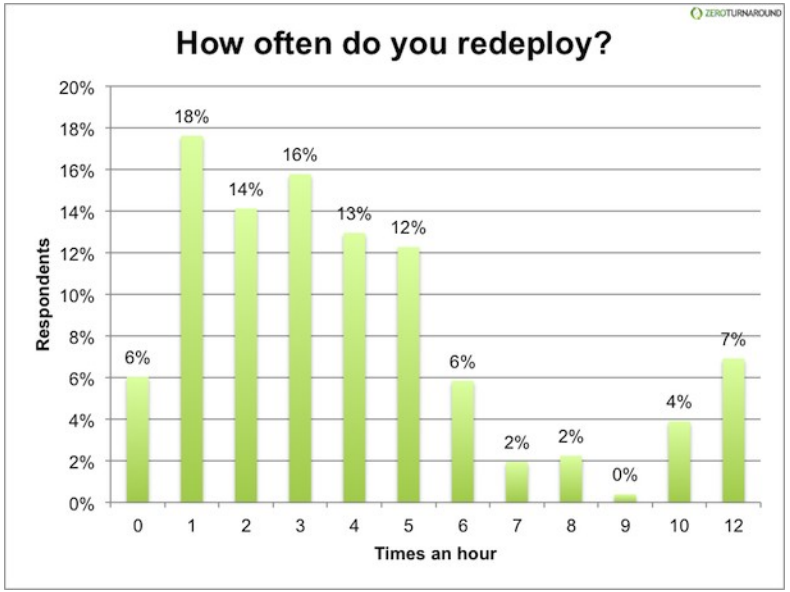
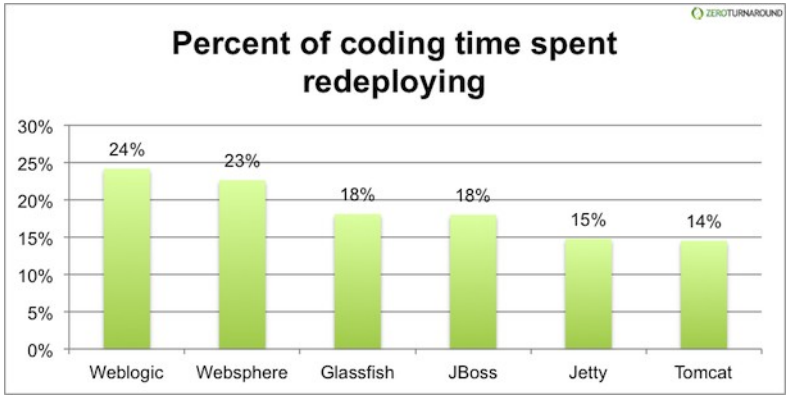
Lets Play

Prasanta Mohanty,
Manish Kumar

Challenges of Java Web App Development

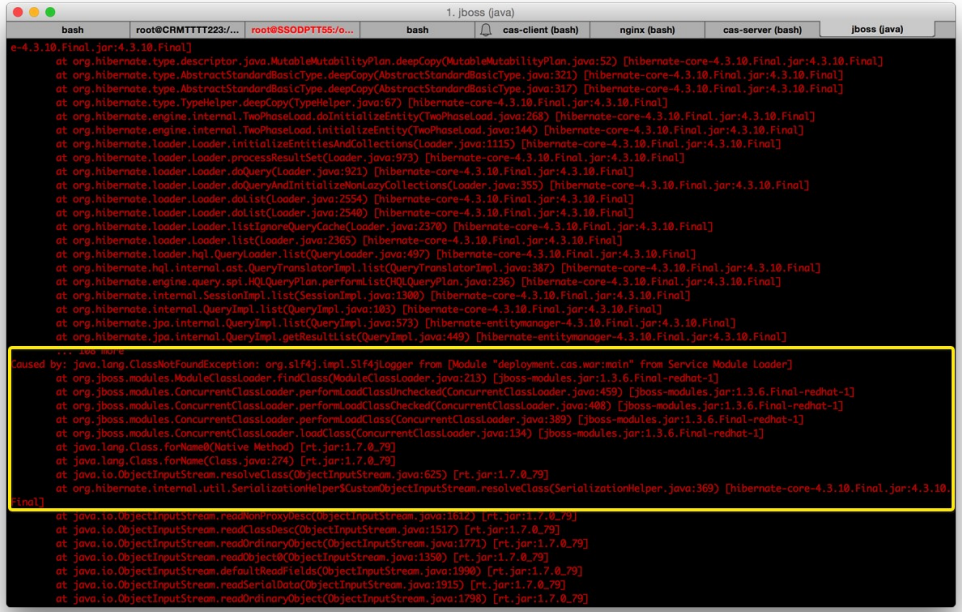
Lots of Time Waiting for Server Redeploys...



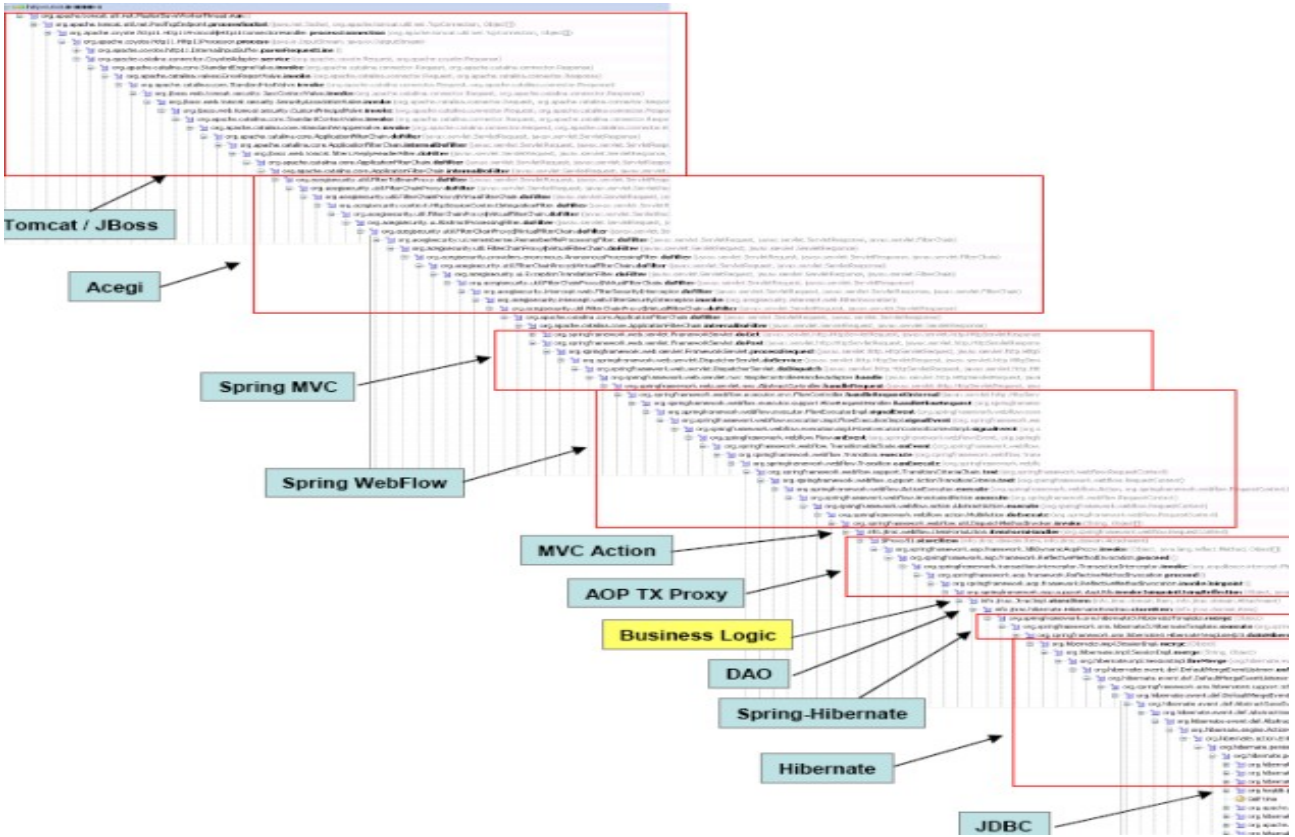


Source: http://zeroturnaround.com/rebellabs/java-ee-productivity-report-2011/#redeploy_times

Long, Ugly Error Messages



Stack Trace of any Error



Crazy XML Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://ja
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://j
    id="WebApp_ID" version="2.5">
    <display-name>library-web</display-name>
    <servlet>
        <servlet-name>CreateUserServlet</servlet-name>
        <servlet-class>library.servlets.CreateUser</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>CreateUserServlet</servlet-name>
        <url-pattern>/create_user</url-pattern>
    </servlet-mapping>
</web-app>
```

Restful URLs

WEB.XML

```
<servlet-mapping>
<servlet-name>springmvc</servlet-name>
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

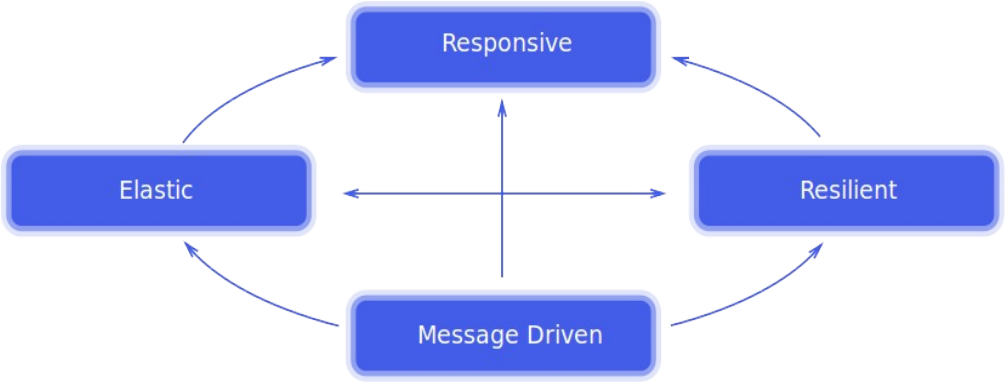
CONTROLLER.JAVA

```
@Controller
@RequestMapping("/people")
public class PeopleController {
@RequestMapping(“entrypoint/{collectionName}”,
method=RequestMethod.GET)
public @ResponseBody String getPeople() {
return GsonFactory.getInstance().toJson(LookupDao.getInstance().getP
}
@RequestMapping(value="{id}", method=RequestMethod.GET)
public @ResponseBody String getPerson(@PathVariable String id) {
return GsonFactory.getInstance().toJson(LookupDao.getInstance().get...
}
```

Modern Web App Development

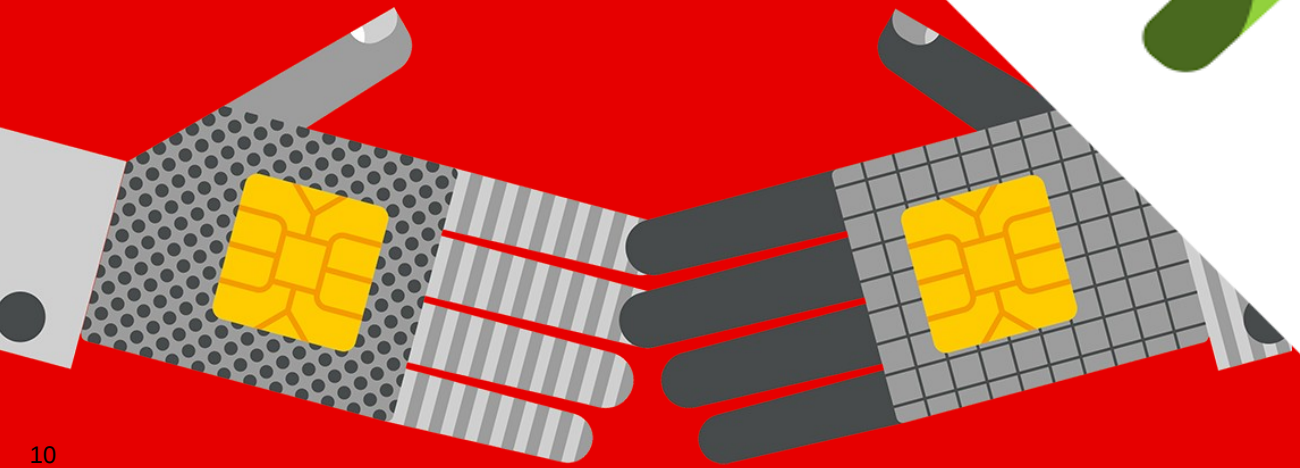
- Mobile
- NoSQL
- Real-Time
- Big Data
- Asynchronous
- Immutability
- Connected Devices
- HTML5 & Java script

The Reactive Manifesto
http://www.ReactiveManifesto.org/



- The Reactive Manifesto is a "bullshit bingo words dictionary", which incorporates **distributed system design best practices**.
- **Definition:-** These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

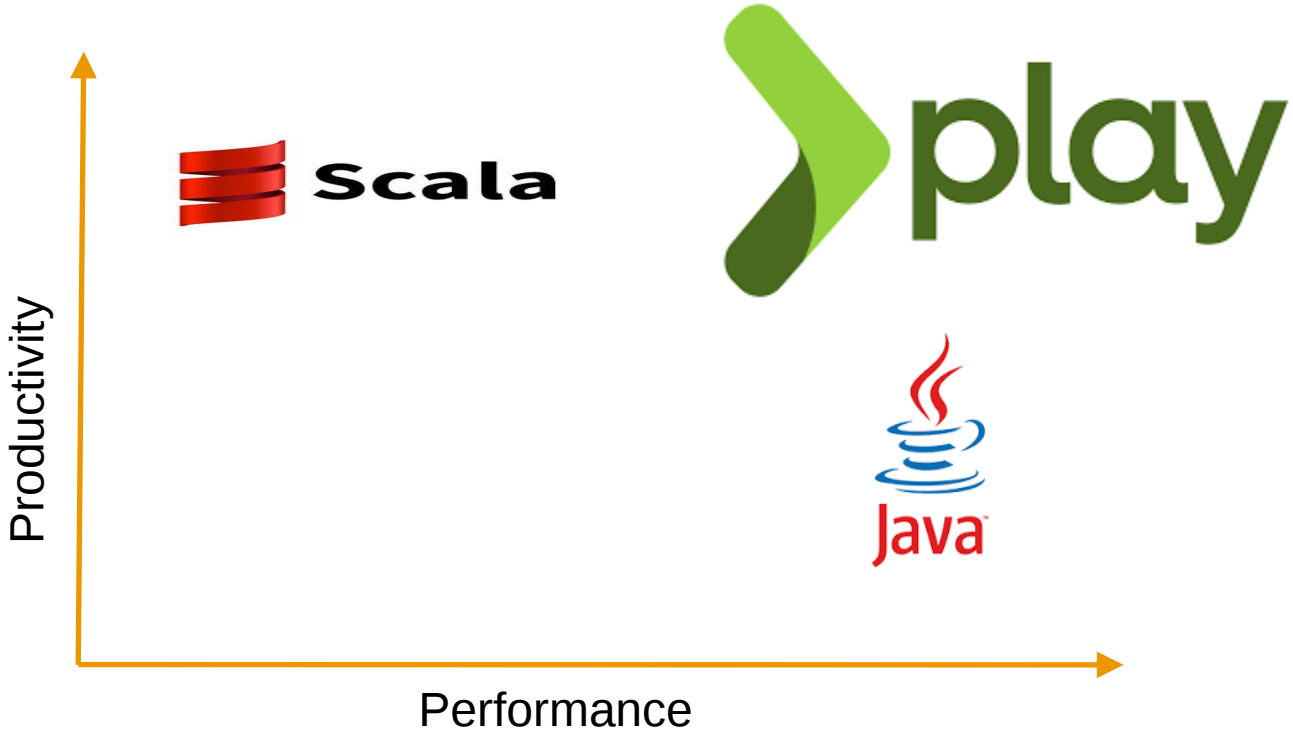
Meet Play Framework



Why is Play Framework so fast ?

- **The short answer:** Play is a stateless, asynchronous, and non-blocking framework that uses an underlying fork-join thread pool to do work stealing for network operations, and can leverage Akka for user level operations.

Goal: Performance + Productivity



No more JEE container

Java EE Web vs Play framework architecture

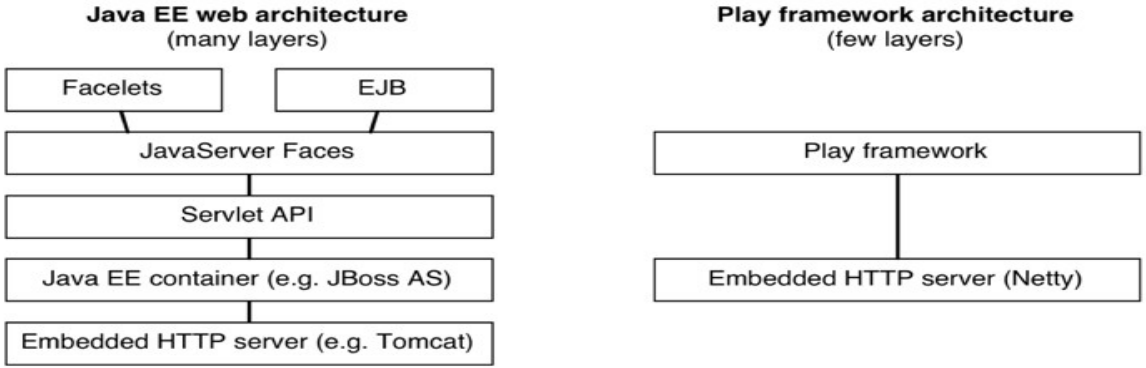


Figure 1.2 Java EE 'lasagne' architecture compared to Play's simplified architecture

Focused on Developer **Productivity**

- Live code changes when you refresh the browser
- More friendly error messages directly in browser
- Type safety in the templates
- Cool console & build tools

Designed for the Modern Web

- RESTful by default
- Auto-compile LESS and CoffeeScript files
- JSON is a first-class citizen
- Web sockets, other HTTP Streaming Support

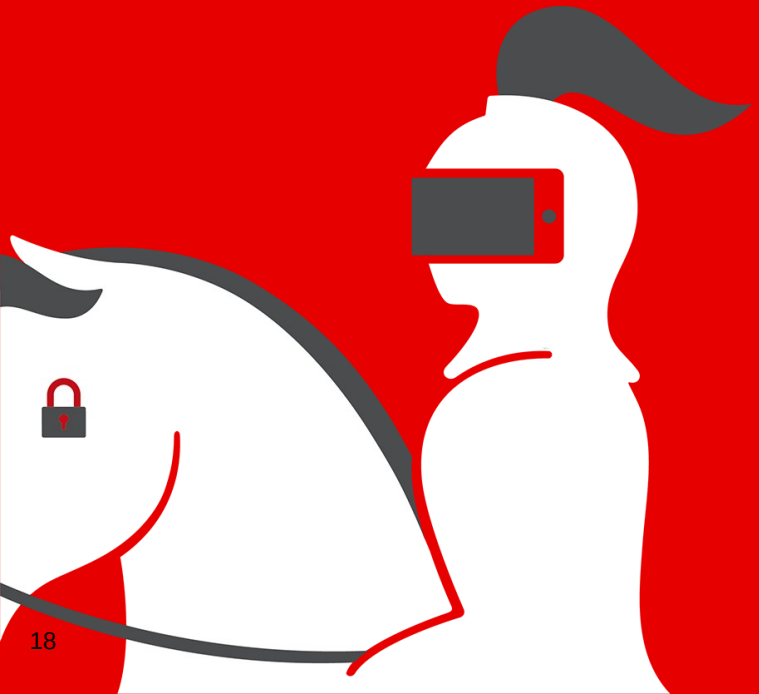
Stateless and Built for Scale

- Forces every aspect of your app to be stateless
- Non-Blocking I/O
- Well-suited for real-time

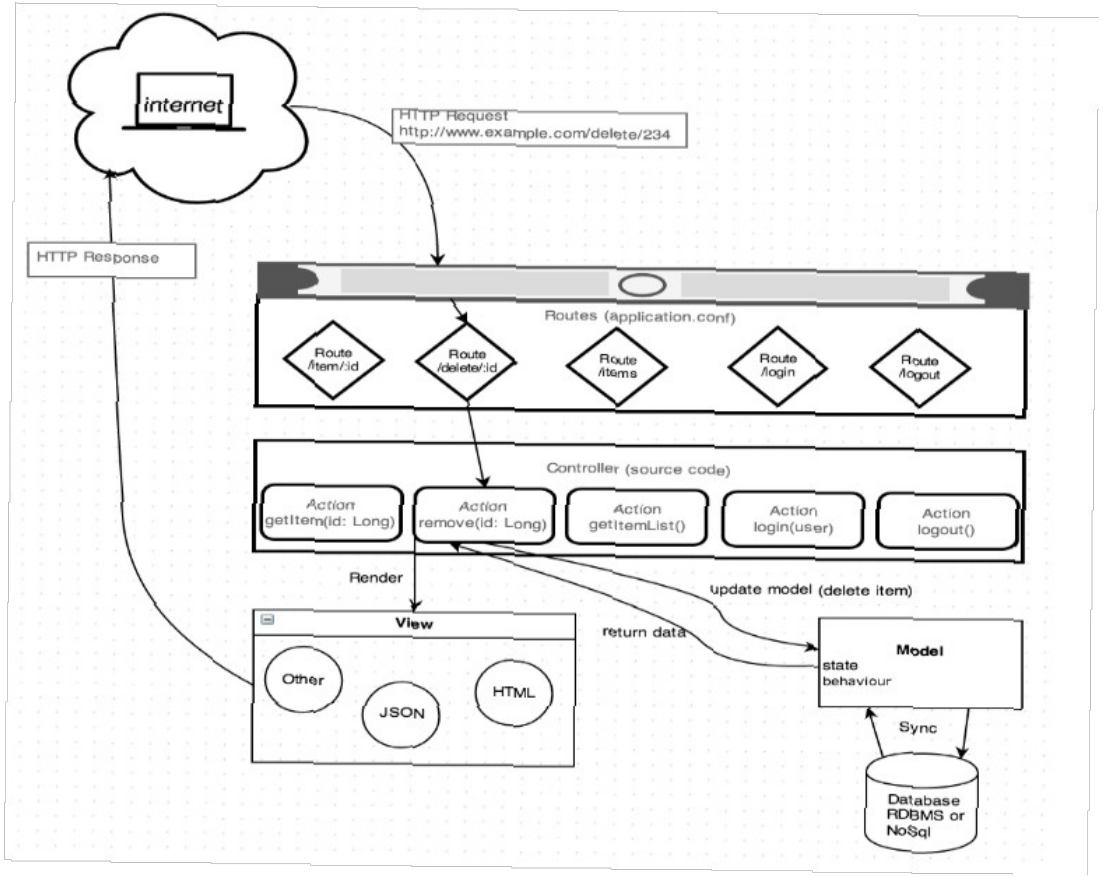
Of course, nothing's perfect

- You can mostly avoid Scala, but not completely (of course, Scala itself is pretty cool)
- For advanced build logic, SBT has a steep learning curve
- Template system works well, but sometimes the functional Paradigm can feel awkward
- Backward compatibility is a major issue

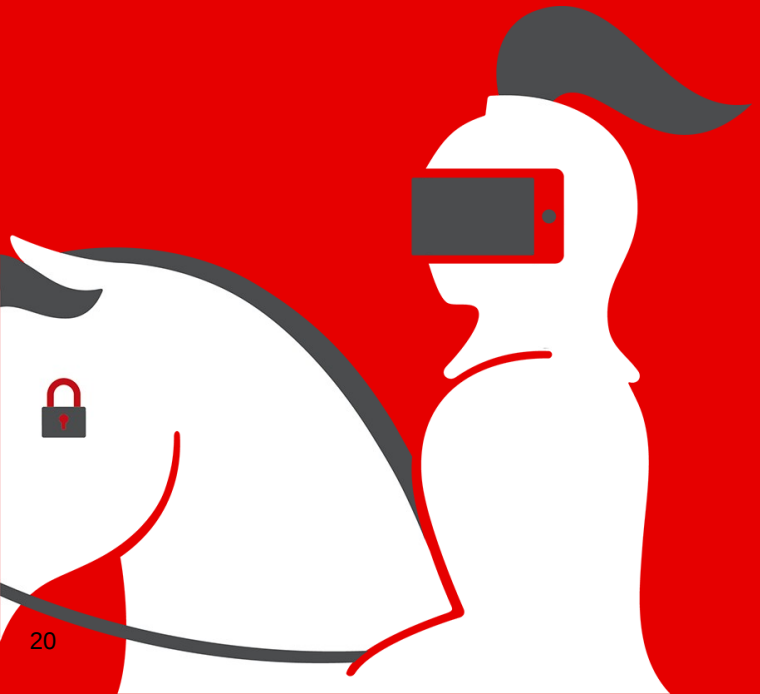
Lets play



Architecture



Code Refresh & Running



Code & Refresh vs. WAR Deployment

- It is possible because Play is stateless and tightly integration between Play and sbt
- Classloader hierarchy manage by sbt
- Tricky (You don't need JRebel)
  - For each request reloader check if there's any code changes
  - Compile changed codes remove the old application classloader
  - create a new application classloader with the updated classes.
  - restart application (only application, no need to restart the JVM)

Project Structure

- app
- public
- conf
- project
- build.sbt

```
app                     → Application sources
 └ assets               → Compiled asset sources
 └ controllers          → Application controllers
 └ models               → Application business layer
 └ views                → Templates
conf                    → Configurations files
 └ application.conf     → Main configuration file
 └ routes               → Routes definition
public                  → Public assets
 └ stylesheets          → CSS files
 └ javascripts          → Javascript files
 └ images               → Image files
project                 → sbt configuration files
 └ Build.scala          → Application build script
 └ plugins.sbt          → sbt plugins
lib                     → Unmanaged libraries
dependencies
logs                    → Standard logs folder
target                  → Generated stuff
test                    → Unit or functional tests
```

Route

- Scala syntax
- Optional parameters
- Reverse routing
  - <full-package-name>.routes.<controller>.<action>
- REST friendly

```
# Home page
GET     /                           controllers.Application.index
```

Controller

- Actions return Result
- Static method
- Use dependency injection (JSR 330 in Play 3)
- HTTP manipulation (body parsers)
  - REST service (JSON and XML)

View

- Type safe template engine
- Scala Template (Twirl inspired by ASP.NET Razor)
- Http form handling

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);
```

Session

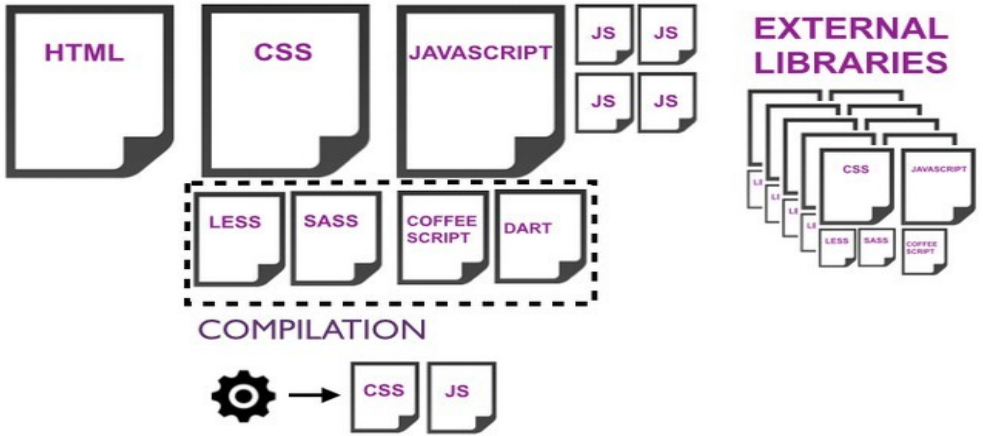
- Session data are not stored in the server
- Session data added to each subsequent HTTP
- Request, using Cookie.
- Cookie are signed with a secret key so the client can’t modify the cookie data or it will be invalidated
- Use cache for sever side session data

SBT Built tool

```
name := """MyNewPlay"""

version := "1.0-SNAPSHOT"

lazy val root = (project in file(".")).enablePlugins(PlayJava, PlayEbean)

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  javaJdbc,
  cache,
  javaWs
)
~
~
~
~
~
```

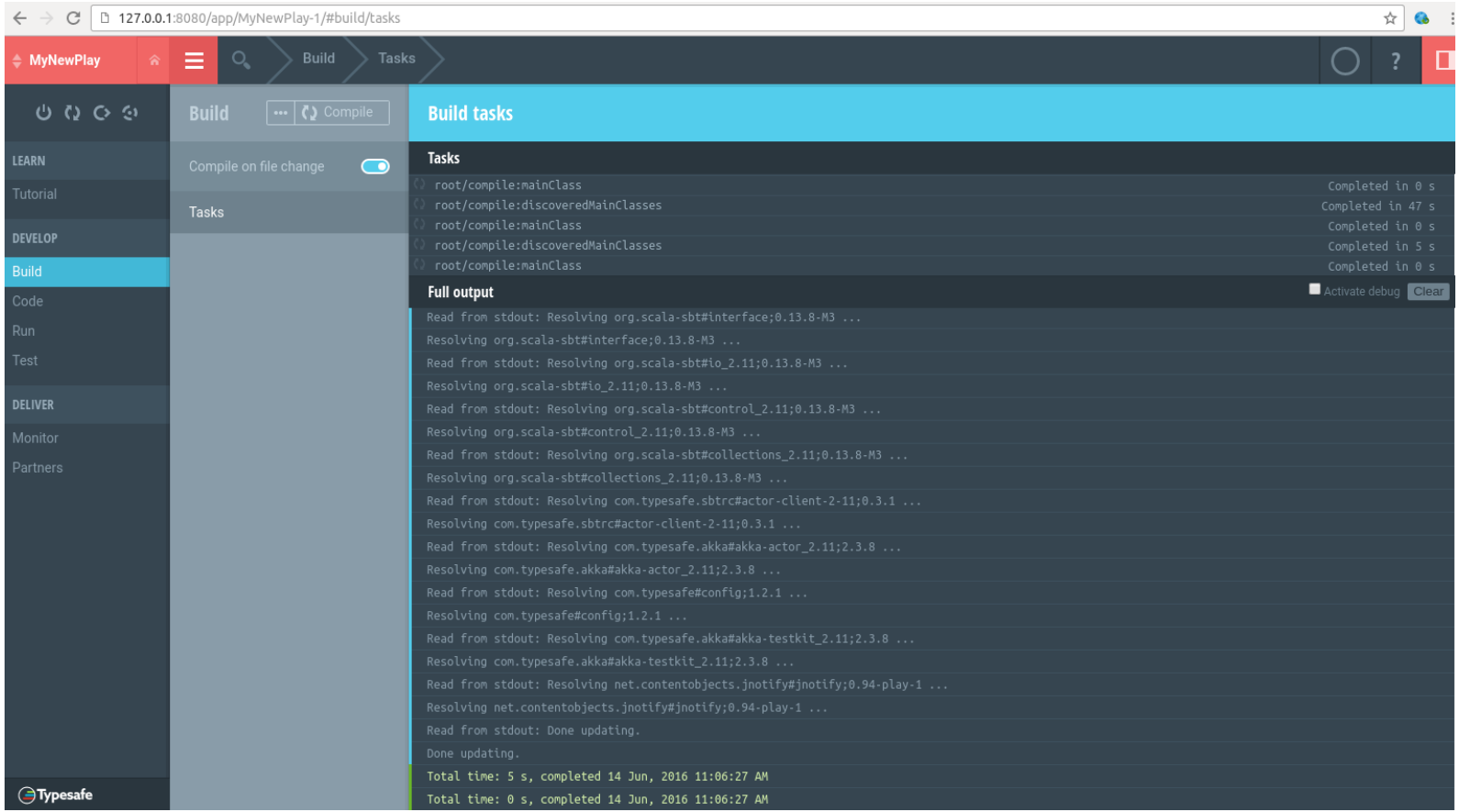
Assets

When you package your application, all assets for the application, including all sub projects, are aggregated into a single jar, in target/my-first-app-1.0.0-assets.jar. This jar is included in the distribution so that your Play application can serve them.

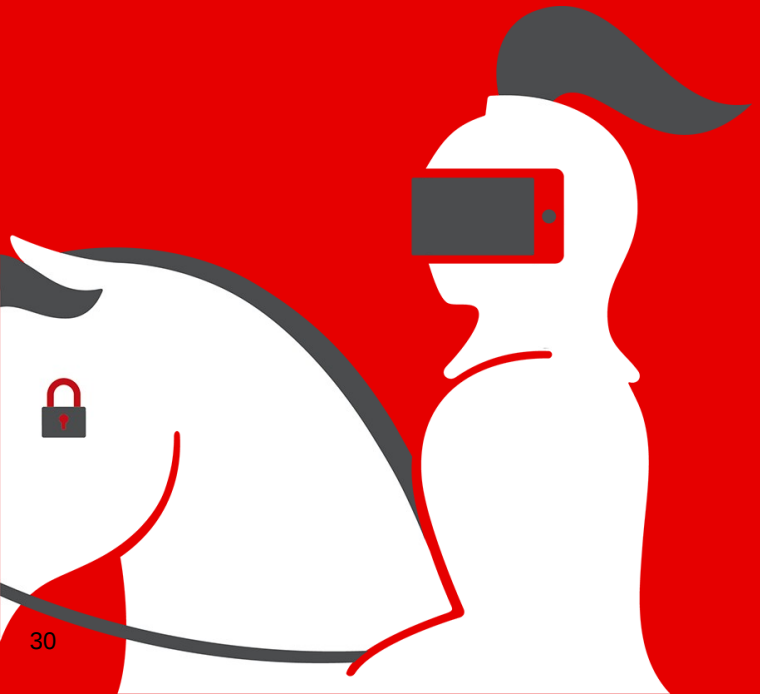


- Public Assets
- WebJars
- Assets controller
  - Etag
  - Gzip
  - Caching
- Managed Assets
  - CoffeeScript
  - LESS CSS
  - Sbt-web plugins
  - js-engine plugin (able to execute plugins written to the Node API either within the JVM via the excellent Trireme)

Activator UI



Testing

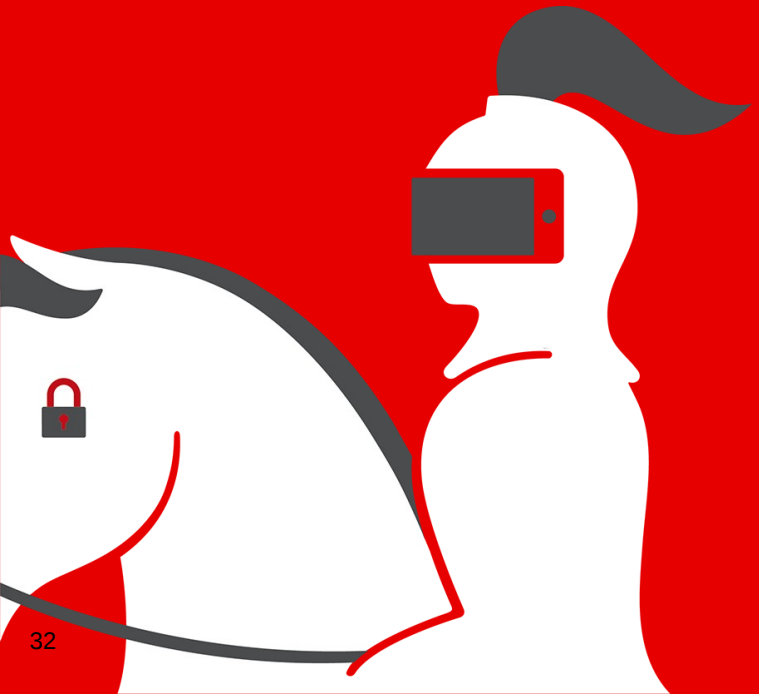


Testing

- Ready to use application stubs for test (several helper methods)
- Junit
- FEST assertions
- Mock
- Test every part of your application (router, action, controller and ...)
- Testing with a browser using FluentLenium (Selenium WebDriver wrapper)

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

Deployment



Deployment

- Compile all class and assets in application
- Activator
  - start
  - stage (without any dependency on Play)
  - dist (produces a ZIP file without any dependency on Play)
- SBT Native Packager plugin(activator universal:package-bin)
  - MSI, OS X disk image, RPM, DEB and ...
- Publishing to a Maven (or Ivy) repository

Play Main Topics

HTTP programming
Asynchronous HTTP programming
The template engine
Form submission and validation
Working with Json
Working with XML
Handling file upload
Accessing an SQL database
Using the Cache
Calling WebServices
Integrating with Akka
Internationalization
Testing your application
Logging

Play Advanced Topics

Dependency injection
HTTP architecture
Advanced routing
Extending Play

Play Pros

- **Dramatically improved developer productivity**
  - Make a change, refresh the page, see the change.
- **Reactive**
  - Play is built on Netty, so it supports non-blocking I/O. This means it's very easy and inexpensive to make remote calls in parallel, which is important for high performance apps in a service oriented architecture.
- **Open Source**
  - Play is open source. They accept pull requests if you need to change something.
- **Amazing error handling**
  - Play has beautiful error handling in dev mode: for both compile and runtime errors.
- **Flexible**
  - Just about everything in Play is pluggable, configurable, and customizable.
- **Modern stack**
  - Play is an MVC stack on top of Netty and Akka and has built-in support for modern web framework: REST, JSON/XML handling, non-blocking I/O, WebSockets, asset compilation (CoffeeScript, less), ORM, NoSQL support, and so on.
- **Java (and Scala)**
  - Use reliable, type-safe languages and leverage JVM performance to scale to many users and many developers.

Play Cons

- **New**
  - Play 2 was a total rewrite of Play 1, so not much could be carried over from the years of Play work before that. This means the community isn't as large as other Java frameworks and there aren't as many Play 2 plugins.
- **Immature**
  - The API is changing, and the best practices aren't well defined.
- **Java + Async**
  - Lots of existing Java libraries are synchronous/blocking, so you have to be careful with which ones you use in a an async/non-blocking environment like Netty.
- **Not a servlet**
  - Breaking away from the servlet spec has advantages, but lots of existing code in the Java world is built around HttpServletRequest, HttpServletResponse, etc. Play uses none of these, so you need to find other libraries or create wrappers.
- **SBT**
  - SBT is written in a way that's hard to understand for Scala experts, let alone Java devs: tons of implicits, wildcard imports, operator overloading, and an odd programming/mental model that makes it tough to follow (or google) the code.