Scala Play! Framework

Web Applications

A web application is client-server application that uses the hyper-text transfer protocol (HTTP).

- HTTP request is sent from client to server
- HTTP response is sent back to client from server
- HTTP is stateless - there is no inherent relationship betwen request/response pairs
    - We simulate sessions (related request/response pairs) by setting cookies on the client.
- Web browsers – Firefox, Chrome – are platforms for clients.
- Web servers – Apache, Tomcat, nginx – are plaforms for servers.

A particular set of web pages running in a browser that communicate with a particular set of web server applications constitutes a web application.

HTTP Protocol

HTTP request message contain a request line, headers, and a body. Each request line specifies a method. Methods we care about:

- GET - get a resource from a server running at a specified URI
- POST
- UPDATE
- DELETE

For example, if you type http://www.gatech.edu/ in your browser's address bar, or follow a hyperlink whose target is http://www.gatech.edu/, you browser will send a GET request that looks something like this:

```
GET http://www.gatech.edu/ HTTP/1.1
```

By the way, the inclusion of the access mechanism `http://` makes the URI above a URL. In gneral, though, it's a waste of mentons to distinguish between URIs and URLs.

See http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

Web Application Structure

Web applications can be arbitrarily rich, but the core functionality of most web applications is to manage resources by implementing four operations:

- Create - create a new instance of a resourece (new email message, new customer account object, etc) - maps to the HTTP POST method.
- Read - read a resource - maps to the HTTP GET method.
- Update - modify a resource - maps to the HTTP PUT method.
- Delete - delete a resource - maps to the HTTP DELETE method.

This paradigm is called “CRUD” and most web frameworks (and RESTful web services) are structured around these operations. In our sample application we’ll see a simple way to map these operations to HTTP methods

Web Application Frameworks

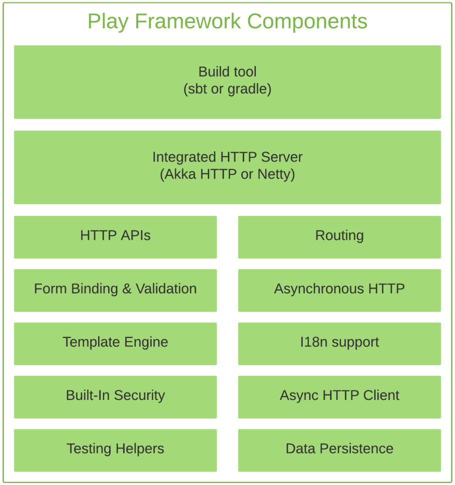
Web frameworks typically provide:

- A model-view-controller (MVC) structure
  - Models house the domain logic
  - Views house the UI elements
  - Controllers service web requests, invoking model code and forwarding to views
- Routes, which map URLs to server files or handler code
- Templates, which dynamically insert server-side data into pages of HTML
- Authentication and authorization of user names, passwords, permissions
- Sessions, which keep track of a user during a single visit to a site

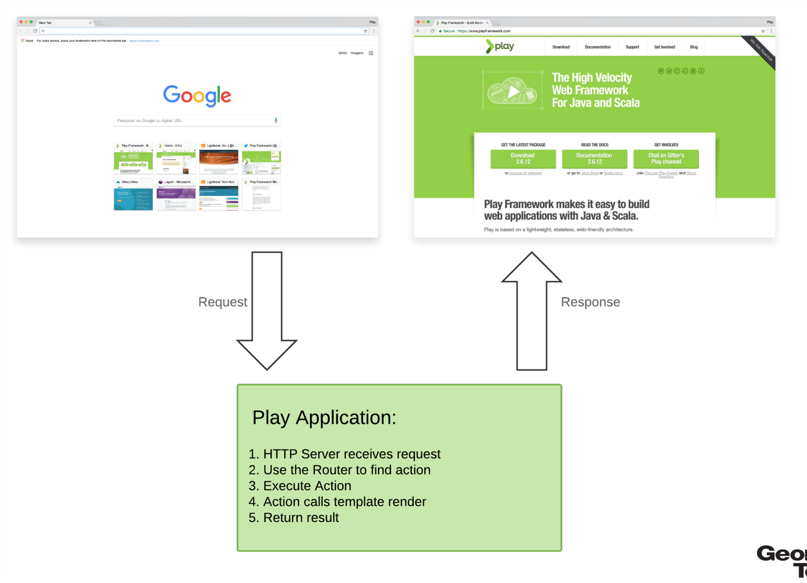
and more . . .

Play! Framework

- Play! is written primarily in Scala but has a Java API as well.
- Play! is built on Akka, making it efficient and limitlessly scalable.



Play! Application Overview



The High Velocity Web Framework For Java and Scala

Play Framework makes it easy to build web applications with Java & Scala.

Play is based on a lightweight, stateless, web-friendly architecture.

Play Application:

1. HTTP Server receives request
2. Use the Router to find action
3. Execute Action
4. Action calls template render
5. Return result

Hello, Play!

We'll create a simple web application from scratch. We'll see all the essential parts of a Play! application and how they fit together.

- Build files
- Directory structure
- A view using a Twirl template
- A controller using an Action
- A route to connect the view and the controller

This tutorial is based on Play's Hellow World Tutorial but builds the application from scratch and removes irrelevant details.

A Build Configuration for Hello, Play!

Create an empty directory called `hello-play`. This will be the project root directory.

- In the project root directory create a `build.sbt` with the following minimal contents:

```
name := """hello-play"""

version := "1.0-SNAPSHOT"

lazy val root = (project in file(".")).enablePlugins(PlayScala)

resolvers += Resolver.sonatypeRepo("snapshots")

scalaVersion := "2.12.8"

libraryDependencies += guice

scalacOptions ++= Seq(
  "-feature",
  "-deprecation",
  "-Xfatal-warnings"
)
```

A Congfiguration for the Build

In the project root directory, create a `project` directory. The `project` directory contains configuration information for the sbt build.

- In the `project` directory, create two files with the following contents:

  `build.properties`

  ```
  sbt.version=1.2.8
  ```

  `plugins.sbt`

  ```
  addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.7.0")
  ```

At this point we should have:

```

build.sbt
project
   build.properties
   plugins.sbt
```

A Layout for Views

In Play! ciews are typically implemented with Twirl templates. We'll create a view in two steps: first we'll create a layout template, then a template for rendering the hello page

- In the project root directory, create a directory named `app/views`
- In the `app/views` directory create a file called `main.scala.html` with the following contents:

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

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>@title</title>
    </head>
    <body>
        @content
    </body>
</html>
```

This template provides a shared layout. Other templates that call this template insert their content inserted into the `@content` portion

A Template for Hello

- In the `app/views` directory create a file called `hello.scala.html` with the following contents:

```
@main("Hello") {
  <section id="top">
      <div class="wrapper">
          <h1>Hello World</h1>
      </div>
  </section>
}
```

Notice that this template takes advantage of Scala's syntactic flexibility: the first argument list uses parentheses and the second argument list uses curly braces.

A Controller

In Play!, controllers consist of actions and are housed in the

- In the project root directory create a directory named `app/controllers`
- In the `app/controllers` directory create a file named `HomeController.scala` with the following contents:

```
package controllers

import javax.inject._
import play.api.mvc._

class HomeController @Inject()(cc: ControllerComponents)
                              (implicit assetsFinder: AssetsFinder)
    extends AbstractController(cc) {
  def hello = Action {
    Ok(views.html.hello())
  }
}
```

There's a lot going on here. For now consider all but the body of the class as boilerplate.

A Route

Play! routes URLs to controller actions via a routes files configuration.

- In the project root directory create a directory named `conf`
- In the `conf` directory create a file named `routes` with the following contents:

```
GET    /hello    controllers.HomeController.hello
```

One last thing. Create an empty file at `conf/application.conf`. Play! won't run if it's not there.

Now you can run your application with sbt:

```
$ sbt run
```

and see the view in your browser at http://localhost:9000/hello