

# Play! Framework

*We are Reactive !*

# Aujourd'hui

Que favorise les nouveaux frameworks web ?

# Aujourd'hui

Que favorise les nouveaux frameworks web ?

1. Asynchrone
2. Stateless
3. Légèreté

# Aujourd'hui

Quels sont les intérêts de ces focalisations ?

# Aujourd'hui

Quels sont les intérêts de ces focalisations ?

1. Disponibilité
2. Montée en charge

Donc faire des applications réparties...

# Présentation

Play! est écrit en Scala

Framework web complet

Rechargement à chaud des classes modifiées

# Scala

Langage fonctionnel

Tourne sur la JVM

Apporte des fonctionnalités absentes de Java

Démarrer avec Play!

# Installation

Dézipper l'archive téléchargée sur le site

Ajouter la variable d'environnement  
`$PLAY_HOME` et modifier `$PATH`

S'assurer d'avoir mis `$JAVA_HOME`

# Hello World!

Exécuter “play new hello”

Rentrer dans le shell play en tapant “play”

Exécuter “run”

# Anatomy

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
build.sbt	→ Application build script
conf	→ Configurations files and other non-compiled resources
├ 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
└ plugins.sbt	→ sbt plugins including the declaration for Play itself
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

# VCS

Le .gitignore standard pour une application  
Play! :

```
logs  
project/project  
project/target  
target  
tmp  
dist  
.cache
```

# Shell

Les opérations sur votre application se font via le shell Play!

Très simple à utiliser (comme celui de MongoDB)

La commande “help” devrait vous aider

# Shell

- Lancer en mode dev : “run”
- Lancer en prod : “start”
- Faire un build : “prod”
- Intégrer avec Eclipse : “eclipse”

# Play!

Actuellement en version 2.2

Favorise MVC

Les api Java sont dans le package *play*

# Play! Framework

# Concepts

Action → Reçoit une requête et produit une réponse HTTP

Controller → Classe héritant de *Controller*

Result → Encapsulation d'une réponse HTTP

# Controller

Représente un contrôleur de l'application

Les méthodes publiques sont static

Déclarées dans le fichier routes

# Example

```
package controllers;
```

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

```
public class Application extends Controller {
```

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

```
}
```

# Le fichier routes

Situé dans le répertoire “application”

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

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

# Les routes dynamiques

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

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

# Valeurs dans les routes

Fixes → GET / controllers.**Application**.show(page = "home")

Valeur par défaut →

GET /clients controllers.**Clients**.list(page: Integer ?= 1)

# Manipuler la réponse HTTP

Accessible dans le controller via *response()*

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

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

# Requête asynchrone

## Les méthodes retournent des *Promise*

```
public static Promise<SimpleResult> index() {  
    Promise<Integer> promiseOfInt = Promise.promise(  
        new Function0<Integer>() {  
            public Integer apply() {  
                return intensiveComputation();  
            }  
        }  
    );  
    return promiseOfInt.map(  
        new Function<Integer, SimpleResult>() {  
            public SimpleResult apply(Integer i) {  
                return ok("Got result: " + i);  
            }  
        }  
    );  
}
```

# Streaming

## Play! facilite le streaming

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

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

# Play! Templates

# Template

Permet la génération dynamique de pages

Inspiré d'ASP .NET

Écrit en scala et compilé

# Template

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

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

```
<ul>
```

```
@for(order <- orders) {
```

```
  <li>@order.getTitle() </li>
```

```
}
```

```
</ul>
```

# Template

Un template est l'équivalent d'une fonction  
Scala

Déclaration des paramètres en haut du fichier

Le caractère “@” délimite le code exécutable

# Import et commentaires

Import de classes :

```
@import utils._
```

Commentaire

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

# Itération

## Le mot-clé : for

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

# Condition

Le mot-clé : if

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

# Bloc réutilisable

## Mot-clé : display

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

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

# Afficher du HTML/XML

Par défaut, le contenu est échappé.

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

# Réutilisabilité

views/main.scala.html

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

views/index.scala.html

```
@main(title = "Home") {
  <h1>Home page</h1>
}
```

Play! : JSON / XML

# Récupérer du JSON

## Utilisation de la bibliothèque jackson

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

# JSON Exemple

## Requête POST :

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

## Résultat

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain; charset=utf-8
```

```
Content-Length: 15
```

```
Hello Olivier
```

# Renvoyer du JSON

```
@BodyParser.Of(BodyParser.Json.class)
public static Result sayHello() {
    JsonNode json = request().body().asJson();
    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);
    }
}
```

# Obtenir du XML

## Utilisation de DOM

```
@BodyParser.Of(Xml.class)
public static Result 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);
        }
    }
}
```

# XML Exemple

## Requête POST :

```
curl --header "Content-type: application/json" --request POST --data '<name>Olivier</name>' http://localhost:9000/sayHello
```

## Résultat

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain; charset=utf-8
```

```
Content-Length: 15
```

```
Hello Olivier
```

# Renvoyer du XML

A écrire soit même ou utiliser une bibliothèque spécifique ...

On peut utiliser JAXB pour transformer automatiquement des objets en String

# Play! : Stockage

*Cache, SQL*

# Play! Cache

Pourquoi mettre en cache ?

# Ajouter dans le cache

## Stocker une donnée

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

## Stocker une donnée pour un temps

```
Cache.set("item.key", frontPageNews, 60 * 15);
```

# Récupérer et supprimer

## Récupérer une donnée

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

## Supprimer une donnée

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

# Sauvegarder les réponses

Permet de mettre en cache les réponses HTTP.  
Quand faut-il l'utiliser ?

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

# Utiliser un SGBD

Configurer la configuration dans `conf/application.conf`

Création d'une base en mémoire nommée play

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

# Récupérer une connexion

## Récupérer un DataSource

```
import play.db.*;  
DataSource ds = DB.getDatasource();
```

## Récupérer une connexion

```
import play.db.*;  
Connection connection = DB.getConnection();
```

# Play! : Configuration

*Dépendances, langues, Global object*

# Gestion des dépendances

Les dépendances peuvent être gérées par Play!  
... ou pas

Systeme proche de celui de Maven ...

# Dépendances gérées

Se fait dans le fichier `build.sbt`

## Ajouter une dépendance

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

## Ajouter un resolver

```
resolvers += "Scala-Tools Maven2 Snapshots Repository" at "http://scala-tools.org/repo-snapshots"
```

# Dépendances non gérées

Ajouter simplement le JAR dans un dossier lib à la racine ...

Quels sont les avantages / inconvénients de ces deux méthodes ?

# Internationalisation

Ajouter les langues dans `conf/application.conf`

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

Les fichiers de langues sont `conf/messages.xxx`

```
conf/messages.fr
```

```
conf/messages.en-US
```

# i18n

## Récupérer un message dans le code avec la langue automatique

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

## Forcer la langue

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

# Template et i18n

Afin de récupérer les messages dans les templates

```
@import play.i18n._  
@Messages.get("key")
```

# Formattage

Mettre des arguments dans les messages

Contenu du message :

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

Passage d'argument :

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

# Langues disponibles

Connaître les langues supportées par l'émetteur de la requête permet de lui donner la bonne langue.

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

# Global Object

Permet de spécifier des actions dans toutes l'application

Il faut définir sa propre classe héritant de `GlobalSettings`

Doit être mis dans le package root, ou placée dans la configuration

# Évènements

## Effectuer des actions à différents cycles

```
public class Global extends GlobalSettings {  
  
    public void onStart(Application app) {  
        Logger.info("Application has started");  
    }  
  
    public void onStop(Application app) {  
        Logger.info("Application shutdown...");  
    }  
}
```

# Gérer les exceptions

La méthode `onError` est appelée lors d'une exception

```
public class Global extends GlobalSettings {  
    public Promise<SimpleResult> onError(RequestHeader request,  
    Throwable t) {  
        return Promise.<SimpleResult>pure(internalServerError(  
            views.html.errorPage.render(t)  
        ));  
    }  
}
```

# Gérer les actions inconnues

Action lorsqu'un client effectue une requête à une adresse inconnue

```
public class Global extends GlobalSettings {  
    public Promise<SimpleResult> onHandlerNotFound  
(RequestHeader request) {  
        return Promise.<SimpleResult>pure(notFound(  
            views.html.notFoundPage.render(request.uri())  
        ));  
    }  
}
```

# Play! : Web Services

# Web Services

Effectuer des requêtes HTTP

Utilise les mêmes classes qu'avant

# Web Services

## Effectuer une requête GET

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

## Effectuer une requête POST

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

# Gestion des erreurs

Si une erreur est lancée lors de la requête, on peut la gérer de manière transparente

```
Promise<WS.Response> homePage = WS.url("http://example.com").get();
Promise<WS.Response> callWithRecover = homePage.recover(new Function<Throwable,
WS.Response>() {
    @Override
    public WS.Response apply(Throwable throwable) throws Throwable {
        return WS.url("http://backup.example.com").get().get(timeout);
    }
});
```

# Configuration du client HTTP

Se fait dans `application.conf`

`# Follow redirects (default true)`

`ws.followRedirects=true`

`# Connection timeout in ms (default 120000)`

`ws.timeout=120000`

`# Whether to use http.proxy* JVM system properties (default true)`

`ws.useProxyProperties=true`

`# A user agent string to set on each request (default none)`

`ws.userAgent="My Play Application"`

# Conclusion

Framework utilisé par de grands acteurs  
(LinkedIn...)

Léger, performant, flexible

Bien supérieur à ce qui est proposé  
actuellement en Java (pour moi)