Play 2.0 for Java developers

2012.04.19

Play 2.0 for Java developers

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

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

Main concepts

Tutorials

1. HTTP Programming

Actions, Controllers and Results

What is an Action?

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

An action is basically a Java method that processes the request parameters, and produces a result to be sent to the client.

```
public static Result index() {
  return ok("Got request " + request() + "!");
}
```

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

Controllers

A controller is nothing more than a class extending `play.mvc.Controller` that groups several action methods.

The simplest syntax for defining an action is a static method with no parameters that returns a `Result` value:

```
public static Result index() {
  return ok("It works!");
}
```

An action method can also have parameters:

```
public static Result index(String name) {
  return ok("Hello" + name);
}
```

These parameters will be resolved by the `Router` and will be filled with values from the request URL. The parameter values can be extracted from either the URL path or the URL query string.

Results

Let's start with simple results: an HTTP result with a status code, a set of HTTP headers and a body to be sent to the web client.

These results are defined by `play.mvc.Result`, and the `play.mvc.Results` class provides several helpers to produce standard HTTP results, such as the `ok` method we used in the previous section:

```
public static Result index() {
  return ok("Hello world!");
}
```

Here are several examples that create various results:

```
Result ok = ok("Hello world!");
Result notFound = notFound();
Result pageNotFound = notFound("<h1>Page not found</h1>").as("text/html");
Result badRequest = badRequest(views.html.form.render(formWithErrors));
Result oops = internalServerError("Oops");
Result anyStatus = status(488, "Strange response type");
```

All of these helpers can be found in the `play.mvc.Results` class.

Redirects are simple results too

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

There are several helpers available to create redirect results:

```
public static Result index() {
  return redirect("/user/home");
}
```

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

```
public static Result index() {
  return temporaryRedirect("/user/home");
}
```

Next: HTTP Routing

HTTP routing

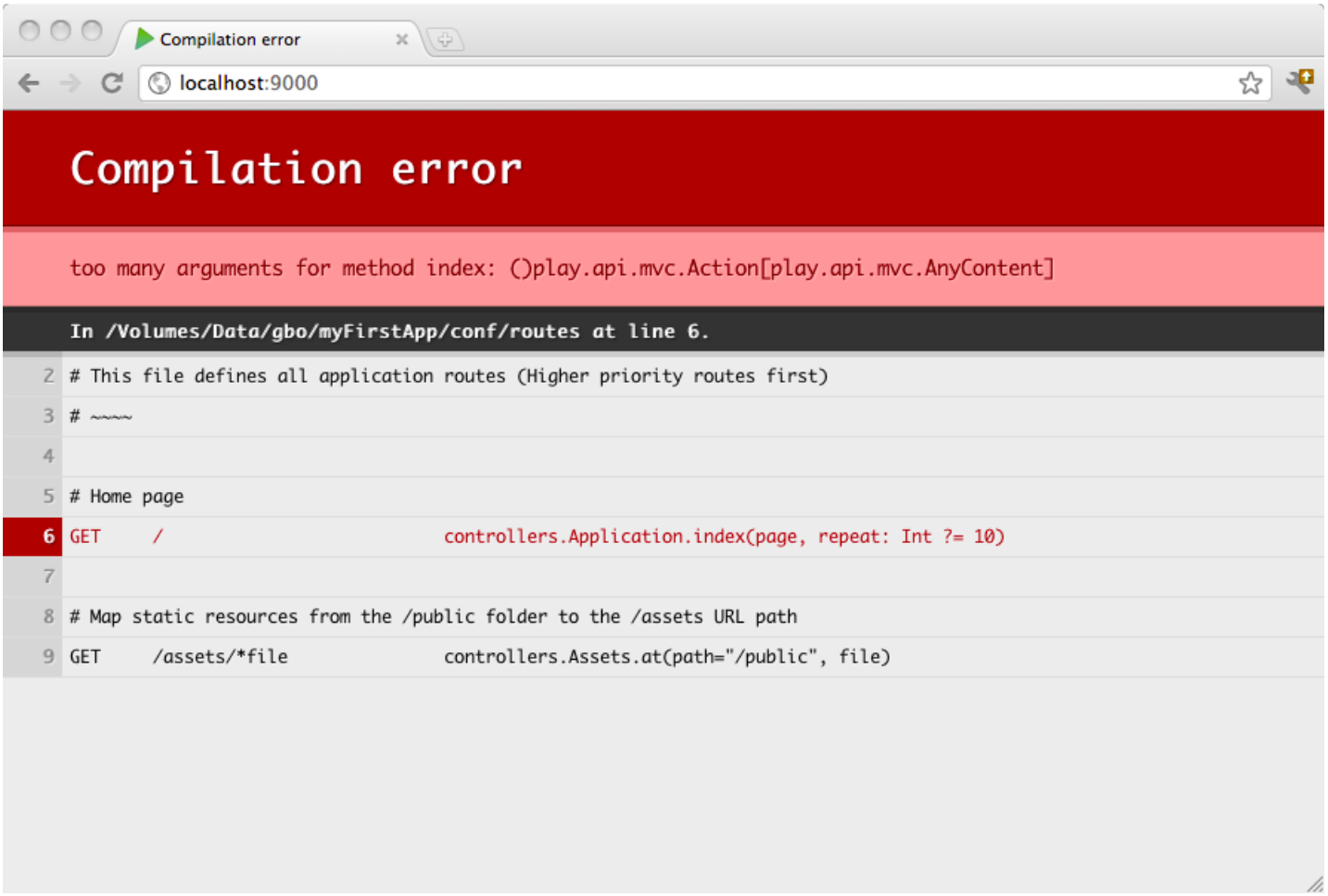
The built-in HTTP router

The router is the component that translates each incoming HTTP request to an action call (a static, public method in a controller class).

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

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

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



The routes file syntax

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

Let's see what a route definition looks like:

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

> Note that in the action call, the parameter type comes after the parameter name, like in Scala.

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

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

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

The HTTP method

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

The URI pattern

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

Static path

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

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

Dynamic parts

If you want to define a route that, say, retrieves a client by id, you need to add a dynamic part:

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

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

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

Dynamic parts spanning several /

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

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

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

Dynamic parts with custom regular expressions

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

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

Call to action generator method

The last part of a route definition is the call. This part must define a valid call to an action method.

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

```
GET   /                     controllers.Application.homePage()
```

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

```
# Extract the page parameter from the path.
# i.e. http://myserver.com/index
GET   /:page                controllers.Application.show(page)
```

Or:

```
# Extract the page parameter from the query string.
# i.e. http://myserver.com/?page=index
GET   /                     controllers.Application.show(page)
```

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

```
public static Result show(String page) {
```

```
  String content = Page.getContentOf(page);
  response().setContentType("text/html");
  return ok(content);
}
```

Parameter types

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

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

Then use the same type for the corresponding action method parameter in the controller:

```
public static Result show(Long id) {
  Client client = Client.findById(id);
  return ok(views.html.Client.show(client));
}
```

> **Note:** The parameter types are specified using a suffix syntax. Also The generic types are specified using the `[]` symbols instead of <>, as in Java. For example, `List[String]` is the same type as the Java `List<String>`.

Parameters with fixed values

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

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

Parameters with default values

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

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

Routing priority

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

Reverse routing

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

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

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

For example, if you create a controller like:

```
package controllers;

import play.*;
import play.mvc.*;

public class Application extends Controller {

  public static Result hello(String name) {
      return ok("Hello " + name + "!");
  }

}
```

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

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

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

```
// Redirect to /hello/Bob
public static Result index() {
    return redirect(controllers.routes.Application.hello("Bob"));
}
```

Next: Manipulating the response

Manipulating the response

Changing the default Content-Type

The result content type is automatically inferred from the Java value you specify as body.

For example:

```
Result textResult = ok("Hello World!");
```

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

```
Result jsonResult = ok(jerksonObject);
```

will set the `Content-Type` header to `application/json`.

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

```
Result htmlResult = ok("<h1>Hello World!</h1>").as("text/html");
```

You can also set the content type on the HTTP response:

```
public static Result index() {
  response().setContentType("text/html");
  return ok("<h1>Hello World!</h1>");
}
```

Setting HTTP response headers

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

```
public static Result index() {
  response().setContentType("text/html");
  response().setHeader(CACHE_CONTROL, "max-age=3600");
  response().setHeader(ETAG, "xxx");
  return ok("<h1>Hello World!</h1>");
}
```

Note that setting an HTTP header will automatically discard any previous value.

Setting and discarding cookies

Cookies are just a special form of HTTP headers, but Play provides a set of helpers to make it easier.

You can easily add a Cookie to the HTTP response:

```
response().setCookie("theme", "blue");
```

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

```
response().discardCookies("theme");
```

Specifying the character encoding for text results

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

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

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

```
public static Result index() {
  response().setContentType("text/html; charset=iso-8859-1");
  return ok("<h1>Hello World!</h1>", "iso-8859-1");
}
```

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 the Flash scope. Data stored in the Session are available during the whole user session, and data stored in the flash scope are only available to the next request.

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

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

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

Reading a Session value

You can retrieve the incoming Session from the HTTP request:

```
public static Result index() {
  String user = session("connected");
  if(user != null) {
    return ok("Hello " + user);
  } else {
    return unauthorized("Oops, you are not connected");
  }
}
```

Storing data into the Session

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

```
public static Result index() {
  session("connected", "user@gmail.com");
  return ok("Welcome!");
}
```

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

```
public static Result index() {
  session.remove("connected");
  return ok("Bye");
}
```

Discarding the whole session

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

```
public static Result index() {
  session().clear();
  return ok("Bye");
}
```

Flash scope

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

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

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

Here are a few examples using the Flash scope:

```
public static Result index() {
  String message = flash("success");
  if(message == null) {
    message = "Welcome!";
  }
  return ok(message);
}

public static Result save() {
  flash("success", "The item has been created");
  return redirect("/home");
}
```

Next: Body parsers

Body parsers

What is a body parser?

An HTTP request (at least for those using the POST and PUT operations) contains a body. This body can be formatted with any format specified in the Content-Type header. A **body parser** transforms this request body into a Java value.

> **Note:** You can't write `BodyParser` implementation directly using Java. Because a Play `BodyParser` must handle the body content incrementaly using an `Iteratee[Array[Byte], A]` it must be implemented in Scala.
>
> However Play provides default `BodyParser`s that should fit most use cases (parsing Json, Xml, Text, uploading files). And you can reuse these default parsers to create your own directly in Java; for example you can provide an RDF parsers based on the Text one.

The `BodyParser` Java API

In the Java API, all body parsers must generate a `play.mvc.Http.RequestBody` value. This value computed by the body parser can then be retrieved via `request().body()`:

```
pulic static Result index() {
  RequestBody body = request().body();
  ok("Got body: " + body);
}
```

You can specify the `BodyParser` to use for a particular action using the `@BodyParser.Of` annotation:

```
@BodyParser.Of(BodyParser.Json.class)
pulic static Result index() {
  RequestBody body = request().body();
  ok("Got json: " + body.asJson());
}
```

The `Http.RequestBody` API

As we just said all body parsers in the Java API will give you a `play.mvc.Http.RequestBody` value. From this body object you can retrieve the request body content in the most appropriate Java type.

> **Note:** The `RequestBody` methods like `asText()` or `asJson()` will return null if the parser used to compute this request body doesn't support this content type. For example in an action method annotated with `@BodyParser.Of(BodyParser.Json.class)`, calling `asXml()` on the generated body will retun null.

Some parsers can provide a most specific type than `Http.RequestBody` (ie. a subclass of `Http.RequestBody`). You can automatically cast the request body into another type using the `as(...)` helper method:

```
@BodyParser.Of(BodyLengthParser.class)
pulic static Result index() {
  BodyLength body = request().body().as(BodyLength.class);
  ok("Request body length: " + body.getLength());
}
```

Default body parser: AnyContent

If you don't specify your own body parser, Play will use the default one guessing the most appropriate content type from the `Content-Type` header:

- **text/plain**: `String`, accessible via `asText()`
- **application/json**: `JsonNode`, accessible via `asJson()`
- **text/xml**: `org.w3c.Document`, accessible via `asXml()`
- **application/form-url-encoded**: `Map<String, String[]>`, accessible via `asFormUrlEncoded()`
- **multipart/form-data**: `Http.MultipartFormData`, accessible via `asMultipartFormData()`
- Any other content type: `Http.RawBuffer`, accessible via `asRaw()`

Example:

```
pulic static Result save() {
  RequestBody body = request().body();
  String textBody = body.asText();

  if(textBody != null) {
    ok("Got: " + text);
  } else {
    badRequest("Expecting text/plain request body");
  }
}
```

Max content length

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

There is a default content length (the default is 100KB).

> **Tip:** The default content size can be defined in `application.conf`:
>
> ```
> parsers.text.maxLength=128K
> ```

You can also specify a maximum content length via the `@BodyParser.of` annotation:

```
// Accept only 10KB of data.
@BodyParser.Of(value = BodyParser.Text.class, maxLength = 10 * 1024)
pulic static Result index() {
  if(request().body().isMaxSizeExceeded()) {
    return badRequest("Too much data!");
  } else {
    ok("Got body: " + request().body().asText());
  }
}
```

> **Next:** Actions composition

Action composition

This chapter introduces several ways to define generic action functionality.

Reminder about actions

Previously, we said that an action is a Java method that returns a `play.mvc.Result` value. Actually, Play manages internally actions as functions. Because Java doesn't support first class functions, an action provided by the Java API is an instance of `play.mvc.Action`:

```
public abstract class Action {

  public abstract Result call(Http.Context ctx);

}
```

Play builds a root action for you that just calls the proper action method. This allows for more complicated action composition.

Composing actions

You can compose the code provided by the action method with another `play.mvc.Action`, using the `@With` annotation:

```
@With(VerboseAction.class)
public static Result index() {
  return ok("It works!");
}
```

Here is the definition of the `VerboseAction`:

```
public class VerboseAction extends Action.Simple {

  public Result call(Http.Context ctx) throws Throwable {
    Logger.info("Calling action for " + ctx);
    return delegate.call(ctx);
  }
}
```

At one point you need to delegate to the wrapped action using `delegate.call(...)`.

You also mix with several actions:

```
@With(Authenticated.class, Cached.class)
public static Result index() {
  return ok("It works!");
}
```

> **Note:** `play.mvc.Security.Authenticated` and `play.cache.Cached` annotations and the corresponding predefined Actions are shipped with Play. See the relevant API documentation for more information.

Defining custom action annotations

You can also mark action composition with your own annotation, which must itself be annotated using

`@With`:

```
@With(VerboseAction.class)
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Verbose {
   boolean value() default true;
}
```

You can then use your new annotation with an action method:

```
@Verbose(false)
public static Result index() {
  return ok("It works!");
}
```

Your `Action` definition retrieves the annotation as configuration:

```
public class VerboseAction extends Action<Verbose> {

  public Result call(Http.Context ctx) {
    if(configuration.value) {
      Logger.info("Calling action for " + ctx);
    }
    return delegate.call(ctx);
  }
}
```

Annotating controllers

You can also put any action composition annotation directly on the `Controller` class. In this case it will be applied to all action methods defined by this controller.

```
@Authenticated
public Admin extends Controller {

  …

}
```

Next: Asynchronous HTTP programming

2. Asynchronous HTTP programming

Handling asynchronous results

Why asynchronous results?

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

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

A `Promise<Result>` will eventually be redeemed with a value of type `Result`. By giving a `Promise<Result>` instead of a normal `Result`, we are able to compute the result quickly without blocking anything. Play will then serve this result as soon as the promise is redeemed.

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

How to create a `Promise<Result>`

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

```
Promise<Double> promiseOfPIValue = computePIAsynchronously();
Promise<Result> promiseOfResult = promiseOfPIValue.map(
  new Function<Double,Result>() {
    public Result apply(Double pi) {
      return ok("PI value computed: " + pi);
    }
  }
);
```

> **Note:** Writing functional composition in Java is really verbose for at the moment, but it should be better when Java supports lambda notation.

Play 2.0 asynchronous API methods give you a `Promise`. This is the case when you are calling an external web service using the `play.libs.WS` API, or if you are using Akka to schedule asynchronous tasks or to communicate with Actors using `play.libs.Akka`.

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

```
Promise<Integer> promiseOfInt = Akka.future(
  new Callable<Integer>() {
    public Integer call() {
      intensiveComputation();
    }
  }
);
```

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

AsyncResult

While we were using `Results.Status` until now, to send an asynchronous result we need an `Results.AsyncResult` that wraps the actual result:

```
public static Result index() {
  Promise<Integer> promiseOfInt = Akka.future(
    new Callable<Integer>() {
      public Integer call() {
        intensiveComputation();
      }
    }
  );
  async(
    promiseOfInt.map(
      new Function<Integer,Result>() {
        public Result apply(Integer i) {
          return ok("Got result: " + i);
        }
      }
    )
  );
}
```

Note: `async()` is an helper method building an `AsyncResult` from a `Promise<Result>`.

Next: Streaming HTTP responses

Streaming HTTP responses

Standard responses and Content-Length header

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

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

```
public static Result index() {
  return ok("Hello World")
}
```

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

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

To be able to compute the `Content-Length` header properly, Play must consume the whole response data and load its content into memory.

Serving files

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

Play provides easy to use helpers to this common task of serving a local file:

```
public static Result index() {
  return ok(new java.io.File("/tmp/fileToServe.pdf"));
}
```

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

Chunked responses

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

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

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

finished receiving data for that chunk. The data transfer is terminated by a final chunk of length zero.

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

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

Let's say that we have a service somewhere that provides a dynamic `InputStream` that computes some data. We can ask Play to stream this content directly using a chunked response:

```
public static Result index() {
  InputStream is = getDynamicStreamSomewhere();
  return ok(is);
}
```

You can also set up your own chunked response builder. The Play Java API supports both text and binary chunked streams (via `String` and `byte[]`):

```
public static index() {
  // Prepare a chunked text stream
  Chunks<String> chunks = new StringChunks() {

    // Called when the stream is ready
    public void onReady(Chunks.Out<String> out) {
      registerOutChannelSomewhere(out);
    }

  }

  // Serves this stream with 200 OK
  ok(chunks);
}
```

The `onReady` method is called when it is safe to write to this stream. It gives you a `Chunks.Out` channel you can write to.

Let's say we have an asynchronous process (like an `Actor`) somewhere pushing to this stream:

```
public void registerOutChannelSomewhere(Chunks.Out<String> out) {
  out.write("kiki");
  out.write("foo");
  out.write("bar");
  out.close();
}
```

We can inspect the HTTP response sent by the server:

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

4
kiki
3
foo
3
bar
0
```

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

Next: Comet sockets

Comet sockets

Using chunked responses to create Comet sockets

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

Let's write a first proof-of-concept: create an enumerator generating `<script>` tags calling the browser `console.log` function:

```
public static Result index() {
  // Prepare a chunked text stream
  Chunks<String> chunks = new StringChunks() {

    // Called when the stream is ready
    public void onReady(Chunks.Out<String> out) {
      out.write("<script>console.log('kiki')</script>");
      out.write("<script>console.log('foo')</script>");
      out.write("<script>console.log('bar')</script>");
      out.close();
    }

  }

  response().setContentType("text/html");

  ok(chunks);
}
```

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

Using the `play.libs.Comet` helper

We provide a Comet helper to handle these comet chunked streams that does almost the same as what we just wrote.

> **Note:** Actually it does more, such as pushing an initial blank buffer data for browser compatibility, and supporting both String and JSON messages.

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

```
public static Result index() {
  Comet comet = new Comet("console.log") {
    public void onConnected() {
      sendMessage("kiki");
      sendMessage("foo");
      sendMessage("bar");
      close();
    }
  };

  ok(comet);
}
```

The forever iframe technique

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

```
public static Result index() {
  Comet comet = new Comet("parent.cometMessage") {
    public void onConnected() {
      sendMessage("kiki");
      sendMessage("foo");
      sendMessage("bar");
      close();
    }
  };

  ok(comet);
}
```

With an HTML page like:

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

<iframe src="/comet"></iframe>
```

Next: WebSockets

WebSockets

Using WebSockets instead of Comet sockets

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

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

> WebSocket is a web technology providing for bi-directional, full-duplex communications channels, over a single Transmission Control Protocol (TCP) socket. The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455.
>
> WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. Because ordinary TCP connections to port numbers other than 80 are frequently blocked by administrators outside of home environments, it can be used as a way to circumvent these restrictions and provide similar functionality with some additional protocol overhead while multiplexing several WebSocket services over a single TCP port. Additionally, it serves a purpose for web applications that require real-time bi-directional communication. Before the implementation of WebSocket, such bi-directional communication was only possible using comet channels; however, a comet is not trivial to implement reliably, and due to the TCP Handshake and HTTP header overhead, it may be inefficient for small messages. The WebSocket protocol aims to solve these problems without compromising security assumptions of the web.
>
> http://en.wikipedia.org/wiki/WebSocket

Handling WebSockets

Until now we were using a simple action method to handle standard HTTP requests and send back standard HTTP results. WebSockets are a totally different beast, and can't be handled via standard actions.

To handle a WebSocket your method must return a `WebSocket` instead of a `Result`:

```
public static WebSocket<String> index() {
  return new WebSocket<String>() {

    // Called when the Websocket Handshake is done.
    public void onReady(WebSocket.In<String> in, WebSocket.Out<String> out) {

      // For each event received on the socket,
      in.onMessage(new Callback<String>() {
         public void invoke(String event) {

           // Log events to the console
           println(event);

         }
      });

      // When the socket is closed.
      in.onClose(new Callback0() {
         public void invoke() {

           println("Disconnected")
```

```
        }
      });

      // Send a single 'Hello!' message
      out.write("Hello!");

    }

  }
}
```

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

When the `WebSocket` is ready, you get both `in` and `out` channels.

It this example, we print each message to console and we send a single **Hello!** message.

> **Tip:** You can test your WebSocket controller on http://websocket.org/echo.html. Just set the location to `ws://localhost:9000`.

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

```
public static WebSocket<String> index() {
  return new WebSocket<String>() {

    public void onReady(WebSocket.In<String> in, WebSocket.Out<String> out) {
      out.write("Hello!");
      out.close()
    }

  }
}
```

Next: The template engine

3. The template engine

The template engine

A type safe template engine based on Scala

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

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

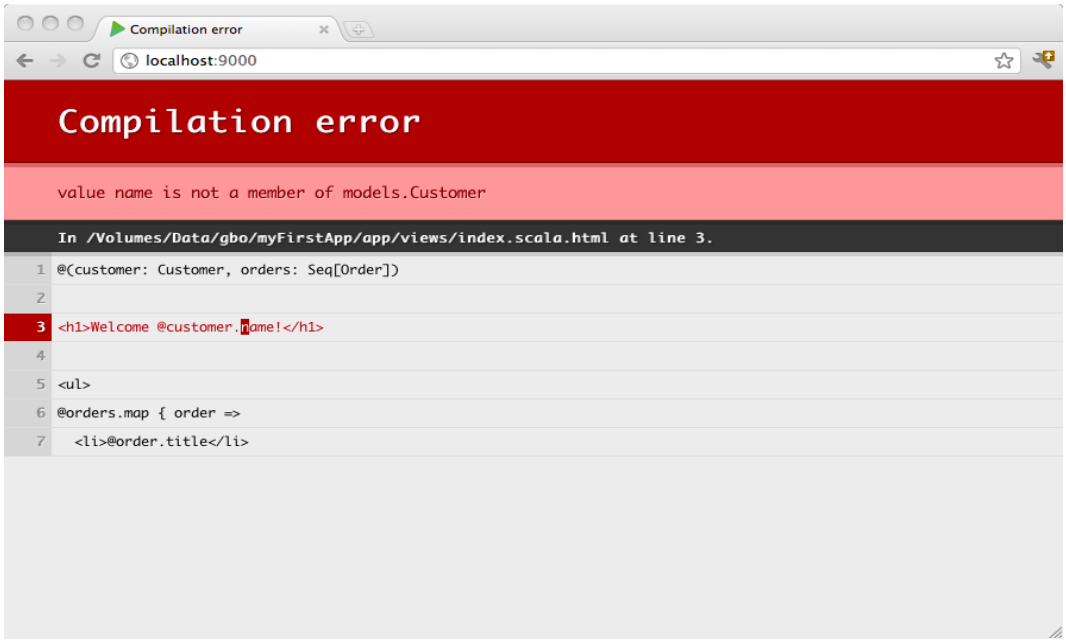
  Note: Even though the template engine uses Scala as expression language, this is not a problem for Java developers. You can almost use it as if the language were Java.

  Remember that a template is not a place to write complex logic. You don't have to write complicated Scala code here. Most of the time you will just access data from your model objects, as follows:

  ```
  myUser.getProfile().getUsername()
  ```

  Parameter types are specified using a suffix syntax. Generic types are specified using the `[]` symbols instead of the usual <> Java syntax. For example, you write `List[String]`, which is the same as `List<String>` in Java.

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



Overview

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

The template system has been designed to feel comfortable to those used to working with HTML, allowing front-end developers to easily work with the templates.

Templates are compiled as standard Scala functions, following a simple naming convention. If you create a `views/Application/index.scala.html` template file, it will generate a `views.html.Application.index` class that has a `render()` method.

For example, here is a simple template:

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

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

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

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

```
Content html = views.html.Application.index.render(customer, orders);
```

Syntax: the magic '@' character

The Scala template uses `@` as the single special character. Every time this character is encountered, it indicates the beginning of a dynamic statement. You are not required to explicitly close the code block - the end of the dynamic statement will be inferred from your code:

```
Hello @customer.getName()!
       ^^^^^^^^^^^^^^^^^^
          Dynamic code
```

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

```
Hello @(customer.getFirstName() + customer.getLastName())!
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                          Dynamic Code
```

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

```
Hello @{val name = customer.getFirstName() + customer.getLastName(); name}!
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                  Dynamic Code
```

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

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

Template parameters

A template is like a function, so it needs parameters, which must be declared at the top of the template file:

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

You can also use default values for parameters:

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

Or even several parameter groups:

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

Iterating

You can use the `for` keyword, in a pretty standard way:

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

If-blocks

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

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

Declaring reusable blocks

You can create reusable code blocks:

```
@display(product: models.Product) = {
  @product.getName() ($@product.getPrice())
}

<ul>
@for(product <- products) {
  @display(product)
}
</ul>
```

Note that you can also declare reusable pure code blocks:

```
@title(text: String) = @{
  text.split(' ').map(_.capitalize).mkString(" ")
}
```

```
<h1>@title("hello world")</h1>
```

> **Note:** Declaring code block this way in a template can be sometime useful but keep in mind that a template is not the best place to write complex logic. It is often better to externalize these kind of code in a Java class (that you can store under the `views/` package as well if your want).

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

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

Declaring reusable values

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

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

Import statements

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

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

@import utils._

...
```

Comments

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

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

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

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

<h1>@msg</h1>
```

Escaping

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

For example to output raw HTML:

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

Next: Common use cases

Common template use cases

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

Layout

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

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

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

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

  <h1>Home page</h1>

}
```

> **Note:** You can use both named parameters (like `@main(title = "Home")` and positional parameters, like `@main("Home")`. Choose whichever is clearer in a specific context.

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

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

Using this from our 'index' template, we have:

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

} {
  <h1>Home page</h1>

}
```

Alternatively, we can declare the sidebar block separately:

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

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

}
```

Tags (they are just functions right?)

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

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

@level match {

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

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

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

}
```

And now let's use it from another template:

```
@import tags._

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

Includes

Again, there's nothing special here. You can just call any other template you like (or in fact any other function, wherever it is defined):

```
<h1>Home</h1>

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

Next: HTTP form submission and validation

4. HTTP form submission and validation

Handling form submission

Defining a form

The `play.data` package contains several helpers to handle HTTP form data submission and validation. The easiest way to handle a form submission is to define a `play.data.Form` that wraps an existing class:

```
public class User {
    public String email;
    public String password;
}
```

```
Form<User> userForm = form(User.class);
```

> **Note:** The underlying binding is done using Spring data binder.

This form can generate a `User` result value from `HashMap<String,String>` data:

```
Map<String,String> anyData = new HashMap();
anyData.put("email", "bob@gmail.com");
anyData.put("password", "secret");

User user = userForm.bind(anyData).get();
```

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

```
User user = userForm.bindFromRequest().get();
```

Defining constraints

You can define additional constraints that will be checked during the binding phase using JSR-303 (Bean Validation) annotations:

```
public class User {

    @Required
    public String email;
    public String password;
}
```

> **Tip:** The `play.data.validation.Constraints` class contains several built-in validation annotations.

You can also define an ad-hoc validation by adding a `validate` method to your top object:

```
public class User {

    @Required
    public String email;
    public String password;

    public String validate() {
        if(authenticate(email,password) == null) {
            return "Invalid email or password";
        }
        return null;
    }
}
```

Handling binding failure

Of course if you can define constraints, then you need to be able to handle the binding errors.

```
if(userForm.hasErrors()) {
    return badRequest(form.render(userForm));
} else {
    User user = userForm.get();
    return ok("Got user " + user);
}
```

Filling a form with initial default values

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

```
userForm.fill(new User("bob@gmail.com", "secret"))
```

Register a custom DataBinder

In case you want to define a mapping from a custom object to a form field string and vice versa you need to register a new Formatter for this object.
For an object like JodaTime's `LocalTime` it could look like this:

```
Formatters.register(LocalTime.class, new Formatters.SimpleFormatter<LocalTime>() {

        private Pattern timePattern = Pattern.compile("([012]?\\\\d)(?:[\\\\s:\\\\._\\\\-]+([0-5]\\\\d))?");

        @Override
        public LocalTime parse(String input, Locale l) throws ParseException {
                Matcher m = timePattern.matcher(input);
                if (!m.find()) throw new ParseException("No valid Input",0);
                int hour = Integer.valueOf(m.group(1));
                int min = m.group(2) == null ? 0 : Integer.valueOf(m.group(2));
                return new LocalTime(hour, min);
        }

        @Override
        public String print(LocalTime localTime, Locale l) {
                return localTime.toString("HH:mm");
        }
});
```

Next: Using the form template helpers

Form template helpers

Play provides several helpers to help you render form fields in HTML templates.

Creating a `<form>` tag

The first helper creates the `<form>` tag. It is a pretty simple helper that automatically sets the `action` and `method` tag parameters according to the reverse route you pass in:

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

}
```

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

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

}
```

Rendering an `<input>` element

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

```
@(myForm: Form[User])

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

    @helper.inputText(myForm("username"))

    @helper.inputPassword(myForm("password"))

}
```

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

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

> **Note:** All extra parameters will be added to the generated HTML, except for ones whose name starts with the _ character. Arguments starting with an underscore are reserved for field constructor argument (which we will see later).

Handling HTML input creation yourself

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

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

Field constructors

A rendered field does not only consist of an `<input>` tag, but may also need a `<label>` and a bunch of other tags used by your CSS framework to decorate the field.

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

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

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

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

Twitter bootstrap field constructor

There is another built-in field constructor that can be used with Twitter Bootstrap .

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

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

This field constructor generates HTML like the following:

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

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

Writing you own field constructor

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

```
@(elements: helper.FieldElements)

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

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

Now create a `FieldConstructor` somewhere, using:

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

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

Handling repeated values

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

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

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

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

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

    @inputText(emailField)

}
```

Use the `min` parameter to display a minimum number of fields, even if the corresponding form data are empty.

> **Next:** Working with JSON

5. Working with JSON

Handling and serving JSON requests

Handling a JSON request

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

By default an action uses an **any content** body parser, which you can use to retrieve the body as JSON (actually as a Jerkson `JsonNode`):

```
public static index sayHello() {
  JsonNode json = request().body().asJson();
  if(json == null) {
    return badRequest("Expecting Json data");
  } else {
    String name = json.findPath("name").getTextValue();
    if(name == null) {
      return badRequest("Missing parameter [name]");
    } else {
      return ok("Hello " + name);
    }
  }
}
```

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

```
@BodyParser.Of(Json.class)
public static index sayHello() {
  String name = json.findPath("name").getTextValue();
  if(name == null) {
    return badRequest("Missing parameter [name]");
  } else {
    return ok("Hello " + name);
  }
}
```

> **Note:** This way, a 400 HTTP response will be automatically returned for non JSON requests.

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

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

It replies with:

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

Hello Guillaume
```

Serving a JSON response

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

```
@BodyParser.Of(Json.class)
public static index sayHello() {
  ObjectNode result = Json.newObject();
  String name = json.findPath("name").getTextValue();
  if(name == null) {
    result.put("status", "KO");
    result.put("message", "Missing parameter [name]");
    return badRequest(result);
  } else {
    result.put("status", "OK");
    result.put("message", "Hello " + name);
    return ok(result);
  }
}
```

Now it replies with:

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

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

Next: Working with XML

6. 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 request body. It must specify the `text/xml` MIME type in its `Content-Type` header.

By default, an action uses an **any content** body parser, which you can use to retrieve the body as XML (actually as a `org.w3c.Document`):

```
public static index sayHello() {
  Document dom = request().body().asXml();
  if(dom == null) {
    return badRequest("Expecting Xml data");
  } else {
    String name = XPath.selectText("//name", dom);
    if(name == null) {
      return badRequest("Missing parameter [name]");
    } else {
      return ok("Hello " + name);
    }
  }
}
```

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

```
@BodyParser.Of(Xml.class)
public static index sayHello() {
  String name = XPath.selectText("//name", dom);
  if(name == null) {
    return badRequest("Missing parameter [name]");
  } else {
    return ok("Hello " + name);
  }
}
```

> **Note:** This way, a 400 HTTP response will be automatically returned for non-XML requests.

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

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

It replies with:

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

Hello Guillaume
```

Serving an XML response

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

```
@BodyParser.Of(Xml.class)
public static index sayHello() {
  String name = XPath.selectText("//name", dom);
  if(name == null) {
    return badRequest("<message \"status\"=\"KO\">Missing parameter [name]</message>");
  } else {
    return ok("<message \"status\"=\"OK\">Hello " + name + "</message>");
  }
}
```

Now it replies with:

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

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

Next: Handling file upload

7. 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 allows to mix standard form data with file attachments.

Start by writing an HTML form:

```
@form(action = routes.Application.upload, 'enctype -> "multipart/form-data") {

    <input type="file" name="picture">

    <p>
        <input type="submit">
    </p>

}
```

Now let's define the `upload` action:

```
public static Result upload() {
  MultipartFormData body = request().body().asMultipartFormData();
  FilePart picture = body.getFile("picture");
  if (picture != null) {
    String fileName = picture.getFilename();
    String contentType = picture.getContentType();
    File file = picture.getFile();
    return ok("File uploaded");
  } else {
    flash("error", "Missing file");
    return redirect(routes.Application.index());
  }
}
```

Direct file upload

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

```
public static Result upload() {
  File file = request().body().asRaw().asFile();
  return ok("File uploaded");
}
```

Next: Accessing an SQL database

8. Accessing an SQL database

Accessing an SQL database

Configuring JDBC connection pools

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

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

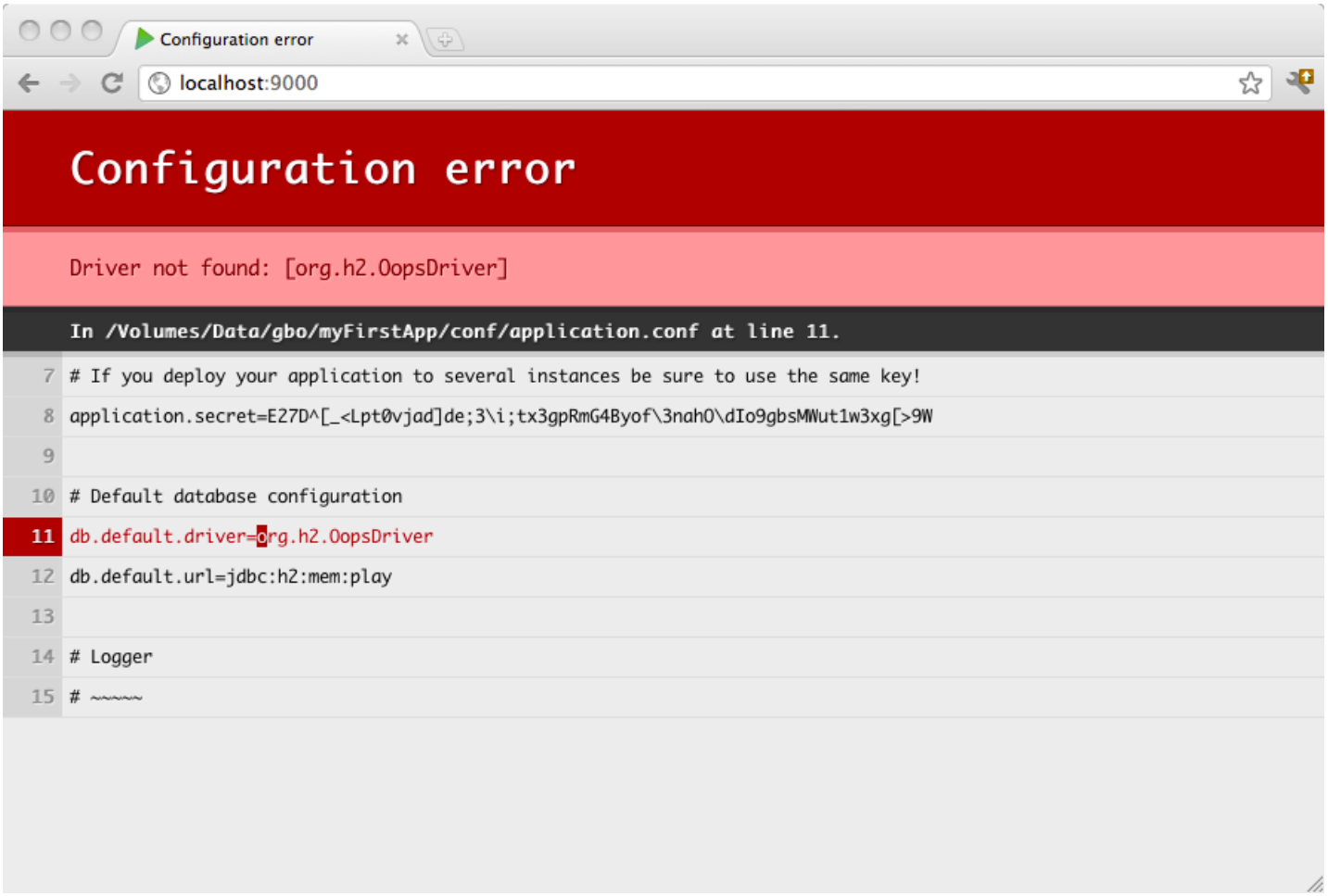
```
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

To configure several data sources:

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

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

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



Accessing the JDBC datasource

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

```
import play.db.*;
```

```
DataSource ds = DB.getDatasource();
```

Obtaining a JDBC connection

You can retrieve a JDBC connection the same way:

```
Connection connection = DB.getConnection();
```

Exposing the datasource through JNDI

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

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

Importing a Database Driver

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

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

```
val appDependencies = Seq(
     // Add your project dependencies here,
     ...
     "mysql" % "mysql-connector-java" % "5.1.18"
     ...
)
```

Next: Using Ebean to access your database

Using the Ebean ORM

Configuring Ebean

Play 2.0 comes with the Ebean ORM. To enable it, add the following line to `conf/application.conf`:

```
ebean.default="models.*"
```

This defines a `default` Ebean server, using the `default` data source, which must be properly configured. You can actually create as many Ebean servers you need, and explicitly define the mapped class for each server.

```
ebean.orders="models.Order,models.OrderItem"
ebean.customers="models.Customer,models.Address"
```

In this example, we have access to two Ebean servers - each using its own database.

> For more information about Ebean, see the Ebean documentation .

Using the play.db.ebean.Model superclass

Play 2.0 defines a convenient superclass for your Ebean model classes. Here is a typical Ebean class, mapped in Play 2.0:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.ebean.*;
import play.data.format.*;
import play.data.validation.*;

@Entity
public class Task extends Model {

  @Id
  @Constraints.Min(10)
  public Long id;

  @Constraints.Required
  public String name;

  public boolean done;

  @Formats.DateTime(pattern="dd/MM/yyyy")
  public Date dueDate = new Date();

  public static Finder<Long,Task> find = new Finder<Long,Task>(
    Long.class, Task.class
  );

}
```

As you can see, we've added a `find` static field, defining a `Finder` for an entity of type `Task` with a `Long` identifier. This helper field is then used to simplify querying our model:

```
// Find all tasks
```

```
List<Task> tasks = Task.find.all();

// Find a task by ID
Task anyTask = Task.find.byId(34L);

// Delete a task by ID
Task.find.ref(34L).delete();

// More complex task query
List<Task> tasks = find.where()
    .ilike("name", "%coco%")
    .orderBy("dueDate asc")
    .findPagingList(25)
    .getPage(1);
```

Transactional actions

By default Ebean will not use transactions. However, you can use any transaction helper provided by Ebean to create a transaction. For example:

```
// run in Transactional scope...
Ebean.execute(new TxRunnable() {
  public void run() {

    // code running in "REQUIRED" transactional scope
    // ... as "REQUIRED" is the default TxType
    System.out.println(Ebean.currentTransaction());

    // find stuff...
    User user = Ebean.find(User.class, 1);
    ...

    // save and delete stuff...
    Ebean.save(user);
    Ebean.delete(order);
    ...
  }
});
```

You can also annotate your action method with `@play.db.ebean.Transactional` to compose your action method with an `Action` that will automatically manage a transaction:

```
@Transactional
public static Result save() {
  ...
}
```

Next: Integrating with JPA

Integrating with JPA

Exposing the datasource through JNDI

JPA requires the datasource to be accessible via JNDI. You can expose any Play-managed datasource via JDNI by adding this configuration in `conf/application.conf`:

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

Adding a JPA implementation to your project

There is no built-in JPA implementation in Play 2.0; you can choose any available implementation. For example, to use Hibernate, just add the dependency to your project:

```
val appDependencies = Seq(
  "org.hibernate" % "hibernate-entitymanager" % "3.6.9.Final"
)
```

Creating a persistence unit

Next you have to create a proper `persistence.xml` JPA configuration file. Put it into the `conf/META-INF` directory, so it will be properly added to your classpath.

Here is a sample configuration file to use with Hibernate:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="defaultPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>DefaultDS</non-jta-data-source>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
        </properties>
    </persistence-unit>

</persistence>
```

Annotating JPA actions with `@Transactional`

Every JPA call must be done in a transaction so, to enable JPA for a particular action, annotate it with `@play.db.jpa.Transactional`. This will compose your action method with a JPA `Action` that manages the transaction for you:

```
@Transactional
public static Result index() {
  ...
}
```

If your action perfoms only queries, you can set the `readOnly` attribute to `true`:

```
@Transactional(readOnly=true)
public static Result index() {
  ...
}
```

Using the `play.db.jpa.JPA` helper

At any time you can retrieve the current entity manager from the `play.db.jpa.JPA` helper class:

```
public static Company findById(Long id) {
  return JPA.em().find(Company.class, id);
}
```

Next: Using the cache

9. 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. You can also provide your own implementation via a plugin.

Accessing the Cache API

The cache API is provided by the `play.cache.Cache` object. This requires a cache plugin to be registered.

> **Note:** The API is intentionally minimal to allow various implementations to be plugged in. If you need a more specific API, use the one provided by your Cache plugin.

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

```
Cache.set("item.key", frontPageNews);
```

You can retrieve the data later:

```
News news = Cache.get("item.key");
```

Caching HTTP responses

You can easily create a smart cached action using standard `Action` composition.

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

Play provides a default built-in helper for the standard case:

```
@Cached("homePage")
public static Result index() {
  return ok("Hello world");
}
```

Caching in templates

You may also access the cache from a view template.

```
@cache.Cache.getOrElse("cached-content", 3600) {
     <div>I'm cached for an hour</div>
}
```

Session cache

Play provides a global cache, whose data are visible to anybody. How would one restrict visibility to a given user? For instance you may want to cache metrics that only apply to a given user.

```
// Generate a unique ID
String uuid=session("uuid");
if(uuid==null) {
        uuid=java.util.UUID.randomUUID().toString();
        session("uuid", uuid);
}

// Access the cache
News userNews = Cache.get(uuid+"item.key");
if(userNews==null) {
        userNews = generateNews(uuid);
        Cache.set(uuid+"item.key", userNews );
}
```

Next: Calling web services

10. Calling WebServices

The Play WS API

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

A call made by `play.libs.ws` should return a `Promise<Ws.Response>`, which you can handle later with Play's asynchronous mechanisms.

Making HTTP calls

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

```
Promise<WS.Response> homePage = WS.url("http://mysite.com").get();
```

Alternatively:

```
Promise<WS.Response> result = WS.url("http://localhost:9001").post("content");
```

Retrieving the HTTP response result

The call is made asynchronously and you need to manipulate it as a `Promise<WS.Response>` to get the actual content. You can compose several promises and end up with a `Promise<Result>` that can be handled directly by the Play server:

```
import play.libs.F.Function;
import play.libs.WS;
import play.mvc.*;

public class Controller extends Controller {

 public static Result feedTitle(String feedUrl) {
    return async(
        WS.url(feedUrl).get().map(
            new Function<WS.Response, Result>() {
                public Result apply(WS.Response response) {
                    return ok("Feed title:" + response.asJson().findPath("title"));
                }
            }
        )
    );
 }
}
```

Next: Integrating with Akka

OpenID Support in Play

OpenID is a protocol for users to access several services with a single account. As a web developer, you can use OpenID to offer users a way to login with an account they already have (their Google account for example). In the enterprise, you can use OpenID to connect to a company's SSO server if it supports it.

The OpenID flow in a nutshell

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

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

OpenID in Play Framework

The OpenID API has two important functions:

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

In any case, you should catch exceptions and if one is thrown redirect back the user to the login page with relevant information.

Extended Attributes

The OpenID of a user gives you his identity. The protocol also support getting extended attributes such as the email address, the first name, the last name…

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

Extended attributes are requested in the redirect URL:

```
Map<String, String> attributes = new HashMap<String, String>();
attributes.put("email", "http://schema.openid.net/contact/email");
OpenID.redirectURL(
  openid,
  routes.Application.openIDCallback.absoluteURL(),
  attributes
);
```

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

11. Intergrating 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 2.0 can work with several containers called `ActorSystems`. An actor system manages the resources it is configured to use in order to run the actors it contains.

A Play application defines a special actor system to be used by the application. This actor system follows the application life-cycle and restarts automatically when the application restarts.

> **Note:** Nothing prevents you from using another actor system from within a Play application. The provided default actor system is just a convenient way to start a few actors without having to set-up your own.

You can access the default application actor system using the `play.libs.Akka` helper:

```
ActorRef myActor = Akka.system().actorOf(new Props(MyActor.class));
```

Configuration

The default actor system configuration is read from the Play application configuration file. For example to configure the default dispatcher of the application actor system, add these lines to the `conf/application.conf` file:

```
akka.default-dispatcher.core-pool-size-max = 64
akka.debug.receive = on
```

> **Note:** You can also configure any other actor system from the same file, just provide a top configuration key.

Converting Akka `Future` to Play `Promise`

When you interact asynchronously with an Akka actor we will get `Future` object. You can easily convert them to play `Promise` using the conversion method provided in `play.libs.Akka.asPromise()`:

```
import static akka.pattern.Patterns.ask;
import play.libs.Akka;
import play.mvc.Result;
import static play.mvc.Results.async;
import play.libs.F.Function;

public static Result index() {
  return async(
    Akka.asPromise(ask(myActor,"hello", 1000)).map(
      new Function<Object,Result>() {
        public Result apply(Object response) {
```

```
          return ok(response.toString());
        }
      }
    )
  );
}
```

Executing a block of code asynchronously

A common use case within Akka is to have some computation performed concurrently without needing the extra utility of an Actor. If you find yourself creating a pool of Actors for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import static play.libs.Akka.future;
import play.libs.F.*;
import java.util.concurrent.Callable;

public static Result index() {
  return async(
    future(new Callable<Integer>() {
      public Integer call() {
        return longComputation();
      }
    }).map(new Function<Integer,Result>() {
      public Result apply(Integer i) {
        return ok("Got " + i);
      }
    })
  );
}
```

Scheduling asynchronous tasks

You can schedule sending messages to actors and executing tasks (functions or `Runnable` instances). You will get a `Cancellable` back that you can call `cancel` on to cancel the execution of the scheduled operation.

For example, to send a message to the `testActor` every 30 minutes:

```
Akka.system().scheduler().schedule(
  Duration.create(0, TimeUnit.MILLISECONDS),
  Duration.create(30, TimeUnit.MINUTES)
  testActor,
  "tick"
)
```

Alternatively, to run a block of code ten seconds from now:

```
Akka.system().scheduler().scheduleOnce(
  Duration.create(10, TimeUnit.SECONDS),
  new Runnable() {
    public void run() {
      file.delete()
    }
  }
);
```

Next: Internationalization

12. Internationalization

Externalising messages and internationalization

Specifying languages supported by your application

The specify your application's languages, you need a valid language code, specified by a valid **ISO Language Code**, optionally followed by a valid **ISO Country Code**. For example, `fr` or `en-US`.

To start, you need to specify the languages that your application supports in its `conf/application.conf` file:

```
application.langs=en,en-US,fr
```

Externalizing messages

You can externalize messages in the `conf/messages.xxx` files.

The default `conf/messages` file matches all languages. You can specify additional language messages files, such as `conf/messages.fr` or `conf/messages.en-US`.

You can retrieve messages for the current language using the `play.api.i18n.Messages` object:

```
String title = Messages.get("home.title")
```

You can also specify the language explicitly:

```
String title = Messages.get(new Lang("fr"), "home.title")
```

> **Note:** If you have a `Request` in the scope, it will provide a default `Lang` value corresponding to the preferred language extracted from the `Accept-Language` header and matching one the application's supported languages.

Formatting messages

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

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

You can then specify parameters as:

```
Messages.get("files.summary", d.files.length, d.name)
```

Retrieving supported languages from an HTTP request

You can retrieve a specific HTTP request's supported languages:

```
public static Result index() {
  return ok(request().acceptLanguages());
}
```

Next: The application Global object

13. The application Global object

Application global settings

The Global object

Defining a `Global` object in your project allows you to handle global settings for your application. This object must be defined in the root package.

```
import play.*;

public class Global extends GlobalSettings {

}
```

Intercepting application start-up and shutdown

You can override the `onStart` and `onStop` operation to be notified of the corresponding application lifecycle events:

```
import play.*;

public class Global extends GlobalSettings {

  @Override
  public void onStart(Application app) {
    Logger.info("Application has started");
  }

  @Override
  public void onStop(Application app) {
    Logger.info("Application shutdown...");
  }

}
```

Providing an application error page

When an exception occurs in your application, the `onError` operation will be called. The default is to use the internal framework error page. You can override this:

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

  @Override
  public Result onError(Throwable t) {
    return internalServerError(
      views.html.errorPage(t)
    );
  }

}
```

Handling action not found

If the framework doesn't find an action method for a request, the `onActionNotFound` operation will be called:

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

  @Override
  public Result onActionNotFound(String uri) {
    return notFound(
      views.html.pageNotFound(uri)
    );
  }

}
```

The `onBadRequest` operation will be called if a route was found, but it was not possible to bind the request parameters:

```
import play.*;
import play.mvc.*;

import static play.mvc.Results.*;

public class Global extends GlobalSettings {

  @Override
  public Result onBadRequest(String uri, String error) {
    return badRequest("Don't try to hack the URI!");
  }

}
```

Next: Intercepting requests

Intercepting requests

Overriding onRequest

One important aspect of the `GlobalSettings` class is that it provides a way to intercept requests and execute business logic before a request is dispatched to an action.

For example:

```
import play.*;

public class Global extends GlobalSettings {

@Override
public Action onRequest(Request request, Method actionMethod) {
   System.out.println("before each request..." + request.toString());
   return super.onRequest(request, actionMethod);
}

}
```

It's also possible to intercept a specific action method. This can be achieved via Action composition.

Next: Testing your application

14. Testing your application

Testing your application

Test source files must be placed in your application's `test` folder. You can run tests from the Play console using the `test` and `test-only` tasks.

Using JUnit

The default way to test a Play 2 application is with JUnit.

```
package test;

import org.junit.*;

import play.mvc.*;
import play.test.*;
import play.libs.F.*;

import static play.test.Helpers.*;
import static org.fest.assertions.Assertions.*;

public class SimpleTest {

    @Test
    public void simpleCheck() {
        int a = 1 + 1;
        assertThat(a).isEqualTo(2);
    }
```

Running in a fake application

If the code you want to test depends on a running application, you can easily create a `FakeApplication` on the fly:

```
@Test
public void findById() {
    running(fakeApplication(), new Runnable() {
       public void run() {
           Computer macintosh = Computer.find.byId(21l);
           assertThat(macintosh.name).isEqualTo("Macintosh");
           assertThat(formatted(macintosh.introduced)).isEqualTo("1984-01-24");
       }
    });
}
```

You can also pass (or override) additional application configuration, or mock any plugin. For example to create a `FakeApplication` using a `default` in-memory database:

```
fakeApplication(inMemoryDatabase())
```

Next: Writing functional tests

Writing functional tests

Testing a template

As a template is a standard Scala function, you can execute it from a test and check the result:

```
@Test
public void renderTemplate() {
    Content html = views.html.index.render("Coco");
    assertThat(contentType(html)).isEqualTo("text/html");
    assertThat(contentAsString(html)).contains("Coco");
}
```

Testing your controllers

You can also retrieve an action reference from the reverse router, such as `controllers.routes.ref.Application.index`. You can then invoke it:

```
@Test
public void callIndex() {
    Result result = callAction(controllers.routes.ref.Application.index("Kiki"));
    assertThat(status(result)).isEqualTo(OK);
    assertThat(contentType(result)).isEqualTo("text/html");
    assertThat(charset(result)).isEqualTo("utf-8");
    assertThat(contentAsString(result)).contains("Hello Kiki");
}
```

Testing the router

Instead of calling the `Action` yourself, you can let the `Router` do it:

```
@Test
public void badRoute() {
    Result result = routeAndCall(fakeRequest(GET, "/xx/Kiki"));
    assertThat(result).isNull();
}
```

Starting a real HTTP server

Sometimes you want to test the real HTTP stack from with your test. You can do this by starting a test server:

```
@Test
public void testInServer() {
  running(testServer(3333), new Callback0() {
      public void invoke() {
         assertThat(
           WS.url("http://localhost:3333").get().get().status
         ).isEqualTo(OK);
      }
  });
}
```

Testing from within a web browser

If you want to test your application from with a Web browser, you can use Selenium WebDriver. Play will start the WebDriver for your, and wrap it in the convenient API provided by FluentLenium.

```
@Test
public void runInBrowser() {
    running(testServer(3333), HTMLUNIT, new Callback<TestBrowser>() {
        public void invoke(TestBrowser browser) {
           browser.goTo("http://localhost:3333");
           assertThat(browser.$("#title").getTexts().get(0)).isEqualTo("Hello Guest");
           browser.$("a").click();
           assertThat(browser.url()).isEqualTo("http://localhost:3333/Coco");
           assertThat(browser.$("#title", 0).getText()).isEqualTo("Hello Coco");
        }
    });
}
```