` and `Writes` of the same type:

```
val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").read[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply _)
```

```
val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val locationFormat: Format[Location] =
  Format(locationReads, locationWrites)
```

### Creating Format using combinators

In the case where your `Reads` and `Writes` are symmetrical (which may not be the case in real applications), you can define a `Format` directly from combinators:

```
implicit val locationFormat: Format[Location] = (
  (JsPath \ "lat").format[Double](min(-90.0) keepAnd max(90.0)) and
  (JsPath \ "long").format[Double](min(-180.0) keepAnd max(180.0))
)(Location.apply, unlift(Location.unapply))
```

**Next:** JSON Transformers

# JSON transformers

Please note this documentation was initially published as an article by Pascal Voitot (@mandubian) on mandubian.com

Now you should know how to validate JSON and convert into any structure you can write in Scala and back to JSON. But as soon as I've begun to use those combinators to write web applications, I almost immediately encountered a case : read JSON from network, validate it and convert it into… JSON.

---

# Introducing JSON *coast-to-coast* design

### Are we doomed to convert JSON to OO?

For a few years now, in almost all web frameworks (except recent JavaScript server side stuff maybe in which JSON is the default data structure), we have been used to get JSON from network and **convert JSON (or even POST/GET data) into OO structures**such as classes (or case classes in Scala). Why?

- For a good reason : **OO structures are "language-native"** and allows **manipulating data with respect to your business logic** in a seamless way while ensuring isolation of business logic from web layers.
- For a more questionable reason : **ORM frameworks talk to DB only with OO structures** and we have (kind of) convinced ourselves that it was impossible to do else… with the well-known good & bad features of ORMs… (not here to criticize those stuff)

## Is OO conversion really the default use case?

In many cases, you don't really need to perform any real business logic with data but validating/transforming before storing or after extracting. Let's take the CRUD case:

- You just get the data from the network, validate them a bit and insert/update into DB.
- In the other way, you just retrieve data from DB and send them outside.

So, generally, for CRUD ops, you convert JSON into a OO structure just because the frameworks are only able to speak OO.

I don't say or pretend you shouldn't use JSON to OO conversion but maybe this is not the most common case and we should keep conversion to OO only when we have real business logic to fulfill.

## New tech players change the way of manipulating JSON

Besides this fact, we have some new DB types such as MongoDB (or CouchDB) accepting document structured data looking almost like JSON trees (_isn't BSON, Binary JSON?_).

With these DB types, we also have new great tools such as ReactiveMongo which provides reactive environment to stream data to and from Mongo in a very natural way.
I've been working with Stephane Godbillon to integrate ReactiveMongo with Play2.1 while writing the Play2-ReactiveMongo module. Besides Mongo facilities for Play2.1, this module provides *Json To/From BSON conversion typeclasses*.

So it means you can manipulate JSON flows to and from DB directly without even converting into OO.

## JSON *coast-to-coast* design

Taking this into account, we can easily imagine the following:

- Receive JSON.
- Validate JSON.
- Transform JSON to fit expected DB document structure.
- Directly send JSON to DB (or somewhere else).

This is exactly the same case when serving data from DB:

- Extract some data from DB as JSON directly.
- Filter/transform this JSON to send only mandatory data in the format expected by the client (e.g you don't want some secure info to go out).
- Directly send JSON to the client.

In this context, we can easily imagine **manipulating a flow of JSON data** from client to DB and back without any (explicit) transformation in anything else than JSON.

Naturally, when you plug this transformation flow on **reactive infrastructure provided by Play2.1**, it suddenly opens new horizons.

This is the so-called (by me) **JSON coast-to-coast design**:

- Don't consider JSON data chunk by chunk but as a **continuous flow of data from client to DB (or else) through server**,
- Treat the **JSON flow like a pipe that you connect to others pipes** while applying modifications, transformations alongside,
- Treat the flow in a **fully asynchronous/non-blocking** way.

This is also one of the reason of being of Play2.1 reactive architecture…

I believe **considering your app through the prism of flows of data changes drastically the way you design** your web apps in general. It may also open new functional scopes that fit today's webapps requirements quite better than classic architecture. Anyway, this is not the subject here ;)

So, as you have deduced by yourself, to be able to manipulate Json flows based on validation and transformation directly, we needed some new tools. JSON combinators were good candidates but they are a bit too generic.

That's why we have created some specialized combinators and API called **JSON transformers** to do that.

# JSON transformers are `Reads[T <: JsValue]`

- You may tell JSON transformers are just `f:JSON => JSON`.
- So a JSON transformer could be simply a `Writes[A <: JsValue]`.
- But, a JSON transformer is not only a function: as we said, we also want to validate JSON while transforming it.
- As a consequence, a JSON transformer is a `Reads[A <: JsValue]`.

Keep in mind that a `Reads[A <: JsValue]` is able to transform and not only to read/validate

# Use `JsValue.transform` instead of `JsValue.validate`

We have provided a function helper in `JsValue` to help people consider a `Reads[T]` is a transformer and not only a validator:

```
JsValue.transform[A <: JsValue](reads: Reads[A]): JsResult[A]
```

This is exactly the same `JsValue.validate(reads)`

# The details

In the code samples below, we'll use the following JSON:

```
{
  "key1" : "value1",
  "key2" : {
    "key21" : 123,
    "key22" : true,
    "key23" : [ "alpha", "beta", "gamma"],
    "key24" : {
      "key241" : 234.123,
      "key242" : "value242"
    }
  },
  "key3" : 234
}
```

# Case 1: Pick JSON value in JsPath

## Pick value as JsValue

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick

scala> json.transform(jsonTransformer)
res9: play.api.libs.json.JsResult[play.api.libs.json.JsValue] =
  JsSuccess(
    ["alpha","beta","gamma"],
    /key2/key23
  )
```

`(__ \ 'key2 \ 'key23).json...`

- All JSON transformers are in `JsPath.json.`
  `(__ \ 'key2 \ 'key23).json.pick`
- `pick` is a `Reads[JsValue]` which picks the value **IN** the given JsPath.
  Here `["alpha","beta","gamma"]`
  `JsSuccess(["alpha","beta","gamma"],/key2/key23)`
- This is a simply successful `JsResult`.
- For info, `/key2/key23` represents the JsPath where data were read but don't care about it, it's mainly used by Play API to compose `JsResult(s)`).
- `["alpha","beta","gamma"]` is just due to the fact that we have overridden `toString`.

**Reminder**
`jsPath.json.pick` gets ONLY the value inside the JsPath

## Pick value as Type

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key23).json.pick[JsArray]
```

```
scala> json.transform(jsonTransformer)
res10: play.api.libs.json.JsResult[play.api.libs.json.JsArray] =
 JsSuccess(
  ["alpha","beta","gamma"],
  /key2/key23
 )
```

`(__ \ 'key2 \ 'key23).json.pick[JsArray]`

- `pick[T]` is a `Reads[T <: JsValue]` which picks the value (as a `JsArray` in our case) **IN** the given `JsPath`

**Reminder**
`jsPath.json.pick[T <: JsValue]` extracts ONLY the typed value inside the `JsPath`

# Case 2: Pick branch following `JsPath`

## Pick branch as `JsValue`

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key24 \ 'key241).json.pickBranch

scala> json.transform(jsonTransformer)
res11: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
 {
  "key2": {
   "key24":{
    "key241":234.123
   }
  }
 },
 /key2/key24/key241
 )
```

`(__ \ 'key2 \ 'key23).json.pickBranch`

- `pickBranch` is a `Reads[JsValue]` which picks the branch from root to given `JsPath`
  `{"key2":{"key24":{"key242":"value242"}}}`
- The result is the branch from root to given JsPath including the JsValue in `JsPath`

**Reminder:**
`jsPath.json.pickBranch` extracts the single branch down to JsPath + the value inside JsPath

# Case 3: Copy a value from input JsPath into a new JsPath

```
import play.api.libs.json._
```

```
val jsonTransformer = (__ \ 'key25 \ 'key251).json.copyFrom( (__ \ 'key2 \ 'key21).json.pick )

scala> json.transform(jsonTransformer)
res12: play.api.libs.json.JsResult[play.api.libs.json.JsObject]
 JsSuccess(
  {
   "key25":{
    "key251":123
   }
  },
  /key2/key21
 )
```

`(__ \ 'key25 \ 'key251).json.copyFrom( reads: Reads[A <: JsValue] )`

- `copyFrom` is a `Reads[JsValue]`
- `copyFrom` reads the JsValue from input JSON using provided Reads[A]
- `copyFrom` copies this extracted JsValue as the leaf of a new branch corresponding to given JsPath `{"key25":{"key251":123}}`
- `copyFrom` reads value `123`
- `copyFrom` copies this value into new branch `(__ \ 'key25 \ 'key251)`

**Reminder:**
`jsPath.json.copyFrom(Reads[A <: JsValue])` reads value from input JSON and creates a new branch with result as leaf

# Case 4: Copy full input Json & update a branch

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key24).json.update(
 __.read[JsObject].map{ o => o ++ Json.obj( "field243" -> "coucou" ) }
)

scala> json.transform(jsonTransformer)
res13: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
  {
   "key1":"value1",
   "key2":{
    "key21":123,
    "key22":true,
    "key23":["alpha","beta","gamma"],
    "key24":{
     "key241":234.123,
     "key242":"value242",
     "field243":"coucou"
```

```
    }
   },
   "key3":234
  },
 )
```

`(__ \ 'key2).json.update(reads: Reads[A < JsValue])`

- Is a `Reads[JsObject]`

`(__ \ 'key2 \ 'key24).json.update(reads)` *does 3 things:*

- Extracts value from input JSON at JsPath `(__ \ 'key2 \ 'key24)`.
- Applies `reads` on this relative value and re-creates a branch `(__ \ 'key2 \ 'key24)` adding result of `reads` as leaf.
- Merges this branch with full input JSON replacing existing branch (so it works only with input `JsObject` and not other type of `JsValue`).

`JsSuccess({…},)`

- Just for info, there is no JsPath as 2nd parameter there because the JSON manipulation was done from Root JsPath

**Reminder:**
`jsPath.json.update(Reads[A <: JsValue])` only works for `JsObject`, copies full input `JsObject` and updates jsPath with provided `Reads[A <: JsValue]`

# Case 5: Put a given value in a new branch

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key24 \ 'key241).json.put(JsNumber(456))

scala> json.transform(jsonTransformer)
res14: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
  {
   "key24":{
    "key241":456
   }
  },
 )
```

`(__ \ 'key24 \ 'key241).json.put( a: => JsValue )`

- Is a Reads[JsObject]

`(__ \ 'key24 \ 'key241).json.put( a: => JsValue )`

- Creates a new branch `(__ \ 'key24 \ 'key241)`
- Puts `a` as leaf of this branch.

`jsPath.json.put( a: => JsValue )`

- Takes a `JsValue` argument passed by name allowing to pass even a closure to it.

`jsPath.json.put`

- Does not care at all about input JSON.

- Simply replace input JSON by given value.

**Reminder: **
`jsPath.json.put( a: => Jsvalue )` creates a new branch with a given value without taking into account input JSON

# Case 6: Prune a branch from input JSON

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2 \ 'key22).json.prune

scala> json.transform(jsonTransformer)
res15: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
   {
    "key1":"value1",
    "key3":234,
    "key2":{
     "key21":123,
     "key23":["alpha","beta","gamma"],
     "key24":{
      "key241":234.123,
      "key242":"value242"
     }
    }
   },
   /key2/key22/key22
 )
```

`(__ \ 'key2 \ 'key22).json.prune`

- Is a `Reads[JsObject]` that works only with JsObject

`(__ \ 'key2 \ 'key22).json.prune`

- Removes given JsPath from input JSON (`key22` has disappeared under `key2`)

Please note the resulting `JsObject` hasn't same keys order as input `JsObject`. This is due to the implementation of `JsObject` and to the merge mechanism. But this is not important since we have overridden `JsObject.equals` method to take this into account.

**Reminder:**
`jsPath.json.prune` only works with JsObject and removes given JsPath form input JSON)
Please note that:
- `prune` doesn't work for recursive JsPath for the time being
- if `prune` doesn't find any branch to delete, it doesn't generate any error and returns unchanged JSON.

# More complicated cases

# Case 7: Pick a branch and update its content in 2 places

```
import play.api.libs.json._
import play.api.libs.json.Reads._

val jsonTransformer = (__ \ 'key2).json.pickBranch(
  (__ \ 'key21).json.update(
    of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }
  ) andThen
  (__ \ 'key23).json.update(
    of[JsArray].map{ case JsArray(arr) => JsArray(arr :+ JsString("delta")) }
  )
)

scala> json.transform(jsonTransformer)
res16: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
  JsSuccess(
    {
      "key2":{
        "key21":133,
        "key22":true,
        "key23":["alpha","beta","gamma","delta"],
        "key24":{
          "key241":234.123,
          "key242":"value242"
        }
      }
    },
    /key2
  )
```

`(__ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])`

- Extracts branch `__ \ 'key2` from input JSON and applies `reads` to the relative leaf of this branch (only to the content).

`(__ \ 'key21).json.update(reads: Reads[A <: JsValue])`

- Updates `(__ \ 'key21)` branch.

`of[JsNumber]`

- Is just a `Reads[JsNumber]`.
- Extracts a JsNumber from `(__ \ 'key21)`.

`of[JsNumber].map{ case JsNumber(nb) => JsNumber(nb + 10) }`

- Reads a JsNumber (_value 123_ in `__ \ 'key21`).
- Uses `Reads[A].map` to increase it by *10* (in immutable way naturally).

`andThen`

- Is just the composition of 2 `Reads[A]`.
- First reads is applied and then result is piped to second reads.

`of[JsArray].map{ case JsArray(arr) => JsArray(arr :+ JsString("delta")`

- Reads a JsArray (_value [alpha, beta, gamma] in `__ \ 'key23`_).
- Uses `Reads[A].map` to append `JsString("delta")` to it.

Please note the result is just the `__ \ 'key2` branch since we picked only this branch

# Case 8: Pick a branch and prune a sub-branch

```
import play.api.libs.json._

val jsonTransformer = (__ \ 'key2).json.pickBranch(
  (__ \ 'key23).json.prune
)

scala> json.transform(jsonTransformer)
res18: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
  JsSuccess(
    {
      "key2":{
        "key21":123,
        "key22":true,
        "key24":{
          "key241":234.123,
          "key242":"value242"
        }
      }
    },
    /key2/key23
  )
```

`(__ \ 'key2).json.pickBranch(reads: Reads[A <: JsValue])`

- Extracts branch `__ \ 'key2` from input JSON and applies `reads` to the relative leaf of this branch (only to the content).

`(__ \ 'key23).json.prune`

- Removes branch `__ \ 'key23` from relative JSON

Please remark the result is just the `__ \ 'key2` branch without `key23` field.

# What about combinators?

I stop there before it becomes boring (if not yet)…

Just keep in mind that you have now a huge toolkit to create generic JSON transformers. You can compose, map, flatmap transformers together into other transformers. So possibilities are almost infinite.

But there is a final point to treat: mixing those great new JSON transformers with previously presented Reads combinators. This is quite trivial as JSON transformers are just `Reads[A <: JsValue]`

Let's demonstrate by writing a **Gizmo to Gremlin** JSON transformer.

Here is Gizmo:

```
val gizmo = Json.obj(
  "name" -> "gizmo",
  "description" -> Json.obj(
    "features" -> Json.arr( "hairy", "cute", "gentle"),
    "size" -> 10,
    "sex" -> "undefined",
    "life_expectancy" -> "very old",
    "danger" -> Json.obj(
      "wet" -> "multiplies",
      "feed after midnight" -> "becomes gremlin"
    )
  ),
  "loves" -> "all"
)
```

Here is Gremlin:

```
val gremlin = Json.obj(
  "name" -> "gremlin",
  "description" -> Json.obj(
    "features" -> Json.arr("skinny", "ugly", "evil"),
    "size" -> 30,
    "sex" -> "undefined",
    "life_expectancy" -> "very old",
    "danger" -> "always"
  ),
  "hates" -> "all"
)
```

Ok let's write a JSON transformer to do this transformation

```
import play.api.libs.json._
import play.api.libs.json.Reads._
import play.api.libs.functional.syntax._

val gizmo2gremlin = (
  (__ \ 'name).json.put(JsString("gremlin")) and
  (__ \ 'description).json.pickBranch(
    (__ \ 'size).json.update( of[JsNumber].map{ case JsNumber(size) => JsNumber(size * 3) } ) and
    (__ \ 'features).json.put( Json.arr("skinny", "ugly", "evil") ) and
    (__ \ 'danger).json.put(JsString("always"))
    reduce
  ) and
  (__ \ 'hates).json.copyFrom( (__ \ 'loves).json.pick )
```

```
) reduce

scala> gizmo.transform(gizmo2gremlin)
res22: play.api.libs.json.JsResult[play.api.libs.json.JsObject] =
 JsSuccess(
  {
   "name":"gremlin",
   "description":{
    "features":["skinny","ugly","evil"],
    "size":30,
    "sex":"undefined",
    "life_expectancy":
    "very old","danger":"always"
   },
   "hates":"all"
  },
 )
```

Here we are ;)
I'm not going to explain all of this because you should be able to understand now.
Just remark:

`(__ \ 'features).json.put(…)` *is after* `(__ \ 'size).json.update` *so that it overwrites original* `(__ \ 'features)`
`(Reads[JsObject] and Reads[JsObject]) reduce`

- It merges results of both `Reads[JsObject]` (JsObject ++ JsObject)
- It also applies the same JSON to both `Reads[JsObject]` unlike `andThen` which injects the result of the first reads into second one.

**Next:** JSON Macro Inception

# JSON Macro Inception

Please note this documentation was initially published as an article by Pascal Voitot (@mandubian) on mandubian.com
**This feature is still experimental because Scala Macros are still experimental in Scala 2.10. If you prefer not using an experimental feature from Scala, please use hand-written Reads/Writes/Format which are strictly equivalent.**

## Writing a default case class Reads/Writes/Format is so boring!

Remember how you write a `Reads[T]` for a case class.

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = (
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
)(Person)
```

So you write 4 lines for this case class.
You know what?
We have had a few complaints from some people who think it's not cool to write a`Reads[TheirClass]` because usually Java JSON frameworks like Jackson or Gson do it behind the curtain without writing anything.
We argued that Play2.1 JSON serializers/deserializers are:

- completely typesafe,
- fully compiled,
- nothing was performed using introspection/reflection at runtime.

But for some, this didn't justify the extra lines of code for case classes.

We believe this is a really good approach so we persisted and proposed:

- JSON simplified syntax
- JSON combinators
- JSON transformers

Added power, but nothing changed for the additional 4 lines.

---

# Let's be minimalist

As we are perfectionist, now we propose a new way of writing the same code:

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

case class Person(name: String, age: Int, lovesChocolate: Boolean)

implicit val personReads = Json.reads[Person]
```

1 line only.
Questions you may ask immediately:

Does it use runtime bytecode enhancement? -> NO

Does it use runtime introspection? -> NO

Does it break type-safety? -> NO

**So what?**
After creating buzzword **JSON coast-to-coast design**, let's call it **JSON INCEPTION**.

# JSON Inception

# Code Equivalence

As explained just before:

```
import play.api.libs.json._
// please note we don't import functional.syntax._ as it is managed by the macro itself

implicit val personReads = Json.reads[Person]

// IS STRICTLY EQUIVALENT TO writing

implicit val personReads = (
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
)(Person)
```

# Inception equation

Here is the equation describing the windy *Inception* concept:
(Case Class INSPECTION) + (Code INJECTION) + (COMPILE Time) = INCEPTION

### *Case Class Inspection*

As you may deduce by yourself, in order to ensure preceding code equivalence, we need :

- to inspect `Person` case class,
- to extract the 3 fields `name`, `age`, `lovesChocolate` and their types,
- to resolve typeclasses implicits,

- to find `Person.apply`.

### *INJECTION?*

No I stop you immediately…

> **Code injection is not dependency injection…**
> No Spring behind inception… No IOC, No DI… No No No ;)

I used this term on purpose because I know that injection is now linked immediately to IOC and Spring. But I'd like to re-establish this word with its real meaning.
Here code injection just means that **we inject code at compile-time into the compiled scala AST** (Abstract Syntax Tree).
So `Json.reads[Person]` is compiled and replaced in the compile AST by:

```
(
  (__ \ 'name).read[String] and
  (__ \ 'age).read[Int] and
  (__ \ 'lovesChocolate).read[Boolean]
)(Person)
```

Nothing less, nothing more…

### *COMPILE-TIME*

Yes everything is performed at compile-time.
No runtime bytecode enhancement.
No runtime introspection.

> As everything is resolved at compile-time, you will have a compile error if you did not import the required implicits for all the types of the fields.

# Json inception is Scala 2.10 Macros

We needed a Scala feature enabling:

- compile-time code enhancement
- compile-time class/implicits inspection

- compile-time code injection
  This is enabled by a new experimental feature introduced in Scala 2.10: Scala Macros

  Scala macros is a new feature (still experimental) with a huge potential. You can :

- introspect code at compile-time based on Scala reflection API,
- access all imports, implicits in the current compile context
- create new code expressions, generate compiling errors and inject them into compile chain.

  Please note that:

- **We use Scala Macros because it corresponds exactly to our requirements.**
- **We use Scala macros as an enabler, not as an end in itself.**
- **The macro is a helper that generates the code you could write by yourself.**
- **It doesn't add, hide unexpected code behind the curtain.**
- **We follow the *no-surprise* principle**

  As you may discover, writing a macro is not a trivial process since your macro code executes in the compiler runtime (or universe).

```
So you write macro code
  that is compiled and executed
  in a runtime that manipulates your code
     to be compiled and executed
     in a future runtime…
```

**That's also certainly why I called it *Inception* ;)**

So it requires some mental exercises to follow exactly what you do. The API is also quite complex and not fully documented yet. Therefore, you must persevere when you begin using macros.

I'll certainly write other articles about Scala macros because there are lots of things to say. This article is also meant **to begin the reflection about the right way to use Scala Macros**. Great power means greater responsability so it's better to discuss all together and establish a few good manners…

# Writes[T] & Format[T]

Please remark that JSON inception just works for structures having `unapply/apply`functions with corresponding input/output types.
Naturally, you can also *incept* `Writes[T]` and `Format[T]`.

# Writes[T]

```
import play.api.libs.json._

implicit val personWrites = Json.writes[Person]
```

# Format[T]

```
import play.api.libs.json._

implicit val personWrites = Json.format[Person]
```

# Special patterns

- **You can define your Reads/Writes in your companion object**
  This is useful because then the implicit Reads/Writes is implicitly infered as soon as you manipulate an instance of your class.

```
import play.api.libs.json._

case class Person(name: String, age: Int)

object Person{
  implicit val personFmt = Json.format[Person]
}
```

- **You can now define Reads/Writes for single-field case class** (known limitation until 2.1-RC2)

```
import play.api.libs.json._

case class Person(names: List[String])

object Person{
  implicit val personFmt = Json.format[Person]
}
```

# Known limitations

- **Don't override apply function in companion object** because then the Macro will have several apply functions and won't choose.
- **Json Macros only work when apply and unapply have corresponding input/output types**: This is naturally the case for case classes. But if you want to the same with a trait, you must implement the same apply/unapply you would have in a case class.
- **Json Macros are known to accept Option/Seq/List/Set & Map[String, _]**. For other generic types, test and if not working, use traditional way of writing Reads/Writes manually.

**Next:** Working with XML

# 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 `application/xml` or `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: application/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="KO">Missing parameter [name]</message>)
  }
}
```

Now it replies with:

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

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

**Next:** Handling file upload

# 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.

**Note**: The HTTP method used to submit the form must be `POST` (not `GET`).

Start by writing an HTML form:

```
@helper.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(s"/tmp/picture/$filename"))
    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`mutipartFormData` parser handles file upload.

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

At last, add a `POST` router

```
POST  /         controllers.Application.upload()
```

# 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/uploaded"))
  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`mutipartFormData` 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`.

**Next:** Accessing an SQL database

# Accessing an SQL database

# Configuring JDBC connection pools

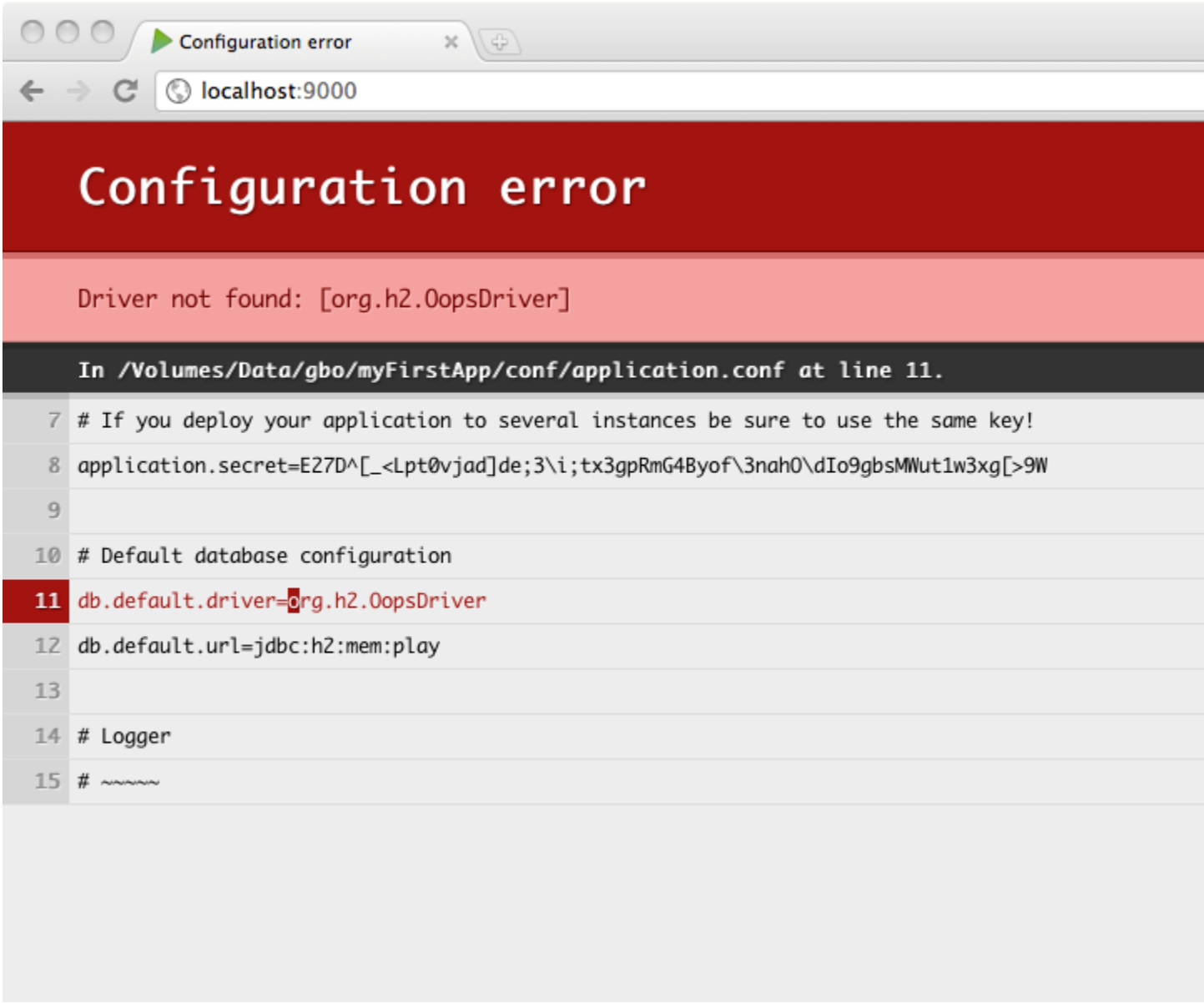
Play provides a plug-in for managing JDBC connection pools. You can configure as many databases as you need.

To enable the database plug-in, add jdbc in your build dependencies :

```
libraryDependencies += jdbc
```

Then you must configure a connection pool in the `conf/application.conf` file. By convention, the default JDBC data source must be called `default` and the corresponding configuration properties are `db.default.driver` and `db.default.url`.

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




**Note:** You likely need to enclose the JDBC URL configuration value with double quotes, since ':' is a reserved character in the configuration syntax.

## H2 database engine connection properties

In memory database:

```
# Default database configuration using H2 database engine in an in-memory mode
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

File based database:

```
# Default database configuration using H2 database engine in a persistent mode
db.default.driver=org.h2.Driver
```

```
db.default.url="jdbc:h2:/path/to/db-file"
```

The details of the H2 database URLs are found from H2 Database Engine Cheat Sheet.

### SQLite database engine connection properties

```
# Default database configuration using SQLite database engine
db.default.driver=org.sqlite.JDBC
db.default.url="jdbc:sqlite:/path/to/db-file"
```

### PostgreSQL database engine connection properties

```
# Default database configuration using PostgreSQL database engine
db.default.driver=org.postgresql.Driver
db.default.url="jdbc:postgresql://database.example.com/playdb"
```

### MySQL database engine connection properties

```
# Default database configuration using MySQL database engine
# Connect to playdb as playdbuser
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://localhost/playdb"
db.default.username=playdbuser
db.default.password="a strong password"
```

# How 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"
```

# Configuring the JDBC Driver

Play is bundled only with an H2 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:

```
libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.34"
```

Or if the driver can't be found from repositories you can drop the driver into your project'sunmanaged dependencies `lib` directory.

# 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 are several ways to retrieve a JDBC connection. The simplest way is:

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

Following code show you a JDBC example very simple, working with MySQL 5.*:

```
package controllers
import play.api.Play.current
import play.api.mvc._
import play.api.db._

object Application extends Controller {

  def index = Action {
    var outString = "Number is "
    val conn = DB.getConnection()
    try {
      val stmt = conn.createStatement
      val rs = stmt.executeQuery("SELECT 9 as testkey ")
      while (rs.next()) {
        outString += rs.getString("testkey")
      }
    } finally {
      conn.close()
    }
    Ok(outString)
  }

}
```

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:

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

For a database other than the default:

```
// access "orders" database instead of "default"
DB.withConnection("orders") { 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's auto-commit to `false` and to manage a transaction for the block:

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

# Selecting and configuring the connection pool

Out of the box, Play provides two database connection pool implementations, HikariCP and BoneCP. **The default is HikariCP**, but this can be changed by setting the `play.db.pool` property:

```
play.db.pool=bonecp
```

The full range of configuration options for connection pools can be found by inspecting the `play.db.prototype` property in Play's JDBC `reference.conf`.

# Testing

For information on testing with databases, including how to setup in-memory databases and, see Testing With Databases.

# Enabling Play database evolutions

Read Evolutions to find out what Play database evolutions are useful for, and follow the setup instructions for using it.

**Next:** Using Slick to access your database

# Using Play Slick

The Play Slick module makes Slick a first-class citizen of Play.

The Play Slick module consists of two features:

- Integration of Slick into Play's application lifecycle.
- Support for Play database evolutions.

Play Slick currently supports Slick 3.1 with Play 2.4, for both Scala 2.10 and 2.11.

Note: This guide assumes you already know both Play 2.4 and Slick 3.1.

## Getting Help

If you are having trouble using Play Slick, check if the FAQ contains the answer. Otherwise, feel free to reach out to play-framework user group. Also, note that if you are seeking help on Slick, the slick user group may be a better place.
Finally, if you prefer to get an answer for your Play and Slick questions in a timely manner, and with a well-defined SLA, you may prefer to get in touch with Typesafe, as it offers commercial support for these technologies.

# About this release

If you have been using a previous version of Play Slick, you will notice that there have been quite a few major changes. It's recommended to read the migration guide for a smooth upgrade.

While, if this is the first time you are using Play Slick, you will appreciate that the integration of Slick in Play is quite austere. Meaning that if you know both Play and Slick, using Play Slick module should be straightforward.

# Setup

Add a library dependency on play-slick:

```
"com.typesafe.play" %% "play-slick" % "1.1.1"
```

The above dependency will also bring along the Slick library as a transitive dependency. This implies you don't need to add an explicit dependency on Slick, but you might still do so if needed. A likely reason for wanting to explicitly define a dependency to Slick is if you want to use a newer version than the one bundled with play-slick. Because Slick trailing dot releases are binary compatible, you won't incur any risk in using a different Slick trailing point release than the one that was used to build play-slick.

## Support for Play database evolutions

Play Slick supports Play database evolutions.

To enable evolutions, you will need the following dependencies:

```
"com.typesafe.play" %% "play-slick" % "1.1.1"
"com.typesafe.play" %% "play-slick-evolutions" % "1.1.1"
```

Note there is no need to add the Play `evolutions` component to your dependencies, as it is a transitive dependency of the `play-slick-evolutions` module.

## JDBC driver dependency

Play Slick module does not bundle any JDBC driver. Hence, you will need to explicitly add the JDBC driver(s) you want to use in your application. For instance, if you would like to use an in-memory database such as H2, you will have to add a dependency to it:

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

# Database Configuration

To have Play Slick module handling the lifecycle of Slick databases, it is important that you never create database's instances explicitly in your code. Rather, you should provide a valid Slick driver and database configuration in your **application.conf** (by convention the default Slick database must be called `default`):

```
# Default database configuration
slick.dbs.default.driver="slick.driver.H2Driver$"
slick.dbs.default.db.driver="org.h2.Driver"
slick.dbs.default.db.url="jdbc:h2:mem:play"
```

First, note that the above is a valid Slick configuration (for the complete list of configuration parameters that you can use to configure a database see the Slick ScalaDoc for Database.forConfig - make sure to expand the `forConfig` row in the doc).
Second, the `slick.dbs` prefix before the database's name is configurable. In fact, you may change it by overriding the value of the configuration key `play.slick.db.config`.
Third, in the above configuration `slick.dbs.default.driver` is used to configure the Slick driver, while `slick.dbs.default.db.driver` is the underlying JDBC driver used by Slick's backend. In the above configuration we are configuring Slick to use H2 database, but Slick supports several other databases. Check the Slick documentationfor a complete list of supported databases, and to find a matching Slick driver.
Slick does not support the `DATABASE_URL` environment variable in the same way as the default Play JBDC connection pool. But starting in version 3.0.3, Slick provides a`DatabaseUrlDataSource` specifically for parsing the environment variable.

```
slick.dbs.default.driver="slick.driver.PostgresDriver$"
slick.dbs.default.db.dataSourceClass = "slick.jdbc.DatabaseUrlDataSource"
slick.dbs.default.db.properties.driver = "org.postgresql.Driver"
```

On some platforms, such as Heroku, you may substitue the `JDBC_DATABASE_URL`, which is in the format `jdbc:vendor://host:port/db?args`, if it is available. For example:

```
slick.dbs.default.driver="slick.driver.PostgresDriver$"
slick.dbs.default.db.driver="org.postgresql.Driver"
slick.dbs.default.db.url=${JDBC_DATABASE_URL}
```

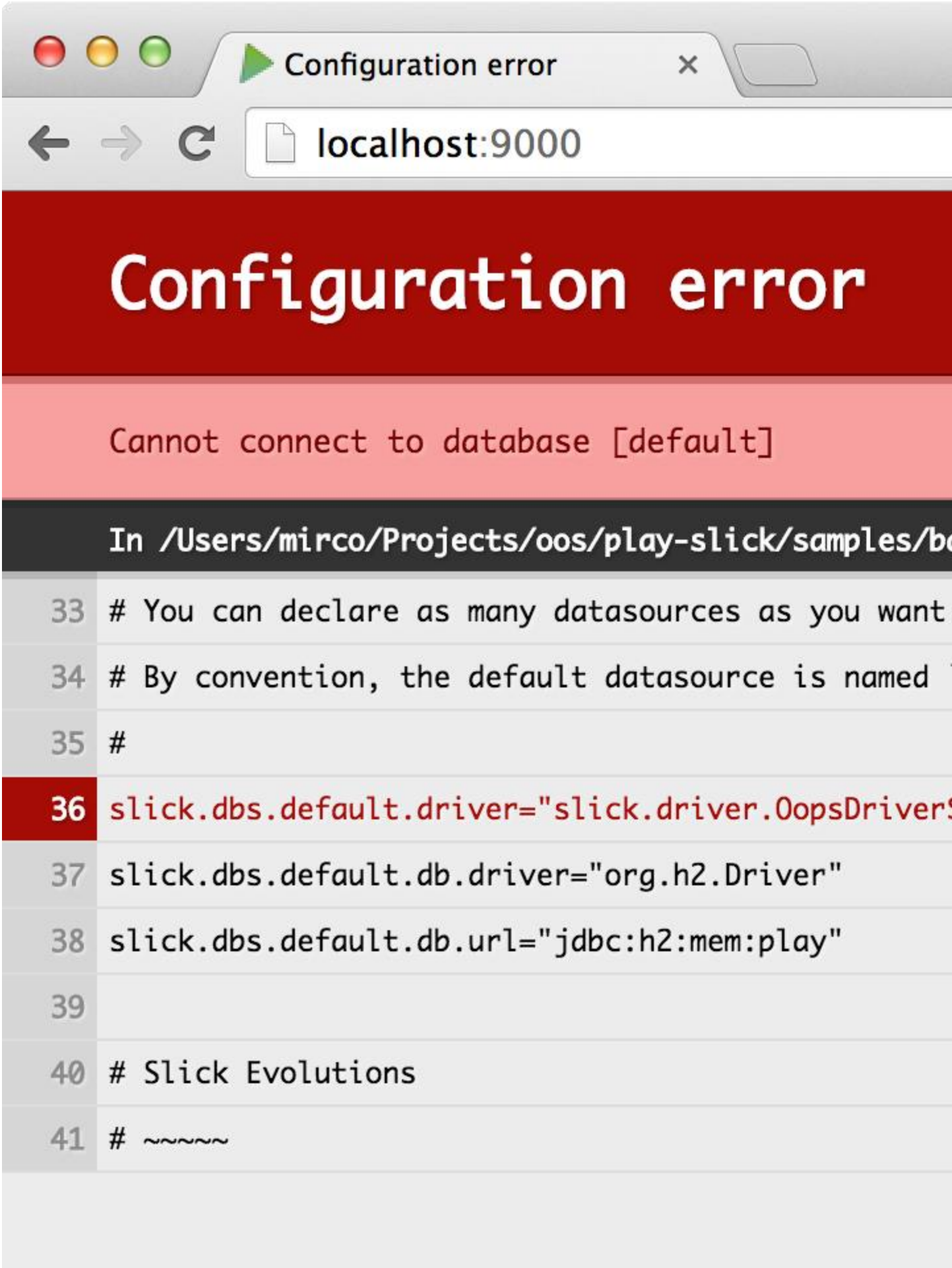
Note: Failing to provide a valid value for both `slick.dbs.default.driver` and `slick.dbs.default.db.driver` will lead to an exception when trying to run your Play application.

To configure several databases:

```
# Orders database
slick.dbs.orders.driver="slick.driver.H2Driver$"
slick.dbs.orders.db.driver="org.h2.Driver"
slick.dbs.orders.db.url="jdbc:h2:mem:play"

# Customers database
slick.dbs.customers.driver="slick.driver.H2Driver$"
slick.dbs.customers.db.driver="org.h2.Driver"
slick.dbs.customers.db.url="jdbc:h2:mem:play"
```

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


Configuration error
localhost:9000
Configuration error
Cannot connect to database [default]
In /Users/mirco/Projects/oos/play-slick/samples/b
33 # You can declare as many datasources as you want
34 # By convention, the default datasource is named
35 #
36 slick.dbs.default.driver="slick.driver.OopsDriver
37 slick.dbs.default.db.driver="org.h2.Driver"
38 slick.dbs.default.db.url="jdbc:h2:mem:play"
39
40 # Slick Evolutions
41 # ~~~~~

Note: Your application will be started only if you provide a valid Slick configuration.

# Usage

After having properly configured a Slick database, you can obtain a `DatabaseConfig` (which is a Slick type bundling a database and driver) in two different ways. Either by using dependency injection, or through a global lookup via the `DatabaseConfigProvider` singleton.

Note: A Slick database instance manages a thread pool and a connection pool. In general, you should not need to shut down a database explicitly in your code (by calling its `close` method), as the Play Slick module takes care of this already.

## DatabaseConfig via Dependency Injection

Here is an example of how to inject a `DatabaseConfig` instance for the default database (i.e., the database named `default` in your configuration):

```
class Application @Inject()(dbConfigProvider: DatabaseConfigProvider) extends Controller {
  val dbConfig = dbConfigProvider.get[JdbcProfile]
```

Injecting a `DatabaseConfig` instance for a different database is also easy. Simply prepend the annotation `@NamedDatabase("<db-name>")` to the `dbConfigProvider` constructor parameter:

```
class Application2 @Inject()(@NamedDatabase("<db-name>") dbConfigProvider: DatabaseConfigProvider)
extends Controller {
```

Of course, you should replace the string `"<db-name>"` with the name of the database's configuration you want to use.

For a full example, have a look at this sample projet.

## DatabaseConfig via Global Lookup

Here is an example of how to lookup a `DatabaseConfig` instance for the default database (i.e., the database named `default` in your configuration):

```
val dbConfig = DatabaseConfigProvider.get[JdbcProfile](Play.current)
```

Looking up a `DatabaseConfig` instance for a different database is also easy. Simply pass the database name:

```
val dbConfig = DatabaseConfigProvider.get[JdbcProfile]("<db-name>")(Play.current)
```

Of course, you should replace the string `"<db-name>"` with the name of the database's configuration you want to use.

For a full example, have a look at this sample projet.

## Running a database query in a Controller

To run a database query in your controller, you will need both a Slick database and driver. Fortunately, from the above we now know how to obtain a Slick `DatabaseConfig`, hence we have what we need to run a database query.

You will need to import some types and implicits from the driver:

```
import dbConfig.driver.api._
```

And then you can define a controller's method that will run a database query:

```
def index(name: String) = Action.async { implicit request =>
  val resultingUsers: Future[Seq[User]] = dbConfig.db.run(Users.filter(_.name === name).result)
  resultingUsers.map(users => Ok(views.html.index(users)))
}
```

That's just like using stock Play and Slick!

# Configuring the connection pool

Read here to find out how to configure the connection pool.

**Next:** Play Slick migration guide

# Play Slick Migration Guide

This is a guide for migrating from Play Slick v0.8 to v1.0 or v1.1.

It assumes you have already migrated your project to use Play 2.4 (see Play 2.4 Migration Guide), that you have read the Slick 3.1 documentation, and are ready to migrate your Play application to use the new Slick Database I/O Actions API.

# Build changes

Update the Play Slick dependency in your sbt build to match the version provided in theSetup section.

## Removed H2 database dependency

Previous releases of Play Slick used to bundle the H2 database library. That's no longer the case. Hence, if you want to use H2 you will need to explicitly add it to your project's dependencies:

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

## Evolutions support in a separate module

Support for database evolutions used to be included with Play Slick. That's no longer the case. Therefore, if you are using evolutions, you now need to add an additional dependency to `play-slick-evolutions` as explained here.

While, if you are not using evolutions, you can now safely remove`evolutionplugin=disabled` from your `application.conf`.

# Database configuration

With the past releases of Slick Play (which used Slick 2.1 or earlier), you used to configure Slick datasources exactly like you would configure Play JDBC datasources. This is no longer the case, and the following configuration will now be **ignored** by Play Slick:

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

There are several reasons for this change. First, the above is not a valid Slick configuration. Second, in Slick 3 you configure not just the datasource, but also both a connection pool and a thread pool. Therefore, it makes sense for Play Slick to use an entirely different path for configuring Slick databases. The default path for Slick configuration is now `slick.dbs`.

Here is how you would need to migrate the above configuration:

```
slick.dbs.default.driver="slick.driver.H2Driver$" # You must provide the required Slick driver!
slick.dbs.default.db.driver=org.h2.Driver
slick.dbs.default.db.url="jdbc:h2:mem:play"
slick.dbs.default.db.user=sa
slick.dbs.default.db.password=""
```

Note: If your database configuration contains settings for the connection pool, be aware that you will need to migrate those settings as well. However, this may be a bit trickier, because Play 2.3 default connection pool used to be BoneCP, while the default Slick 3 connection pool is HikariCP.
Read here for how to configure the connection pool.

# Automatic Slick driver detection

Play Slick used to automatically infer the needed Slick driver from the datasource configuration. This feature was removed, hence you must provide the Slick driver to use, for each Slick database configuration, in your **application.conf**.

The rationale for removing this admittedly handy feature is that we want to accept only valid Slick configurations. Furthermore, it's not always possible to automatically detect the correct Slick driver from the database configuration (if this was possible, then Slick would already provide such functionality).

Therefore, you will need to make the following changes:

- Each of your Slick database configuration must provide the Slick driver (see here for an example of how to migrate your database configuration).
- Remove all imports to `import play.api.db.slick.Config.driver.simple._`.
- Read here for how to lookup the Slick driver and database instances (which are needed to use the new Slick 3 Database I/O Actions API).

# `DBAction` and `DBSessionRequest` were removed

Play Slick used to provide a `DBAction` that was useful for:

- Conveniently pass the Slick `Session` into your Action method.
- Execute the action's body, and hence any blocking call to the database, in a separate thread pool.
- Limiting the number of blocking requests queued in the thread pool (useful to limit application's latency)

`DBAction` was indeed handy when using Slick 2.1. However, with the new Slick 3 release, we don't need it anymore. The reason is that Slick 3 comes with a new asynchronous API (a.k.a., Database I/O Actions API) that doesn't need the user to manipulate neither a `Session` nor a `Connection`. This makes `DBSessionRequest` and `DBAction`, together with its close friends `CurrentDBAction` and `PredicatedDBAction`, completely obsolete, which is why they have been removed.

Having said that, migrating your code should be as simple as changing all occurrences of `DBAction` and friends, with the standard Play `Action.async`. Click here for an example.

# Thread Pool

Play Slick used to provide a separate thread pool for executing controller's actions requiring to access a database. Slick 3 already does exactly this, hence there is no longer a need for Play Slick to create and manage additional thread pools. It follows that the below configuration parameter are effectively obsolete and should be removed from your **applications.conf**:

```
db.$dbName.maxQueriesPerRequest
slick.db.execution.context
```

The parameter `db.$dbName.maxQueriesPerRequest` was used to limit the number of tasks queued in the thread pool. In Slick 3 you can reach similar results by tuning the configuration parameters `numThreads` and `queueSize`. Read the Slick ScalaDoc for Database.forConfig (make sure to expand the `forConfig` row in the doc).

While the parameter `slick.db.execution.context` was used to name the thread pools created by Play Slick. In Slick 3, each thread pool is named using the Slick database configuration path, i.e., if in your **application.conf** you have provided a Slick configuration for the database named `default`, then Slick will create a thread pool named `default` for executing the database action on the default database. Note that the name used for the thread pool is not configurable.

# `Profile` was removed

The trait `Profile` was removed and you can use instead `HasDatabaseConfigProvider` or `HasDatabaseConfig` with similar results.
The trait to use depend on what approach you select to retrieve a Slick database and driver (i.e., an instance of `DatabaseConfig`). If you decide to use dependency injection, then `HasDatabaseConfigProvider` will serve you well. Otherwise, use `HasDatabaseConfig`.
Read here for a discussion of how to use dependency injection vs global lookup to retrieve an instance of `DatabaseConfig`.

# `Database` was removed

The object `Database` was removed. To retrieve a Slick database and driver (i.e., an instance of `DatabaseConfig`) read here.

# `Config` was removed

The `Config` object, together with `SlickConfig` and `DefaultSlickConfig`, were removed. These abstractions are simply not needed. If you used to call `Config.driver` or `Config.datasource` to retrieve the Slick driver and database, you should now use `DatabaseConfigProvider`. Read here for details.

# `SlickPlayIteratees` was removed

If you were using `SlickPlayIteratees.enumerateSlickQuery` to stream data from the database, you will be happy to know that doing so became a lot easier. Slick 3 implements the reactive-streams (Service Provider Interface), and Play 2.4 provides a utility class to handily convert a reactive stream into a Play enumerator.
In Slick, you can obtain a reactive stream by calling the method `stream` on a Slick database instance (instead of the eager `run`). To convert the stream into an enumerator simply call `play.api.libs.streams.Streams.publisherToEnumerator`, passing the stream in argument.
For a full example, have a look at this sample project.

# DDL support was removed

Previous versions of Play Slick included a DDL plugin which would read your Slick tables definitions, and automatically creates schema updates on reload. While this is an interesting and useful feature, the underlying implementation was fragile, and and relied on the assumption that your tables would be accessible via a module (i.e., a Scala `object`). This coding pattern was possible because Play Slick allowed to import the Slick driver available via a top-level import. However, because support for automatic detection of the Slick driver was removed, you will not declare a top-level import for the Slick driver. This implies that Slick tables will no longer be accessible via a module. This fact breaks the assumption made in the initial implementation of the DDL plugin, and it's the reason why the feature was removed.

The consequence of the above is that you are in charge of creating and managing your project's database schema. Therefore, whenever you make a change a Slick table in the code, make sure to also update the database schema. If you find it tedious to manually keep in sync your database schema and the related table definition in your code, you may want to have a look at the code generation feature available in Slick.

**Next:** Play Slick advanced topics

# Play Slick Advanced Topics

## Connection Pool

With Slick 3 release, Slick starts and controls both a connection pool and a thread pool for optimal asynchronous execution of your database actions.

In Play Slick we have decided to let Slick be in control of creating and managing the connection pool (the default connection pool used by Slick 3 is HikariCP), which means that to tune the connection pool you will need to look at the Slick ScalaDoc forDatabase.forConfig(make sure to expand the `forConfig` row in the doc). In fact, be aware that any value you may pass for setting the Play connection pool (e.g., under the key `play.db.default.hikaricp`) is simply not picked up by Slick, and hence effectively ignored.
Also, note that as stated in the Slick documentation, a reasonable default for the connection pool size is calculated from the thread pool size. In fact, you should only need to tune `numThreads` and `queueSize` in most cases, for each of your database configuration.
Finally, it's worth mentioning that while Slick allows using a different connection pool thanHikariCP(though, Slick currently only offers built-in support for HikariCP, and requires

you to provide an implementation of JdbcDataSourceFactory if you want to use a different connection pool), Play Slick currently doesn't allow using a different connection pool than HikariCP.

Note: Changing the value of `play.db.pool` won't affect what connection pool Slick is using. Furthermore, be aware that any configuration under `play.db` is not considered by Play Slick.

# Thread Pool

With Slick 3.0 release, Slick starts and controls both a thread pool and a connection pool for optimal asynchronous execution of your database actions.

For optimal execution, you may need to tune the `numThreads` and `queueSize`parameters, for each of your database configuration. Refer to the Slick documentationfor details.

**Next:** Play Slick FAQ

# Play Slick FAQ

## What version should I use?

Have a look at the compatibility matrix to know what version you should be using.

## `play.db.pool` is ignored

It's indeed the case. Changing the value of `play.db.pool` won't affect what connection pool Slick is going to use. The reason is simply that Play Slick module currently doesn't support using a different connection pool than HikariCP.

## Changing the connection pool used by Slick

While Slick allows using a different connection pool than HikariCP(though, Slick currently only offers built-in support for HikariCP, and requires you to provide an implementation of JdbcDataSourceFactory if you want to use a different connection pool), Play Slick currently doesn't allow using a different connection pool than HikariCP. If you find yourself needing this feature, you can try to drop us a note on playframework-dev.

# A binding to `play.api.db.DBApi` was already configured

If you get the following exception when starting your Play application:

```
1) A binding to play.api.db.DBApi was already configured at
play.api.db.slick.evolutions.EvolutionsModule.bindings:
Binding(interface play.api.db.DBApi to ConstructionTarget(class
play.api.db.slick.evolutions.internal.DBApiAdapter) in interface javax.inject.Singleton).
 at play.api.db.DBModule.bindings(DBModule.scala:25):
Binding(interface play.api.db.DBApi to ProviderConstructionTarget(class play.api.db.DBApiProvider))
```

It is very likely that you have enabled the jdbc plugin, and that doesn't really make sense if you are using Slick for accessing your databases. To fix the issue simply remove the Play *jdbc* component from your project's build.

Another possibility is that there is another Play module that is binding DBApi to some other concrete implementation. This means that you are still trying to use Play Slick together with another Play module for database access, which is likely not what you want.

# Play throws `java.lang.ClassNotFoundException: org.h2.tools.Server`

If you get the following exception when starting your Play application:

```
java.lang.ClassNotFoundException: org.h2.tools.Server
        at java.net.URLClassLoader$1.run(URLClassLoader.java:372)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        ...
```

It means you are trying to use a H2 database, but have forgot to add a dependency to it in your project's build. Fixing the problem is simple, just add the missing dependency in your project's build, e.g.,

```
"com.h2database" % "h2" % "${H2_VERSION}" // replace `${H2_VERSION}` with an actual version number
```

**Next:** Using Anorm to access your database

# 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 configure it as explained on the Scala database page .

# 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.

---

# Add Anorm to your project

You will need to add Anorm and JDBC plugin to your dependencies :

```
libraryDependencies ++= Seq(
  jdbc,
  "com.typesafe.play" %% "anorm" % "2.4.0"
)
```

# 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._
import play.api.db.DB

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()
```

If you are inserting data that has an auto-generated `Long` primary key, you can call`executeInsert()`.

```
val id: Option[Long] =
  SQL("insert into City(name, country) values ({name}, {country})")
  .on('name -> "Cambridge", 'country -> "New Zealand").executeInsert()
```

When key generated on insertion is not a single `Long`, `executeInsert` can be passed a`ResultSetParser` to return the correct key.

```
import anorm.SqlParser.str

val id: List[String] =
  SQL("insert into City(name, country) values ({name}, {country})")
  .on('name -> "Cambridge", 'country -> "New Zealand")
  .executeInsert(str.+) // insertion returns a list of at least one string keys
```

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 a value to them:

```
SQL(
  """
    select * from Country c
```

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

You can also use string interpolation to pass parameters (see details thereafter).

In case several columns are found with same name in query result, for example columns named `code` in both `Country` and `CountryLanguage` tables, there can be ambiguity. By default a mapping like following one will use the last column:

```
import anorm.{ SQL, SqlParser }

val code: String = SQL(
  """
    select * from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = {countryCode}
  """)
  .on("countryCode" -> "FRA").as(SqlParser.str("code").single)
```

If `Country.Code` is 'First' and `CountryLanguage` is 'Second', then in previous example `code` value will be 'Second'. Ambiguity can be resolved using qualified column name, with table name:

```
import anorm.{ SQL, SqlParser }

val code: String = SQL(
  """
    select * from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = {countryCode}
  """)
  .on("countryCode" -> "FRA").as(SqlParser.str("Country.code").single)
// code == "First"
```

When a column is aliased, typically using SQL `AS`, its value can also be resolved. Following example parses column with `country_lang` alias.

```
import anorm.{ SQL, SqlParser }

val lang: String = SQL(
  """
    select l.language AS country_lang from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where c.code = {countryCode}
  """).on("countryCode" -> "FRA").
    as(SqlParser.str("country_lang").single)
```

Columns can also be specified by position, rather than name:

```
import anorm.SqlParser.{ str, float }
// Parsing column by name or position
val parser =
  str("name") ~ float(3) /* third column as float */ map {
```

```
  case name ~ f => (name -> f)
 }

val product: (String, Float) = SQL("SELECT * FROM prod WHERE id = {id}").
 on('id -> "p").as(parser.single)
```

`java.util.UUID` can be used as parameter, in which case its string value is passed to statement.

## SQL queries using String Interpolation

Since Scala 2.10 supports custom String Interpolation there is also a 1-step alternative to `SQL(queryString).on(params)` seen before. You can abbreviate the code as:

```
val name = "Cambridge"
val country = "New Zealand"

SQL"insert into City(name, country) values ($name, $country)"
```

It also supports multi-line string and inline expresions:

```
val lang = "French"
val population = 10000000
val margin = 500000

val code: String = SQL"""
  select * from Country c
   join CountryLanguage l on l.CountryCode = c.Code
   where l.Language = $lang and c.Population >= ${population - margin}
   order by c.Population desc limit 1"""
 .as(SqlParser.str("Country.code").single)
```

This feature tries to make faster, more concise and easier to read the way to retrieve data in Anorm. Please, feel free to use it wherever you see a combination of `SQL().on()` functions (or even an only `SQL()` without parameters).

By using `#$value` instead of `$value`, interpolated value will be part of the prepared statement, rather being passed as a parameter when executing this SQL statement (e.g. `#$cmd` and `#$table` in example bellow).

```
val cmd = "SELECT"
val table = "Test"

SQL"""#$cmd * FROM #$table WHERE id = ${"id1"} AND code IN (${Seq(2, 5)})"""

// prepare the SQL statement, with 1 string and 2 integer parameters:
// SELECT * FROM Test WHERE id = ? AND code IN (?, ?)
```

# Streaming results

Query results can be processed row per row, not having all loaded in memory.

In the following example we will count the number of country rows.

```
val countryCount: Either[List[Throwable], Long] =
  SQL"Select count(*) as c from Country".fold(0L) { (c, _) => c + 1 }
```

In previous example, either it's the successful `Long` result (right), or the list of errors (left).

Result can also be partially processed:

```
val books: Either[List[Throwable], List[String]] =
  SQL("Select name from Books").foldWhile(List[String]()) { (list, row) =>
    if (list.size == 100) (list -> false) // stop with `list`
    else (list := row[String]("name")) -> true // continue with one more name
  }
```

It's possible to use a custom streaming:

```
import anorm.{ Cursor, Row }

@annotation.tailrec
def go(c: Option[Cursor], l: List[String]): List[String] = c match {
  case Some(cursor) => {
    if (l.size == 100) l // custom limit, partial processing
    else {
      go(cursor.next, l :+ cursor.row[String]("name"))
    }
  }
  case _ => l
}

val books: Either[List[Throwable], List[String]] =
  SQL("Select name from Books").withResult(go(_, List.empty[String]))
```

The parsing API can be used with streaming, using `RowParser` on each cursor `.row`. The previous example can be updated with row parser.

```
import scala.util.{ Try, Success => TrySuccess, Failure }

// bookParser: anorm.RowParser[Book]

@annotation.tailrec
def go(c: Option[Cursor], l: List[Book]): Try[List[Book]] = c match {
  case Some(cursor) => {
    if (l.size == 100) l // custom limit, partial processing
    else {
      val parsed: Try[Book] = cursor.row.as(bookParser)

      parsed match {
        case TrySuccess(book) => // book successfully parsed from row
          go(cursor.next, l :+ book)
        case Failure(f) => /* fails to parse a book */ Failure(f)
      }
    }
  }
  case _ => l
}
```

```
val books: Either[List[Throwable], Try[List[Book]]] =
  SQL("Select name from Books").withResult(go(_, List.empty[Book]))

books match {
  case Left(streamingErrors) => ???
  case Right(Failure(parsingError)) => ???
  case Right(TrySuccess(listOfBooks)) => ???
}
```

## Multi-value support

Anorm parameter can be multi-value, like a sequence of string.
In such case, values will be prepared to be passed to JDBC.

```
// With default formatting (", " as separator)
SQL("SELECT * FROM Test WHERE cat IN ({categories})").
  on('categories -> Seq("a", "b", "c")
// -> SELECT * FROM Test WHERE cat IN ('a', 'b', 'c')

// With custom formatting
import anorm.SeqParameter
SQL("SELECT * FROM Test t WHERE {categories}").
  on('categories -> SeqParameter(
    values = Seq("a", "b", "c"), separator = " OR ",
    pre = "EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name=",
    post = ")"))
/* ->
SELECT * FROM Test t WHERE
EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='a')
OR EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='b')
OR EXISTS (SELECT NULL FROM j WHERE t.id=j.id AND name='c')
*/
```

On purpose multi-value parameter must strictly be declared with one of supported types (`List`, 'Seq`,`Set`,`SortedSet`,`Stream`,`Vector`and`SeqParameter`). Value of a subtype must be passed as parameter with supported:

```
val seq = IndexedSeq("a", "b", "c")
// seq is instance of Seq with inferred type IndexedSeq[String]

// Wrong
SQL"SELECT * FROM Test WHERE cat in ($seq)"
// Erroneous - No parameter conversion for IndexedSeq[T]

// Right
SQL"SELECT * FROM Test WHERE cat in (${seq: Seq[String]})"

// Right
val param: Seq[String] = seq
SQL"SELECT * FROM Test WHERE cat in ($param)"
```

In case parameter type is JDBC array (`java.sql.Array`), its value can be passed as`Array[T]`, as long as element type `T` is a supported one.

```
val arr = Array("fr", "en", "ja")
SQL"UPDATE Test SET langs = $arr".execute()
```

A column can also be multi-value if its type is JDBC array (`java.sql.Array`), then it can be mapped to either array or list (`Array[T]` or `List[T]`), provided type of element (`T`) is also supported in column mapping.

```
import anorm.SQL
import anorm.SqlParser.{ scalar, * }

// array and element parser
import anorm.Column.{ columnToArray, stringToArray }

val res: List[Array[String]] =
  SQL("SELECT str_arr FROM tbl").as(scalar[Array[String]].*)
```

Convenient parsing functions is also provided for arrays with `SqlParser.array[T](...)` and `SqlParser.list[T](...)`.

## Batch update

When you need to execute SQL statement several times with different arguments, batch query can be used (e.g. to execute a batch of insertions).

```
import anorm.BatchSql

val batch = BatchSql(
  "INSERT INTO books(title, author) VALUES({title}, {author}",
  Seq[NamedParameter]("title" -> "Play 2 for Scala", "author" -> Peter Hilton"),
  Seq[NamedParameter]("title" -> "Learning Play! Framework 2",
    "author" -> "Andy Petrella"))

val res: Array[Int] = batch.execute() // array of update count
```

Batch update must be called with at least one list of parameter. If a batch is executed with the mandatory first list of parameter being empty (e.g. `Nil`), only one statement will be executed (without parameter), which is equivalent to `SQL(statement).executeUpdate()`.

## Edge cases

Passing anything different from string or symbol as parameter name is now deprecated. For backward compatibility, you can activate `anorm.features.parameterWithUntypedName`.

```
import anorm.features.parameterWithUntypedName // activate

val untyped: Any = "name" // deprecated
SQL("SELECT * FROM Country WHERE {p}").on(untyped -> "val")
```

Type of parameter value should be visible, to be properly set on SQL statement. Using value as `Any`, explicitly or due to erasure, leads to compilation error `No implicit view available from Any => anorm.ParameterValue`.

```
// Wrong #1
val p: Any = "strAsAny"
SQL("SELECT * FROM test WHERE id={id}").
  on('id -> p) // Erroneous - No conversion Any => ParameterValue

// Right #1
val p = "strAsString"
```

```
SQL("SELECT * FROM test WHERE id={id}").on('id -> p)

// Wrong #2
val ps = Seq("a", "b", 3) // inferred as Seq[Any]
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
  on('a -> ps(0), // ps(0) - No conversion Any => ParameterValue
    'b -> ps(1),
    'c -> ps(2))

// Right #2
val ps = Seq[anorm.ParameterValue]("a", "b", 3) // Seq[ParameterValue]
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
  on('a -> ps(0), 'b -> ps(1), 'c -> ps(2))

// Wrong #3
val ts = Seq( // Seq[(String -> Any)] due to _2
  "a" -> "1", "b" -> "2", "c" -> 3)

val nps: Seq[NamedParameter] = ts map { t =>
  val p: NamedParameter = t; p
  // Erroneous - no conversion (String,Any) => NamedParameter
}

SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").on(nps :_*)

// Right #3
val nps = Seq[NamedParameter]( // Tuples as NamedParameter before Any
  "a" -> "1", "b" -> "2", "c" -> 3)
SQL("SELECT * FROM test WHERE (a={a} AND b={b}) OR c={c}").
  on(nps: _*) // Fail - no conversion (String,Any) => NamedParameter
```

For backward compatibility, you can activate such unsafe parameter conversion, accepting untyped `Any` value, with `anorm.features.anyToStatement`.

```
import anorm.features.anyToStatement

val d = new java.util.Date()
val params: Seq[NamedParameter] = Seq("mod" -> d, "id" -> "idv")
// Values as Any as heterogenous

SQL("UPDATE item SET last_modified = {mod} WHERE id = {id}").on(params:_*)
```

It's not recommanded because moreover hiding implicit resolution issues, as untyped it could lead to runtime conversion error, with values are passed on statement using `setObject`.

In previous example, `java.util.Date` is accepted as parameter but would with most databases raise error (as it's not valid JDBC type).

In some cases, some JDBC drivers returns a result set positioned on the first row rather than before this first row (e.g. stored procedured with Oracle JDBC driver).

To handle such edge-case, `.withResultSetOnFirstRow(true)` can be used as following.

```
SQL("EXEC stored_proc {arg}").on("arg" -> "val").withResultSetOnFirstRow(true)
SQL"""EXEC stored_proc ${"val"}""".withResultSetOnFirstRow(true)
```

# 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 transforms 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 WHERE id = {i}").
  on("i" -> "id").map({
    case Row("France", _) => France()
    case Row(name:String, pop:Int) if(pop > 1000000) => BigCountry(name)
    case Row(name:String, _) => SmallCountry(name)
  }).list
```

# Using for-comprehension

Row parser can be defined as for-comprehension, working with SQL result type. It can be useful when working with lot of column, possibly to work around case class limit.

```
import anorm.SqlParser.{ str, int }

val parser = for {
  a <- str("colA")
  b <- int("colB")
} yield (a -> b)

val parsed: (String, Int) = SELECT("SELECT * FROM Test").as(parser.single)
```

# Retrieving data along with execution context

Moreover data, query execution involves context information like SQL warnings that may be raised (and may be fatal or not), especially when working with stored SQL procedure.

Way to get context information along with query data is to use `executeQuery()`:

```
import anorm.SqlQueryResult

val res: SqlQueryResult = SQL("EXEC stored_proc {code}").
```

```
  on('code -> code).executeQuery()

// Check execution context (there warnings) before going on
val str: Option[String] =
  res.statementWarning match {
    case Some(warning) =>
      warning.printStackTrace()
      None

    case _ => res.as(scalar[String].singleOpt) // go on row parsing
  }
```

# Working with optional/nullable values

If a column in database can contain `Null` values, you need to parse it as an `Option`type. For example, the `indepYear` of the `Country` table is nullable, so you need to match it as`Option[Int]`:

```
case class Info(name: String, year: Option[Int])

val parser = str("name") ~ get[Option[Int]]("indepYear") map {
  case n ~ y => Info(n, y)
}

val res: List[Info] = SQL("Select name,indepYear from Country").as(parser.*)
```

If you try to match this column as `Int` it won't be able to parse `Null` values. 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]`.

A nullable parameter is also passed as `Option[T]`, `T` being parameter base type (see*Parameters* section thereafter).

Passing directly `None` for a NULL value is not supported, as inferred as`Option[Nothing]` (`Nothing` being unsafe for a parameter value). In this case,`Option.empty[T]` must be used.

```
// OK:

SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> Some("Title"))

val title1 = Some("Title1")
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> title1)

val title2: Option[String] = None
// None inferred as Option[String] on assignment
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> title2)
```

```
// Not OK:
SQL("INSERT INTO Test(title) VALUES({title})").on("title" -> None)

// OK:
SQL"INSERT INTO Test(title) VALUES(${Option.empty[String]})"
```

# 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._`

## Getting a single result

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)
```

If expected single result is optional (0 or 1 row), then `scalar` parser can be combined with `singleOpt`:

```
val name: Option[String] =
  SQL"SELECT name FROM Country WHERE code = $code" as scalar[String].singleOpt
```

## Getting a single optional result

Let's say you want to retrieve the country_id from the country name, but the query might return null. We'll use the singleOpt parser :

```
val countryId: Option[Long] =
  SQL("SELECT country_id FROM Country C WHERE C.country='France'")
  .as(scalar[Long].singleOpt)
```

## Getting a more complex result

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 rather 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:

```
val parser = 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(parser.flatten.*)
```

A `RowParser` can be combined with any function to applied it with extracted columns.

```
import anorm.SqlParser.{ int, str, to }

def display(name: String, population: Int): String =
  s"The population in $name is of $population."

val parser = str("name") ~ int("population") map (to(display _))
```

**Note:** The mapping function must be partially applied (syntax `fn _`) when given `to` the parser (see SLS 6.26.2, 6.26.5 - Eta expansion).

If list should not be empty, `parser.+` can be used instead of `parser.*`.

Anorm is providing parser combinators other than the most common `~` one: `~>`, `<~`.

```
import anorm.{ SQL, SqlParser }, SqlParser.{ int, str }

// Combinator ~>
val String = SQL("SELECT * FROM test").as((int("id") ~> str("val")).single)
  // row has to have an int column 'id' and a string 'val' one,
  // keeping only 'val' in result

val Int = SQL("SELECT * FROM test").as((int("id") <~ str("val")).single)
  // row has to have an int column 'id' and a string 'val' one,
  // keeping only 'id' in result
```

## A more complicated example

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 MySQL world sample database, you will get:

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

# JDBC mappings

As already seen in this documentation, Anorm provides builtins converters between JDBC and JVM types.

## Column parsers

Following table describes which JDBC numeric types (getters on `java.sql.ResultSet`, first column) can be parsed to which Java/Scala types (e.g. integer column can be read as double value).

| ↓JDBC / JVM→ | BigDecimal[1] | BigInteger[2] | Boolean | Byte | Double | Float | Int | Long | Short |
|---|---|---|---|---|---|---|---|---|---|
| BigDecimal[1] | Yes | Yes | No | No | Yes | No | Yes | Yes | No |
| BigInteger[2] | Yes | Yes | No | No | Yes | Yes | Yes | Yes | No |
| Boolean | No | No | Yes | Yes | No | No | Yes | Yes | Yes |
| Byte | Yes | No | No | Yes | Yes | Yes | No | No | Yes |
| Double | Yes | No | No | No | Yes | No | No | No | No |
| Float | Yes | No | No | No | Yes | Yes | No | No | No |
| Int | Yes | Yes | No | No | Yes | Yes | Yes | Yes | No |
| Long | Yes | Yes | No | No | No | No | Yes | Yes | No |
| Short | Yes | No | No | Yes | Yes | Yes | No | No | Yes |

- 1. Types `java.math.BigDecimal` and `scala.math.BigDecimal`.
- 1. Types `java.math.BigInteger` and `scala.math.BigInt`.

The second table shows mappings for the other supported types.

| ↓JDBC / JVM→ | Array[T][3] | Char | List[3] | String | UUID[4] |
|---|---|---|---|---|---|
| Array[5] | Yes | No | Yes | No | No |
| Clob | No | Yes | No | Yes | No |
| Iterable[6] | Yes | No | Yes | No | No |

| ↓JDBC / JVM→ | Array[T][3] | Char | List[3] | String | UUID[4] |
|---|---|---|---|---|---|
| Long | No | No | No | No | No |
| String | No | Yes | No | Yes | Yes |
| UUID | No | No | No | No | Yes |

- 1. Array which type `T` of elements is supported.
- 1. Type `java.util.UUID`.
- 1. Type `java.sql.Array`.
- 1. Type `java.lang.Iterable[_]`.

Optional column can be parsed as `Option[T]`, as soon as `T` is supported.

Binary data types are also supported.

| ↓JDBC / JVM→ | Array[Byte] | InputStream[1] |
|---|---|---|
| Array[Byte] | Yes | Yes |
| Blob[2] | Yes | Yes |
| Clob[3] | No | No |
| InputStream[4] | Yes | Yes |
| Reader[5] | No | No |

- 1. Type `java.io.InputStream`.
- 1. Type `java.sql.Blob`.
- 1. Type `java.sql.Clob`.
- 1. Type `java.io.Reader`.

CLOBs/TEXTs can be extracted as so:

```
SQL("Select name,summary from Country")().map {
  case Row(name: String, summary: java.sql.Clob) => name -> summary
}
```

Here we specifically chose to use `map`, as we want an exception if the row isn't in the format we expect.

Extracting binary data is similarly possible:

```
SQL("Select name,image from Country")().map {
  case Row(name: String, image: Array[Byte]) => name -> image
}
```

For types where column support is provided by Anorm, convenient functions are available to ease writing custom parsers. Each of these functions parses column either by name or index (> 1).

```
import anorm.SqlParser.str // String function

str("column")
str(1/* columnIndex)
```

| Type | Function |
|---|---|
| Array[Byte] | byteArray |
| Boolean | bool |
| Byte | byte |
| Date | date |
| Double | double |
| Float | float |
| InputStream[1] | binaryStream |
| Int | int |
| Long | long |
| Short | short |

| Type | Function |
|---|---|
| String | str |

- 1. Type `java.io.InputStream`.

The Joda and Java 8 temporal types are also supported.

| ↓JDBC / JVM→ | Date[1] | DateTime[2] | Instant[3] |
|---|---|---|---|
| Date | Yes | Yes | Yes |
| Long | Yes | Yes | Yes |
| Timestamp | Yes | Yes | Yes |
| Timestamp wrapper[5] | Yes | Yes | Yes |

- 1. Type `java.util.Date`.
- 1. Types `org.joda.time.DateTime`, `java.time.LocalDateTime` and `java.time.ZonedDateTime`.
- 1. Type `org.joda.time.Instant` and `java.time.Instant` (see Java 8).
- 1. Any type with a getter `getTimestamp` returning a `java.sql.Timestamp`.

It's possible to add custom mapping, for example if underlying DB doesn't support boolean datatype and returns integer instead. To do so, you have to provide a new implicit conversion for `Column[T]`, where `T` is the target Scala type:

```
import anorm.Column

// Custom conversion from JDBC column to Boolean
implicit def columnToBoolean: Column[Boolean] =
  Column.nonNull1 { (value, meta) =>
    val MetaDataItem(qualified, nullable, clazz) = meta
    value match {
      case bool: Boolean => Right(bool) // Provided-default case
      case bit: Int      => Right(bit == 1) // Custom conversion
      case _             => Left(TypeDoesNotMatch(s"Cannot convert $value:
${value.asInstanceOf[AnyRef].getClass} to Boolean for column $qualified"))
    }
  }
```

## Parameters

The following table indicates how JVM types are mapped to JDBC parameter types:

| JVM | JDBC | Nullable |
| --- | --- | --- |
| Array[T][1] | Array[2] with `T` mapping for each element | Yes |
| BigDecimal[3] | BigDecimal | Yes |
| BigInteger[4] | BigDecimal | Yes |
| Boolean[5] | Boolean | Yes |
| Byte[6] | Byte | Yes |
| Char[7]/String | String | Yes |
| Date/Timestamp | Timestamp | Yes |
| Double[8] | Double | Yes |
| Float[9] | Float | Yes |
| Int[10] | Int | Yes |
| List[T] | Multi-value[11], with `T` mapping for each element | No |
| Long[12] | Long | Yes |
| Object[13] | Object | Yes |
| Option[T] | `T` being type if some defined value | No |
| Seq[T] | Multi-value, with `T` mapping for each element | No |
| Set[T][14] | Multi-value, with `T` mapping for each element | No |
| Short[15] | Short | Yes |

| JVM | JDBC | Nullable |
| --- | --- | --- |
| SortedSet[T][16] | Multi-value, with `T` mapping for each element | No |
| Stream[T] | Multi-value, with `T` mapping for each element | No |
| UUID | String[17] | No |
| Vector | Multi-value, with `T` mapping for each element | No |

- 1. Type Scala `Array[T]`.
- 1. Type `java.sql.Array`.
- 1. Types `java.math.BigDecimal` and `scala.math.BigDecimal`.
- 1. Types `java.math.BigInteger` and `scala.math.BigInt`.
- 1. Types `Boolean` and `java.lang.Boolean`.
- 1. Types `Byte` and `java.lang.Byte`.
- 1. Types `Char` and `java.lang.Character`.
- 1. Types compatible with `java.util.Date`, and any wrapper type with `getTimestamp: java.sql.Timestamp`.
- 1. Types `Double` and `java.lang.Double`.
- 1. Types `Float` and `java.lang.Float`.
- 1. Types `Int` and `java.lang.Integer`.
- 1. Types `Long` and `java.lang.Long`.
- 1. Type `anorm.Object`, wrapping opaque object.
- 1. Multi-value parameter, with one JDBC placeholder (`?`) added for each element.
- 1. Type `scala.collection.immutable.Set`.
- 1. Types `Short` and `java.lang.Short`.

1. Type `scala.collection.immutable.SortedSet`.

- 1. Not-null value extracted using `.toString`.

Passing `None` for a nullable parameter is deprecated, and typesafe `Option.empty[T]` must be use instead.

Large and stream parameters are also supported.

| JVM | JDBC |
| --- | --- |
| Array[Byte] | Long varbinary |
| Blob[1] | Blob |
| InputStream[2] | Long varbinary |
| Reader[3] | Long varchar |

- 1. Type `java.sql.Blob`
- 1. Type `java.io.InputStream`
- 1. Type `java.io.Reader`

Joda and Java 8 temporal types are supported as parameters:

| JVM | JDBC |
| --- | --- |
| DateTime[1] | Timestamp |
| Instant[2] | Timestamp |
| LocalDateTime[3] | Timestamp |
| ZonedDateTime[4] | Timestamp |

- 1. Type `org.joda.time.DateTime`.
- 1. Type `org.joda.time.Instant` and `java.time.Instant`.
- 1. Type `org.joda.time.LocalDateTime`.

1. Type `org.joda.time.ZonedDateTime`

Custom or specific DB conversion for parameter can also be provided:

```
import java.sql.PreparedStatement
import anorm.ToStatement

// Custom conversion to statement for type T
implicit def customToStatement: ToStatement[T] = new ToStatement[T] {
  def set(statement: PreparedStatement, i: Int, value: T): Unit =
    ??? // Sets |value| on |statement|
}
```

If involved type accept `null` value, it must be appropriately handled in conversion.
The`NotNullGuard` trait can be used to explicitly refuse `null` values in parameter conversion: `new ToStatement[T] with NotNullGuard { /* ... */ }`.

DB specific parameter can be explicitly passed as opaque value.
In this case at your own risk, `setObject` will be used on statement.

```
val anyVal: Any = myVal
SQL("UPDATE t SET v = {opaque}").on('opaque -> anorm.Object(anyVal))
```

**Next:** Integrating with other database access libraries

# Integrating with other database libraries

You can use any **SQL** database access library you like with Play, and 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", O PrimaryKey, O AutoInc)
  def name = column[String]("name", O 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
```

**Next:** Using the Cache

# 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.

# Importing the Cache API

Add `cache` into your dependencies list. For example, in `build.sbt`:

```
libraryDependencies ++= Seq(
  cache,
  ...
)
```

# Accessing the Cache API

The cache API is provided by the CacheApi object, and can be injected into your component like any other dependency. For example:

```
import play.api.cache._
import play.api.mvc._
import javax.inject.Inject

class Application @Inject() (cache: CacheApi) extends Controller {

}
```

**Note:** The API is intentionally minimal to allow several implementation 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 either store data in cache:

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

And then retrieve it later:

```
val maybeUser: Option[User] = cache.get[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.getOrElse[User]("item.key") {
  User.findById(connectedUser)
}
```

You can specify an expiry duration by passing a duration, by default the duration is infinite:

```
import scala.concurrent.duration._

cache.set("item.key", connectedUser, 5.minutes)
```

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

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

# Accessing different caches

It is possible to access different caches. The default cache is called `play`, and can be configured by creating a file called `ehcache.xml`. Additional caches may be configured with different configurations, or even implementations.

If you want to access multiple different ehcache caches, then you'll need to tell Play to bind them in `application.conf`, like so:

```
play.cache.bindCaches = ["db-cache", "user-cache", "session-cache"]
```

Now to access these different caches, when you inject them, use the NamedCachequalifier on your dependency, for example:

```
import play.api.cache._
import play.api.mvc._
import javax.inject.Inject

class Application @Inject()(
   @NamedCache("session-cache") sessionCache: CacheApi
) extends Controller {

}
```

# 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.

The Cached class helps you build cached actions.

```
import play.api.cache.Cached
import javax.inject.Inject

class Application @Inject() (cached: Cached) extends Controller {

}
```

You can cache the result of an action using a fixed key like `"homePage"`.

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

If results vary, you can cache each result using a different key. In this example, each user has a different cached result.

```
def userProfile = Authenticated {
  user =>
    cached(req => "profile." + user) {
      Action {
```

```
      Ok(views.html.profile(User.find(user)))
    }
  }
}
```

## Control caching

You can easily control what you want to cache or what you want to exclude from the cache.

You may want to only cache 200 Ok results.

```
def get(index: Int) = cached.status(_ => "/resource/"+ index, 200) {
  Action {
    if (index > 0) {
      Ok(Json.obj("id" -> index))
    } else {
      NotFound
    }
  }
}
```

Or cache 404 Not Found only for a couple of minutes

```
def get(index: Int) = {
  val caching = cached
    .status(_ => "/resource/"+ index, 200)
    .includeStatus(404, 600)

  caching {
    Action {
      if (index % 2 == 1) {
        Ok(Json.obj("id" -> index))
      } else {
        NotFound
      }
    }
  }
}
```

# Custom implementations

It is possible to provide a custom implementation of the CacheApi that either replaces, or sits along side the default implementation.
To replace the default implementation, you'll need to disable the default implementation by setting the following in `application.conf`:

```
play.modules.disabled += "play.api.cache.EhCacheModule"
```

Then simply implement `CacheApi` and bind it in the DI container.
To provide an implementation of the cache API in addition to the default implementation, you can either create a custom qualifier, or reuse the `NamedCache` qualifier to bind the implementation.

**Next:** Calling WebServices

# The Play WS API

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

There are two important parts to using the WS API: making a request, and processing the response. We'll discuss how to make both GET and POST HTTP requests first, and then show how to process the response from WS. Finally, we'll discuss some common use cases.

---

# Making a Request

To use WS, first add `ws` to your `build.sbt` file:

```
libraryDependencies ++= Seq(
  ws
)
```

Now any controller or component that wants to use WS will have to declare a dependency on the `WSClient`:

```
import javax.inject.Inject
import scala.concurrent.Future

import play.api.mvc._
import play.api.libs.ws._

class Application @Inject() (ws: WSClient) extends Controller {

}
```

We've called the `WSClient` instance `ws`, all the following examples will assume this name. To build an HTTP request, you start with `ws.url()` to specify the URL.

```
val request: WSRequest = ws.url(url)
```

This returns a WSRequest that you can use to specify various HTTP options, such as setting headers. You can chain calls together to construct complex requests.

```
val complexRequest: WSRequest =
  request.withHeaders("Accept" -> "application/json")
    .withRequestTimeout(10000)
    .withQueryString("search" -> "play")
```

You end by calling a method corresponding to the HTTP method you want to use. This ends the chain, and uses all the options defined on the built request in the `WSRequest`.

```
val futureResponse: Future[WSResponse] = complexRequest.get()
```

This returns a `Future[WSResponse]` where the Response contains the data returned from the server.

## Request with authentication

If you need to use HTTP authentication, you can specify it in the builder, using a username, password, and an AuthScheme. Valid case objects for the AuthScheme are`BASIC`, `DIGEST`, `KERBEROS`, `NONE`, `NTLM`, and `SPNEGO`.

```
ws.url(url).withAuth(user, password, WSAuthScheme.BASIC).get()
```

## Request with follow redirects

If an HTTP call results in a 302 or a 301 redirect, you can automatically follow the redirect without having to make another call.

```
ws.url(url).withFollowRedirects(true).get()
```

## Request with query parameters

Parameters can be specified as a series of key/value tuples.

```
ws.url(url).withQueryString("paramKey" -> "paramValue").get()
```

## Request with additional headers

Headers can be specified as a series of key/value tuples.

```
ws.url(url).withHeaders("headerKey" -> "headerValue").get()
```

If you are sending plain text in a particular format, you may want to define the content type explicitly.

```
ws.url(url).withHeaders("Content-Type" -> "application/xml").post(xmlString)
```

## Request with virtual host

A virtual host can be specified as a string.

```
ws.url(url).withVirtualHost("192.168.1.1").get()
```

## Request with timeout

If you wish to specify a request timeout, you can use `withRequestTimeout` to set a value in milliseconds. A value of `-1` can be used to set an infinite timeout.

```
ws.url(url).withRequestTimeout(5000).get()
```

## Submitting form data

To post url-form-encoded data a `Map[String, Seq[String]]` needs to be passed into`post`.

```
ws.url(url).post(Map("key" -> Seq("value")))
```

## Submitting JSON data

The easiest way to post JSON data is to use the JSON library.

```
import play.api.libs.json._
```

```
val data = Json.obj(
  "key1" -> "value1",
  "key2" -> "value2"
)
val futureResponse: Future[WSResponse] = ws.url(url).post(data)
```

## Submitting XML data

The easiest way to post XML data is to use XML literals. XML literals are convenient, but not very fast. For efficiency, consider using an XML view template, or a JAXB library.

```
val data = <person>
  <name>Steve</name>
  <age>23</age>
</person>
val futureResponse: Future[WSResponse] = ws.url(url).post(data)
```

# Processing the Response

Working with the Response is easily done by mapping inside the Future.

The examples given below have some common dependencies that will be shown once here for brevity.

Whenever an operation is done on a `Future`, an implicit execution context must be available - this declares which thread pool the callback to the future should run in. The default Play execution context is often sufficient:

```
implicit val context = play.api.libs.concurrent.Execution.Implicits.defaultContext
```

The examples also use the folowing case class for serialization / deserialization:

```
case class Person(name: String, age: Int)
```

## Processing a response as JSON

You can process the response as a JSON object by calling `response.json`.

```
val futureResult: Future[String] = ws.url(url).get().map {
  response =>
    (response.json \ "person" \ "name").as[String]
}
```

The JSON library has a useful feature that will map an implicit `Reads[T]` directly to a class:

```
import play.api.libs.json._

implicit val personReads = Json.reads[Person]

val futureResult: Future[JsResult[Person]] = ws.url(url).get().map {
  response => (response.json \ "person").validate[Person]
}
```

## Processing a response as XML

You can process the response as an XML literal by calling `response.xml`.

```
val futureResult: Future[scala.xml.NodeSeq] = ws.url(url).get().map {
  response =>
    response.xml \ "message"
}
```

## Processing large responses

Calling `get()` or `post()` will cause the body of the request to be loaded into memory before the response is made available. When you are downloading with large, multi-gigabyte files, this may result in unwelcome garbage collection or even out of memory errors.

`WS` lets you use the response incrementally by using an iteratee.

The `stream()` and `getStream()` methods on `WSRequest` return `Future[(WSResponseHeaders, Enumerator[Array[Byte]])]`. The enumerator contains the response body.

Here is a trivial example that uses an iteratee to count the number of bytes returned by the response:

```
import play.api.libs.iteratee._

// Make the request
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
  ws.url(url).getStream()

val bytesReturned: Future[Long] = futureResponse.flatMap {
  case (headers, body) =>
    // Count the number of bytes returned
    body |>>> Iteratee.fold(0l) { (total, bytes) =>
      total + bytes.length
    }
}
```

Of course, usually you won't want to consume large bodies like this, the more common use case is to stream the body out to another location. For example, to stream the body to a file:

```
import play.api.libs.iteratee._

// Make the request
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
  ws.url(url).getStream()

val downloadedFile: Future[File] = futureResponse.flatMap {
  case (headers, body) =>
    val outputStream = new FileOutputStream(file)

    // The iteratee that writes to the output stream
    val iteratee = Iteratee.foreach[Array[Byte]] { bytes =>
      outputStream.write(bytes)
    }
```

```
    // Feed the body into the iteratee
    (body |>>> iteratee).andThen {
      case result =>
        // Close the output stream whether there was an error or not
        outputStream.close()
        // Get the result or rethrow the error
        result.get
    }.map(_ => file)
}
```

Another common destination for response bodies is to stream them through to a response that this server is currently serving:

```
def downloadFile = Action.async {

  // Make the request
  ws.url(url).getStream().map {
    case (response, body) =>

      // Check that the response was successful
      if (response.status == 200) {

        // Get the content type
        val contentType = response.headers.get("Content-Type").flatMap(_.headOption)
          .getOrElse("application/octet-stream")

        // If there's a content length, send that, otherwise return the body chunked
        response.headers.get("Content-Length") match {
          case Some(Seq(length)) =>
            Ok.feed(body).as(contentType).withHeaders("Content-Length" -> length)
          case _ =>
            Ok.chunked(body).as(contentType)
        }
      } else {
        BadGateway
      }
  }
}
```

`POST` and `PUT` calls require manually calling the `withMethod` method, eg:

```
val futureResponse: Future[(WSResponseHeaders, Enumerator[Array[Byte]])] =
  ws.url(url).withMethod("PUT").withBody("some body").stream()
```

# Common Patterns and Use Cases

## Chaining WS calls

Using for comprehensions is a good way to chain WS calls in a trusted environment. You should use for comprehensions together with Future.recover to handle possible failure.

```
val futureResponse: Future[WSResponse] = for {
  responseOne <- ws.url(urlOne).get()
  responseTwo <- ws.url(responseOne.body).get()
  responseThree <- ws.url(responseTwo.body).get()
} yield responseThree

futureResponse.recover {
  case e: Exception =>
    val exceptionData = Map("error" -> Seq(e.getMessage))
    ws.url(exceptionUrl).post(exceptionData)
}
```

### Using in a controller

When making a request from a controller, you can map the response to a `Future[Result]`. This can be used in combination with Play's `Action.async` action builder, as described in Handling Asynchronous Results.

```
def wsAction = Action.async {
  ws.url(url).get().map { response =>
    Ok(response.body)
  }
}
status(wsAction(FakeRequest())) must_== OK
```

# Using WSClient

WSClient is a wrapper around the underlying AsyncHttpClient. It is useful for defining multiple clients with different profiles, or using a mock.

You can define a WS client directly from code without having it injected by WS, and then use it implicitly with `WS.clientUrl()`:

```
import play.api.libs.ws.ning._

implicit val sslClient = NingWSClient()
// close with sslClient.close() when finished with client
val response = WS.clientUrl(url).get()
```

NOTE: if you instantiate a NingWSClient object, it does not use the WS module lifecycle, and so will not be automatically closed in `Application.onStop`. Instead, the client must be manually shutdown using `client.close()` when processing has completed. This will release the underlying ThreadPoolExecutor used by AsyncHttpClient. Failure to close the client may result in out of memory exceptions (especially if you are reloading an application frequently in development mode).

or directly:

```
val response = sslClient.url(url).get()
```

Or use a magnet pattern to match up certain clients automatically:

```
object PairMagnet {
  implicit def fromPair(pair: (WSClient, java.net.URL)) =
    new WSRequestMagnet {
      def apply(): WSRequest = {
        val (client, netUrl) = pair
        client.url(netUrl.toString)
      }
    }
}

import scala.language.implicitConversions
import PairMagnet._

val exampleURL = new java.net.URL(url)
val response = WS.url(ws -> exampleURL).get()
```

By default, configuration happens in `application.conf`, but you can also set up the builder directly from configuration:

```
import com.typesafe.config.ConfigFactory
import play.api._
import play.api.libs.ws._
import play.api.libs.ws.ning._

val configuration = Configuration.reference ++ Configuration(ConfigFactory.parseString(
  """
    |ws.followRedirects = true
  """.stripMargin))

// If running in Play, environment should be injected
val environment = Environment(new File("."), this.getClass.getClassLoader, Mode.Prod)

val parser = new WSConfigParser(configuration, environment)
val config = new NingWSClientConfig(wsClientConfig = parser.parse())
val builder = new NingAsyncHttpClientConfigBuilder(config)
```

You can also get access to the underlying async client.

```
import com.ning.http.client.AsyncHttpClient

val client: AsyncHttpClient = ws.underlying
```

This is important in a couple of cases. WS has a couple of limitations that require access to the client:

- `WS` does not support multi part form upload directly. You can use the underlying client with RequestBuilder.addBodyPart.
- `WS` does not support streaming body upload. In this case, you should use the`FeedableBodyGenerator` provided by AsyncHttpClient.

# Configuring WS

Use the following properties in `application.conf` to configure the WS client:

- `play.ws.followRedirects`: Configures the client to follow 301 and 302 redirects*(default is **true**)*.
- `play.ws.useProxyProperties`: To use the system http proxy settings(http.proxyHost, http.proxyPort) *(default is **true**)*.
- `play.ws.useragent`: To configure the User-Agent header field.
- `play.ws.compressionEnabled`: Set it to true to use gzip/deflater encoding *(default is**false**)*.

## Configuring WS with SSL

To configure WS for use with HTTP over SSL/TLS (HTTPS), please see Configuring WS SSL.

## Configuring Timeouts

There are 3 different timeouts in WS. Reaching a timeout causes the WS request to interrupt.

- `play.ws.timeout.connection`: The maximum time to wait when connecting to the remote host *(default is **120 seconds**)*.
- `play.ws.timeout.idle`: The maximum time the request can stay idle (connection is established but waiting for more data) *(default is **120 seconds**)*.
- `play.ws.timeout.request`: The total time you accept a request to take (it will be interrupted even if the remote host is still sending data) *(default is **120 seconds**)*.

The request timeout can be overridden for a specific connection with `withRequestTimeout()` (see "Making a Request" section).

## Configuring AsyncHttpClientConfig

The following advanced settings can be configured on the underlying AsyncHttpClientConfig.

Please refer to the AsyncHttpClientConfig Documentation for more information.

- `play.ws.ning.allowPoolingConnection`
- `play.ws.ning.allowSslConnectionPool`
- `play.ws.ning.ioThreadMultiplier`
- `play.ws.ning.maxConnectionsPerHost`
- `play.ws.ning.maxConnectionsTotal`
- `play.ws.ning.maxConnectionLifeTime`
- `play.ws.ning.idleConnectionInPoolTimeout`
- `play.ws.ning.webSocketIdleTimeout`
- `play.ws.ning.maxNumberOfRedirects`
- `play.ws.ning.maxRequestRetry`
- `play.ws.ning.disableUrlEncoding`

**Next:** Connecting to OpenID services

# 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 to redirect the user.
3. The user confirms the authorization on his OpenID provider, and gets redirected back to 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).

# Usage

To use OpenID, first add `ws` to your `build.sbt` file:

```
libraryDependencies ++= Seq(
  ws
)
```

Now any controller or component that wants to use OpenID will have to declare a dependency on the OpenIdClient:

```
import javax.inject.Inject
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import play.api.libs.openid._

class Application @Inject() (openIdClient: OpenIdClient) extends Controller {

}
```

We've called the `OpenIdClient` instance `openIdClient`, all the following examples will assume this name.

# OpenID in Play

The OpenID API has two important functions:

- `OpenIdClient.redirectURL` calculates the URL where you should redirect the user. It involves fetching the user's OpenID page asynchronously, this is why it returns a`Future[String]`. If the OpenID is invalid, the returned `Future` will fail.
- `OpenIdClient.verifiedId` needs a `RequestHeader` and inspects it to establish the user information, including his verified OpenID. It will do a call to the OpenID server asynchronously to check the authenticity of the information, returning a future ofUserInfo. If the information is not correct or if the server check is false (for example if the redirect URL has been forged), the returned `Future` will fail.
  If the `Future` fails, you can define a fallback, which redirects back the user to the login page or return a `BadRequest`.

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

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

def loginPost = Action.async { implicit request =>
  Form(single(
    "openid" -> nonEmptyText
  )).bindFromRequest.fold({ error =>
    Logger.info("bad request " + error.toString)
    Future.successful(BadRequest(error.toString))
  }, { openId =>
    openIdClient.redirectURL(openId, routes.Application.openIdCallback.absoluteURL())
      .map(url => Redirect(url))
      .recover { case t: Throwable => Redirect(routes.Application.login)}
  })
}

def openIdCallback = Action.async { implicit request =>
  openIdClient.verifiedId(request).map(info => Ok(info.id + "\n" + info.attributes))
    .recover {
    case t: Throwable =>
      // 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:

```
openIdClient.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.

**Next:** Accessing resources protected by OAuth

# 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.

# Usage

To use OAuth, first add `ws` to your `build.sbt` file:

```
libraryDependencies ++= Seq(
  ws
)
```

# 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),
    true)

  def authenticate = Action { request =>
    request.getQueryString("oauth_verifier").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)
    }
  }
```

```
}
object Application extends Controller {

 def timeline = Action.async { implicit request =>
  Twitter.sessionTokenPair match {
   case Some(credentials) => {
    WS.url("https://api.twitter.com/1.1/statuses/home_timeline.json")
     .sign(OAuthCalculator(Twitter.KEY, credentials))
     .get
     .map(result => Ok(result.json))
   }
   case _ => Future.successful(Redirect(routes.Twitter.authenticate))
  }
 }
}
```

**Next:** Integrating with Akka

# 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 can work with several containers called actor systems. 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.

## Writing actors

To start using Akka, you need to write an actor. Below is a simple actor that simply says hello to whoever asks it to.

```
import akka.actor._

object HelloActor {
 def props = Props[HelloActor]
```

```
  case class SayHello(name: String)
}

class HelloActor extends Actor {
  import HelloActor._

  def receive = {
    case SayHello(name: String) =>
      sender() ! "Hello, " + name
  }
}
```

This actor follows a few Akka conventions:

- The messages it sends/receives, or its *protocol*, are defined on its companion object
- It also defines a `props` method on its companion object that returns the props for creating it

## Creating and using actors

To create and/or use an actor, you need an `ActorSystem`. This can be obtained by declaring a dependency on an ActorSystem, like so:

```
import play.api.mvc._
import akka.actor._
import javax.inject._

import actors.HelloActor

@Singleton
class Application @Inject() (system: ActorSystem) extends Controller {

  val helloActor = system.actorOf(HelloActor.props, "hello-actor")

  //...
}
```

The `actorOf` method is used to create a new actor. Notice that we've declared this controller to be a singleton. This is necessary since we are creating the actor and storing a reference to it, if the controller was not scoped as singleton, this would mean a new actor would be created every time the controller was created, which would ultimate throw an exception because you can't have two actors in the same system with the same name.

## Asking things of actors

The most basic thing that you can do with an actor is send it a message. When you send a message to an actor, there is no response, it's fire and forget. This is also known as the*tell* pattern.

In a web application however, the *tell* pattern is often not useful, since HTTP is a protocol that has requests and responses. In this case, it is much more likely that you will want to use the *ask* pattern. The ask pattern returns a `Future`, which you can then map to your own result type.

Below is an example of using our `HelloActor` with the ask pattern:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.duration._
import akka.pattern.ask
implicit val timeout = 5.seconds

def sayHello(name: String) = Action.async {
  (helloActor ? SayHello(name)).mapTo[String].map { message =>
    Ok(message)
  }
}
```

A few things to notice:

- The ask pattern needs to be imported, and then this provides a `?` operator on the actor.
- The return type of the ask is a `Future[Any]`, usually the first thing you will want to do after asking actor is map that to the type you are expecting, using the `mapTo` method.
- An implicit timeout is needed in scope - the ask pattern must have a timeout. If the actor takes longer than that to respond, the returned future will be completed with a timeout error.

# Dependency injecting actors

If you prefer, you can have Guice instantiate your actors and bind actor refs to them for your controllers and components to depend on.

For example, if you wanted to have an actor that depended on the Play configuration, you might do this:

```
import akka.actor._
import javax.inject._
import play.api.Configuration

object ConfiguredActor {
  case object GetConfig
}

class ConfiguredActor @Inject() (configuration: Configuration) extends Actor {
  import ConfiguredActor._

  val config = configuration.getString("my.config").getOrElse("none")

  def receive = {
    case GetConfig =>
      sender() ! config
  }
}
```

Play provides some helpers to help providing actor bindings. These allow the actor itself to be dependency injected, and allows the actor ref for the actor to be injected into other components. To bind an actor using these helpers, create a module as described in

thedependency injection documentation, then mix in the `AkkaGuiceSupport` trait and use the `bindActor` method to bind the actor:

```
import com.google.inject.AbstractModule
import play.api.libs.concurrent.AkkaGuiceSupport

import actors.ConfiguredActor

class MyModule extends AbstractModule with AkkaGuiceSupport {
  def configure = {
    bindActor[ConfiguredActor]("configured-actor")
  }
}
```

This actor will both be named `configured-actor`, and will also be qualified with the`configured-actor` name for injection. You can now depend on the actor in your controllers and other components:

```
import play.api.mvc._
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import javax.inject._
import actors.ConfiguredActor._
import scala.concurrent.ExecutionContext
import scala.concurrent.duration._

@Singleton
class Application @Inject() (@Named("configured-actor") configuredActor: ActorRef)
                    (implicit ec: ExecutionContext) extends Controller {

  implicit val timeout: Timeout = 5.seconds

  def getConfig = Action.async {
    (configuredActor ? GetConfig).mapTo[String].map { message =>
      Ok(message)
    }
  }
}
```

## Dependency injecting child actors

The above is good for injecting root actors, but many of the actors you create will be child actors that are not bound to the lifecycle of the Play app, and may have additional state passed to them.

In order to assist in dependency injecting child actors, Play utilises Guice'sAssistedInject support.

Let's say you have the following actor, which depends configuration to be injected, plus a key:

```
import akka.actor._
```

```
import javax.inject._
import com.google.inject.assistedinject.Assisted
import play.api.Configuration

object ConfiguredChildActor {
  case object GetConfig

  trait Factory {
    def apply(key: String): Actor
  }
}

class ConfiguredChildActor @Inject() (configuration: Configuration,
    @Assisted key: String) extends Actor {
  import ConfiguredChildActor._

  val config = configuration.getString(key).getOrElse("none")

  def receive = {
    case GetConfig =>
      sender() ! config
  }
}
```

Note that the `key` parameter is declared to be `@Assisted`, this tells that it's going to be manually provided.

We've also defined a `Factory` trait, this takes the `key`, and returns an `Actor`. We won't implement this, Guice will do that for us, providing an implementation that not only passes our `key` parameter, but also locates the `Configuration` dependency and injects that.

Since the trait just returns an `Actor`, when testing this actor we can inject a factory that returns any actor, for example this allows us to inject a mocked child actor, instead of the actual one.

Now, the actor that depends on this can extend `InjectedActorSupport`, and it can depend on the factory we created:

```
import akka.actor._
import javax.inject._
import play.api.libs.concurrent.InjectedActorSupport

object ParentActor {
  case class GetChild(key: String)
}

class ParentActor @Inject() (
    childFactory: ConfiguredChildActor.Factory
) extends Actor with InjectedActorSupport {
  import ParentActor._

  def receive = {
    case GetChild(key: String) =>
      val child: ActorRef = injectedChild(childFactory(key), key)
      sender() ! child
```

```
  }
}
```

It uses the `injectedChild` to create and get a reference to the child actor, passing in the key.

Finally, we need to bind our actors. In our module, we use the `bindActorFactory`method to bind the parent actor, and also bind the child factory to the child implementation:

```
import com.google.inject.AbstractModule
import play.api.libs.concurrent.AkkaGuiceSupport

import actors._

class MyModule extends AbstractModule with AkkaGuiceSupport {
  def configure = {
    bindActor[ParentActor]("parent-actor")
    bindActorFactory[ConfiguredChildActor, ConfiguredChildActor.Factory]
  }
}
```

This will get Guice to automatically bind an instance of`ConfiguredChildActor.Factory`, which will provide an instance of `Configuration`to `ConfiguredChildActor` when it's instantiated.

# 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.actor.default-dispatcher.fork-join-executor.parallelism-max = 64
akka.actor.debug.receive = on
```

For Akka logging configuration, see configuring logging.

## Changing configuration prefix

In case you want to use the `akka.*` settings for another Akka actor system, you can tell Play to load its Akka settings from another location.

```
play.akka.config = "my-akka"
```

Now settings will be read from the `my-akka` prefix instead of the `akka` prefix.

```
my-akka.actor.default-dispatcher.fork-join-executor.pool-size-max = 64
my-akka.actor.debug.receive = on
```

## Built-in actor system name

By default the name of the Play actor system is `application`. You can change this via an entry in the `conf/application.conf`:

```
play.akka.actor-system = "custom-name"
```

**Note:** This feature is useful if you want to put your play application ActorSystem in an Akka cluster.

# 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 300 microseconds:

```
import scala.concurrent.duration._

val cancellable = system.scheduler.schedule(
  0.microseconds, 300.microseconds, testActor, "tick")
```

**Note:** This example uses implicit conversions defined in `scala.concurrent.duration`to convert numbers to `Duration` objects with various time units.

Similarly, to run a block of code 10 milliseconds from now:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
system.scheduler.scheduleOnce(10.milliseconds) {
  file.delete()
}
```

# Using your own Actor system

While we recommend you use the built in actor system, as it sets up everything such as the correct classloader, lifecycle hooks, etc, there is nothing stopping you from using your own actor system. It is important however to ensure you do the following:

- Register a stop hook to shut the actor system down when Play shuts down
- Pass in the correct classloader from the Play Environment otherwise Akka won't be able to find your applications classes
- Ensure that either you change the location that Play reads it's akka configuration from using `play.akka.config`, or that you don't read your akka configuration from the default `akka` config, as this will cause problems such as when the systems try to bind to the same remote ports

**Next:** Internationalization

# 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:

```
play.i18n.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.i18n.Messages` argument retrieved from the current scope. This implicit value contains both the language to use and (essentially) the internationalized messages.

The simplest way to get such an implicit value is to use the `I18nSupport` trait. For instance you can use it as follows in your controllers:

```
import play.api.i18n.I18nSupport
class MyController(val messagesApi: MessagesApi) extends Controller with I18nSupport {
  // ...
```

The `I18nSupport` trait gives you an implicit `Messages` value as long as there is a `Lang` or a `RequestHeader` in the implicit scope.

**Note:** If you have a `RequestHeader` in the implicit scope, it will use the preferred language extracted from the `Accept-Language` header and matching one of the `MessagesApi` supported languages. You should add a `Messages` implicit parameter to your template like this: `@()(implicit messages: Messages)`.

**Note:** Also, Play "knows" out of the box how to inject a `MessagesApi` value (that uses the `DefaultMessagesApi` implementation), so you can just annotate your controller with the `@javax.inject.Inject` annotation and let Play automatically wire the components for you.

# Messages format

Messages are 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)
```

# Notes on apostrophes

Since Messages uses `java.text.MessageFormat`, please be aware that single quotes are used as a meta-character for escaping parameter substitutions.

For example, if you have the following messages defined:

```
info.error=You aren't logged in!
example.formatting=When using MessageFormat, '{0}' is replaced with the first parameter.
```

you should expect the following results:

```
Messages("info.error") == "You aren't logged in!"
Messages("example.formatting") == "When using MessageFormat, '{0}' is replaced with the first parameter."
```

# 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(", "))
}
```

**Next:** Testing your application

# Testing your application

Writing tests for your application can be an involved process. Play offers integration with both ScalaTest and specs2 and provides helpers and application stubs to make testing your application as easy as possible. For the details of using your preferred test framework with Play, see the pages on ScalaTest or specs2.

- Testing your Application with ScalaTest
- Testing your Application with specs2

# Advanced testing

Play provides a number of helpers for testing specific parts of an application.

- Testing using Guice dependency injection
- Testing with databases

**Next:** Testing with ScalaTest

# Testing your application with ScalaTest

Writing tests for your application can be an involved process. Play provides helpers and application stubs, and ScalaTest provides an integration library, ScalaTest + Play, to make testing your application as easy as possible.

# Overview

The location for tests is in the “test” folder.

You can run tests from the Play console.

- To run all tests, run `test`.
- To run only one test class, run `test-only` followed by the name of the class, i.e.,`test-only my.namespace.MySpec`.
- To run only the tests that have failed, run `test-quick`.
- To run tests continually, run a command with a tilde in front, i.e. `~test-quick`.
- To access test helpers such as `FakeApplication` in console, run `test:console`.

Testing in Play is based on SBT, and a full description is available in the testing SBTchapter.

# Using ScalaTest + Play

To use *ScalaTest + Play*, you’ll need to add it to your build, by changing `build.sbt` like this:

```
libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "2.2.1" % "test",
  "org.scalatestplus" %% "play" % "1.4.0-M3" % "test",
)
```

You do not need to add ScalaTest to your build explicitly. The proper version of ScalaTest will be brought in automatically as a transitive dependency of *ScalaTest + Play*. You will,

however, need to select a version of *ScalaTest + Play* that matches your Play version. You can do so by checking the Versions, Versions, Versions page for*ScalaTest + Play*.
In *ScalaTest + Play*, you define test classes by extending the `PlaySpec` trait. Here's an example:

```
import collection.mutable.Stack
import org.scalatestplus.play._

class StackSpec extends PlaySpec {

 "A Stack" must {
  "pop values in last-in-first-out order" in {
   val stack = new Stack[Int]
   stack.push(1)
   stack.push(2)
   stack.pop() mustBe 2
   stack.pop() mustBe 1
  }
  "throw NoSuchElementException if an empty stack is popped" in {
   val emptyStack = new Stack[Int]
   a [NoSuchElementException] must be thrownBy {
    emptyStack.pop()
   }
  }
 }
}
```

You can alternatively define your own base classes instead of using `PlaySpec`.
You can run your tests with Play itself, or in IntelliJ IDEA (using the Scala plugin) or in Eclipse (using the Scala IDE and the ScalaTest Eclipse plugin). Please see theIDE page for more details.

## Matchers

`PlaySpec` mixes in ScalaTest's `MustMatchers`, so you can write assertions using ScalaTest's matchers DSL:

```
import play.api.test.Helpers._

"Hello world" must endWith ("world")
```

For more information, see the documentation for `MustMatchers`.

## Mockito

You can use mocks to isolate unit tests against external dependencies. For example, if your class depends on an external `DataService` class, you can feed appropriate data to your class without instantiating a `DataService` object.
ScalaTest provides integration with Mockito via its `MockitoSugar` trait.
To use Mockito, mix `MockitoSugar` into your test class and then use the Mockito library to mock dependencies:

```
case class Data(retrievalDate: java.util.Date)

trait DataService {
 def findData: Data
```

```
}
import org.scalatest._
import org.scalatest.mock.MockitoSugar
import org.scalatestplus.play._

import org.mockito.Mockito._

class ExampleMockitoSpec extends PlaySpec with MockitoSugar {

  "MyService#isDailyData" should {
    "return true if the data is from today" in {
      val mockDataService = mock[DataService]
      when(mockDataService.findData) thenReturn Data(new java.util.Date())

      val myService = new MyService() {
        override def dataService = mockDataService
      }

      val actual = myService.isDailyData
      actual mustBe true
    }
  }
}
```

Mocking is especially useful for testing the public methods of classes. Mocking objects and private methods is possible, but considerably harder.

# Unit Testing Models

Play does not require models to use a particular database data access layer. However, if the application uses Anorm or Slick, then frequently the Model will have a reference to database access internally.

```
import anorm._
import anorm.SqlParser._

case class User(id: String, name: String, email: String) {
  def roles = DB.withConnection { implicit connection =>
    ...
  }
}
```

For unit testing, this approach can make mocking out the `roles` method tricky.

A common approach is to keep the models isolated from the database and as much logic as possible, and abstract database access behind a repository layer.

```
case class Role(name:String)
```

```
case class User(id: String, name: String, email:String)
trait UserRepository {
  def roles(user:User) : Set[Role]
}
class AnormUserRepository extends UserRepository {
  import anorm._
  import anorm.SqlParser._

  def roles(user:User) : Set[Role] = {
    ...
  }
}
```

and then access them through services:

```
class UserService(userRepository : UserRepository) {

  def isAdmin(user:User) : Boolean = {
    userRepository.roles(user).contains(Role("ADMIN"))
  }
}
```

In this way, the `isAdmin` method can be tested by mocking out the `UserRepository` reference and passing it into the service:

```
class UserServiceSpec extends PlaySpec with MockitoSugar {

  "UserService#isAdmin" should {
    "be true when the role is admin" in {
      val userRepository = mock[UserRepository]
      when(userRepository.roles(any[User])) thenReturn Set(Role("ADMIN"))

      val userService = new UserService(userRepository)

      val actual = userService.isAdmin(User("11", "Steve", "user@example.org"))
      actual mustBe true
    }
  }
}
```

# Unit Testing Controllers

When defining controllers as objects, they can be trickier to unit test. In Play this can be alleviated by dependency injection. Another way to finesse unit testing with a controller declared as a object is to use a trait with an explicitly typed self reference to the controller:

```
trait ExampleController {
  this: Controller =>

  def index() = Action {
    Ok("ok")
  }
}
```

```
object ExampleController extends Controller with ExampleController
```

and then test the trait:

```
import scala.concurrent.Future

import org.scalatest._
import org.scalatestplus.play._

import play.api.mvc._
import play.api.test._
import play.api.test.Helpers._

class ExampleControllerSpec extends PlaySpec with Results {

 class TestController() extends Controller with ExampleController

 "Example Page#index" should {
  "should be valid" in {
   val controller = new TestController()
   val result: Future[Result] = controller.index().apply(FakeRequest())
   val bodyText: String = contentAsString(result)
   bodyText mustBe "ok"
  }
 }
}
```

When testing POST requests with, for example, JSON bodies, you won't be able to use the pattern shown above (`apply(fakeRequest)`); instead you should use `call()` on the `testController`:

```
trait WithControllerAndRequest {
 val testController = new Controller with ApiController

 def fakeRequest(method: String = "GET", route: String = "/") = FakeRequest(method, route)
  .withHeaders(
   ("Date", "2014-10-05T22:00:00"),
   ("Authorization", "username=bob;hash=foobar==")
 )
}

"REST API" should {
 "create a new user" in new WithControllerAndRequest {
  val request = fakeRequest("POST", "/user").withJsonBody(Json.parse(
   s"""{"first_name": "Alice",
    | "last_name": "Doe",
    | "credentials": {
    |  "username": "alice",
    |  "password": "secret"
    | }
    |}""".stripMargin))
  val apiResult = call(testController.createUser, request)
  status(apiResult) mustEqual CREATED
  val jsonResult = contentAsJson(apiResult)
```

```
    ObjectId.isValid((jsonResult \ "id").as[String]) mustBe true

    // now get the real thing from the DB and check it was created with the correct values:
    val newbie = Dao().findByUsername("alice").get
    newbie.id.get.toString mustEqual (jsonResult \ "id").as[String]
    newbie.firstName mustEqual "Alice"
  }
}
```

# Unit Testing EssentialAction

Testing `Action` or `Filter` can require to test an an `EssentialAction` (more information about what an EssentialAction is)
For this, the test `Helpers.call` can be used like that:

```
class ExampleEssentialActionSpec extends PlaySpec {

  "An essential action" should {
    "can parse a JSON body" in {
      val action: EssentialAction = Action { request =>
        val value = (request.body.asJson.get \ "field").as[String]
        Ok(value)
      }

      val request = FakeRequest(POST, "/").withJsonBody(Json.parse("""{ "field": "value" }"""))

      val result = call(action, request)

      status(result) mustEqual OK
      contentAsString(result) mustEqual "value"
    }
  }
}
```

**Next:** Writing functional tests with ScalaTest

# Writing functional tests with ScalaTest

Play provides a number of classes and convenience methods that assist with functional testing. Most of these can be found either in the `play.api.test` package or in the`Helpers` object. The *ScalaTest + Play* integration library builds on this testing support for ScalaTest.
You can access all of Play's built-in test support and *ScalaTest + Play* with the following imports:

```
import org.scalatest._
import play.api.test._
import play.api.test.Helpers._
import org.scalatestplus.play._
```

# FakeApplication

Play frequently requires a running `Application` as context: it is usually provided from `play.api.Play.current`.

To provide an environment for tests, Play provides a `FakeApplication` class which can be configured with a different `Global` object, additional configuration, or even additional plugins.

```
val fakeApplicationWithGlobal = FakeApplication(withGlobal = Some(new GlobalSettings() {
  override def onStart(app: Application) { println("Hello world!") }
}))
```

If all or most tests in your test class need a `FakeApplication`, and they can all share the same `FakeApplication`, mix in trait `OneAppPerSuite`. You can access the `FakeApplication` from the `app` field. If you need to customize the `FakeApplication`, override `app` as shown in this example:

```
class ExampleSpec extends PlaySpec with OneAppPerSuite {

  // Override app if you need a FakeApplication with other than
  // default parameters.
  implicit override lazy val app: FakeApplication =
    FakeApplication(
      additionalConfiguration = Map("ehcacheplugin" -> "disabled")
    )

  "The OneAppPerSuite trait" must {
    "provide a FakeApplication" in {
      app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
    }
    "start the FakeApplication" in {
      Play.maybeApplication mustBe Some(app)
    }
  }
}
```

If you need each test to get its own `FakeApplication`, instead of sharing the same one, use `OneAppPerTest` instead:

```
class ExampleSpec extends PlaySpec with OneAppPerTest {

  // Override app if you need a FakeApplication with other than
  // default parameters.
  implicit override def newAppForTest(td: TestData): FakeApplication =
    FakeApplication(
      additionalConfiguration = Map("ehcacheplugin" -> "disabled")
    )

  "The OneAppPerTest trait" must {
```

```
    "provide a new FakeApplication for each test" in {
      app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
    }
    "start the FakeApplication" in {
      Play.maybeApplication mustBe Some(app)
    }
  }
}
```

The reason *ScalaTest + Play* provides both `OneAppPerSuite` and `OneAppPerTest` is to allow you to select the sharing strategy that makes your tests run fastest. If you want application state maintained between successive tests, you'll need to use`OneAppPerSuite`. If each test needs a clean slate, however, you could either use`OneAppPerTest` or use `OneAppPerSuite`, but clear any state at the end of each test. Furthermore, if your test suite will run fastest if multiple test classes share the same application, you can define a master suite that mixes in `OneAppPerSuite` and nested suites that mix in `ConfiguredApp`, as shown in the example in the documentation for`ConfiguredApp`. You can use whichever strategy makes your test suite run the fastest.

# Testing with a server

Sometimes you want to test with the real HTTP stack. If all tests in your test class can reuse the same server instance, you can mix in `OneServerPerSuite` (which will also provide a new `FakeApplication` for the suite):

```
class ExampleSpec extends PlaySpec with OneServerPerSuite {

  // Override app if you need a FakeApplication with other than
  // default parameters.
  implicit override lazy val app: FakeApplication =
    FakeApplication(
      additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
      withRoutes = {
        case ("GET", "/") => Action { Ok("ok") }
      }
    )

  "test server logic" in {
    val myPublicAddress =  s"localhost:$port"
    val testPaymentGatewayURL = s"http://$myPublicAddress"
    // The test payment gateway requires a callback to this server before it returns a result...
    val callbackURL = s"http://$myPublicAddress/callback"
    // await is from play.api.test.FutureAwaits
    val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
callbackURL).get())

    response.status mustBe (OK)
  }
}
```

If all tests in your test class require separate server instance, use `OneServerPerTest` instead (which will also provide a new `FakeApplication` for the suite):

```
class ExampleSpec extends PlaySpec with OneServerPerTest {

 // Override newAppForTest if you need a FakeApplication with other than
 // default parameters.
 override def newAppForTest(testData: TestData): FakeApplication =
  new FakeApplication(
    additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
    withRoutes = {
      case ("GET", "/") => Action { Ok("ok") }
    }
  )

 "The OneServerPerTest trait" must {
  "test server logic" in {
    val myPublicAddress =  s"localhost:$port"
    val testPaymentGatewayURL = s"http://$myPublicAddress"
    // The test payment gateway requires a callback to this server before it returns a result...
    val callbackURL = s"http://$myPublicAddress/callback"
    // await is from play.api.test.FutureAwaits
    val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
callbackURL).get())

    response.status mustBe (OK)
  }
 }
}
```

The `OneServerPerSuite` and `OneServerPerTest` traits provide the port number on which the server is running as the `port` field. By default this is 19001, however you can change this either overriding `port` or by setting the system property `testserver.port`. This can be useful for integrating with continuous integration servers, so that ports can be dynamically reserved for each build.

You can also customize the `FakeApplication` by overriding `app`, as demonstrated in the previous examples.

Lastly, if allowing multiple test classes to share the same server will give you better performance than either the `OneServerPerSuite` or `OneServerPerTest` approaches, you can define a master suite that mixes in `OneServerPerSuite` and nested suites that mix in `ConfiguredServer`, as shown in the example in the documentation for`ConfiguredServer`.

# Testing with a web browser

The *ScalaTest + Play* library builds on ScalaTest's Selenium DSL to make it easy to test your Play applications from web browsers.

To run all tests in your test class using a same browser instance, mix `OneBrowserPerSuite` into your test class. You'll also need to mix in a `BrowserFactory` trait that will provide a Selenium web driver: one of `ChromeFactory`, `FirefoxFactory`, `HtmlUnitFactory`, `InternetExplorerFactory`, `SafariFactory`.

In addition to mixing in a `BrowserFactory`, you will need to mix in a `ServerProvider` trait that provides a `TestServer`: one of `OneServerPerSuite`, `OneServerPerTest`, or `ConfiguredServer`.

For example, the following test class mixes in `OneServerPerSuite` and `HtmUnitFactory`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with OneBrowserPerSuite with
HtmlUnitFactory {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
  FakeApplication(
    additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
    withRoutes = {
      case ("GET", "/testing") =>
        Action(
          Results.Ok(
            "<html>" +
              "<head><title>Test Page</title></head>" +
              "<body>" +
              "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
              "</body>" +
              "</html>"
          ).as("text/html")
        )
    }
  )

 "The OneBrowserPerTest trait" must {
  "provide a web driver" in {
    go to (s"http://localhost:$port/testing")
    pageTitle mustBe "Test Page"
    click on find(name("b")).value
    eventually { pageTitle mustBe "scalatest" }
  }
 }
}
```

If each of your tests requires a new browser instance, use `OneBrowserPerTest` instead. As with `OneBrowserPerSuite`, you'll need to also mix in a `ServerProvider` and `BrowserFactory`:

```
class ExampleSpec extends PlaySpec with OneServerPerTest with OneBrowserPerTest with HtmlUnitFactory
{

 // Override newAppForTest if you need a FakeApplication with other than
```

```
  // default parameters.
  override def newAppForTest(testData: TestData): FakeApplication =
    new FakeApplication(
      additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
      withRoutes = {
        case ("GET", "/testing") =>
          Action(
            Results.Ok(
              "<html>" +
                "<head><title>Test Page</title></head>" +
                "<body>" +
                "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
                "</body>" +
                "</html>"
            ).as("text/html")
          )
      }
    )

  "The OneBrowserPerTest trait" must {
    "provide a web driver" in {
      go to (s"http://localhost:$port/testing")
      pageTitle mustBe "Test Page"
      click on find(name("b")).value
      eventually { pageTitle mustBe "scalatest" }
    }
  }
}
```

If you need multiple test classes to share the same browser instance, mix `OneBrowserPerSuite` into a master suite and `ConfiguredBrowser` into multiple nested suites. The nested suites will all share the same web browser. For an example, see the documentation for trait `ConfiguredBrowser`.

# Running the same tests in multiple browsers

If you want to run tests in multiple web browsers, to ensure your application works correctly in all the browsers you support, you can use traits `AllBrowsersPerSuite` or `AllBrowsersPerTest`. Both of these traits declare a `browsers` field of type `IndexedSeq[BrowserInfo]` and an abstract `sharedTests` method that takes a `BrowserInfo`. The `browsers` field indicates which browsers you want your tests to run in. The default is Chrome, Firefox, Internet Explorer, `HtmlUnit`, and Safari. You can override `browsers` if the default doesn't fit your needs. You place tests you want run in multiple browsers in the `sharedTests` method, placing the name of the browser at the end of each test name. (The browser name is

available from the `BrowserInfo` passed into `sharedTests`.) Here's an example that uses `AllBrowsersPerSuite`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerSuite {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
   FakeApplication(
     additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
     withRoutes = {
       case ("GET", "/testing") =>
         Action(
           Results.Ok(
             "<html>" +
               "<head><title>Test Page</title></head>" +
               "<body>" +
               "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
               "</body>" +
               "</html>"
           ).as("text/html")
         )
     }
   )

 def sharedTests(browser: BrowserInfo) = {
   "The AllBrowsersPerSuite trait" must {
     "provide a web driver " + browser.name in {
       go to (s"http://localhost:$port/testing")
       pageTitle mustBe "Test Page"
       click on find(name("b")).value
       eventually { pageTitle mustBe "scalatest" }
     }
   }
 }
}
```

All tests declared by `sharedTests` will be run with all browsers mentioned in the`browsers` field, so long as they are available on the host system. Tests for any browser that is not available on the host system will be canceled automatically. Note that you need to append the `browser.name` manually to the test name to ensure each test in the suite has a unique name (which is required by ScalaTest). If you leave that off, you'll get a duplicate-test-name error when you run your tests.

`AllBrowsersPerSuite` will create a single instance of each type of browser and use that for all the tests declared in `sharedTests`. If you want each test to have its own, brand new browser instance, use `AllBrowsersPerTest` instead:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerTest {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
   FakeApplication(
```

```
    additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
    withRoutes = {
      case ("GET", "/testing") =>
        Action(
          Results.Ok(
            "<html>" +
              "<head><title>Test Page</title></head>" +
              "<body>" +
              "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
              "</body>" +
              "</html>"
          ).as("text/html")
        )
    }
  )

 def sharedTests(browser: BrowserInfo) = {
   "The AllBrowsersPerTest trait" must {
     "provide a web driver"  + browser.name in {
       go to (s"http://localhost:$port/testing")
       pageTitle mustBe "Test Page"
       click on find(name("b")).value
       eventually { pageTitle mustBe "scalatest" }
     }
   }
 }
}
```

Although both `AllBrowsersPerSuite` and `AllBrowsersPerTest` will cancel tests for unavailable browser types, the tests will show up as canceled in the output. To can clean up the output, you can exclude web browsers that will never be available by overriding `browsers`, as shown in this example:

```
class ExampleOverrideBrowsersSpec extends PlaySpec with OneServerPerSuite with AllBrowsersPerSuite {

 override lazy val browsers =
   Vector(
     FirefoxInfo(firefoxProfile),
     ChromeInfo
   )

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
   FakeApplication(
     additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
     withRoutes = {
       case ("GET", "/testing") =>
         Action(
           Results.Ok(
             "<html>" +
               "<head><title>Test Page</title></head>" +
               "<body>" +
               "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
```

```
            "</body>" +
            "</html>"
          ).as("text/html")
        )
    }
  )

 def sharedTests(browser: BrowserInfo) = {
  "The AllBrowsersPerSuite trait" must {
    "provide a web driver"  + browser.name in {
      go to (s"http://localhost:$port/testing")
      pageTitle mustBe "Test Page"
      click on find(name("b")).value
      eventually { pageTitle mustBe "scalatest" }
    }
  }
 }
}
```

The previous test class will only attempt to run the shared tests with Firefox and Chrome (and cancel tests automatically if a browser is not available).

# PlaySpec

`PlaySpec` provides a convenience “super Suite” ScalaTest base class for Play tests, You get `WordSpec`, `MustMatchers`, `OptionValues`,
and `WsScalaTestClient`automatically by extending `PlaySpec`:

```
class ExampleSpec extends PlaySpec with OneServerPerSuite with  ScalaFutures with IntegrationPatience {

 // Override app if you need a FakeApplication with other than
 // default parameters.
 implicit override lazy val app: FakeApplication =
   FakeApplication(
     additionalConfiguration = Map("ehcacheplugin" -> "disabled"),
     withRoutes = {
       case ("GET", "/testing") =>
         Action(
           Results.Ok(
             "<html>" +
               "<head><title>Test Page</title></head>" +
               "<body>" +
               "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
               "</body>" +
               "</html>"
           ).as("text/html")
         )
     }
   )

 "WsScalaTestClient's" must {
```

```
  "wsUrl works correctly" in {
    val futureResult = wsUrl("/testing").get
    val body = futureResult.futureValue.body
    val expectedBody =
      "<html>" +
        "<head><title>Test Page</title></head>" +
        "<body>" +
        "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
        "</body>" +
        "</html>"
    assert(body == expectedBody)
  }

  "wsCall works correctly" in {
    val futureResult = wsCall(Call("get", "/testing")).get
    val body = futureResult.futureValue.body
    val expectedBody =
      "<html>" +
        "<head><title>Test Page</title></head>" +
        "<body>" +
        "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
        "</body>" +
        "</html>"
    assert(body == expectedBody)
  }
 }
}
```

You can mix any of the previously mentioned traits into `PlaySpec`.

# When different tests need different fixtures

In all the test classes shown in previous examples, all or most tests in the test class required the same fixtures. While this is common, it is not always the case. If different tests in the same test class need different fixtures, mix in trait `MixedFixtures`. Then give each individual test the fixture it needs using one of these no-arg functions: App,Server, Chrome, Firefox, HtmlUnit, InternetExplorer, or Safari.
You cannot mix `MixedFixtures` into `PlaySpec` because `MixedFixtures`requires a ScalaTest `fixture.Suite` and `PlaySpec` is just a regular `Suite`. If you want a convenient base class for mixed fixtures, extend `MixedPlaySpec`instead. Here's an example:

```
// MixedPlaySpec already mixes in MixedFixtures
class ExampleSpec extends MixedPlaySpec {

 // Some helper methods
 def fakeApp[A](elems: (String, String)*) = FakeApplication(additionalConfiguration = Map(elems:_*),
```

```
  withRoutes = {
    case ("GET", "/testing") =>
      Action(
        Results.Ok(
          "<html>" +
            "<head><title>Test Page</title></head>" +
            "<body>" +
            "<input type='button' name='b' value='Click Me' onclick='document.title=\"scalatest\"' />" +
            "</body>" +
            "</html>"
        ).as("text/html")
      )
  })
def getConfig(key: String)(implicit app: Application) = app.configuration.getString(key)

// If a test just needs a FakeApplication, use "new App":
"The App function" must {
  "provide a FakeApplication" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new App(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
}

// If a test needs a FakeApplication and running TestServer, use "new Server":
"The Server function" must {
  "provide a FakeApplication" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new Server(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
  import Helpers._
  "send 404 on a bad request" in new Server {
    import java.net._
    val url = new URL("http://localhost:" + port + "/boom")
    val con = url.openConnection().asInstanceOf[HttpURLConnection]
    try con.getResponseCode mustBe 404
    finally con.disconnect()
  }
}

// If a test needs a FakeApplication, running TestServer, and Selenium
// HtmlUnit driver use "new HtmlUnit":
"The HtmlUnit function" must {
  "provide a FakeApplication" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
```

```
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled"))
{
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new HtmlUnit(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
  import Helpers._
  "send 404 on a bad request" in new HtmlUnit {
    import java.net._
    val url = new URL("http://localhost:" + port + "/boom")
    val con = url.openConnection().asInstanceOf[HttpURLConnection]
    try con.getResponseCode mustBe 404
    finally con.disconnect()
  }
  "provide a web driver" in new HtmlUnit(fakeApp()) {
    go to ("http://localhost:" + port + "/testing")
    pageTitle mustBe "Test Page"
    click on find(name("b")).value
    eventually { pageTitle mustBe "scalatest" }
  }
 }

 // If a test needs a FakeApplication, running TestServer, and Selenium
 // Firefox driver use "new Firefox":
 "The Firefox function" must {
  "provide a FakeApplication" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new Firefox(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
  import Helpers._
  "send 404 on a bad request" in new Firefox {
    import java.net._
    val url = new URL("http://localhost:" + port + "/boom")
    val con = url.openConnection().asInstanceOf[HttpURLConnection]
    try con.getResponseCode mustBe 404
    finally con.disconnect()
  }
  "provide a web driver" in new Firefox(fakeApp()) {
    go to ("http://localhost:" + port + "/testing")
    pageTitle mustBe "Test Page"
    click on find(name("b")).value
    eventually { pageTitle mustBe "scalatest" }
  }
 }
```

```
// If a test needs a FakeApplication, running TestServer, and Selenium
// Safari driver use "new Safari":
"The Safari function" must {
  "provide a FakeApplication" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new Safari(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
  import Helpers._
  "send 404 on a bad request" in new Safari {
    import java.net._
    val url = new URL("http://localhost:" + port + "/boom")
    val con = url.openConnection().asInstanceOf[HttpURLConnection]
    try con.getResponseCode mustBe 404
    finally con.disconnect()
  }
  "provide a web driver" in new Safari(fakeApp()) {
    go to ("http://localhost:" + port + "/testing")
    pageTitle mustBe "Test Page"
    click on find(name("b")).value
    eventually { pageTitle mustBe "scalatest" }
  }
}

// If a test needs a FakeApplication, running TestServer, and Selenium
// Chrome driver use "new Chrome":
"The Chrome function" must {
  "provide a FakeApplication" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
    app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
  }
  "make the FakeApplication available implicitly" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
    getConfig("ehcacheplugin") mustBe Some("disabled")
  }
  "start the FakeApplication" in new Chrome(fakeApp("ehcacheplugin" -> "disabled")) {
    Play.maybeApplication mustBe Some(app)
  }
  import Helpers._
  "send 404 on a bad request" in new Chrome {
    import java.net._
    val url = new URL("http://localhost:" + port + "/boom")
    val con = url.openConnection().asInstanceOf[HttpURLConnection]
    try con.getResponseCode mustBe 404
    finally con.disconnect()
  }
  "provide a web driver" in new Chrome(fakeApp()) {
    go to ("http://localhost:" + port + "/testing")
    pageTitle mustBe "Test Page"
    click on find(name("b")).value
    eventually { pageTitle mustBe "scalatest" }
```

```
    }
  }

  // If a test needs a FakeApplication, running TestServer, and Selenium
  // InternetExplorer driver use "new InternetExplorer":
  "The InternetExplorer function" must {
    "provide a FakeApplication" in new InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
      app.configuration.getString("ehcacheplugin") mustBe Some("disabled")
    }
    "make the FakeApplication available implicitly" in new InternetExplorer(fakeApp("ehcacheplugin" ->
"disabled")) {
      getConfig("ehcacheplugin") mustBe Some("disabled")
    }
    "start the FakeApplication" in new InternetExplorer(fakeApp("ehcacheplugin" -> "disabled")) {
      Play.maybeApplication mustBe Some(app)
    }
    import Helpers._
    "send 404 on a bad request" in new InternetExplorer {
      import java.net._
      val url = new URL("http://localhost:" + port + "/boom")
      val con = url.openConnection().asInstanceOf[HttpURLConnection]
      try con.getResponseCode mustBe 404
      finally con.disconnect()
    }
    "provide a web driver" in new InternetExplorer(fakeApp()) {
      go to ("http://localhost:" + port + "/testing")
      pageTitle mustBe "Test Page"
      click on find(name("b")).value
      eventually { pageTitle mustBe "scalatest" }
    }
  }

  // If a test does not need any special fixtures, just
  // write "in { () => ..."
  "Any old thing" must {
    "be doable without much boilerplate" in { () =>
      1 + 1 mustEqual 2
    }
  }
}
```

# 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 new App {
  val html = views.html.index("Coco")

  contentAsString(html) must include ("Hello Coco")
}
```

# Testing a controller

You can call any `Action` code by providing a `FakeRequest`:

```
import scala.concurrent.Future

import org.scalatest._
import org.scalatestplus.play._

import play.api.mvc._
import play.api.test._
import play.api.test.Helpers._

class ExampleControllerSpec extends PlaySpec with Results {

 class TestController() extends Controller with ExampleController

 "Example Page#index" should {
  "should be valid" in {
   val controller = new TestController()
   val result: Future[Result] = controller.index().apply(FakeRequest())
   val bodyText: String = contentAsString(result)
   bodyText mustBe "ok"
  }
 }
}
```

Technically, you don't need `WithApplication` here, although it wouldn't hurt anything to have it.

# Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
"respond to the index Action" in new App(fakeApplication) {
 val Some(result) = route(FakeRequest(GET, "/Bob"))

 status(result) mustEqual OK
 contentType(result) mustEqual Some("text/html")
 charset(result) mustEqual Some("utf-8")
 contentAsString(result) must include ("Hello Bob")
}
```

# Testing a model

If you are using an SQL database, you can replace the database connection with an in-memory instance of an H2 database using `inMemoryDatabase`.

```
val appWithMemoryDatabase = FakeApplication(additionalConfiguration = inMemoryDatabase("test"))
"run an application" in new App(appWithMemoryDatabase) {

 val Some(macintosh) = Computer.findById(21)
```

```
  macintosh.name mustEqual "Macintosh"
  macintosh.introduced.value mustEqual "1984-01-24"
}
```

# Testing WS calls

If you are calling a web service, you can use `WSTestClient`. There are two calls available, `wsCall` and `wsUrl` that will take a Call or a string, respectively. Note that they expect to be called in the context of `WithApplication`.

```
wsCall(controllers.routes.Application.index()).get()
wsUrl("http://localhost:9000").get()
```

**Next:** Testing with specs2

# Testing your application with specs2

Writing tests for your application can be an involved process. Play provides a default test framework for you, and provides helpers and application stubs to make testing your application as easy as possible.

# Overview

The location for tests is in the “test” folder. There are two sample test files created in the test folder which can be used as templates.

You can run tests from the Play console.

- To run all tests, run `test`.
- To run only one test class, run `test-only` followed by the name of the class i.e. `test-only my.namespace.MySpec`.
- To run only the tests that have failed, run `test-quick`.
- To run tests continually, run a command with a tilde in front, i.e. `~test-quick`.
- To access test helpers such as `FakeApplication` in console, run `test:console`.

Testing in Play is based on SBT, and a full description is available in the testing SBTchapter.

# Using specs2

To use Play's specs2 support, add the Play specs2 dependency to your build as a test scoped dependency:

```
libraryDependencies += specs2 % Test
```

In specs2, tests are organized into specifications, which contain examples which run the system under test through various different code paths.
Specifications extend the `Specification` trait and are using the should/in format:

```
import org.specs2.mutable._

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")
  }
 }
}
```

Specifications can be run in either IntelliJ IDEA (using the Scala plugin) or in Eclipse (using the Scala IDE). Please see the IDE page for more details.
NOTE: Due to a bug in the presentation compiler, tests must be defined in a specific format to work with Eclipse:

- The package must be exactly the same as the directory path.
- The specification must be annotated with `@RunWith(classOf[JUnitRunner])`.

Here is a valid specification for Eclipse:

```
package models // this file must be in a directory called "models"

import org.specs2.mutable._
import org.specs2.runner._
import org.junit.runner._

@RunWith(classOf[JUnitRunner])
class ApplicationSpec extends Specification {
  ...
}
```

## Matchers

When you use an example, you must return an example result. Usually, you will see a statement containing a `must`:

```
"Hello world" must endWith("world")
```

The expression that follows the `must` keyword are known as `matchers`. Matchers return an example result, typically Success or Failure. The example will not compile if it does not return a result.
The most useful matchers are the match results. These are used to check for equality, determine the result of Option and Either, and even check if exceptions are thrown.
There are also optional matchers that allow for XML and JSON matching in tests.

## Mockito

Mocks are used to isolate unit tests against external dependencies. For example, if your class depends on an external `DataService` class, you can feed appropriate data to your class without instantiating a `DataService` object.
Mockito is integrated into specs2 as the default mocking library.

To use Mockito, add the following import:

```
import org.specs2.mock._
```

You can mock out references to classes like so:

```
trait DataService {
  def findData: Data
}

case class Data(retrievalDate: java.util.Date)
import org.specs2.mock._
import org.specs2.mutable._

import java.util._

class ExampleMockitoSpec extends Specification with Mockito {

  "MyService#isDailyData" should {
    "return true if the data is from today" in {
      val mockDataService = mock[DataService]
      mockDataService.findData returns Data(retrievalDate = new java.util.Date())

      val myService = new MyService() {
        override def dataService = mockDataService
      }

      val actual = myService.isDailyData
      actual must equalTo(true)
    }
  }

}
```

Mocking is especially useful for testing the public methods of classes. Mocking objects and private methods is possible, but considerably harder.

# Unit Testing Models

Play does not require models to use a particular database data access layer. However, if the application uses Anorm or Slick, then frequently the Model will have a reference to database access internally.

```
import anorm._
import anorm.SqlParser._

case class User(id: String, name: String, email: String) {
  def roles = DB.withConnection { implicit connection =>
    ...
  }
}
```

For unit testing, this approach can make mocking out the `roles` method tricky.

A common approach is to keep the models isolated from the database and as much logic as possible, and abstract database access behind a repository layer.

```
case class Role(name:String)

case class User(id: String, name: String, email:String)
trait UserRepository {
  def roles(user:User) : Set[Role]
}
class AnormUserRepository extends UserRepository {
  import anorm._
  import anorm.SqlParser._

  def roles(user:User) : Set[Role] = {
    ...
  }
}
```

and then access them through services:

```
class UserService(userRepository : UserRepository) {

  def isAdmin(user:User) : Boolean = {
    userRepository.roles(user).contains(Role("ADMIN"))
  }
}
```

In this way, the `isAdmin` method can be tested by mocking out the `UserRepository`reference and passing it into the service:

```
object UserServiceSpec extends Specification with Mockito {

  "UserService#isAdmin" should {
    "be true when the role is admin" in {
```

```
    val userRepository = mock[UserRepository]
    userRepository.roles(any[User]) returns Set(Role("ADMIN"))

    val userService = new UserService(userRepository)
    val actual = userService.isAdmin(User("11", "Steve", "user@example.org"))
    actual must beTrue
   }
  }
}
```

# Unit Testing Controllers

When defining controllers as objects, they can be trickier to unit test. In Play this can be alleviated by dependency injection. Another way to finesse unit testing with a controller declared as a object is to use a trait with an explicitly typed self reference to the controller:

```
trait ExampleController {
  this: Controller =>

  def index() = Action {
    Ok("ok")
  }
}

object ExampleController extends Controller with ExampleController
```

and then test the trait:

```
import play.api.mvc._
import play.api.test._
import scala.concurrent.Future

object ExampleControllerSpec extends PlaySpecification with Results {

  class TestController() extends Controller with ExampleController

  "Example Page#index" should {
    "should be valid" in {
      val controller = new TestController()
      val result: Future[Result] = controller.index().apply(FakeRequest())
      val bodyText: String = contentAsString(result)
      bodyText must be equalTo "ok"
    }
  }
}
```

# Unit Testing EssentialAction

Testing `Action` or `Filter` can require to test an `EssentialAction` (more information about what an EssentialAction is)

For this, the test `Helpers.call` can be used like that:

```
object ExampleEssentialActionSpec extends PlaySpecification {

 "An essential action" should {
  "can parse a JSON body" in {
    val action: EssentialAction = Action { request =>
      val value = (request.body.asJson.get \ "field").as[String]
      Ok(value)
    }

    val request = FakeRequest(POST, "/").withJsonBody(Json.parse("""{ "field": "value" }"""))

    val result = call(action, request)

    status(result) mustEqual OK
    contentAsString(result) mustEqual "value"
  }
 }
}
```

**Next:** Writing functional tests with specs2

# Writing functional tests with specs2

Play provides a number of classes and convenience methods that assist with functional testing. Most of these can be found either in the `play.api.test` package or in the `Helpers` object.

You can add these methods and classes by importing the following:

```
import play.api.test._
import play.api.test.Helpers._
```

# FakeApplication

Play frequently requires a running `Application` as context: it is usually provided from `play.api.Play.current`.
To provide an environment for tests, Play provides a `FakeApplication` class which can be configured with a different Global object, additional configuration, or even additional plugins.

```
val fakeApplicationWithGlobal = FakeApplication(withGlobal = Some(new GlobalSettings() {
 override def onStart(app: Application) { println("Hello world!") }
}))
```

# WithApplication

To pass in an application to an example, use `WithApplication`. An explicit`Application` can be passed in, but a default `FakeApplication` is provided for convenience.
Because `WithApplication` is a built in `Around` block, you can override it to provide your own data population:

```
abstract class WithDbData extends WithApplication {
  override def around[T: AsResult](t: => T): Result = super.around {
    setupData()
    t
  }

  def setupData() {
    // setup data
  }
}

"Computer model" should {

  "be retrieved by id" in new WithDbData {
    // your test code
  }
  "be retrieved by email" in new WithDbData {
    // your test code
  }
}
```

# WithServer

Sometimes you want to test the real HTTP stack from within your test, in which case you can start a test server using `WithServer`:

```
"test server logic" in new WithServer(app = fakeApplicationWithBrowser, port = testPort) {
  // The test payment gateway requires a callback to this server before it returns a result...
  val callbackURL = s"http://$myPublicAddress/callback"

  // await is from play.api.test.FutureAwaits
  val response = await(WS.url(testPaymentGatewayURL).withQueryString("callbackURL" ->
callbackURL).get())

  response.status must equalTo(OK)
}
```

The `port` value contains the port number the server is running on. By default this is 19001, however you can change this either by passing the port into `WithServer`, or by setting the system property `testserver.port`. This can be useful for integrating with continuous integration servers, so that ports can be dynamically reserved for each build.

A `FakeApplication` can also be passed to the test server, which is useful for setting up custom routes and testing WS calls:

```
val appWithRoutes = FakeApplication(withRoutes = {
 case ("GET", "/") =>
  Action {
   Ok("ok")
  }
})

"test WS logic" in new WithServer(app = appWithRoutes, port = 3333) {
 await(WS.url("http://localhost:3333").get()).status must equalTo(OK)
}
```

# WithBrowser

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 byFluentLenium using `WithBrowser`. Like `WithServer`, you can change the port,`Application`, and you can also select the web browser to use:

```
val fakeApplicationWithBrowser = FakeApplication(withRoutes = {
 case ("GET", "/") =>
  Action {
   Ok(
    """
     |<html>
     |<body>
     |  <div id="title">Hello Guest</div>
     |  <a href="/login">click me</a>
     |</body>
     |</html>
    """.stripMargin) as "text/html"
  }
 case ("GET", "/login") =>
  Action {
   Ok(
    """
     |<html>
     |<body>
     |  <div id="title">Hello Coco</div>
     |</body>
     |</html>
    """.stripMargin) as "text/html"
  }
})

"run in a browser" in new WithBrowser(webDriver = WebDriverFactory(HTMLUNIT), app =
fakeApplicationWithBrowser) {
 browser.goTo("/")

 // Check the page
 browser.$("#title").getTexts().get(0) must equalTo("Hello Guest")
```

```
  browser.$("a").click()

  browser.url must equalTo("/login")
  browser.$("#title").getTexts().get(0) must equalTo("Hello Coco")
}
```

# PlaySpecification

`PlaySpecification` is an extension of `Specification` that excludes some of the mixins provided in the default specs2 specification that clash with Play helpers methods. It also mixes in the Play test helpers and types for convenience.

```
object ExamplePlaySpecificationSpec extends PlaySpecification {
  "The specification" should {

    "have access to HeaderNames" in {
      USER_AGENT must be_===("User-Agent")
    }

    "have access to Status" in {
      OK must be_===(200)
    }
  }
}
```

# Testing a view template

Since a template is a standard Scala function, you can execute it from your test, and check the result:

```
"render index template" in new WithApplication {
  val html = views.html.index("Coco")

  contentAsString(html) must contain("Hello Coco")
}
```

# Testing a controller

You can call any `Action` code by providing a `FakeRequest`:

```
"respond to the index Action" in {
  val result = controllers.Application.index()(FakeRequest())

  status(result) must equalTo(OK)
  contentType(result) must beSome("text/plain")
  contentAsString(result) must contain("Hello Bob")
}
```

Technically, you don't need `WithApplication` here, although it wouldn't hurt anything to have it.

# Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
"respond to the index Action" in new WithApplication(fakeApplication) {
 val Some(result) = route(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")
}
```

# Testing a model

If you are using an SQL database, you can replace the database connection with an in-memory instance of an H2 database using `inMemoryDatabase`.

```
val appWithMemoryDatabase = FakeApplication(additionalConfiguration = inMemoryDatabase("test"))
"run an application" in new WithApplication(appWithMemoryDatabase) {

 val Some(macintosh) = Computer.findById(21)

 macintosh.name must equalTo("Macintosh")
 macintosh.introduced must beSome.which(_ must beEqualTo("1984-01-24"))
}
```

**Next:** Testing with Guice

# Testing with Guice

If you're using Guice for dependency injection then you can directly configure how components and applications are created for tests. This includes adding extra bindings or overriding existing bindings.

# GuiceApplicationBuilder

GuiceApplicationBuilder provides a builder API for configuring the dependency injection and creation of an Application.

## Environment

The Environment, or parts of the environment such as the root path, mode, or class loader for an application, can be specified. The configured environment will be used for loading the

application configuration, it will be used when loading modules and passed when deriving bindings from Play modules, and it will be injectable into other components.

```
import play.api.inject.guice.GuiceApplicationBuilder
val application = new GuiceApplicationBuilder()
  .in(Environment(new File("path/to/app"), classLoader, Mode.Test))
  .build
val application = new GuiceApplicationBuilder()
  .in(new File("path/to/app"))
  .in(Mode.Test)
  .in(classLoader)
  .build
```

## Configuration

Additional configuration can be added. This configuration will always be in addition to the configuration loaded automatically for the application. When existing keys are used the new configuration will be preferred.

```
val application = new GuiceApplicationBuilder()
  .configure(Configuration("a" -> 1))
  .configure(Map("b" -> 2, "c" -> "three"))
  .configure("d" -> 4, "e" -> "five")
  .build
```

The automatic loading of configuration from the application environment can also be overridden. This will completely replace the application configuration. For example:

```
val application = new GuiceApplicationBuilder()
  .loadConfig(env => Configuration.load(env))
  .build
```

## Bindings and Modules

The bindings used for dependency injection are completely configurable. The builder methods support Play Modules and Bindings and also Guice Modules.

### *Additional bindings*

Additional bindings, via Play modules, Play bindings, or Guice modules, can be added:

```
import play.api.inject.bind
val injector = new GuiceApplicationBuilder()
  .bindings(new ComponentModule)
  .bindings(bind[Component].to[DefaultComponent])
  .injector
```

### *Override bindings*

Bindings can be overridden using Play bindings, or modules that provide bindings. For example:

```
val application = new GuiceApplicationBuilder()
  .overrides(bind[Component].to[MockComponent])
```

```
  .build
```

### *Disable modules*

Any loaded modules can be disabled by class name:

```
val injector = new GuiceApplicationBuilder()
  .disable[ComponentModule]
  .injector
```

### *Loaded modules*

Modules are automatically loaded from the classpath based on the`play.modules.enabled` configuration. This default loading of modules can be overridden. For example:

```
val injector = new GuiceApplicationBuilder()
  .load(
    new play.api.inject.BuiltinModule,
    bind[Component].to[DefaultComponent]
  ).injector
```

# GuiceInjectorBuilder

GuiceInjectorBuilder provides a builder API for configuring Guice dependency injection more generally. This builder does not load configuration or modules automatically from the environment like `GuiceApplicationBuilder`, but provides a completely clean state for adding configuration and bindings. The common interface for both builders can be found in GuiceBuilder. A Play Injector is created. Here's an example of instantiating a component using the injector builder:

```
import play.api.inject.guice.GuiceInjectorBuilder
import play.api.inject.bind
val injector = new GuiceInjectorBuilder()
  .configure("key" -> "value")
  .bindings(new ComponentModule)
  .overrides(bind[Component].to[MockComponent])
  .injector

val component = injector.instanceOf[Component]
```

# Overriding bindings in a functional test

Here is a full example of replacing a component with a mock component for testing. Let's start with a component, that has a default implementation and a mock implementation for testing:

```
trait Component {
```

```
  def hello: String
}

class DefaultComponent extends Component {
  def hello = "default"
}

class MockComponent extends Component {
  def hello = "mock"
}
```

This component is loaded automatically using a module:

```
import play.api.{ Environment, Configuration }
import play.api.inject.Module

class ComponentModule extends Module {
  def bindings(env: Environment, conf: Configuration) = Seq(
    bind[Component].to[DefaultComponent]
  )
}
```

And the component is used in a controller:

```
import play.api.mvc._
import javax.inject.Inject

class Application @Inject() (component: Component) extends Controller {
  def index() = Action {
    Ok(component.hello)
  }
}
```

To build an `Application` to use in functional tests we can simply override the binding for the component:

```
import play.api.inject.guice.GuiceApplicationBuilder
import play.api.inject.bind
val application = new GuiceApplicationBuilder()
  .overrides(bind[Component].to[MockComponent])
  .build
```

The created application can be used with the functional testing helpers for Specs2 andScalaTest.

**Next:** Testing with databases

# Testing with databases

While it is possible to write functional tests using ScalaTest or specs2 that test database access code by starting up a full application including the database, starting up a full

application is not often desirable, due to the complexity of having many more components started and running just to test one small part of your application.

Play provides a number of utilities for helping to test database access code that allow it to be tested with a database but in isolation from the rest of your app. These utilities can easily be used with either ScalaTest or specs2, and can make your database tests much closer to lightweight and fast running unit tests than heavy weight and slow functional tests.

---

# Using a database

To connect to a database, at a minimum, you just need database driver name and the url of the database, using the `Databases` companion object. For example, to connect to MySQL, you might use the following:

```
import play.api.db.Databases

val database = Databases(
  driver = "com.mysql.jdbc.Driver",
  url = "jdbc:mysql://localhost/test"
)
```

This will create a database connection pool for the MySQL `test` database running on`localhost`, with the name `default`. The name of the database is only used internally by Play, for example, by other features such as evolutions, to load resources associated with that database.

You may want to specify other configuration for the database, including a custom name, or configuration properties such as usernames, passwords and the various connection pool configuration items that Play supports, by supplying a custom name parameter and/or a custom config parameter:

```
import play.api.db.Databases

val database = Databases(
  driver = "com.mysql.jdbc.Driver",
  url = "jdbc:mysql://localhost/test",
  name = "mydatabase",
  config = Map(
    "user" -> "test",
    "password" -> "secret"
  )
)
```

After using a database, since the database is typically backed by a connection pool that holds open connections and may also have running threads, you need to shut it down. This is done by calling the `shutdown` method:

```
database.shutdown()
```

Manually creating the database and shutting it down is useful if you're using a test framework that runs startup/shutdown code around each test or suite. Otherwise it's recommended that you let Play manage the connection pool for you.

## Allowing Play to manage the database for you

Play also provides a `withDatabase` helper that allows you to supply a block of code to execute with a database connection pool managed by Play. Play will ensure that it is correctly shutdown after the block of code finishes executing:

```
import play.api.db.Databases

Databases.withDatabase(
  driver = "com.mysql.jdbc.Driver",
  url = "jdbc:mysql://localhost/test"
) { database =>
  val connection = database.getConnection()
  // ...
}
```

Like the `Database.apply` factory method, `withDatabase` also allows you to pass a custom `name` and `config` map if you please.

Typically, using `withDatabase` directly from every test is an excessive amount of boilerplate code. It is recommended that you create your own helper to remove this boiler plate that your test uses. For example:

```
import play.api.db.{Database, Databases}

def withMyDatabase[T](block: Database => T) = {
  Databases.withDatabase(
    driver = "com.mysql.jdbc.Driver",
    url = "jdbc:mysql://localhost/test",
    name = "mydatabase",
    config = Map(
      "user" -> "test",
      "password" -> "secret"
    )
  )(block)
}
```

Then it can be easily used in each test with minimal boilerplate:

```
withMyDatabase { database =>
  val connection = database.getConnection()
  // ...
}
```

**Tip:** You can use this to externalise your test database configuration, using environment variables or system properties to configure what database to use and how to connect to it. This allows for maximum flexibility for developers to have their own environments set up the way they please, as well as for CI systems that provide particular environments that may differ to development.

## Using an in-memory database

Some people prefer not to require infrastructure such as databases to be installed in order to run tests. Play provides simple helpers to create an H2 in-memory database for these purposes:

```
import play.api.db.Databases

val database = Databases.inMemory()
```

The in-memory database can be configured, by supplying a custom name, custom URL arguments, and custom connection pool configuration. The following shows supplying the `MODE` argument to tell H2 to emulate `MySQL`, as well as configuring the connection pool to log all statements:

```
import play.api.db.Databases

val database = Databases.inMemory(
  name = "mydatabase",
  urlOptions = Map(
    "MODE" -> "MYSQL"
  ),
  config = Map(
    "logStatements" -> true
  )
)
```

As with the generic database factory, ensure you always shut the in-memory database connection pool down:

```
database.shutdown()
```

If you're not using a test frameworks before/after capabilities, you may want Play to manage the in-memory database lifecycle for you, this is straightforward using `withInMemory`:

```
import play.api.db.Databases

Databases.withInMemory() { database =>
  val connection = database.getConnection()

  // ...
}
```

Like `withDatabase`, it is recommended that to reduce boilerplate code, you create your own method that wraps the `withInMemory` call:

```
import play.api.db.{Database, Databases}

def withMyDatabase[T](block: Database => T) = {
  Databases.withInMemory(
    name = "mydatabase",
    urlOptions = Map(
      "MODE" -> "MYSQL"
    ),
    config = Map(
```

```
    "logStatements" -> true
  )
 )(block)
}
```

# Applying evolutions

When running tests, you will typically want your database schema managed for your database. If you're already using evolutions, it will often make sense to reuse the same evolutions that you use in development and production in your tests. You may also want to create custom evolutions just for testing. Play provides some convenient helpers to apply and manage evolutions without having to run a whole Play application.

To apply evolutions, you can use `applyEvolutions` from the `Evolutions` companion object:

```
import play.api.db.evolutions._

Evolutions.applyEvolutions(database)
```

This will load the evolutions from the classpath in the `evolutions/<databasename>`directory, and apply them.

After a test has run, you may want to reset the database to its original state. If you have implemented your evolutions down scripts in such a way that they will drop all the database tables, you can do this simply by calling the `cleanupEvolutions` method:

```
Evolutions.cleanupEvolutions(database)
```

## Custom evolutions

In some situations you may want to run some custom evolutions in your tests. Custom evolutions can be used by using a custom `EvolutionsReader`. The simplest of these is the `SimpleEvolutionsReader`, which is an evolutions reader that takes a preconfigured map of database names to sequences of `Evolution` scripts, and can be constructed using the convenient methods on the `SimpleEvolutionsReader` companion object. For example:

```
import play.api.db.evolutions._

Evolutions.applyEvolutions(database, SimpleEvolutionsReader.forDefault(
  Evolution(
    1,
    "create table test (id bigint not null, name varchar(255));",
    "drop table test;"
  )
))
```

Cleaning up custom evolutions is done in the same way as cleaning up regular evolutions, using the `cleanupEvolutions` method:

```
Evolutions.cleanupEvolutions(database)
```

Note though that you don't need to pass the custom evolutions reader here, this is because the state of the evolutions is stored in the database, including the down scripts which will be used to tear down the database.

Sometimes it will be impractical to put your custom evolution scripts in code. If this is the case, you can put them in the test resources directory, under a custom path using the `ClassLoaderEvolutionsReader`. For example:

```
import play.api.db.evolutions._

Evolutions.applyEvolutions(database, ClassLoaderEvolutionsReader.forPrefix("testdatabase/"))
```

This will load evolutions, in the same structure and format as is done for development and production, from `testdatabase/evolutions/<databasename>/<n>.sql`.

## Allowing Play to manage evolutions

The `applyEvolutions` and `cleanupEvolutions` methods are useful if you're using a test framework to manage running the evolutions before and after a test. Play also provides a convenient `withEvolutions` method to manage it for you, if this lighter weight approach is desired:

```
import play.api.db.evolutions._

Evolutions.withEvolutions(database) {
  val connection = database.getConnection()

  // ...
}
```

Naturally, `withEvolutions` can be combined with `withDatabase` or `withInMemory` to reduce boilerplate code, allowing you to define a function that both instantiates the database and runs evolutions for you:

```
import play.api.db.{Database, Databases}
import play.api.db.evolutions._

def withMyDatabase[T](block: Database => T) = {

  Databases.withInMemory(
    urlOptions = Map(
      "MODE" -> "MYSQL"
    ),
    config = Map(
      "logStatements" -> true
    )
  ) { database =>

    Evolutions.withEvolutions(database, SimpleEvolutionsReader.forDefault(
      Evolution(
        1,
        "create table test (id bigint not null, name varchar(255));",
        "drop table test;"
      )
    )) {
```

```
    block(database)

  }
 }
}
```

Having defined the custom database management method for our tests, we can now use them in a straight forward manner:

```
withMyDatabase { database =>
 val connection = database.getConnection()
 connection.prepareStatement("insert into test values (10, 'testing')").execute()

 connection.prepareStatement("select * from test where id = 10")
  .executeQuery().next() must_== true
}
```

**Next:** Testing web service clients

# Testing web service clients

A lot of code can go into writing a web service client - preparing the request, serializing and deserializing the bodies, setting the correct headers. Since a lot of this code works with strings and weakly typed maps, testing it is very important. However testing it also presents some challenges. Some common approaches include:

## Test against the actual web service

This of course gives the highest level of confidence in the client code, however it is usually not practical. If it's a third party web service, there may be rate limiting in place that prevents your tests from running (and running automated tests against a third party service is not considered being a good netizen). It may not be possible to set up or ensure the existence of the necessary data that your tests require on that service, and your tests may have undesirable side effects on the service.

## Test against a test instance of the web service

This is a little better than the previous one, however it still has a number of problems. Many third party web services don't provide test instances. It also means your tests depend on the test instance being running, meaning that test service could cause your build to fail. If the test instance is behind a firewall, it also limits where the tests can be run from.

## Mock the http client

This approach gives the least confidence in the test code - often this kind of testing amounts to testing no more than that the code does what it does, which is of no value. Tests against mock web service clients show that the code runs and does certain things, but gives no confidence as to whether anything that the code does actually correlates to valid HTTP requests being made.

## Mock the web service

This approach is a good compromise between testing against the actual web service and mocking the http client. Your tests will show that all the requests it makes are valid HTTP requests, that serialisation/deserialisation of bodies work, etc, but they will be entirely self contained, not depending on any third party services.

Play provides some helper utilities for mocking a web service in tests, making this approach to testing a very viable and attractive option.

---

# Testing a GitHub client

As an example, let's say you've written a GitHub client, and you want to test it. The client is very simple, it just allows you to look up the names of the public repositories:

```
import javax.inject.Inject
import play.api.libs.ws.WSClient
import play.api.libs.concurrent.Execution.Implicits.defaultContext
import scala.concurrent.Future

class GitHubClient(ws: WSClient, baseUrl: String) {
  @Inject def this(ws: WSClient) = this(ws, "https://api.github.com")

  def repositories(): Future[Seq[String]] = {
    ws.url(baseUrl + "/repositories").get().map { response =>
      (response.json \\ "full_name").map(_.as[String])
    }
  }
}
```

Note that it takes the GitHub API base URL as a parameter - we'll override this in our tests so that we can point it to our mock server.

To test this, we want an embedded Play server that will implement this endpoint. We can do that using the `Server` `withRouter` helper in combination with the String Interpolating Routing DSL:

```
import play.api.libs.json._
import play.api.mvc._
import play.api.routing.sird._
import play.core.server.Server

Server.withRouter() {
 case GET(p"/repositories") => Action {
  Results.Ok(Json.arr(Json.obj("full_name" -> "octocat/Hello-World")))
 }
} { implicit port =>
```

The `withRouter` method takes a block of code that takes as input the port number that the server starts on. By default, Play starts the server on a random free port - this means that you don't need to worry about resource contention on build servers or assigning ports to tests, but it means that your code does need to be told which port is going to be used.
Now to test the GitHub client, we need a `WSClient` for it. Play provides a `WsTestClient`trait that has some factory methods for creating test clients.
The `withClient` takes an implicit port, this is handy to use in combination with the `Server.withRouter` method.
The client that the `WsTestClient.withClient` method creates here is a special client - if you give it a relative URL, then it will default the hostname to `localhost` and the port number to the port number passed in implicitly. Using this, we can simply set the base url for our GitHub client to be an empty String.

Putting it all together, we have this:

```
import play.core.server.Server
import play.api.routing.sird._
import play.api.mvc._
import play.api.libs.json._
import play.api.test._

import scala.concurrent.Await
import scala.concurrent.duration._

import org.specs2.mutable.Specification
import org.specs2.time.NoTimeConversions

object GitHubClientSpec extends Specification with NoTimeConversions {

 "GitHubClient" should {
  "get all repositories" in {

   Server.withRouter() {
    case GET(p"/repositories") => Action {
     Results.Ok(Json.arr(Json.obj("full_name" -> "octocat/Hello-World")))
    }
   } { implicit port =>
    WsTestClient.withClient { client =>
     val result = Await.result(
```

```
          new GitHubClient(client, "").repositories(), 10.seconds)
        result must_== Seq("octocat/Hello-World")
      }
    }
  }
 }
}
```

## Returning files

In the previous example, we built the json manually for the mocked service. It often will be better to capture an actual response from the service your testing, and return that. To assist with this, Play provides a `sendResource` method that allows easily creating results from files on the classpath.

So after making a request on the actual GitHub API, create a file to store it in the test resources directory. The test resources directory is either `test/resources` if you're using a Play directory layout, or `src/test/resources` if you're using a standard sbt directory layout. In this case, we'll call it `github/repositories.json`, and it will contain the following:

```
[
  {
    "id": 1296269,
    "owner": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url": "https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url": "https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "description": "This your first repo!",
    "private": false,
    "fork": false,
    "url": "https://api.github.com/repos/octocat/Hello-World",
    "html_url": "https://github.com/octocat/Hello-World"
  }
]
```

You may decide to modify it to suit your testing needs, for example, if your GitHub client used the URLs in the above response to make requests to other endpoints, you might remove the `https://api.github.com` prefix from them so that they too are relative, and will automatically be routed to localhost on the right port by the test client.

Now, modify the router to serve this resource:

```
import play.api.mvc._
import play.api.routing.sird._
import play.api.test._
import play.core.server.Server

Server.withRouter() {
  case GET(p"/repositories") => Action {
    Results.Ok.sendResource("github/repositories.json")
  }
} { implicit port =>
```

Note that Play will automatically set a content type of `application/json` due to the filename's extension of `.json`.

## Extracting setup code

The tests implemented so far are fine if you only have one test you want to run, but if you have many methods that you want to test, it may make more sense to extract the mock client setup code out into one helper method. For example, we could define a`withGitHubClient` method:

```
import play.api.mvc._
import play.api.routing.sird._
import play.core.server.Server
import play.api.test._

def withGitHubClient[T](block: GitHubClient => T): T = {
  Server.withRouter() {
    case GET(p"/repositories") => Action {
      Results.Ok.sendResource("github/repositories.json")
    }
  } { implicit port =>
    WsTestClient.withClient { client =>
      block(new GitHubClient(client, ""))
    }
  }
}
```

And then using it in a test looks like:

```
withGitHubClient { client =>
  val result = Await.result(client.repositories(), 10.seconds)
  result must_== Seq("octocat/Hello-World")
}
```

**Next:** Logging

# The Logging API

Using logging in your application can be useful for monitoring, debugging, error tracking, and business intelligence. Play provides an API for logging which is accessed through the `Logger` object and uses Logback as the logging engine.

# Logging architecture

The logging API uses a set of components that help you to implement an effective logging strategy.

### *Logger*

Your application can define `Logger` instances to send log message requests.
Each`Logger` has a name which will appear in log messages and is used for configuration.

Loggers follow a hierarchical inheritance structure based on their naming. A logger is said to be an ancestor of another logger if its name followed by a dot is the prefix of descendant logger name. For example, a logger named “com.foo” is the ancestor of a logger named “com.foo.bar.Baz.” All loggers inherit from a root logger. Logger inheritance allows you to configure a set of loggers by configuring a common ancestor.

Play applications are provided a default logger named “application” or you can create your own loggers. The Play libraries use a logger named “play”, and some third party libraries will have loggers with their own names.

### *Log levels*

Log levels are used to classify the severity of log messages. When you write a log request statement you will specify the severity and this will appear in generated log messages.

This is the set of available log levels, in decreasing order of severity.

- `OFF` - Used to turn off logging, not as a message classification.
- `ERROR` - Runtime errors, or unexpected conditions.
- `WARN` - Use of deprecated APIs, poor use of API, ‘almost’ errors, other runtime situations that are undesirable or unexpected, but not necessarily “wrong”.
- `INFO` - Interesting runtime events such as application startup and shutdown.
- `DEBUG` - Detailed information on the flow through the system.
- `TRACE` - Most detailed information.

In addition to classifying messages, log levels are used to configure severity thresholds on loggers and appenders. For example, a logger set to level `INFO` will log any request of level `INFO` or higher (`INFO`, `WARN`, `ERROR`) but will ignore requests of lower severities (`DEBUG`, `TRACE`). Using `OFF` will ignore all log requests.

### *Appenders*

The logging API allows logging requests to print to one or many output destinations called "appenders." Appenders are specified in configuration and options exist for the console, files, databases, and other outputs.

Appenders combined with loggers can help you route and filter log messages. For example, you could use one appender for a logger that logs useful data for analytics and another appender for errors that is monitored by an operations team.

Note: For further information on architecture, see the Logback documentation.

# Using Loggers

First import the `Logger` class and companion object:

```
import play.api.Logger
```

### *The default Logger*

The `Logger` object is your default logger and uses the name "application." You can use it to write log request statements:

```
// Log some debug info
Logger.debug("Attempting risky calculation.")

try {
  val result = riskyCalculation

  // Log result if successful
  Logger.debug(s"Result=$result")
} catch {
  case t: Throwable => {
    // Log error with message and Throwable.
    Logger.error("Exception with riskyCalculation", t)
  }
}
```

Using Play's default logging configuration, these statements will produce console output similar to this:

```
[debug] application - Attempting risky calculation.
[error] application - Exception with riskyCalculation
java.lang.ArithmeticException: / by zero
    at controllers.Application$.controllers$Application$$riskyCalculation(Application.scala:32) ~[classes/:na]
    at controllers.Application$$anonfun$test$1.apply(Application.scala:18) [classes/:na]
    at controllers.Application$$anonfun$test$1.apply(Application.scala:12) [classes/:na]
```

```
    at play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390) [play_2.10-2.3-M1.jar:2.3-M1]
    at play.api.mvc.ActionBuilder$$anonfun$apply$17.apply(Action.scala:390) [play_2.10-2.3-M1.jar:2.3-M1]
```

Note that the messages have the log level, logger name, message, and stack trace if a Throwable was used in the log request.

### *Creating your own loggers*

Although it may be tempting to use the default logger everywhere, it's generally a bad design practice. Creating your own loggers with distinct names allows for flexibile configuration, filtering of log output, and pinpointing the source of log messages.

You can create a new logger using the `Logger.apply` factory method with a name argument:

```
val accessLogger: Logger = Logger("access")
```

A common strategy for logging application events is to use a distinct logger per class using the class name. The logging API supports this with a factory method that takes a class argument:

```
val logger: Logger = Logger(this.getClass())
```

### *Logging patterns*

Effective use of loggers can help you achieve many goals with the same tool:

```
import scala.concurrent.Future
import play.api.Logger
import play.api.mvc._

trait AccessLogging {

  val accessLogger = Logger("access")

  object AccessLoggingAction extends ActionBuilder[Request] {

    def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
      accessLogger.info(s"method=${request.method} uri=${request.uri} remote-
address=${request.remoteAddress}")
      block(request)
    }
  }
}

object Application extends Controller with AccessLogging {

  val logger = Logger(this.getClass())

  def index = AccessLoggingAction {
    try {
```

```
      val result = riskyCalculation
      Ok(s"Result=$result")
    } catch {
      case t: Throwable => {
        logger.error("Exception with riskyCalculation", t)
        InternalServerError("Error in calculation: " + t.getMessage())
      }
    }
  }
}
```

This example uses action composition to define an `AccessLoggingAction` that will log request data to a logger named “access.” The `Application` controller uses this action and it also uses its own logger (named after its class) for application events. In configuration you could then route these loggers to different appenders, such as an access log and an application log.

The above design works well if you want to log request data for only specific actions. To log all requests, it’s better to use a filter:

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import play.api.Logger
import play.api.mvc._
import play.api._

object AccessLoggingFilter extends Filter {

  val accessLogger = Logger("access")

  def apply(next: (RequestHeader) => Future[Result])(request: RequestHeader): Future[Result] = {
    val resultFuture = next(request)

    resultFuture.foreach(result => {
      val msg = s"method=${request.method} uri=${request.uri} remote-address=${request.remoteAddress}" +
        s" status=${result.header.status}";
      accessLogger.info(msg)
    })

    resultFuture
  }
}

object Global extends WithFilters(AccessLoggingFilter) {

  override def onStart(app: Application) {
    Logger.info("Application has started")
  }

  override def onStop(app: Application) {
    Logger.info("Application has stopped")
  }
}
```

In the filter version we've added the response status to the log request by logging when the `Future[Result]` completes.

# Configuration

See configuring logging for details on configuration.

**Next:** Advanced topics

# 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 the 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: using a rich set of adapters and transformers to change the shape of the source or the consumer - construct your own or start with primitives.
- Being able to stop data being sent mid-way through, and being informed when source is done sending data.
- Non blocking, reactive and allowing control over resource consumption (Thread, Memory)

# Iteratees

An Iteratee is a consumer - it describes the way input will be consumed to produce some value. An Iteratee is a consumer that returns a value it computes after being fed enough input.

```
// an iteratee that consumes String chunks 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](folder: Step[E, A] => Future[B]): Future[B]
```

where the `Step` object has 3 states :

```
object Step {
  case class Done[+A, E](a: A, remaining: Input[E]) extends Step[E, A]
  case class Cont[E, +A](k: Input[E] => Iteratee[E, A]) extends Step[E, A]
  case class Error[E](msg: String, input: Input[E]) extends Step[E, Nothing]
}
```

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 in the state `Done`, then I'll 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 in the state `Cont`, then I'll 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 in the state `Error`, then I'll take the error message of type `String` and the input that caused it and eventually produce a B.

Depending on the state of the iteratee, `fold` will produce the appropriate `B` using the corresponding passed-in function.

To sum up, an 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 `El[E]` containing some actual input, an `Empty` chunk or an `EOF` representing the end of the stream.
  For example, `Input[String]` can be `El("Hello!")`, Empty, or EOF
- `Future[A]` represents, as its name indicates, a future value of type `A`. This means that it is initially empty and will eventually be filled in ("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 Future is a very nice primitive for synchronization and composing async calls, and is explained further at the ScalaAsync 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 a `1:Int` and returning `Empty` as the remaining value from the last `Input[String]`

```
val doneIteratee = new Iteratee[String,Int] {
  def fold[B](folder: Step[String,Int] => Future[B])(implicit ec: ExecutionContext) : Future[B] =
    folder(Step.Done(1, Input.Empty))
}
```

As shown above, this is easily done by calling the appropriate `apply` function, in our case that of `Done`, with the necessary information.

To use this iteratee we will make use of the `Future` that holds a promised value.

```
def folder(step: Step[String,Int]):Future[Option[Int]] = step match {
  case Step.Done(a, e) => future(Some(a))
  case Step.Cont(k) => future(None)
  case Step.Error(msg,e) => future(None)
}

val eventuallyMaybeResult: Future[Option[Int]] = doneIteratee.fold(folder)

eventuallyMaybeResult.onComplete(i => println(i))
```

of course to see what is inside the `Future` when it is redeemed we use `onComplete`

```
// will eventually print 1
eventuallyMaybeResult.onComplete(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[String,Int](1, Input.Empty)
```

Creating a `Done` iteratee is simple, and sometimes useful, but it 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](folder: Step[String,Int] => Future[B])(implicit ec: ExecutionContext): Future[B] = {
    folder(Step.Cont {
      case Input.EOF => Done(0, Input.EOF) //Assuming 0 for default value
      case Input.Empty => this
      case Input.El(e) => Done(e.toInt,Input.EOF)
    })
  }
}

def folder(step: Step[String,Int]):Future[Int] = step match {
  case Step.Done(a, _) => future(a)
  case Step.Cont(k) => k(Input.EOF).fold({
    case Step.Done(a1, _) => Future.successful(a1)
    case _ => throw new Exception("Erroneous or diverging iteratee")
  })
  case _ => throw new Exception("Erroneous iteratee")
}
```

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(100,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 returns 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 would 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 would 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 the `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 that it gives one the opportunity to treat input chunks asynchronously.

Of course one should be worried now about how hard it would 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.

**Next:** Enumerators

# Handling data streams reactively

# Enumerators

If an iteratee represents the consumer, or sink, of input, an `Enumerator` is the source that pushes input into a given iteratee. As the name suggests, it enumerates some input into the iteratee and eventually returns the new state of that iteratee. This can be easily seen looking at the `Enumerator`'s signature:

```
trait Enumerator[E] {

 /**
  * Apply this Enumerator to an Iteratee
  */
 def apply[A](i: Iteratee[E, A]): Future[Iteratee[E, A]]

}
```

An `Enumerator[E]` takes an `Iteratee[E,A]` which is any iteratee that consumes`Input[E]` and returns a `Future[Iteratee[E,A]]` which eventually gives the new state of the iteratee.

We can go ahead and manually implement `Enumerator` instances by consequently calling the iteratee's fold method, or use one of the provided `Enumerator` creation methods. For instance we can create an `Enumerator[String]` that pushes a list of strings into an iteratee, like the following:

```
val enumerateUsers: Enumerator[String] = {
  Enumerator("Guillaume", "Sadek", "Peter", "Erwan")
}
```

Now we can apply it to the consume iteratee we created before:

```
val consume = Iteratee.consume[String]()
val newIteratee: Future[Iteratee[String,String]] = enumerateUsers(consume)
```

To terminate the iteratee and extract the computed result we pass `Input.EOF`.

An`Iteratee` carries a `run` method that does just this. It pushes an `Input.EOF` and returns a `Future[A]`, ignoring left input if any.

```
// We use flatMap since newIteratee is a promise,
```

```
// and run itself return a promise
val eventuallyResult: Future[String] = newIteratee.flatMap(i => i.run)

//Eventually print the result
eventuallyResult.onSuccess { case x => println(x) }

// Prints "GuillaumeSadekPeterErwan"
```

You might notice here that an `Iteratee` will eventually produce a result (returning a promise when calling fold and passing appropriate calbacks), and a `Future` eventually produces a result. Then a `Future[Iteratee[E,A]]` can be viewed as `Iteratee[E,A]`. Indeed this is what `Iteratee.flatten` does, Let's apply it to the previous example:

```
//Apply the enumerator and flatten then run the resulting iteratee
val newIteratee = Iteratee.flatten(enumerateUsers(consume))

val eventuallyResult: Future[String] = newIteratee.run

//Eventually print the result
eventuallyResult.onSuccess { case x => println(x) }

// Prints "GuillaumeSadekPeterErwan"
```

An `Enumerator` has some symbolic methods that can act as operators, which can be useful in some contexts for saving some parentheses. For example, the `|>>` method works exactly like apply:

```
val eventuallyResult: Future[String] = {
  Iteratee.flatten(enumerateUsers |>> consume).run
}
```

Since an `Enumerator` pushes some input into an iteratee and eventually return a new state of the iteratee, we can go on pushing more input into the returned iteratee using another `Enumerator`. This can be done either by using the `flatMap` function on `Future`s or more simply by combining `Enumerator` instances using the `andThen` method, as follows:

```
val colors = Enumerator("Red","Blue","Green")

val moreColors = Enumerator("Grey","Orange","Yellow")

val combinedEnumerator = colors.andThen(moreColors)

val eventuallyIteratee = combinedEnumerator(consume)
```

As for apply, there is a symbolic version of the `andThen` called `>>>` that can be used to save some parentheses when appropriate:

```
val eventuallyIteratee = {
  Enumerator("Red","Blue","Green") >>>
  Enumerator("Grey","Orange","Yellow") |>>
  consume
}
```

We can also create `Enumerator`s for enumerating files contents:

```
val fileEnumerator: Enumerator[Array[Byte]] = {
  Enumerator.fromFile(new File("path/to/some/file"))
}
```

Or more generally enumerating a `java.io.InputStream` using `Enumerator.fromStream`. It is important to note that input won't be read until the iteratee this `Enumerator` is applied on is ready to take more input.

Actually both methods are based on the more generic `Enumerator.generateM` that has the following signature:

```
def generateM[E](e: => Future[Option[E]]) = {
  ...
}
```

This method defined on the `Enumerator` object is one of the most important methods for creating `Enumerator`s from imperative logic. Looking closely at the signature, this method takes a callback function `e: => Future[Option[E]]` that will be called each time the iteratee this `Enumerator` is applied to is ready to take some input.

It can be easily used to create an `Enumerator` that represents a stream of time values every 100 millisecond using the opportunity that we can return a promise, like the following:

```
Enumerator.generateM {
  Promise.timeout(Some(new Date), 100 milliseconds)
}
```

In the same manner we can construct an `Enumerator` that would fetch a url every some time using the `WS` api which returns, not surprisingly a `Future`

Combining this, callback Enumerator, with an imperative `Iteratee.foreach` we can println a stream of time values periodically:

```
val timeStream = Enumerator.generateM {
  Promise.timeout(Some(new Date), 100 milliseconds)
}

val printlnSink = Iteratee.foreach[Date](date => println(date))

timeStream |>> printlnSink
```

Another, more imperative, way of creating an `Enumerator` is by using `Concurrent.unicast` which once it is ready will give a `Channel` interface on which defined methods `push` and `end`:

```
val enumerator = Concurrent.unicast[String](onStart = channel => {
  channel.push("Hello")
  channel.push("World")
})

enumerator |>> Iteratee.foreach(println)
```

The `onStart` function will be called each time the `Enumerator` is applied to an `Iteratee`. In some applications, a chatroom for instance, it makes sense to assign the `enumerator` to a synchronized global value (using STMs for example) that will contain a list of listeners. `Concurrent.unicast` accepts two other functions, `onComplete` and `onError`.

One more interesting method is the `interleave` or `>-` method which as the name says, itrerleaves two Enumerators. For reactive `Enumerator`s Input will be passed as it happens from any of the interleaved `Enumerator`s

# Enumerators à la carte

Now that we have several interesting ways of creating `Enumerator`s, we can use these together with composition methods `andThen` / `>>>` and `interleave` / `>-` to compose`Enumerator`s on demand.

Indeed one interesting way of organizing a streamful application is by creating primitive`Enumerator`s and then composing a collection of them. Let's imagine doing an application for monitoring systems:

```
object AvailableStreams {

  val cpu: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

  val memory: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

  val threads: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

  val heap: Enumerator[JsValue] = Enumerator.generateM(/* code here */)

}

val physicalMachine = AvailableStreams.cpu >- AvailableStreams.memory
val jvm = AvailableStreams.threads >- AvailableStreams.heap

def usersWidgetsComposition(prefs: Preferences) = {
  // do the composition dynamically
}
```

Now, it is time to adapt and transform `Enumerator`s and `Iteratee`s using …`Enumeratee`s!

**Next:** Enumeratees

# Handling data streams reactively

# The realm of Enumeratees

'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 am Enumeratee using the map method on Enumeratee
val toInt: Enumeratee[String,Int] = Enumeratee.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 am 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: Future[Iteratee[String,Iteratee[Int,Int]]] = {
  Enumerator("1","2","3","4") |>> adaptedIteratee
}

val flattenAndRun:Future[Iteratee[Int,Int]] = Iteratee.flatten(afterPushingStrings).run

val originalIteratee = Iteratee.flatten(flattenAndRun)

val moreInts: Future[Iteratee[Int,Int]] = Enumerator(5,6,7) |>> originalIteratee

val sumFuture:Future[Int] = Iteratee.flatten(moreInts).run

sumFuture onSuccess {
 case s => println(s)// 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 straightforward, 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 statisfies 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],Array[Byte]] = {
 Iteratee.consume[Array[Byte]]()
}

val limitTo100: Enumeratee[Array[Byte],Array[Byte]] = {
 Enumeratee.take[Array[Byte]](100)
}

val limitedFillInMemory: Iteratee[Array[Byte],Array[Byte]] = {
 limitTo100 &>> 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`:

```
val fillInMemory: Iteratee[Array[Byte],Array[Byte]] = {
 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],Array[Byte]] = {
 limitTo100 &>> 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.

**Next:** HTTP architecture

# Introduction to Play HTTP API

## What is EssentialAction?

The EssentialAction is the new simpler type replacing the old Action[A]. To understand EssentialAction we need to understand the Play architecture.

The core of Play2 is really small, surrounded by a fair amount of useful APIs, services and structure to make Web Programming tasks easier.

Basically, Play2 is an API that abstractly have the following type:

```
RequestHeader -> Array[Byte] -> Result
```

The above computation takes the request header `RequestHeader`, then takes the request body as `Array[Byte]` and produces a `Result`.

Now this type presumes putting request body entirely into memory (or disk), even if you only want to compute a value out of it, or better forward it to a storage service like Amazon S3.

We rather want to receive request body chunks as a stream and be able to process them progressively if necessary.

What we need to change is the second arrow to make it receive its input in chunks and eventually produce a result. There is a type that does exactly this, it is called `Iteratee` and takes two type parameters.
`Iteratee[E,R]` is a type of arrow that will take its input in chunks of type `E` and eventually return `R`. For our API we need an Iteratee that takes chunks of `Array[Byte]` and eventually return a `Result`. So we slightly modify the type to be:

```
RequestHeader -> Iteratee[Array[Byte],Result]
```

For the first arrow, we are simply using the Function[From,To] which could be type aliased with `=>`:

```
RequestHeader => Iteratee[Array[Byte],Result]
```

Now if I define an infix type alias for `Iteratee[E,R]`:
`type ==>[E,R] = Iteratee[E,R]` then I can write the type in a funnier way:

```
RequestHeader => Array[Byte] ==> Result
```

And this should read as: Take the request headers, take chunks of `Array[Byte]` which represent the request body and eventually return a `Result`. This exactly how the`EssentialAction` type is defined:

```
trait EssentialAction extends (RequestHeader => Iteratee[Array[Byte], Result])
```

The `Result` type, on the other hand, can be abstractly thought of as the response headers and the body of the response:

```
case class Result(headers: ResponseHeader, body:Array[Byte])
```

But, what if we want to send the response body progressively to the client without filling it entirely into memory. We need to improve our type. We need to replace the body type from an `Array[Byte]` to something that produces chunks of `Array[Byte]`.
We already have a type for this and is called `Enumerator[E]` which means that it is capable of producing chunks of `E`, in our case `Enumerator[Array[Byte]]`:

```
case class Result(headers:ResponseHeaders, body:Enumerator[Array[Byte]])
```

If we don't have to send the response progressively we still can send the entire body as a single chunk.

We can stream and write any type of data to socket as long as it is convertible to an`Array[Byte]`, that is what `Writeable[E]` insures for a given type 'E':

```
case class Result[E](headers:ResponseHeaders, body:Enumerator[E])(implicit writeable:Writeable[E])
```

# Bottom Line

The essential Play2 HTTP API is quite simple:

```
RequestHeader -> Iteratee[Array[Byte],Result]
```

or the funnier

```
RequestHeader => Array[Byte] ==> Result
```

Which reads as the following: Take the `RequestHeader` then take chunks of`Array[Byte]` and return a response. A response consists of `ResponseHeaders` and a body which is chunks of values convertible to `Array[Byte]` to be written to the socket represented in the `Enumerator[E]` type.

**Next:** HTTP filters

# Filters

Play provides a simple filter API for applying global filters to each request.

# Filters vs action composition

The filter API is intended for cross cutting concerns that are applied indiscriminately to all routes. For example, here are some common use cases for filters:

- Logging/metrics collection
- GZIP encoding
- Security headers

In contrast, action composition is intended for route specific concerns, such as authentication and authorisation, caching and so on. If your filter is not one that you want applied to every route, consider using action composition instead, it is far more powerful. And don't forget that you can create your own action builders that compose your own custom defined sets of actions to each route, to minimise boilerplate.

# A simple logging filter

The following is a simple filter that times and logs how long a request takes to execute in Play framework:

```
import play.api.Logger
import play.api.mvc._
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits.defaultContext

class LoggingFilter extends Filter {

 def apply(nextFilter: RequestHeader => Future[Result])
         (requestHeader: RequestHeader): Future[Result] = {

   val startTime = System.currentTimeMillis

   nextFilter(requestHeader).map { result =>

     val endTime = System.currentTimeMillis
     val requestTime = endTime - startTime

     Logger.info(s"${requestHeader.method} ${requestHeader.uri} " +
       s"took ${requestTime}ms and returned ${result.header.status}")

     result.withHeaders("Request-Time" -> requestTime.toString)
   }
 }
}
```

Let's understand what's happening here. The first thing to notice is the signature of the `apply` method. It's a curried function, with the first parameter, `nextFilter`, being a function that takes a request header and produces a result, and the second parameter, `requestHeader`, being the actual request header of the incoming request. The `nextFilter` parameter represents the next action in the filter chain. Invoking it will cause the action to be invoked. In most cases you will probably want to invoke this at some point in your future. You may decide to not invoke it if for some reason you want to block the request.

We save a timestamp before invoking the next filter in the chain. Invoking the next filter returns a `Future[Result]` that will redeemed eventually. Take a look at the Handling asynchronous results chapter for more details on asynchronous results. We then manipulate the `Result` in the `Future` by calling the `map` method with a closure that takes a `Result`. We calculate the time it took for the request, log it and send it back to the client in the response headers by calling `result.withHeaders("Request-Time" -> requestTime.toString)`.

# Using filters

The simplest way to use a filter is to provide an implementation of the `HttpFilters` trait in the root package:

```
import javax.inject.Inject
import play.api.http.HttpFilters
import play.filters.gzip.GzipFilter

class Filters @Inject() (
  gzip: GzipFilter,
  log: LoggingFilter
) extends HttpFilters {

  val filters = Seq(gzip, log)
}
```

If you want to have different filters in different environments, or would prefer not putting this class in the root package, you can configure where Play should find the class by setting `play.http.filters` in `application.conf` to the fully qualified class name of the class. For example:

```
play.http.filters=com.example.MyFilters
```

# Where do filters fit in?

Filters wrap the action after the action has been looked up by the router. This means you cannot use a filter to transform a path, method or query parameter to impact the router. However you can direct the request to a different action by invoking that action directly from the filter, though be aware that this will bypass the rest of the filter chain. If you do need to

modify the request before the router is invoked, a better way to do this would be to place your logic in `Global.onRouteRequest` instead.

Since filters are applied after routing is done, it is possible to access routing information from the request, via the `tags` map on the `RequestHeader`. For example, you might want to log the time against the action method. In that case, you might update the`logTime` method to look like this:

```
import play.api.mvc.{Result, RequestHeader, Filter}
import play.api.{Logger, Routes}
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits.defaultContext

object LoggingFilter extends Filter {
  def apply(nextFilter: RequestHeader => Future[Result])
           (requestHeader: RequestHeader): Future[Result] = {

    val startTime = System.currentTimeMillis

    nextFilter(requestHeader).map { result =>

      val action = requestHeader.tags(Routes.ROUTE_CONTROLLER) +
        "." + requestHeader.tags(Routes.ROUTE_ACTION_METHOD)
      val endTime = System.currentTimeMillis
      val requestTime = endTime - startTime

      Logger.info(s"${action} took ${requestTime}ms" +
        s" and returned ${result.header.status}")

      result.withHeaders("Request-Time" -> requestTime.toString)
    }
  }
}
```

Routing tags are a feature of the Play router. If you use a custom router, or return a custom action in `Global.onRouteRequest`, these parameters may not be available.

# More powerful filters

Play provides a lower level filter API called `EssentialFilter` which gives you full access to the body of the request. This API allows you to wrap EssentialAction with another action. Here is the above filter example rewritten as an `EssentialFilter`:

```
import play.api.Logger
import play.api.mvc._
import play.api.libs.concurrent.Execution.Implicits.defaultContext

class LoggingFilter extends EssentialFilter {
  def apply(nextFilter: EssentialAction) = new EssentialAction {
    def apply(requestHeader: RequestHeader) = {

      val startTime = System.currentTimeMillis
```

```
    nextFilter(requestHeader).map { result =>

      val endTime = System.currentTimeMillis
      val requestTime = endTime - startTime

      Logger.info(s"${requestHeader.method} ${requestHeader.uri}" +
        s" took ${requestTime}ms and returned ${result.header.status}")
      result.withHeaders("Request-Time" -> requestTime.toString)

    }
  }
}
```

The key difference here, apart from creating a new `EssentialAction` to wrap the passed in `next` action, is when we invoke next, we get back an `Iteratee`. You could wrap this in an `Enumeratee` to do some transformations if you wished. We then `map` the result of the iteratee and thus handle it.

Although it may seem that there are two different filter APIs, there is only one,`EssentialFilter`. The simpler `Filter` API in the earlier examples extends`EssentialFilter`, and implements it by creating a new `EssentialAction`. The passed in callback makes it appear to skip the body parsing by creating a promise for the`Result`, while the body parsing and the rest of the action are executed asynchronously.

**Next:** HTTP request handlers

# HTTP Request Handlers

Play provides a range of abstractions for routing requests to actions, providing routers and filters to allow most common needs. Sometimes however an application will have more advanced needs that aren't met by Play's abstractions. When this is the case, applications can provide custom implementations of Play's lowest level HTTP pipeline API, the `HttpRequestHandler`.

Providing a custom `HttpRequestHandler` should be a last course of action. Most custom needs can be met through implementing a custom router or a filter.

# Implementing a custom request handler

The `HttpRequestHandler` trait has one method to be implemented,`handlerForRequest`. This takes the request to get a handler for, and returns a tuple of a`RequestHeader` and a `Handler`.

The reason why a request header is returned is so that information can be added to the request, for example, routing information. In this way, the router is able to tag requests with routing information, such as which route matched the request, which can be useful for monitoring or even for injecting cross cutting functionality.

A very simple request handler that simply delegates to a router might look like this:

```
import javax.inject.Inject
import play.api.http._
import play.api.mvc._
import play.api.routing.Router

class SimpleHttpRequestHandler @Inject() (router: Router) extends HttpRequestHandler {
  def handlerForRequest(request: RequestHeader) = {
    router.routes.lift(request) match {
      case Some(handler) => (request, handler)
      case None => (request, Action(Results.NotFound))
    }
  }
}
```

# Extending the default request handler

In most cases you probably won't want to create a new request handler from scratch, you'll want to build on the default one. This can be done by extendingDefaultHttpRequestHandler. The default request handler provides a number of methods that can be overridden, this allows you to implement your custom functionality without reimplementing the code to tag requests, handle errors, etc.

One use case for a custom request handler may be that you want to delegate to a different router, depending on what host the request is for. Here is an example of how this might be done:

```
import javax.inject.Inject
import play.api.http._
import play.api.mvc.RequestHeader

class VirtualHostRequestHandler @Inject() (errorHandler: HttpErrorHandler,
    configuration: HttpConfiguration, filters: HttpFilters,
    fooRouter: foo.Routes, barRouter: bar.Routes
  ) extends DefaultHttpRequestHandler(
    fooRouter, errorHandler, configuration, filters
  ) {

  override def routeRequest(request: RequestHeader) = {
    request.host match {
```

```
    case "foo.example.com" => fooRouter.routes.lift(request)
    case "bar.example.com" => barRouter.routes.lift(request)
    case _ => super.routeRequest(request)
   }
  }
}
```

# Configuring the http request handler

A custom http handler can be supplied by creating a class in the root package called `RequestHandler` that implements `HttpRequestHandler`.

If you don't want to place your request handler in the root package, or if you want to be able to configure different request handlers for different environments, you can do this by configuring the `play.http.requestHandler` configuration property in `application.conf`:

```
play.http.requestHandler = "com.example.RequestHandler"
```

## Performance notes

The http request handler that Play uses if none is configured is one that delegates to the legacy `GlobalSettings` methods. This may have a performance impact as it will mean your application has to do many lookups out of Guice to handle a single request. If you are not using a `Global` object, then you don't need this, instead you can configure Play to use the default http request handler:

```
play.http.requestHandler = "play.api.http.DefaultHttpRequestHandler"
```

**Next:** Dependency injection

# Runtime Dependency Injection

Dependency injection is a way that you can separate your components so that they are not directly dependent on each other, rather, they get injected into each other.

Out of the box, Play provides runtime dependency injection based on JSR 330. Runtime dependency injection is so called because the dependency graph is created, wired and validated at runtime. If a dependency cannot be found for a particular component, you won't get an error until you run your application. In contrast, Play also supports compile time dependency injection, where errors in the dependency graph are detected and thrown at compile time.

The default JSR 330 implementation that comes with Play is Guice, but other JSR 330 implementations can be plugged in.

# Declaring dependencies

If you have a component, such as a controller, and it requires some other components as dependencies, then this can be declared using the @Inject annotation.
The `@Inject`annotation can be used on fields or on constructors, we recommend that you use it on constructors, for example:

```
import javax.inject._
import play.api.libs.ws._

class MyComponent @Inject() (ws: WSClient) {
  // ...
}
```

Note that the `@Inject` annotation must come after the class name but before the constructor parameters, and must have parenthesis.

# Dependency injecting controllers

There are two ways to make Play use dependency injected controllers.

## Injected routes generator

By default, Play will generate a static router, that assumes that all actions are static methods. By configuring Play to use the injected routes generator, you can get Play to generate a router that will declare all the controllers that it routes to as dependencies, allowing your controllers to be dependency injected themselves.

We recommend always using the injected routes generator, the static routes generator exists primarily as a tool to aid migration so that existing projects don't have to make all their controllers non static at once.

To enable the injected routes generator, add the following to your build settings in`build.sbt`:

```
routesGenerator := InjectedRoutesGenerator
```

When using the injected routes generator, prefixing the action with an `@` symbol takes on a special meaning, it means instead of the controller being injected directly, a `Provider`of the controller will be injected. This allows, for example, prototype controllers, as well as an option for breaking cyclic dependencies.

### Injected actions

If using the static routes generator, you can indicate that an action has an injected controller by prefixing the action with `@`, like so:

```
GET     /some/path          @controllers.Application.index
```

# Component lifecycle

The dependency injection system manages the lifecycle of injected components, creating them as needed and injecting them into other components. Here's how component lifecycle works:

- **New instances are created every time a component is needed**. If a component is used more than once, then, by default, multiple instances of the component will be created. If you only want a single instance of a component then you need to mark it as a singleton.
- **Instances are created lazily when they are needed**. If a component is never used by another component, then it won't be created at all. This is usually what you want. For most components there's no point creating them until they're needed. However, in some cases you want components to be started up straight away or even if they're not used by another component. For example, you might want to send a message to a remote system or warm up a cache when the application starts. You can force a component to be created eagerly by using an eager binding.
- **Instances are *not* automatically cleaned up**, beyond normal garbage collection. Components will be garbage collected when they're no longer referenced, but the framework won't do anything special to shut down the component, like calling a `close` method. However, Play provides a special type of component, called the `ApplicationLifecycle` which lets you register components to shut down when the application stops.

# Singletons

Sometimes you may have a component that holds some state, such as a cache, or a connection to an external resource, or a component might be expensive to create. In these cases it may be important that there is only be one instance of that component. This can be achieved using the @Singleton annotation:

```
import javax.inject._

@Singleton
class CurrentSharePrice {
  @volatile private var price = 0

  def set(p: Int) = price = p
  def get = price
}
```

# Stopping/cleaning up

Some components may need to be cleaned up when Play shuts down, for example, to stop thread pools. Play provides an ApplicationLifecycle component that can be used to register hooks to stop your component when Play shuts down:

```
import scala.concurrent.Future
import javax.inject._
import play.api.inject.ApplicationLifecycle

@Singleton
class MessageQueueConnection @Inject() (lifecycle: ApplicationLifecycle) {
  val connection = connectToMessageQueue()
  lifecycle.addStopHook { () =>
    Future.successful(connection.stop())
  }

  //...
}
```

The `ApplicationLifecycle` will stop all components in reverse order from when they were created. This means any components that you depend on can still safely be used in your components stop hook, since because you depend on them, they must have been created before your component was, and therefore won't be stopped until after your component is stopped.

**Note:** It's very important to ensure that all components that register a stop hook are singletons. Any non singleton components that register stop hooks could potentially be a source of memory leaks, since a new stop hook will be registered each time the component is created.

# Providing custom bindings

It is considered good practice to define an trait for a component, and have other classes depend on that trait, rather than the implementation of the component. By doing that, you can inject different implementations, for example you inject a mock implementation when testing your application.

In this case, the DI system needs to know which implementation should be bound to that trait. The way we recommend that you declare this depends on whether you are writing a Play application as an end user of Play, or if you are writing library that other Play applications will consume.

## Play applications

We recommend that Play applications use whatever mechanism is provided by the DI framework that the application is using. Although Play does provide a binding API, this API is somewhat limited, and will not allow you to take full advantage of the power of the framework you're using.

Since Play provides support for Guice out of the box, the examples below show how to provide bindings for Guice.

### *Binding annotations*

The simplest way to bind an implementation to an interface is to use the Guice@ImplementedBy annotation. For example:

```
import com.google.inject.ImplementedBy

@ImplementedBy(classOf[EnglishHello])
trait Hello {
  def sayHello(name: String): String
}

class EnglishHello extends Hello {
  def sayHello(name: String) = "Hello " + name
}
```

### *Programmatic bindings*

In some more complex situations, you may want to provide more complex bindings, such as when you have multiple implementations of the one trait, which are qualified by@Named annotations. In these cases, you can implement a custom Guice Module:

```
import com.google.inject.AbstractModule
import com.google.inject.name.Names

class HelloModule extends AbstractModule {
  def configure() = {

    bind(classOf[Hello])
      .annotatedWith(Names.named("en"))
      .to(classOf[EnglishHello])

    bind(classOf[Hello])
      .annotatedWith(Names.named("de"))
      .to(classOf[GermanHello])
  }
}
```

To register this module with Play, append it's fully qualified class name to the`play.modules.enabled` list in `application.conf`:

```
play.modules.enabled += "modules.HelloModule"
```

### *Configurable bindings*

Sometimes you might want to read the Play `Configuration` or use a `ClassLoader`when you configure Guice bindings. You can get access to these objects by adding them to your module's constructor.

In the example below, the `Hello` binding for each language is read from a configuration file. This allows new `Hello` bindings to be added by adding new settings in your`application.conf` file.

```
import com.google.inject.AbstractModule
import com.google.inject.name.Names
import play.api.{ Configuration, Environment }
```

```
class HelloModule(
  environment: Environment,
  configuration: Configuration) extends AbstractModule {
  def configure() = {
    // Expect configuration like:
    // hello.en = "myapp.EnglishHello"
    // hello.de = "myapp.GermanHello"
    val helloConfiguration: Configuration =
      configuration.getConfig("hello").getOrElse(Configuration.empty)
    val languages: Set[String] = helloConfiguration.subKeys
    // Iterate through all the languages and bind the
    // class associated with that language. Use Play's
    // ClassLoader to load the classes.
    for (l <- languages) {
      val bindingClassName: String = helloConfiguration.getString(l).get
      val bindingClass: Class[_ <: Hello] =
        environment.classLoader.loadClass(bindingClassName)
        .asSubclass(classOf[Hello])
      bind(classOf[Hello])
        .annotatedWith(Names.named(l))
        .to(bindingClass)
    }
  }
}
```

**Note:** In most cases, if you need to access `Configuration` when you create a component, you should inject the `Configuration` object into the component itself or into the component's `Provider`. Then you can read the `Configuration` when you create the component. You usually don't need to read `Configuration` when you create the bindings for the component.

### *Eager bindings*

In the code above, new `EnglishHello` and `GermanHello` objects will be created each time they are used. If you only want to create these objects once, perhaps because they're expensive to create, then you should use the `@Singleton` annotation asdescribed above. If you want to create them once and also create them *eagerly* when the application starts up, rather than lazily when they are needed, then you can Guice's eager singleton binding.

```
import com.google.inject.AbstractModule
import com.google.inject.name.Names

class HelloModule extends AbstractModule {
  def configure() = {

    bind(classOf[Hello])
      .annotatedWith(Names.named("en"))
      .to(classOf[EnglishHello]).asEagerSingleton

    bind(classOf[Hello])
      .annotatedWith(Names.named("de"))
      .to(classOf[GermanHello]).asEagerSingleton
  }
```

```
}
```

Eager singletons can be used to start up a service when an application starts. They are often combined with a shutdown hook so that the service can clean up its resources when the application stops.

## Play libraries

If you're implementing a library for Play, then you probably want it to be DI framework agnostic, so that your library will work out of the box regardless of which DI framework is being used in an application. For this reason, Play provides a lightweight binding API for providing bindings in a DI framework agnostic way.

To provide bindings, implement a Module to return a sequence of the bindings that you want to provide. The `Module` trait also provides a DSL for building bindings:

```
import play.api.inject._

class HelloModule extends Module {
  def bindings(environment: Environment,
               configuration: Configuration) = Seq(
    bind[Hello].qualifiedWith("en").to[EnglishHello],
    bind[Hello].qualifiedWith("de").to[GermanHello]
  )
}
```

This module can be registered with Play automatically by appending it to the `play.modules.enabled` list in `reference.conf`:

```
play.modules.enabled += "com.example.HelloModule"
```

- The `Module` `bindings` method takes a Play `Environment` and `Configuration`. You can access these if you want to configure the bindings dynamically.
- Module bindings support eager bindings. To declare an eager binding, add `.eagerly` at the end of your `Binding`.

In order to maximise cross framework compatibility, keep in mind the following things:

- Not all DI frameworks support just in time bindings. Make sure all components that your library provides are explicitly bound.
- Try to keep binding keys simple - different runtime DI frameworks have very different views on what a key is and how it should be unique or not.

## Excluding modules

If there is a module that you don't want to be loaded, you can exclude it by appending it to the `play.modules.disabled` property in `application.conf`:

```
play.modules.disabled += "play.api.db.evolutions.EvolutionsModule"
```

# Advanced: Extending the GuiceApplicationLoader

Play's runtime dependency injection is bootstrapped by the `GuiceApplicationLoader` class. This class loads all the modules, feeds the modules into Guice, then uses Guice to create the application. If you want to control how Guice initializes the application then you can extend the `GuiceApplicationLoader` class. There are several methods you can override, but you'll usually want to override the `builder` method. This method reads the `ApplicationLoader.Context` and creates a `GuiceApplicationBuilder`. Below you can see the standard implementation for `builder`, which you can change in any way you like. You can find out how to use the `GuiceApplicationBuilder` in the section about testing with Guice.

```
import play.api.ApplicationLoader
import play.api.Configuration
import play.api.inject._
import play.api.inject.guice._

class CustomApplicationLoader extends GuiceApplicationLoader() {
  override def builder(context: ApplicationLoader.Context): GuiceApplicationBuilder = {
    val extra = Configuration("a" -> 1)
    initialBuilder
      .in(context.environment)
      .loadConfig(extra ++ context.initialConfiguration)
      .overrides(overrides(context): _*)
  }
}
```

When you override the `ApplicationLoader` you need to tell Play. Add the following setting to your `application.conf`:

```
play.application.loader = "modules.CustomApplicationLoader"
```

You're not limited to using Guice for dependency injection. By overriding the `ApplicationLoader` you can take control of how the application is initialized. Find out more in the next section.

**Next:** Compile time dependency injection

# Compile Time Dependency Injection

Out of the box, Play provides a mechanism for runtime dependency injection - that is, dependency injection where dependencies aren't wired until runtime. This approach has both advantages and disadvantages, the main advantages being around minimisation of boilerplate code, the main disadvantage being that the construction of the application is not validated at compile time.

An alternative approach that is popular in Scala development is to use compile time dependency injection. At its simplest, compile time DI can be achieved by manually constructing and wiring dependencies. Other more advanced techniques and tools exist, such as macro based autowiring tools, implicit auto wiring techniques, and various forms of the cake pattern. All of these can be easily implemented on top of constructors and manual wiring, so Play's support for compile time dependency injection is provided by providing public constructors and factory methods as API.

In addition to providing public constructors and factory methods, all of Play's out of the box modules provide some traits that implement a lightweight form of the cake pattern, for convenience. These are built on top of the public constructors, and are completely optional. In some applications, they will not be appropriate to use, but in many applications, they will be a very convenient mechanism to wiring the components provided by Play. These traits follow a naming convention of ending the trait name with `Components`, so for example, the default HikariCP based implementation of the DB API provides a trait called HikariCPComponents.

In the examples below, we will show how to wire a Play application manually using the built in component helper traits. By reading the source code of these traits it should be trivial to adapt this to any compile time dependency injection technique you please.

# Current application

One aim of dependency injection is to eliminate global state, such as singletons. Play 2 was designed with an assumption of global state. Play 3 will hopefully remove this global state, however that is a major breaking task. In the meantime, Play will be a bit of a hybrid state, with some parts not using global state, and other parts using global state.

By using dependency injection throughout your application, you should be able to ensure though that your components can be tested in isolation, not requiring starting an entire application to run a single test.

# Application entry point

Every application that runs on the JVM needs an entry point that is loaded by reflection - even if your application starts itself, the main class is still loaded by reflection, and its main method is located and invoked using reflection.

In Play's dev mode, the JVM and HTTP server used by Play must be kept running between restarts of your application. To implement this, Play provides anApplicationLoader trait that you can implement. The application loader is constructed and invoked every time the application is reloaded, to load the application.

This trait's load method takes as an argument the application loader Context, which contains all the components required by a Play application that outlive the application itself and cannot be constructed by the application itself. A number of these components exist specifically for the purposes of providing functionality in dev mode, for example, the source mapper allows the Play error handlers to render the source code of the place that an exception was thrown.

The simplest implementation of this can be provided by extending the PlayBuiltInComponentsFromContext abstract class. This class takes the context, and provides all the built in components, based on that context. The only thing you need to provide is a router for Play to route requests to. Below is the simplest application that can be created in this way, using a null router:

```
import play.api._
import play.api.ApplicationLoader.Context
import play.api.routing.Router

class MyApplicationLoader extends ApplicationLoader {
  def load(context: Context) = {
    new MyComponents(context).application
  }
}

class MyComponents(context: Context) extends BuiltInComponentsFromContext(context) {
  lazy val router = Router.empty
}
```

To configure Play to use this application loader, configure the`play.application.loader` property to point to the fully qualified class name in the`application.conf` file:

```
play.application.loader=MyApplicationLoader
```

# Providing a router

By default Play will generate a static router that requires all of your actions to be objects. Play however also supports generating a router than can be dependency injected, this can be enabled by adding the following configuration to your `build.sbt`:

```
routesGenerator := InjectedRoutesGenerator
```

When you do this, Play will generate a router with a constructor that accepts each of the controllers and included routers from your routes file, in the order they appear in your routes file. The routers constructor will also, as its first argument, accept an`HttpErrorHandler`, which is used to handle parameter binding errors. The primary constructor will also accept a

prefix String as the last argument, but an overloaded constructor that defaults this to `"/"` will also be provided.

The following routes:

```
GET     /                   controllers.Application.index
GET     /foo                controllers.Application.foo
->      /bar                bar.Routes
GET     /assets/*file       controllers.Assets.at(path = "/public", file)
```

Will produce a router with the following constructor signatures:

```
class Routes(
  override val errorHandler: play.api.http.HttpErrorHandler,
  Application_0: controllers.Application,
  bar_Routes_0: bar.Routes,
  Assets_1: controllers.Assets,
  val prefix: String
) extends GeneratedRouter {

  def this(
    errorHandler: play.api.http.HttpErrorHandler,
    Application_0: controllers.Application,
    bar_Routes_0: bar.Routes,
    Assets_1: controllers.Assets
  ) = this(Application_0, bar_Routes_0, Assets_1, "/")
  ...
}
```

Note that the naming of the parameters is intentionally not well defined (and in fact the index that is appended to them is random, depending on hash map ordering), so you should not depend on the names of these parameters.

To use this router in an actual application:

```
import play.api._
import play.api.ApplicationLoader.Context
import router.Routes

class MyApplicationLoader extends ApplicationLoader {
  def load(context: Context) = {
    new MyComponents(context).application
  }
}

class MyComponents(context: Context) extends BuiltInComponentsFromContext(context) {

  lazy val router = new Routes(httpErrorHandler, applicationController, barRoutes, assets)

  lazy val barRoutes = new bar.Routes(httpErrorHandler)
  lazy val applicationController = new controllers.Application()
```

```
  lazy val assets = new controllers.Assets(httpErrorHandler)
}
```

# Using other components

As described before, Play provides a number of helper traits for wiring in other components. For example, if you wanted to use the messages module, you can mix inI18nComponents into your components cake, like so:

```
import play.api.i18n._

class MyComponents(context: Context) extends BuiltInComponentsFromContext(context)
                         with I18nComponents {
  lazy val router = Router.empty

  lazy val myComponent = new MyComponent(messagesApi)
}

class MyComponent(messages: MessagesApi) {
  // ...
}
```

**Next:** Advanced routing

# String Interpolating Routing DSL

Play provides a DSL for defining embedded routers called the *String Interpolating Routing DSL*, or sird for short. This DSL has many uses, including embedding a light weight Play server, providing custom or more advanced routing capabilities to a regular Play application, and mocking REST services for testing.

Sird is based on a string interpolated extractor object. Just as Scala supports interpolating parameters into strings for building strings (and any object for that matter), such as `s"Hello $to"`, the same mechanism can also be used to extract parameters out of strings, for example in case statements.

The DSL lives in the `play.api.routing.sird` package. Typically, you will want to import this package, as well as a few other packages:

```
import play.api.mvc._
import play.api.routing._
import play.api.routing.sird._
```

A simple example of its use is:

```
val router = Router.from {
```

```
  case GET(p"/hello/$to") => Action {
    Results.Ok(s"Hello $to")
  }
}
```

In this case, the `$to` parameter in the interpolated path pattern will extract a single path segment for use in the action. The `GET` extractor extracts requests with the `GET` method. It takes a `RequestHeader` and extracts the same `RequestHeader` parameter, it's only used as a convenient filter. Other method extractors, including `POST`, `PUT` and `DELETE` are also supported.

Like Play's compiled router, sird supports matching multi path segment parameters, this is done by postfixing the parameter with `*`:

```
val router = Router.from {
  case GET(p"/assets/$file*") =>
    Assets.versioned(path = "/public", file = file)
}
```

Regular expressions are also supported, by postfixing the parameter with a regular expression in angled brackets:

```
val router = Router.from {
  case GET(p"/items/$id<[0-9]+>") => Action {
    Results.Ok(s"Item $id")
  }
}
```

Query parameters can also be extracted, using the `?` operator to do further extractions on the request, and using the `q` extractor:

```
val router = Router.from {
  case GET(p"/search" ? q"query=$query") => Action {
    Results.Ok(s"Searching for $query")
  }
}
```

While `q` extracts a required query parameter as a `String`, `q_?` or `q_o` if using Scala 2.10 extracts an optional query parameter as `Option[String]`:

```
val router = Router.from {
  case GET(p"/items" ? q_o"page=$page") => Action {
    val thisPage = page.getOrElse("1")
    Results.Ok(s"Showing page $thisPage")
  }
}
```

Likewise, `q_*` or `q_s` can be used to extract a sequence of multi valued query parameters:

```
val router = Router.from {
  case GET(p"/items" ? q_s"tag=$tags") => Action {
    val allTags = tags.mkString(", ")
    Results.Ok(s"Showing items tagged: $allTags")
  }
}
```

Multiple query parameters can be extracted using the `&` operator:

```
val router = Router.from {
  case GET(p"/items" ? q_o"page=$page"
```

```
   & q_o"per_page=$perPage") => Action {
   val thisPage = page.getOrElse("1")
   val pageLength = perPage.getOrElse("10")

   Results.Ok(s"Showing page $thisPage of length $pageLength")
  }
}
```

Since sird is just a regular extractor object (built by string interpolation), it can be combined with any other extractor object, including extracting its sub parameters even further. Sird provides some useful extractors for some of the most common types out of the box, namely `int`, `long`, `float`, `double` and `bool`:

```
val router = Router.from {
  case GET(p"/items/${int(id)}") => Action {
    Results.Ok(s"Item $id")
  }
}
```

In the above, `id` is of type `Int`. If the `int` extractor failed to match, then of course, the whole pattern will fail to match.

Similarly, the same extractors can be used with query string parameters, including multi value and optional query parameters. In the case of optional or multi value query parameters, the match will fail if any of the values present can't be bound to the type, but no parameters present doesn't cause the match to fail:

```
val router = Router.from {
  case GET(p"/items" ? q_o"page=${int(page)}") => Action {
    val thePage = page.getOrElse(1)
    Results.Ok(s"Items page $thePage")
  }
}
```

To further the point that these are just regular extractor objects, you can see here that you can use all other features of a `case` statement, including `@` syntax and if statements:

```
val router = Router.from {
  case rh @ GET(p"/items/${idString @ int(id)}" ?
    q"price=${int(price)}")
    if price > 200 =>
    Action {
      Results.Ok(s"Expensive item $id")
    }
}
```

**Next:** Javascript routing

# Javascript Routing

The play router is able to generate Javascript code to handle routing from Javascript running client side back to your application. The Javascript router aids in refactoring your application. If you change the structure of your URLs or parameter names your Javascript gets automatically updated to use that new structure.

# Generating a Javascript router

The first step to using Play's Javascript router is to generate it. The router will only expose the routes that you explicitly declare thus minimising the size of the Javascript code.

There are two ways to generate a Javascript router. One is to embed the router in the HTML page using template directives. The other is to generate Javascript resources in an action that can be downloaded, cached and shared between pages.

## Embedded router

An embedded router can be generated using the `@javascriptRouter` directive inside a Scala template. This is typically done inside the main decorating template.

```
@helper.javascriptRouter("jsRoutes")(
  routes.javascript.Users.list,
  routes.javascript.Users.get
)
```

The first parameter is the name of the global variable that the router will be placed in. The second parameter is the list of Javascript routes that should be included in this router. In order to use this function, your template must have an implicit RequestHeader in scope. For example this can be made available by adding `(implicit req: RequestHeader)` to the end of your parameter declarations.

## Router resource

A router resource can be generated by creating an action that invokes the router generator. It has a similar syntax to embedding the router in a template:

```
def javascriptRoutes = Action { implicit request =>
  Ok(
    JavaScriptReverseRouter("jsRoutes")(
      routes.javascript.Users.list,
      routes.javascript.Users.get
    )
  ).as("text/javascript")
}
```

Then, add the corresponding route:

```
GET /javascriptRoutes  controllers.Application.javascriptRoutes
```

Having implemented this action, and adding it to your routes file, you can then include it as a resource in your templates:

```
<script type="text/javascript" src="@routes.Application.javascriptRoutes"></script>
```

# Using the router

Using jQuery as an example, making a call is as simple as:

```
$.ajax(jsRoutes.controllers.Users.get(someId))
  .done( /*...*/ )
  .fail( /*...*/ );
```

The router also makes a few other properties available including the `url` and the `type`(the HTTP method). For example the above call to jQuery's ajax function can also be made like:

```
var r = jsRoutes.controllers.Users.get(someId);
$.ajax({url: r.url, type: r.type, success: /*...*/, error: /*...*/ });
```

The above approach is required where other properties need setting such as success, error, context etc.

The `absoluteURL` and the `webSocketURL` are methods (not properties) which return the complete url string. A Websocket connection can be made like:

```
var r = jsRoutes.controllers.Users.list();
var ws = new WebSocket(r.webSocketURL());
ws.onmessage = function(msg) {
      /*...*/
};
```

# jQuery ajax method support

**Note:** Built-in support for jQuery's ajax function will be removed in a future release. This section on the built-in support is provided for reference purposes only. Please do not use the router's ajax function in new code and consider upgrading existing code as soon as possible. The previous section on using the router documents how jQuery should be used.

If jQuery isn't your thing, or if you'd like to decorate the jQuery ajax method in some way, you can provide a function to the router to use to perform ajax queries. This function must accept the object that is passed to the `ajax` router method, and should expect the router to have set the `type` and `url` properties on it to the appropriate method and url for the router request.

To define this function, in your action pass the `ajaxMethod` method parameter, eg:

```
Routes.javascriptRouter("jsRoutes", Some("myAjaxFunction") ...
```

**Next:** Extending Play

# Writing Plugins

**Note**: Plugins are deprecated. Instead, use Modules.

In the context of the Play runtime, a plugin is a class that is able to plug into the Play lifecycle, and also allows sharing components in a non static way in your application.

Not every library that adds functionality to Play is or needs to be a plugin in this context - a library that provides a custom filter for example does not need to be a plugin.

Similarly, plugins don't necessarily imply that they are reusable between applications, it is often very useful to implement a plugin locally within an application, in order to hook into the Play lifecycle and share components in your code.

---

# Implementing plugins

Implementing a plugin requires two steps. The first is to implement the `play.api.Plugin` interface:

```
package plugins

import play.api.{Plugin, Application}

class MyPlugin extends Plugin {
  val myComponent = new MyComponent()

  override def onStart() = {
    myComponent.start()
  }

  override def onStop() = {
    myComponent.stop()
  }

  override def enabled = true
}
```

The next step is to register this with Play. This can be done by creating a file called `play.plugins` and placing it in the root of the classloader. In a typical Play app, this means putting it in the `conf` folder:

```
2000:plugins.MyPlugin
```

Each line in the `play.plugins` file contains a number followed by the fully qualified name of the plugin to load. The number is used to control lifecycle ordering, lower numbers will be started first and stopped last. Multiple plugins can be declared in the one file, and any lines started with `#` are treated as comments.

Choosing the right number for ordering for a plugin is important, it needs to fit in appropriate according to what other plugins it depends on. The plugins that Play uses use the following ordering numbers:

- *100* - Utilities that have no dependencies, such as the messages plugin
- *200* - Database connection pools
- *300-500* - Plugins that depend on the database, such as JPA, Ebean and evolutions
- *600* - The Play cache plugin
- *700* - The WS plugin
- *1000* - The Akka plugin
- *10000* - The Global plugin, which invokes the `Global.onStart` and `Global.onStop` methods. This plugin is intended to execute last.

# Accessing plugins

Plugins can be accessed via the `plugin` method on `play.api.Application`:

```
import play.api.Play
import play.api.Play.current

val myComponent = Play.application.plugin[MyPlugin]
  .getOrElse(throw new RuntimeException("MyPlugin not loaded"))
  .myComponent
```

# Actor example

A common use case for using plugins is to create and share actors around the application. This can be done by implementing an actors plugin:

```
package actors

import play.api._
import play.api.libs.concurrent.Akka
import akka.actor._
import javax.inject.Inject

class Actors @Inject() (implicit app: Application) extends Plugin {
  lazy val myActor = Akka.system.actorOf(MyActor.props, "my-actor")
}

object Actors {
  def myActor: ActorRef = Play.current.plugin[Actors]
    .getOrElse(throw new RuntimeException("Actors plugin not loaded"))
    .myActor
}
```

Note the `Actors` companion object methods that allow easy access to the `ActorRef` for each actor, instead of code having to use the plugins API directly.
The plugin can then be registered in `play.plugins`:

```
1100:actors.Actors
```

The reason `1100` was chosen for the ordering was because this plugin depends on the Akka plugin, and so must start after that.

**Next:** Embedding Play

# Embedding a Play server in your application

While Play apps are most commonly used as their own container, you can also embed a Play server into your own existing application. This can be used in conjunction with the Twirl template compiler and Play routes compiler, but these are of course not necessary, a common use case for embedding a Play application will be because you only have a few very simple routes.

The simplest way to start an embedded Play server is to use the `NettyServer` factory methods. If all you need to do is provide some straightforward routes, you may decide to use the String Interpolating Routing DSL in combination with the `fromRouter` method:

```
import play.core.server._
import play.api.routing.sird._
import play.api.mvc._

val server = NettyServer.fromRouter() {
  case GET(p"/hello/$to") => Action {
    Results.Ok(s"Hello $to")
  }
}
```

By default, this will start a server on port 9000 in prod mode. You can configure the server by passing in a `ServerConfig`:

```
import play.core.server._
import play.api.routing.sird._
import play.api.mvc._

val server = NettyServer.fromRouter(ServerConfig(
  port = Some(19000),
  address = "127.0.0.1"
)) {
  case GET(p"/hello/$to") => Action {
    Results.Ok(s"Hello $to")
  }
}
```

You may want to customise some of the components that Play provides, for example, the HTTP error handler. A simple way of doing this is by using Play's components traits,

the `NettyServerComponents` trait is provided for this purpose, and can be conveniently combined with `BuiltInComponents` to build the application that it requires:

```
import play.core.server._
import play.api.routing.Router
import play.api.routing.sird._
import play.api.mvc._
import play.api.BuiltInComponents
import play.api.http.DefaultHttpErrorHandler
import scala.concurrent.Future

val components = new NettyServerComponents with BuiltInComponents {

  lazy val router = Router.from {
    case GET(p"/hello/$to") => Action {
      Results.Ok(s"Hello $to")
    }
  }

  override lazy val httpErrorHandler = new DefaultHttpErrorHandler(environment,
    configuration, sourceMapper, Some(router)) {

    override protected def onNotFound(request: RequestHeader, message: String) = {
      Future.successful(Results.NotFound("Nothing was found!"))
    }
  }
}
val server = components.server
```

In this case, the server configuration can be overridden by overriding the `serverConfig` property.

To stop the server once you've started it, simply call the `stop` method:

```
server.stop()
```

**Note:** Play requires an application secret to be configured in order to start. This can be configured by providing an `application.conf` file in your application, or using the `play.crypto.secret` system property.

**Next:** Play for Java developers