Playing with the play Framework

Contents

- How to Create a basic web CRUD application in Play in both: Java and Scala
- Lightly touch on some of the other features of play if time remaining.

Github Contents

```
git@github.com:dhinojosa/play-study.git
```

Downloading Play

- **`http://www.playframework.com`**

What is Play?

- Web Framework
- Part of the Typesafe Stack.
- Painfully Easy

What is Play?

- Built on Akka
  - Massive Library
  - Concurrent and Fault Tolerant Applications
  - Actors, STM, Self-Healing

What is Play?

- Built on Config
  - Configuration Library
  - `https://github.com/typesafehub/config`
  - Built upon a JSON superset, called HOCON

What is Play?

- Built on Config
  - Configuration Library
  - `https://github.com/typesafehub/config`
  - Built upon a JSON superset, called HOCON

HOCON ANGRY!

HOCON

"Human-Optimized Config Object Notation"

HOCON

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

HOCON

```
db {
  default.driver=org.h2.Driver
  default.url="jdbc:h2:mem:play"
  default.user=sa
  default.password=""
}
```

HOCON

```
db {
  default{
    driver=org.h2.Driver
    url="jdbc:h2:mem:play"
    user=sa
    password=""
  }
}
```

What is Play?

- Built on Logback for Logging
- Built on Netty
  - NIO Based
  - `https://netty.io`
  - Asynchronous Event-Driven Network Application Framework

Setting up the environment

- Download to favorite location (e.g. `~/java/play-2.1.0`)
- Map PLAY_HOME: `export PLAY_HOME= ~/java/play-2.1.0`
- Append to PATH (without the bin): `export PATH=$PATH:$SCALA_HOME/bin: $JAVA_HOME/bin:...:$PLAY_HOME: $SBT_HOME/bin`

Getting Started

- **`% play new <application name>`**
- e.g. **`% play new rocknroll`**
- Select Java Application or Scala Application.

Demo

Creating Application

Folder Structure

```
app                      → Application sources
 └ assets                → Compiled asset sources
    └ stylesheets        → Typically LESS CSS sources
    └ javascripts        → Typically CoffeeScript sources
 └ controllers           → Application controllers
 └ models                → Application business layer
 └ views                 → Templates
conf                     → Configurations files and other non-compiled resources (on classpath)
 └ application.conf      → Main configuration file
 └ routes                → Routes definition
public                   → Public assets
 └ stylesheets           → CSS files
 └ javascripts           → Javascript files
 └ images                → Image files
project                  → sbt configuration files
 └ build.properties      → Marker for sbt project
 └ Build.scala           → Application build script
 └ plugins.sbt           → sbt plugins
lib                      → Unmanaged libraries dependencies
logs                     → Standard logs folder
 └ application.log       → Default log file
target                   → Generated stuff
 └ scala-2.10.0
    └ cache
    └ classes            → Compiled class files
    └ classes_managed    → Managed class files (templates, ...)
    └ resource_managed   → Managed resources (less, ...)
    └ src_managed        → Generated sources (templates, ...)
test                     → source folder for unit or functional tests
```

Command Line Launch

- `% play run // default to port 9000`
- `% play “run 9000” // being explicit`
- `% play “run 10133” // different port`

Play Console

- Hyped up SBT (Simple Build Tool) console.
- Contains standard SBT commands
- Contains many additional commands used for play
- **`% play`**

Running in Play Console

- `% play // go into console`
- `> run // default to port 9000`
- `> run 9000 //being explicit`
- `> run 10133 //different port`

About Development

- Any change in the development will recompile code and pages automatically
- Compiling is somewhat intensive on the Scala side, and for good reason.

Other console commands

- `test //test`
- `~test //test (triggered)`
- `compile //compile`
- `~compile //compile (triggered)`
- `console //scala console`
- `clean //clean up compiled code`
- `clean-all //clean everything (play)`
- more...

Setting up your IDE (Simplified)

- Play/SBT commands:
  - `eclipse / eclipse with-source=true`
  - `idea / idea with-sources=yes`
- **http://www.playframework.com/documentation/2.1.0/IDE**

Setting up your IDE (Simplified)

- Play/SBT commands:
  - **`eclipse / eclipse with-source=true`**
  - **`idea / idea with-sources=yes`**
- **`http://www.playframework.com/documentation/2.1.0/IDE`**

Other console commands

- `test //test`
- `~test //test (triggered)`
- `compile //compile`
- `~compile //compile (triggered)`
- `console //scala console`
- more...

Web Application Four Basic Steps

- Create a business method in an **`class`** (Java) or **`object`** (Scala) that extends **`Controller`**.
- Return a **`Result`** from that method: **`ok()`**, **`redirect()`**, **`notFound()`**, **`forbidden()`**, **`badRequest()`**, **`created()`**, **`internalServerError()`**
- Add a route in **`conf/routes`** file.
- Create a page that will be referenced from any **`Result`** possibly returned.

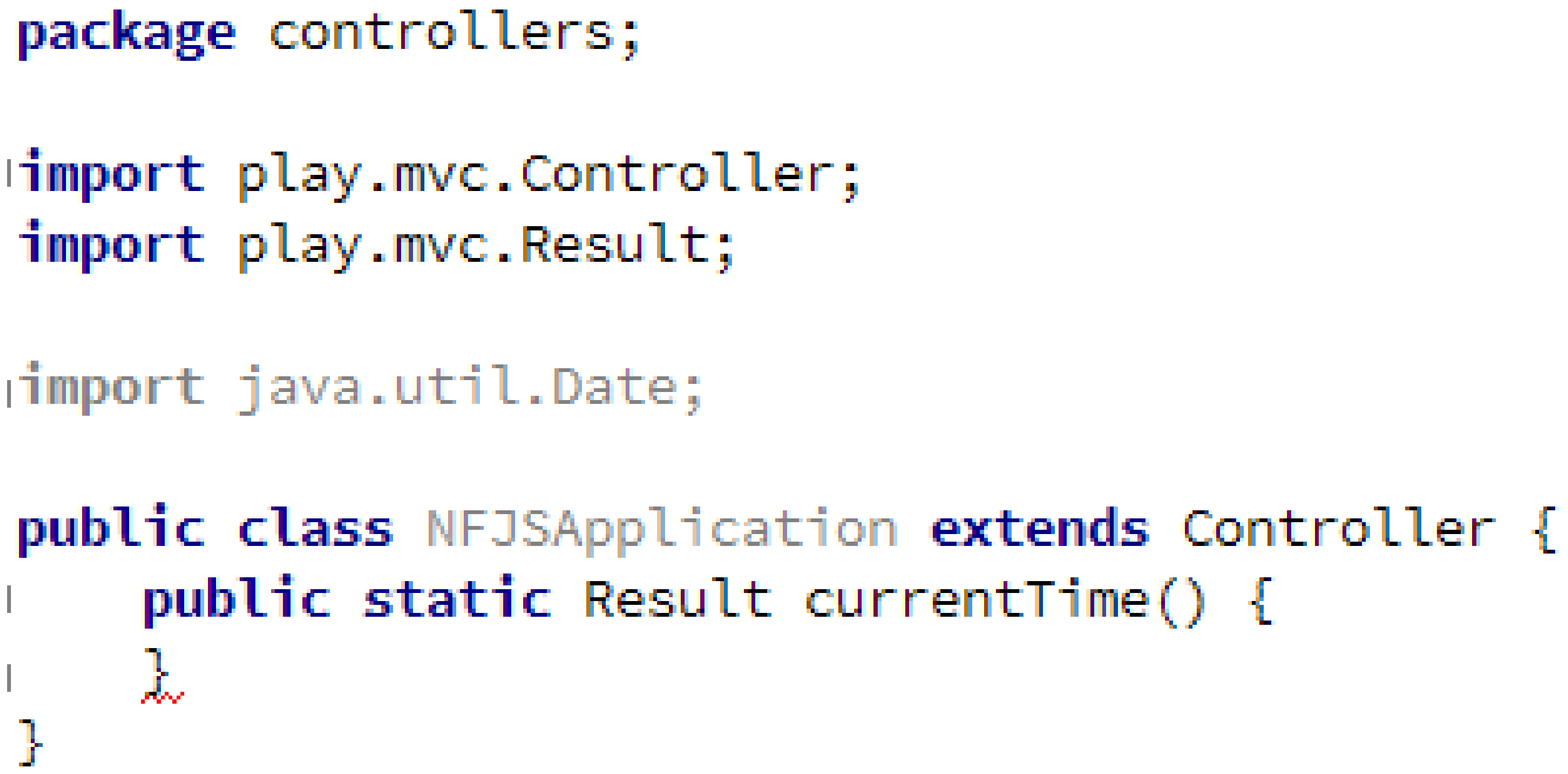
Demo

- Display ad-hoc development in Play

First, in Java...

Step 1: Java

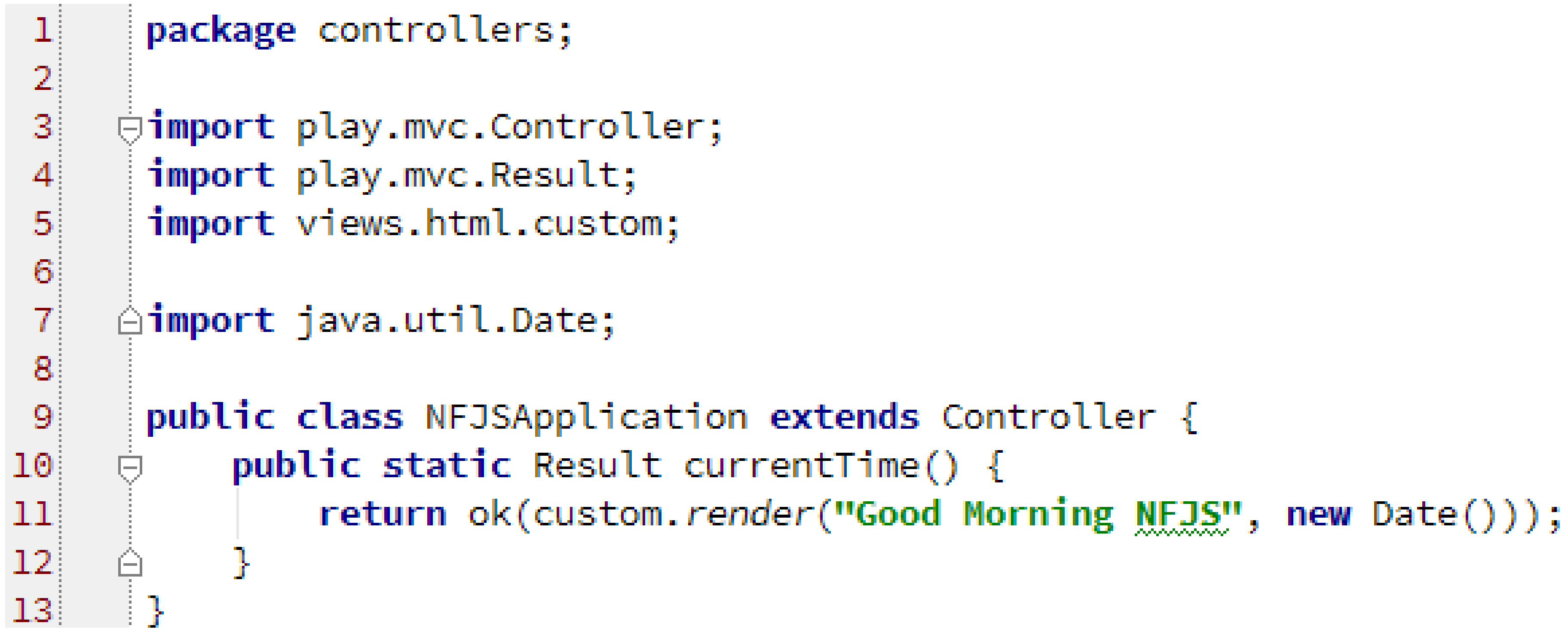
- Create a business method in an **`class`** (Java) or **`object`** (Scala) that extends **`Controller`** and place it in controllers folder.



```
package controllers;

import play.mvc.Controller;
import play.mvc.Result;

import java.util.Date;

public class NFJSApplication extends Controller {
    public static Result currentTime() {
    }
}
```

Step 2: Java

- Return a **`Result`** from that method

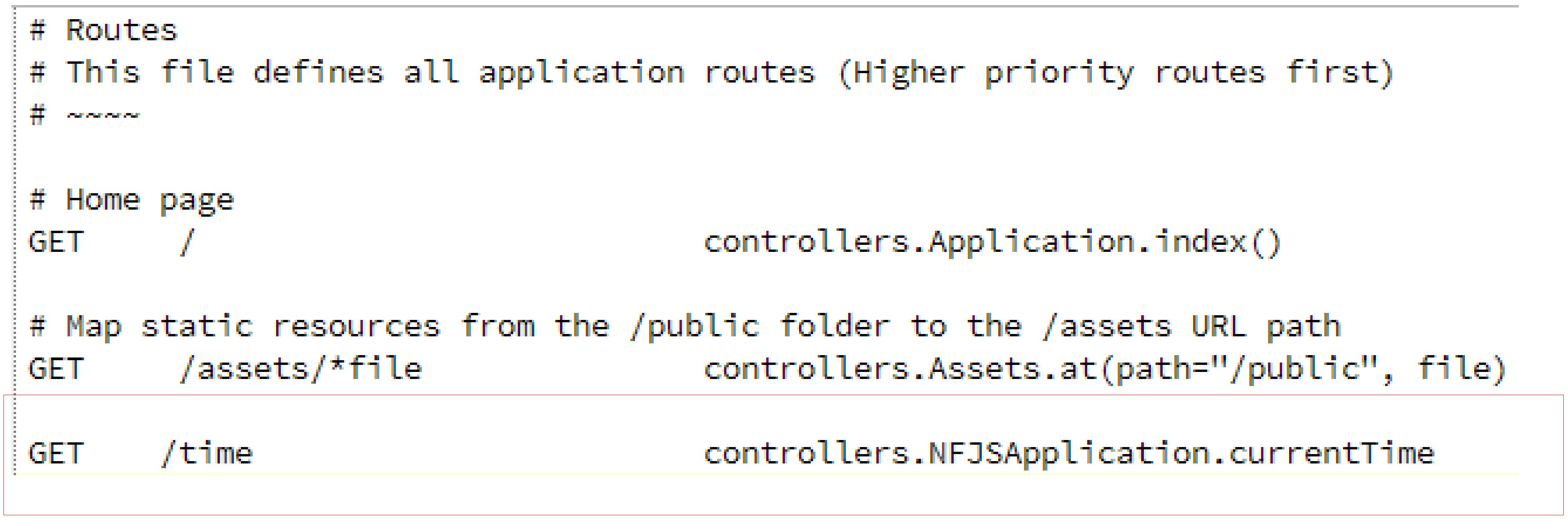


```java
package controllers;

import play.mvc.Controller;
import play.mvc.Result;
import views.html.custom;

import java.util.Date;

public class NFJSApplication extends Controller {
    public static Result currentTime() {
        return ok(custom.render("Good Morning NFJS", new Date()));
    }
}
```

NOTE: CUSTOM WILL REFER TO A PAGE CALLED 'CUSTOM.SCALA.HTML', BUT THAT IS NOT CREATED YET.

Step 3: Java

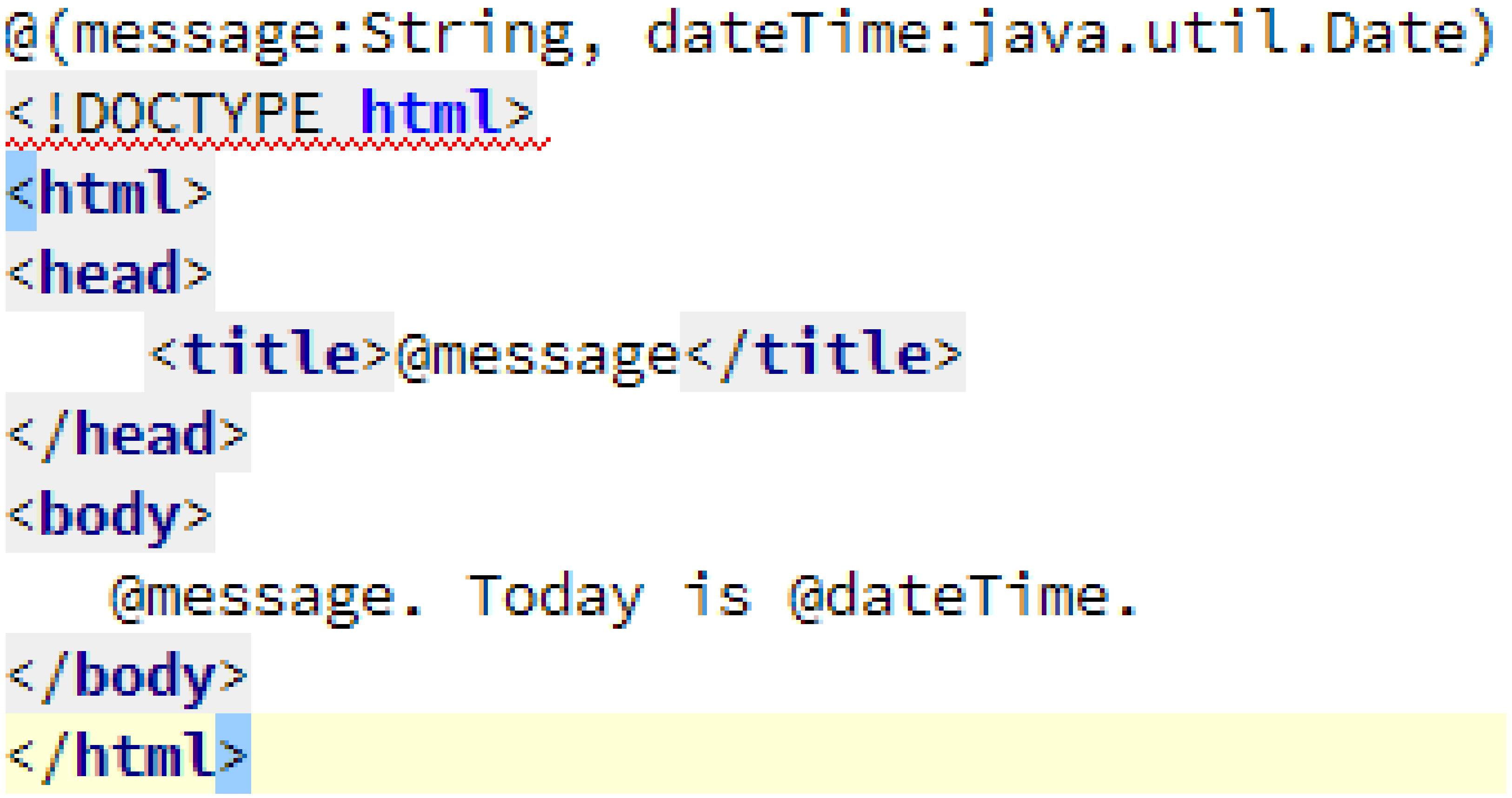
- Add a route in **`conf/routes`** file



```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Home page
GET     /                                   controllers.Application.index()

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file                       controllers.Assets.at(path="/public", file)

GET    /time                                controllers.NFJSApplication.currentTime
```

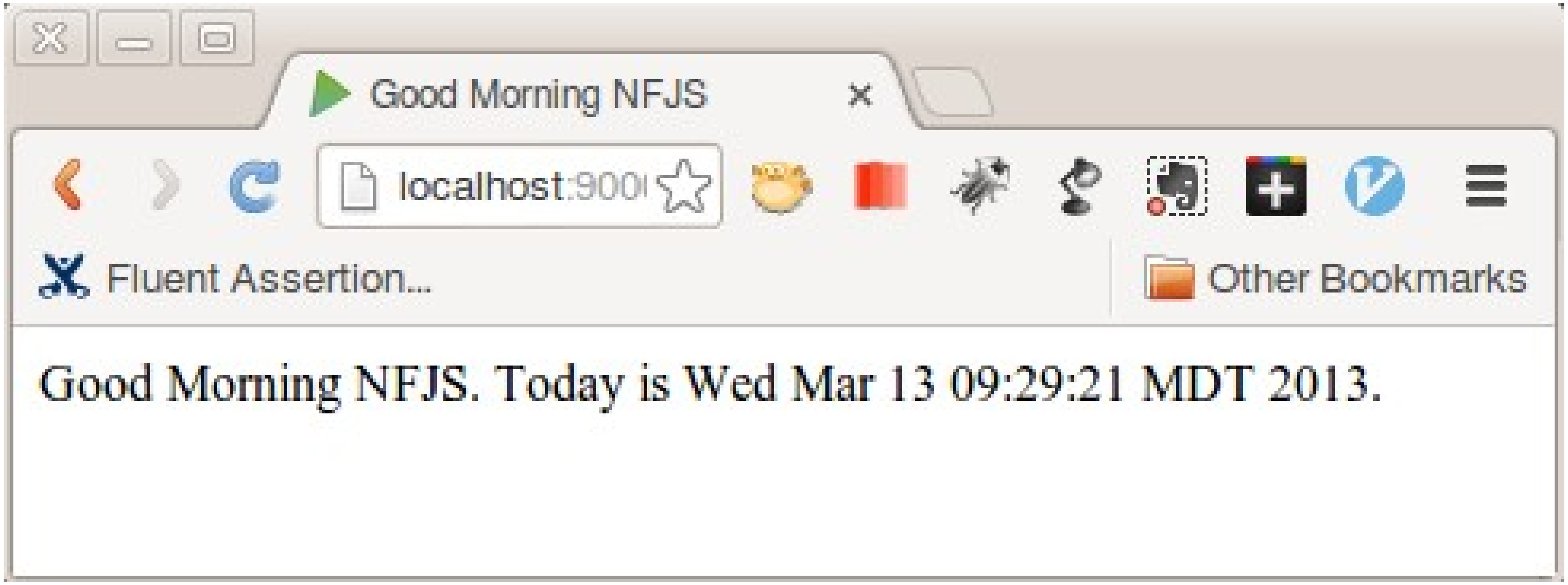
Step 4: Java

- Create a page that will be referenced from any **`Result`** possibly returned.(**`custom.scala.html`**)



```
@(message:String, dateTime:java.util.Date)
<!DOCTYPE html>
<html>
<head>
    <title>@message</title>
</head>
<body>
   @message. Today is @dateTime.
</body>
</html>
```

Java Success!

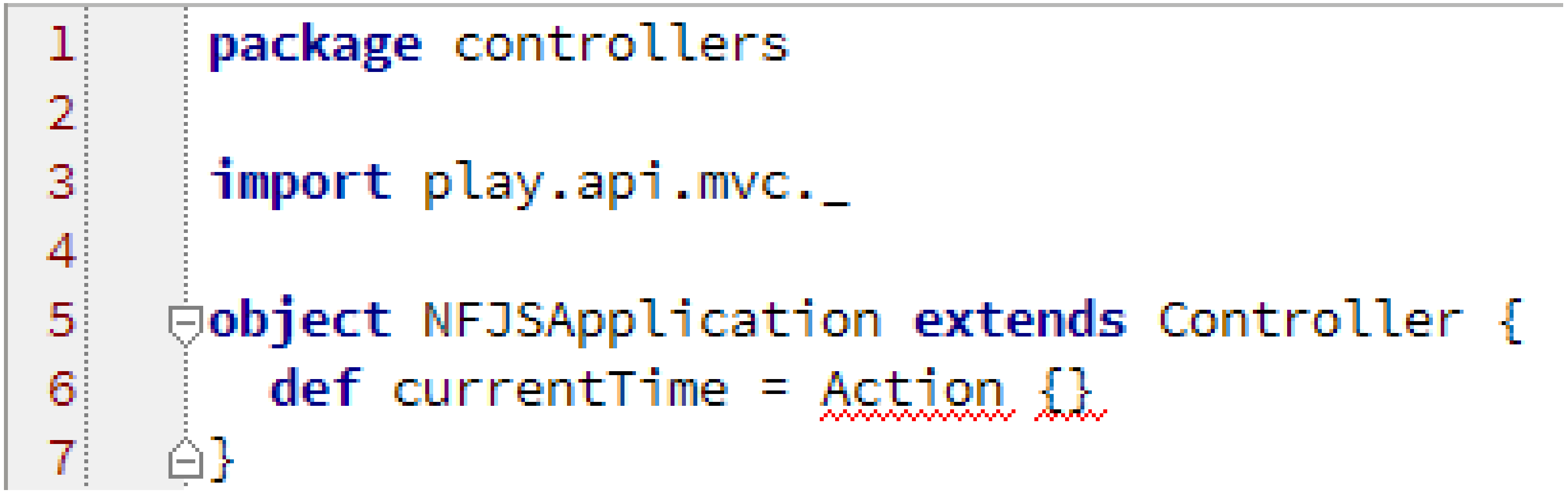


Good Morning NFJS. Today is Wed Mar 13 09:29:21 MDT 2013.

Now in Scala...

Step 1: Scala

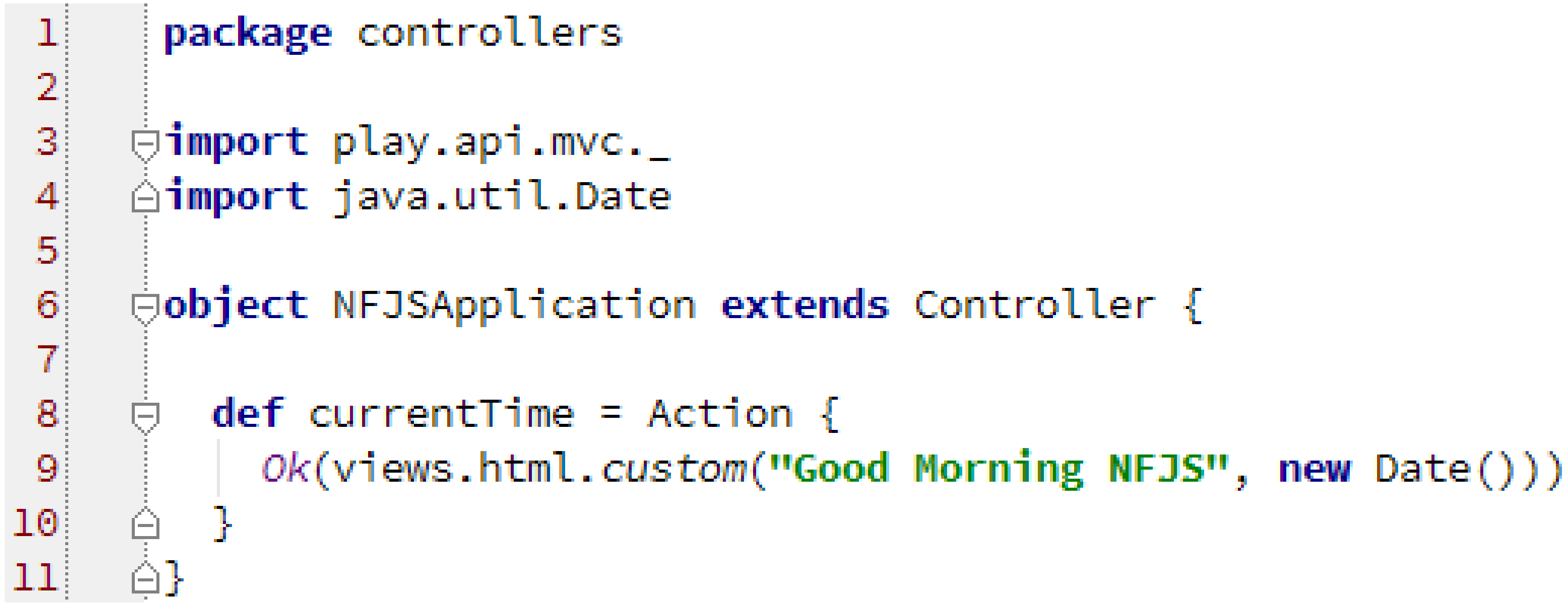
- Create a business method in an **`class`** (Java) or **`object`** (Scala) that extends **`Controller`** and place it in controllers folder.



```scala
package controllers

import play.api.mvc._

object NFJSApplication extends Controller {
  def currentTime = Action {}
}
```

Step 2: Scala

- Return a `Result` from that method

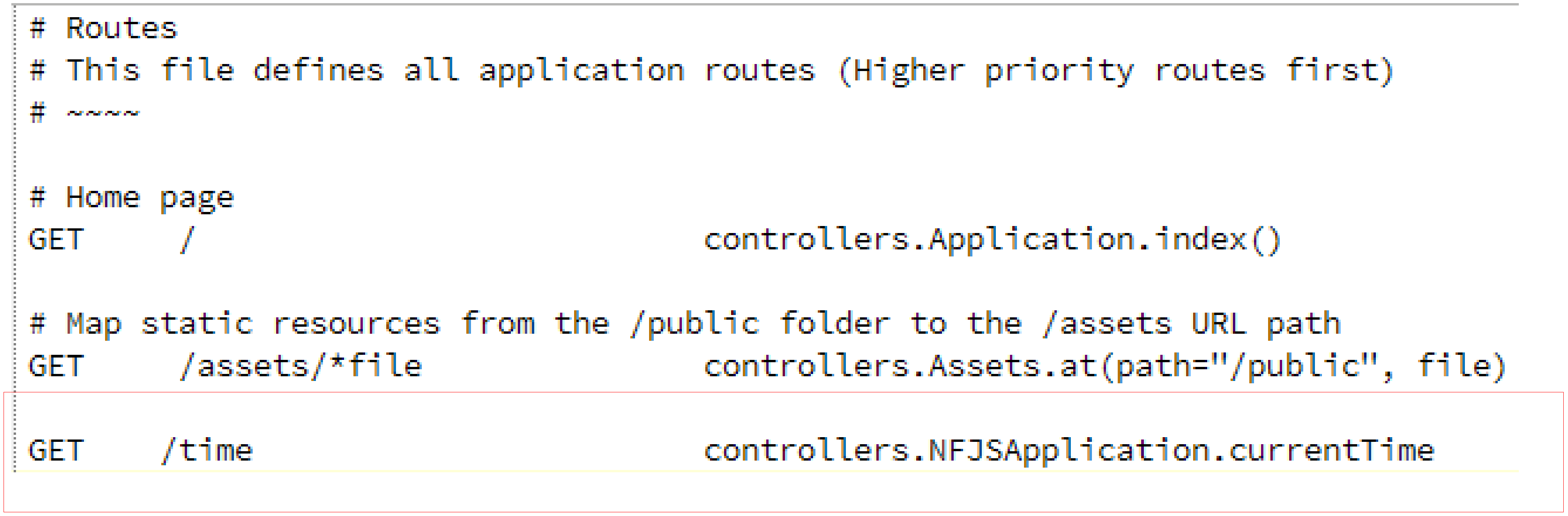


```scala
package controllers

import play.api.mvc._
import java.util.Date

object NFJSApplication extends Controller {

  def currentTime = Action {
    Ok(views.html.custom("Good Morning NFJS", new Date()))
  }
}
```

NOTE: CUSTOM WILL REFER TO A PAGE CALLED 'CUSTOM.SCALA.HTML', BUT THAT IS NOT CREATED YET.

Step 3: Scala

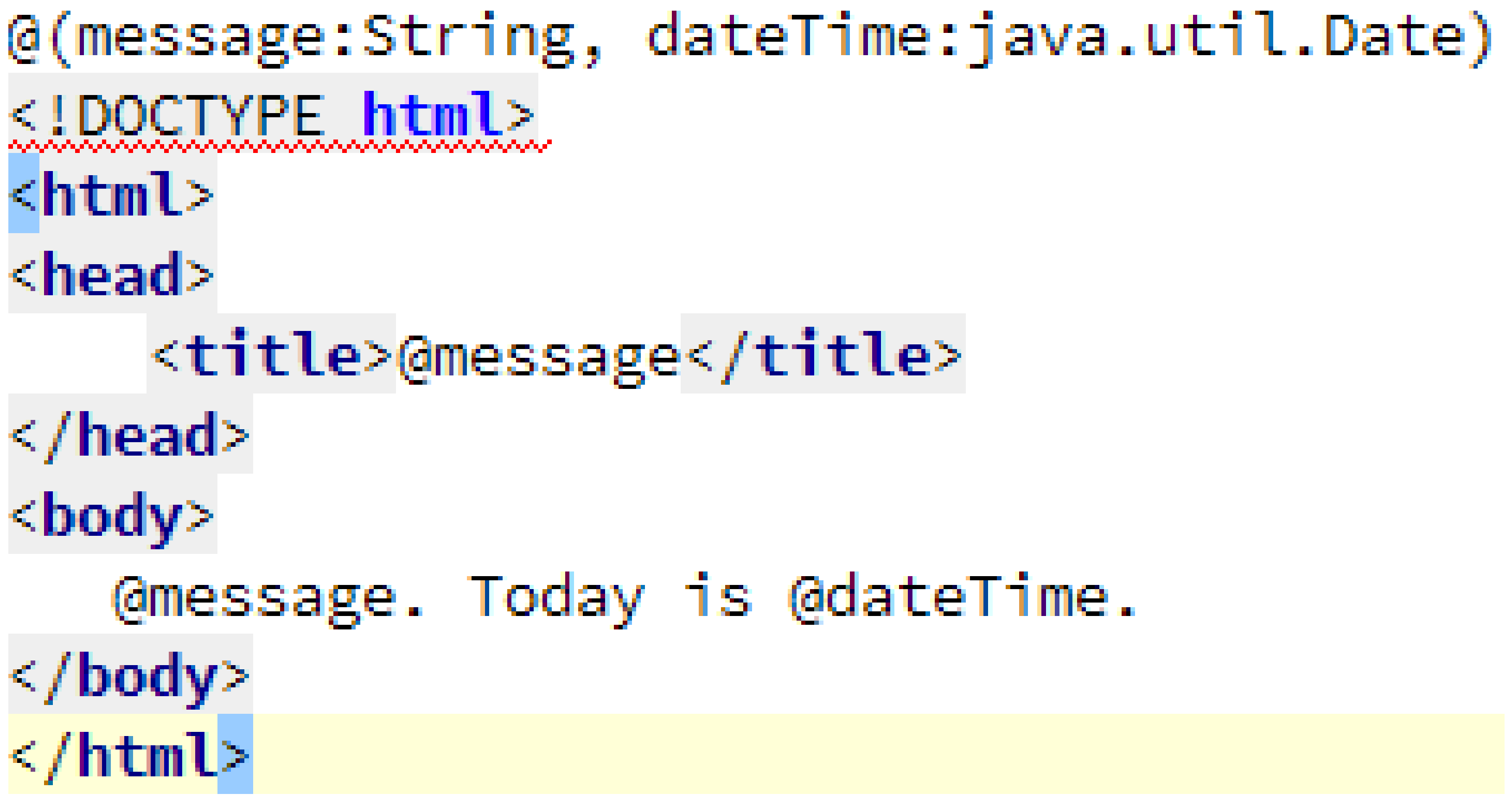
- Add a route in **`conf/routes`** file



```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Home page
GET     /                           controllers.Application.index()

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file               controllers.Assets.at(path="/public", file)

GET    /time                        controllers.NFJSApplication.currentTime
```

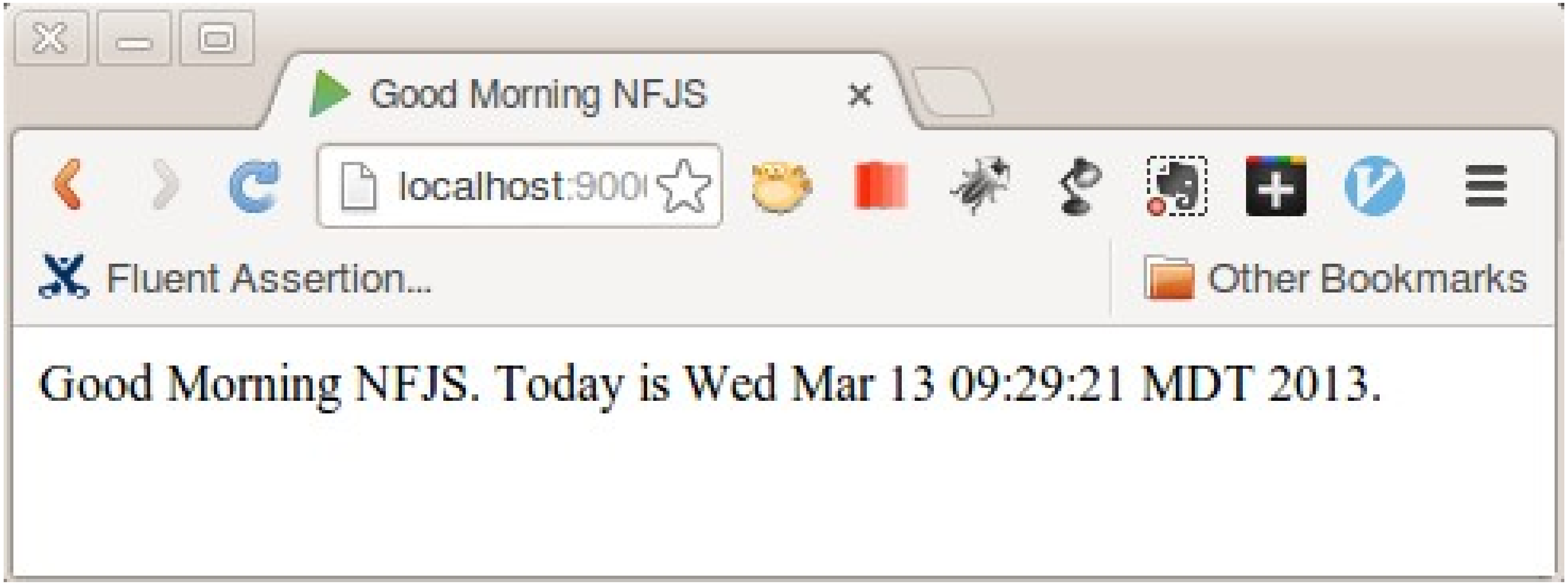
Step 4: Scala

- Create a page that will be referenced from any **`Result`** possibly returned.(**`custom.scala.html`**)



```
@(message:String, dateTime:java.util.Date)
<!DOCTYPE html>
<html>
<head>
    <title>@message</title>
</head>
<body>
   @message. Today is @dateTime.
</body>
</html>
```

Scala Success!



Good Morning NFJS. Today is Wed Mar 13 09:29:21 MDT 2013.

Templating

Working Pages

- Everything under the covers in Scala
- Use templating inside of the HTML file for display logic only.
- Since it is Scala based, generic types are not `<>` but `[]`, e.g. `List[String]`

Templating

- They can generate any text based format file
- A file called **`views/custom.scala.html`** it will generate a class called **`views.html.custom`** with a **`render()`** method.
- This can now be treated like an object with a reference. **`Html customPage = views.html.custom.render(welcome, date)`**

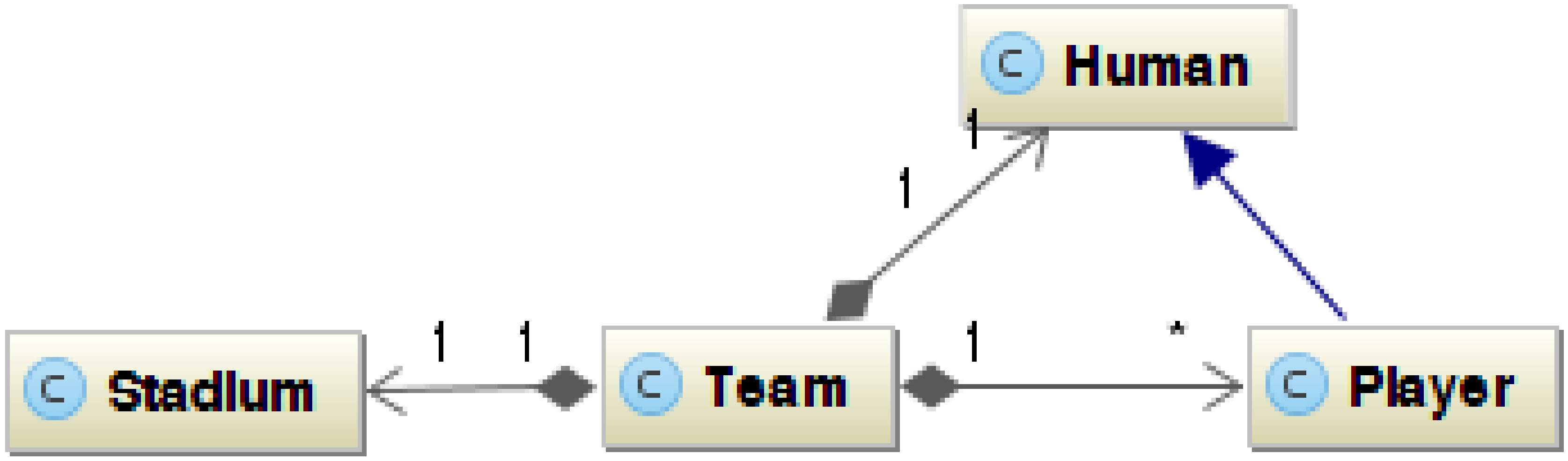
Templating

- The `@` character is analogous to the `$` character in JSTL, and the `#` character in JSF.

Templating

- Given a football team with many players, one coach, and one stadium.
- Given a player has a position.
- Given a player and a coach has a first name, and a last
- Given a team has a name
- Given a stadium has a city and a state

Templating



Getting Simple Values

```
@(team: models.Team)
<!DOCTYPE html>    `
<html>
<head>
    <title>My Favorite Team @team.getName()</title>
</head>
<body>
   Team: @team.getName()
</body>
</html>
```

Demo

Integrating Simple Values

```
@(team: models.Player)

@(player.getFirstName() + “ “ + player.getLastName())
```

Integrating Simple Multiblock Values

```
@(team: models.Player)

@{val fullName = player.getFirstName() + “ “ +
player.getLastName(); fullName}
```

Conditional

```
@if(team.players.isEmpty()) {
  <h1>No Team Members</h1>
} else {
  <h1>@team.players.size() players online!</h1>
}
```

For Loop Iteration

```
<h2>Using for loop</h2>
@for(p <- players) {
    <p>@(p.getFirstName() + " " + p.getLastName())</p>
}
```

Keep in Mind Scala

- All the method that methods

Map Iteration

```
<h2>Using Map</h2>
@players.map { p =>
  <p>@(p.getFirstName() + " " + p.getLastName())</p>
}
```

Map and If Iteration

```
<h2>Using Map and If</h2>
@players.map { p =>
  @if(p.getLastName().size > 5) {
    <p>@(p.getFirstName() + " " + p.getLastName())</p>
  }
}
```

Filter Iteration

```
<h2>Using Filter</h2>
@players.filter(x => x.getLastName().size > 5).map{p =>
   <p>@(p.getFirstName() + " " + p.getLastName())</p>
}
```

Map and mkString() Iteration

```
<h2>Using Map with mkString</h2>
@players.map{p =>
  @p.getLastName()
}.mkString(":")
```

Filter Iteration

```
<h2>Using Filter</h2>
@players.filter(x => x.getLastName().size > 5).map{p =>
   <p>@(p.getFirstName() + " " + p.getLastName())</p>
}
```

Map and If Iteration

```
<h2>Using Map and If</h2>
@players.map { p =>
  @if(p.getLastName().size > 5) {
    <p>@(p.getFirstName() + " " + p.getLastName())</p>
  }
}
```

Map Iteration

```
<h2>Using Map</h2>
@players.map { p =>
  <p>@(p.getFirstName() + " " + p.getLastName())</p>
}
```

Reuseable Blocks

```
@fullName(p:Player) = {
  @(p.getFirstName() + " " + p.getLastName())
}

<h2>Using for loop</h2>
@for(p <- players) {
    <p>@(fullName(p))</p>
}

<h2>Using Map</h2>
@players.map { p =>
  <p>@(fullName(p))</p>
}
```

Defining Blocks

```
@defining(player.getFirstName() + " " +
player.getLastName()) {
   fullName =>
      <div>@fullName is online now!</div>
}
```

Comments

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

No More Need for Tiles

```
@(title: String)(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-1.9.0.min.js")"
              type="text/javascript"></script>
    </head>
    <body>
        @content
    </body>
</html>
```

No More Need for Tiles

```
@(title: String)(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-1.9.0.min.js")"
              type="text/javascript"></script>
    </head>
    <body>
        @content
    </body>
</html>
```

Let's take templating further!

```
@(title: String)(leftMenu:Html)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        //Image and javascript junk
        <title>@title</title>
    </head>
    <body>
        <table id="layout">
          <tr>
             <td width="20%">@leftMenu</td>
             <td width="80%">@content</td>
          </tr>
        </table>
    </body>
</html>
```

Why does that work?

Currying

```
def foo(x:Int)(y:Int)(z:Int) = {
           x + y + z
} // (x: Int)(y: Int)(z: Int)Int


val a = foo _ //Int => (Int => (Int => Int)) = <function1>
val b = a(4)  //Int => (Int => Int) = <function1>
val c = b(2) //Int => Int = <function1>
val d = c(1) //7
```

Templating is Currying!

```
@(title: String)(leftMenu:Html)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        //Image and javascript junk
        <title>@title</title>
    </head>
    <body>
        <table id="layout">
          <tr>
             <td width="20%">@leftMenu</td>
             <td width="80%">@content</td>
          </tr>
        </table>
    </body>
</html>
```

FUNCTIONAL PROGRAMMING FTW!

In Depth-Routing

- `conf/routes` stores all the routing files

In Depth-Routing

- `conf/routes` stores all the routing files
- All the famous HTTP method calls are available:
  - GET, POST, PUT, DELETE, HEAD
- Three elements defined:
  - The HTTP method
  - URI Mapping
  - Associated Class/Method
- If there is a conflict, first one declared wins

Value Routing

- `GET /player/:id controllers.Players.view(id: Long)`
- `:id` is dynamic
- In Scala the variable name comes before the type

Value Routing

- `GET /player/:id controllers.Players.view(id: Long)`
- `:id` is dynamic
- In Scala the variable name comes before the type

How does Routing Map (Java)?

- `GET /player/:id controllers.Players.view(id: Long)`

```
public static Result view(Long id) {
  return ok(views.html.Client.show(playerDAO.find(id));
}
```

How does Routing Map (Scala)?

- `GET /player/:id controllers.Players.view(id: Long)`

```
val view(id:Long) = Action { request =>
  Ok(views.html.Player.show(playerDAO.find(id))
}
```

Regex Routing

- `GET /player/$id<[0-9]+> controllers.Players.view(id: Long)`
- `:id` is dynamic
- `:id` is defined by regular expression

Path Routing

- `GET /player/files/*path controllers.Players.file(path)`
- `path` is dynamic
- The path is defined what is after `/player/files/`
- e.g. Given: `/player/files/td/record.xls` then Path is `/td/record.xls`
- e.g. Given: `/player/files/running/qb/record.xml` then path is /`running/qb/record.xml`

Path Routing

- `GET /player/files/*path controllers.Players.file(path)`
- `path` is dynamic
- The path is defined what is after `/player/files/`
- e.g. Given: `/player/files/td/record.xls` then path is `/td/record.xls`
- e.g. Given: `/player/files/running/qb/record.xml` then path is /`running/qb/record.xml`

Default Routing

- **`GET   /         controllers.blog.show(page = "home")`**
- `GET   /:page    controllers.blog.show(page)`

Query Parameters

```
# Pagination links, like /blog?page=3

GET   /blog  controllers.Blog.list(page: Integer ?= 1)
```

Optional Query Parameters

```
# Optional. /players/list?position=QB

GET /players/list controllers.Player.list(position ?= null)
```

Reverse Routing

```
<!DOCTYPE html>
<html>
<head>
    <title>Index of options</title>
</head>
<body>
       <a href=
           "@controllers.routes.NFJSApplication.favoriteTeam">
           View my favorite team
        </a>
</body>
</html>
```

Inside of `conf/routes`

```
GET /team     controllers.NFJSApplication.favoriteTeam
GET /players  controllers.NFJSApplication.favoritePlayers
GET /players2 controllers.NFJSApplication.favoritePlayers2
GET /player/  controllers.NFJSApplication.index
```

Reverse Routing, How its Rendered

```
<!DOCTYPE html>
<html>
<head>
    <title>Index of options</title>
</head>
<body>
       <a href="/team">
           View my favorite team
       </a>
</body>
</html>
```

Inside of `conf/routes`

```
GET /team     controllers.NFJSApplication.favoriteTeam
GET /players  controllers.NFJSApplication.favoritePlayers
GET /players2 controllers.NFJSApplication.favoritePlayers2
GET /player/  controllers.NFJSApplication.index
```

Forms in Play

Forms in Java

```
Form<Player> form = Form.form(Player.class)
```

```
import static play.data.Form.form;

Form<Player> form = form(Player.class)
```

Forms in Java

```
public static Result preCreate() {
  playerForm = form(Player.class);
  return ok(
     views.html.player.
     create.render(playerForm));
}
```

Setting up the form

```
<form action="/player/create" method="POST">
    <div>First Name:
       <input type="text" id="firstName"
              name="firstName"/></div>
    <div>Last Name:
       <input type="text"
              name="lastName"/></div>
    <input type="submit"/>
</form>
```

Setting up the form with helpers

```
@helper.form(action =
             controllers.player.routes.
             PlayerController.create()) {
    @helper.inputText(form("firstName"))
    @helper.inputPassword(form("lastName"))
    <input type="submit"/>
}
```

Setting up the route

```
GET     /player/create
controllers.player.PlayerController.preCreate

POST    /player/create
controllers.player.PlayerController.create
```

Validation of Forms in Java

```
public class Player {

    @Required
    public String firstName;

    @Required
    public String lastName;
}
```

Processing the Form for errors

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

No Model, No Problem

```
public static Result hello(){
    DynamicForm requestData =
         form().bindFromRequest();
    String firstname =
         requestData.get("firstname");
    String lastname =
         requestData.get("lastname");
    return ok("Hello " + firstname + " " +
lastname);
}
```

Filling in default information

```
playerForm = playerForm.fill(new Player("Tom",
"Brady"))
```

Forms in Scala

Simple Forms in Scala

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

val playerForm = Form(
  tuple(
    "firstName" -> text,
    "lastName" -> text
  )
)
```

Complicated Forms in Scala

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

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

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

Complicated Forms in Scala

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

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

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

Constrained Forms in Scala

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

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

Ad-hoc constraints in Java

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

Processing Forms

```
loginForm.bindFromRequest.fold(
  formWithErrors =>
  BadRequest(views.html.login(formWithErrors)),
  value =>
  Redirect(
 routes.HomeController.home).flashing("message"
-> "Welcome!" + value.firstName)
)
```

Processing Forms

```
loginForm.bindFromRequest.fold(
  formWithErrors =>
  BadRequest(views.html.login(formWithErrors)),
  value =>
  Redirect(
 routes.HomeController.home).flashing("message"
-> "Welcome!" + value.firstName)
)
```

Autofilling a form before sending through

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

Nested Values in Forms

```
case class User(name: String, address: Address)
case class Address(street: String, city:
String)

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

FIELD NAMES ARE CALLED: ADDRESS.STREET, ADDRESS.CITY

Repeated Values in Forms

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

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

Field names are called: emails[0], emails[1], emails[2]

Optional Values in Forms

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

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

None if the field email is either not in Payload or is an ""

Ignored Values in Forms

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

val userForm = Form(
  mapping(
    "id" -> ignored(1234),
    "name" -> text,
    "email" -> optional(email)
  )(User.apply, User.unapply)
)
```

Other Overview Items

JSON

- Both languages use Jackson API
- Receive and Generate JSON
- Scala enjoys functional programming to make things easier.
- Scala makes use of Reads, Writes types to carry the processing load.

Databases

- On Java side, Ebean, and JPA can be used.
- On the Scala side, Anorm, can be used.
- There are different ORMs available in Scala:
  - Squeryl
  - ScalaQuery
  - Slick (Typesafe)
- Casbah is a MongoDB Scala Driver

Play other features

- Caching using EH Cache
- Internationalization
- Integration with Akka
- Global Object
- Testing Frameworks
- File Uploads
- XML Manipulation in Java uses W3CDocument
- XML Manipulation in Scala uses internal XML Nodes