Microservices

Webservices with Scala

Content

1. Introduction to Scala

2. Introduction to Play Framework 2

First things first

- You can do all our exercises with Java, but the solution will be handed out as Scala code
- The primary goal is not to be a scala developer
- BUT you should hear and know the concepts of Scala

Scala

What is Scalable language (Scala) and why?

- First release in 2003 after Martin Odersky decided to make a better Java
- Scala is a JVM language like Kotlin, Clojure and JRuby etc..
- Bytecode is an instruction set of one-byte opcodes executed by the JVM
- Statically typed
- simple lightweight syntax but not easy to to learn
- Object-oriented and functional
- Fully interoperable with Java

Why Scala?

```
public class Time {                         // Java
  private final int hours;
  private final int minutes;
  public Time(int hours, int minutes) {
    this.hours = hours;
    this.minutes = minutes;
  }
  public int getHours() {
    return hours;
  }
  public int getMinutes() {
    return minutes;
  }
}

case class Time(val hours: Int, val minutes: Int) // Scala
```

Immutable & Mutable Values

```
Welcome to the Ammonite Repl 1.2.1
(Scala 2.12.6 Java 1.8.0_181)
If you like Ammonite, please support our development at www.patreon.com/lihaoyi
@ val msg = "Hello, world!"
msg: String = "Hello, world!"

@ msg = "New value!"
cmd1.sc:1: reassignment to val
val res1 = msg = "New value!"
               ^
Compilation Failed

@ var mutableMsg = "Hello Wrld!"
mutableMsg: String = "Hello Wrld!"

@ muta
mutableMsg
@ mutableMsg = "Hello World!"            ^
```

- **An immutable value is defined with the *val* keyword.**
- **A mutable value is defined with *var* keyword.**

Type Inference

```
@ val msg = "Microservices"
msg: String = "Microservices"

@ val msg: String = "Microservices"
msg: String = "Microservices"

@ val msg: Double = "Microservices"
cmd5.sc:1: type mismatch;
 found   : String("Microservices")
 required: Double
val msg: Double = "Microservices"
                  ^
Compilation Failed
```

- Scala Compiler can infer the type
- Provide your types for public API

Scala (OO)

- Classes and traits
- Singleton objects are first-class objects
- Single Inheritance, but multiple traits can be mixed in
- If a singleton object and a class or trait share the same name, package and file, they are called <u>companions</u>

```
class World(name: String) // Can be instantiated

object World { // Creates only one single instace of a class by its Name
    val name = "Earth"
    def getNewInstance = new World(name)
} // access the string with: World.name
```

Class constructors & parameters

- Class definition is the signature
- Class body is the implementation
- Each class gets a constructor automatically

```
@ class MyClass
defined class MyClass

@ class MyClassWithPrimaryConstructor {
  println("init")
  }
defined class MyClassWithPrimaryConstructor

@ new MyClassWithPrimaryConstructor
init
res7: MyClassWithPrimaryConstructor = ammonite.$sess.cmd6$MyClassWithPrimaryConstructor@14b789f6

@ class Greeter(who: String){
  println(who)
  }
defined class Greeter

@ new Greeter("Tobias")
Tobias
res9: Greeter = ammonite.$sess.cmd8$Greeter@6ba060f3
```

Fields & Methods

- Methods are class members providing operations
- Fields are class members keeping state
- Use *var* to define a mutable field
- Scala lets you pick the right tool for the job
- Best practice: Prefer immutable objects, var only for specific use cases

```
@ class Sample(name: String){
  println(s"Init with $name")
  def method(x : Int) = x * x
  var field = 1
  }
defined class Sample
```

Infix & Postfix Operators

- Operators are methods used in operator notation
- Operator notation means omitting dots and parentheses
- Methods with one parameter can be used in infix notation
- Methods without parameters can be used in postfix notation
- In general, avoid using postfix notation

```
@ class Rational(n: Int, d: Int) {
    require( d != 0 )
    val numer: Int = n
    val denom: Int = d
    override def toString = numer + (if (denom == 1) "" else ("/"+denom))
    // default methods
    def +(that: Rational): Rational = new Rational( numer * that.denom + that.numer * denom, denom * that.denom )
  }
defined class Rational

@ val r = new Rational(2,3)
r: Rational = 2/3

@ val f = new Rational(3,4)
f: Rational = 3/4

@ r + f
res16: Rational = 17/12
```

Arguments

- Default arguments let you omit trailing arguments
- Leading arguments can be omitted by giving trailing arguments by name
- You can always give all arguments by name and you can also mix

```
@ def contact(name: String = "", firstname: String = "", mail: String = "unknown") = s"$firstname $name's mail is $mail"
defined function contact

@ contact("Jonas", "Tobias", "tobias.jonas@fh-rosenheim.de")
res18: String = "Tobias Jonas's mail is tobias.jonas@fh-rosenheim.de"

@ contact(name = "Jonas", mail = "tobias.jonas@fh-rosenheim.de")
res19: String = " Jonas's mail is tobias.jonas@fh-rosenheim.de"
```

Packages & Imports

- Packages organize non trivial code bases
- Use *import* if you dont want to use the fqn (= fully qualified name)
- Use the underscore* _* to import all members of a package
- Rename imported objects with =>
- Import multiple Classes with *{ .. }*

```
package de.innfactory.microservices.training
import de.innfactory.ms.Object //import Object
import de.innfactory.ms.commons._ //import all
import de.innfactory.ms2.{ Object => Obj }
```

Case Classes

```
@ case class Person(name: String, firstname: String)
defined class Person

@ Person("Kurfer", "Peter")
res21: Person = Person("Kurfer", "Peter") //Person.apply("Kurfer", "Peter")

// Internally Scala generates a Object with an apply method
// Works for every Object --> Syntactic Sugar
```

- **Create new instances without *new***
- **Compiler creates nice *toString*, *equals* and *hashCode* implementations**
- **Class parameters are promoted to immutable fields automatically**
- ***copy* method is automatically implemented**
- **Use case classes in pattern matching (covered a little later)**

Collections

- Scala collection library is very comprehensive
- Each collection has a companion object with an *apply* method
- Abstract, Mutable, Immutable Version available
- Type parameters are declared in square brackets
- Type arguments can be inferred or given explicitly

```
@ Vector(1,2,3)
res23: Vector[Int] = Vector(1, 2, 3)

@ Set(0, 1, "a")
res24: Set[Any] = Set(1, 2, "a")

@ Tuple2("Hello", "World") // same like ("Hello", "World") or "Hello" -> "World"
res25: (String, String) = ("Hello", "World")

@ Map(1 -> "Hello", 2 -> "World", 3 -> "!")
res28: Map[Int, String] = Map(1 -> "Hello", 2 -> "World", 3 -> "!")
```

Inheritance

- Scala supports inheritance
- Each class, except for *Any*, has <u>exactly one superclass</u>
- *Sealed* classes can only be extended within the same source file
- Use *final* to prevent a class from being extended
- Use *super* to access the superclass members
- Use *lazy* keyword to defer initialization until first usage
- Abstract classes cannot be instantiated

Traits

- JVM has only single class inheritance
- Scala introduces traits to overcome this limitation
  - Inherit from exactly one superclass
  - Mix-in multiple traits
- Use *with* to mix-in a trait

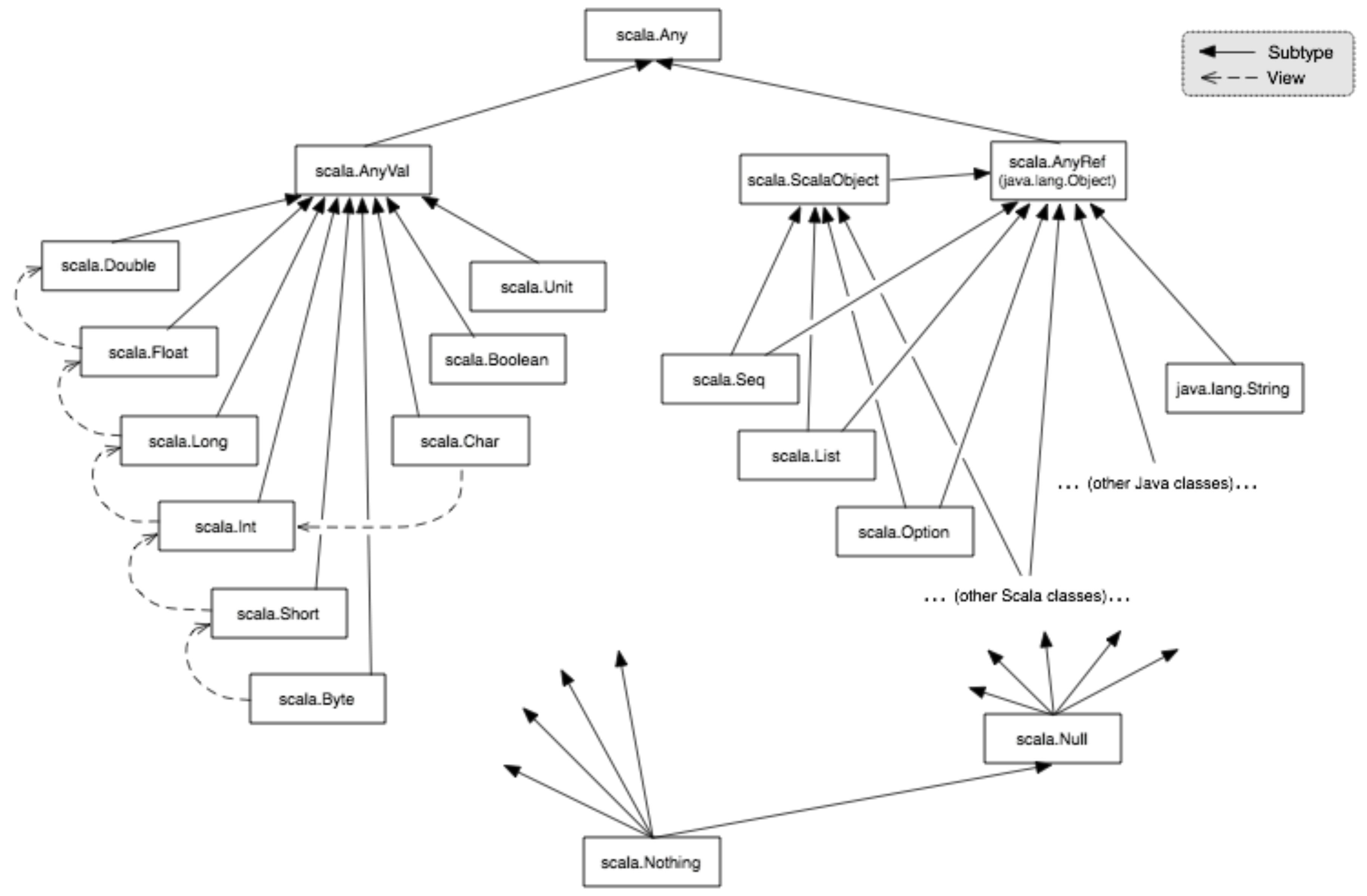
```scala
abstract class Animal

trait Flyer {
    def fly: String = "I'm flying!"
}

trait Swimmer {
  def swim: String = "I'm swimming!"
}

class Bird extends Animal with Flyer
class Fish extends Animal with Swimmer
class Duck extends Bird with Swimmer
```

Type Hierarchy



Pattern Matching

- Expressions are matched against a pattern
  - *case* declares a match pattern
  - *pattern* is one of various pattern types
  - *result* is an arbitrary expression
  - If *pattern* matches, *result* will be evaluated and returned
- Difference to the *switch case* statement of C or Java

```
@ def isAfternoon(any: Any) = any match {
      case Time(h, m) if h >=12 => s"Yes, it is $h:$m in the afternoon"
      case Time(h, m) => s"No, it is $h:$m in the morning"
      case _ => s"$any is no time!"
    }
defined function isAfternoon

@ isAfternoon(Time(13, 30))
res42: String = "Yes, it is 13:30 in the afternoon"

@ isAfternoon(Time(5, 30))
res43: String = "No, it is 5:30 in the morning"
```

Optional Values

- Optional is a better alternative to null in Java
- ADT with *Some* value or *None* as singleton object
- Handle it with Pattern Matching or higher ordered functions
- *getOrElse* extracts the wrapped value or returns a default

```
val sSome = Some("String")
val sNone = None
sSome.getOrElse("no String found")
sNone.getOrElse("no String found")

sSome match {
    case Some(value) => s"Value $value found"
    case _ => "No value found"
}
```

More concepts and knowledge we haven't covered

The following concepts are importand for a real application, but you don't need them for our Microservice samples.

Application and Main Methods, Qualified Access Modifiers, Testing and Mocking Frameworks, Collection Deep Dive, Functional Programming Basics, Higher order functions, Function literals, Function types, Important higher order functions, for Expressions and Generators, Filters, String Interpolation, Handling Failures with Try or Exceptions, Meta Programming, ...

Exercise 1

Your first steps with Scala

play

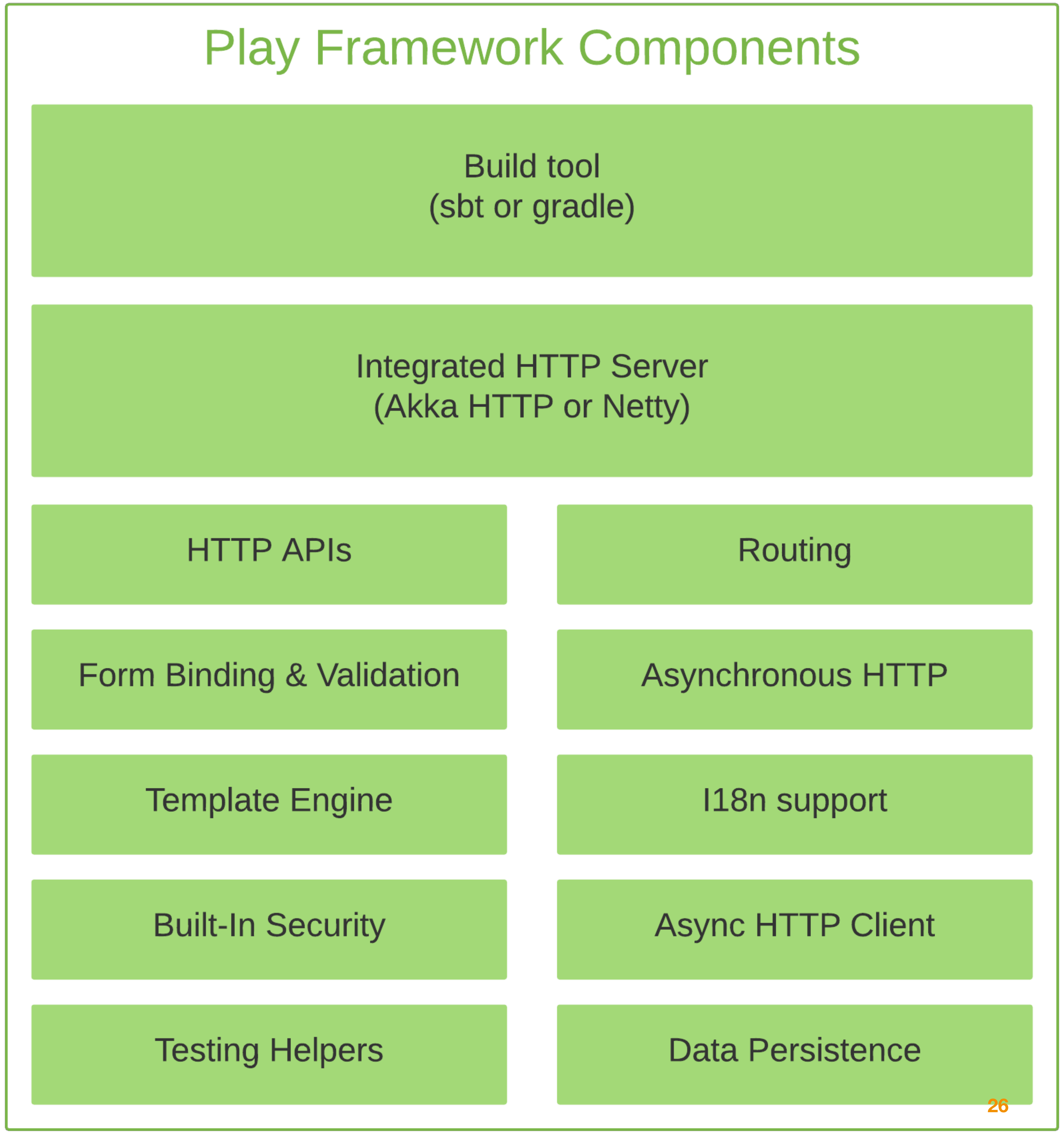
Introduction to Play Framework[1]

- Based on a lightweight, stateless, web-friendly, non blicking architecture
- Built on Akka, provides predictable and minimal resources consumption (CPU, memory, threads) for highly-scalable apllications
- You can easily call other backend services like a event sourced akka endpoint
- Lots of built in features for fast development
- Follows MVC architecture (we'll use it as plain REST Endpoints

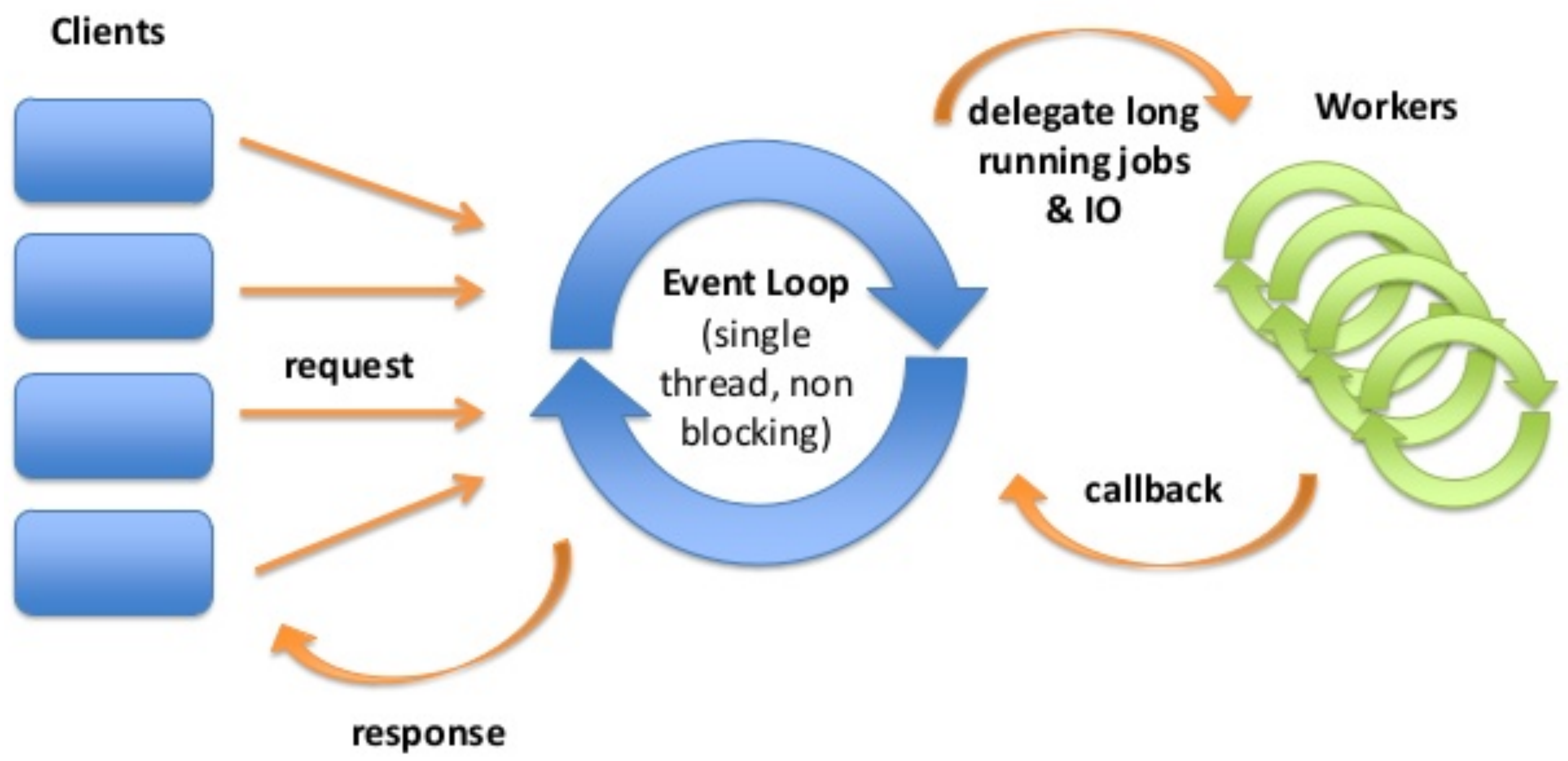
[1] Most sources are from the official documentation

Features of Play Framework

- Strong focus on productivit. Fast turnaround
- Hot reloading: Fix the bug and hit reload will recompile the change
- Type safe all the way, even templates and route files
- Use Scala or Java
- Easy to learn
- Evented Non-blocking I/O server (like node.js, akka, netty)
- Play Provides full stack
  - Websocket support
  - Template Engine for Views
  - Testing engine



Evented Server (Reactor pattern)


Clients
request
response
Event Loop (single thread, non blocking)
delegate long running jobs & IO
Workers
callback

The Play application layout

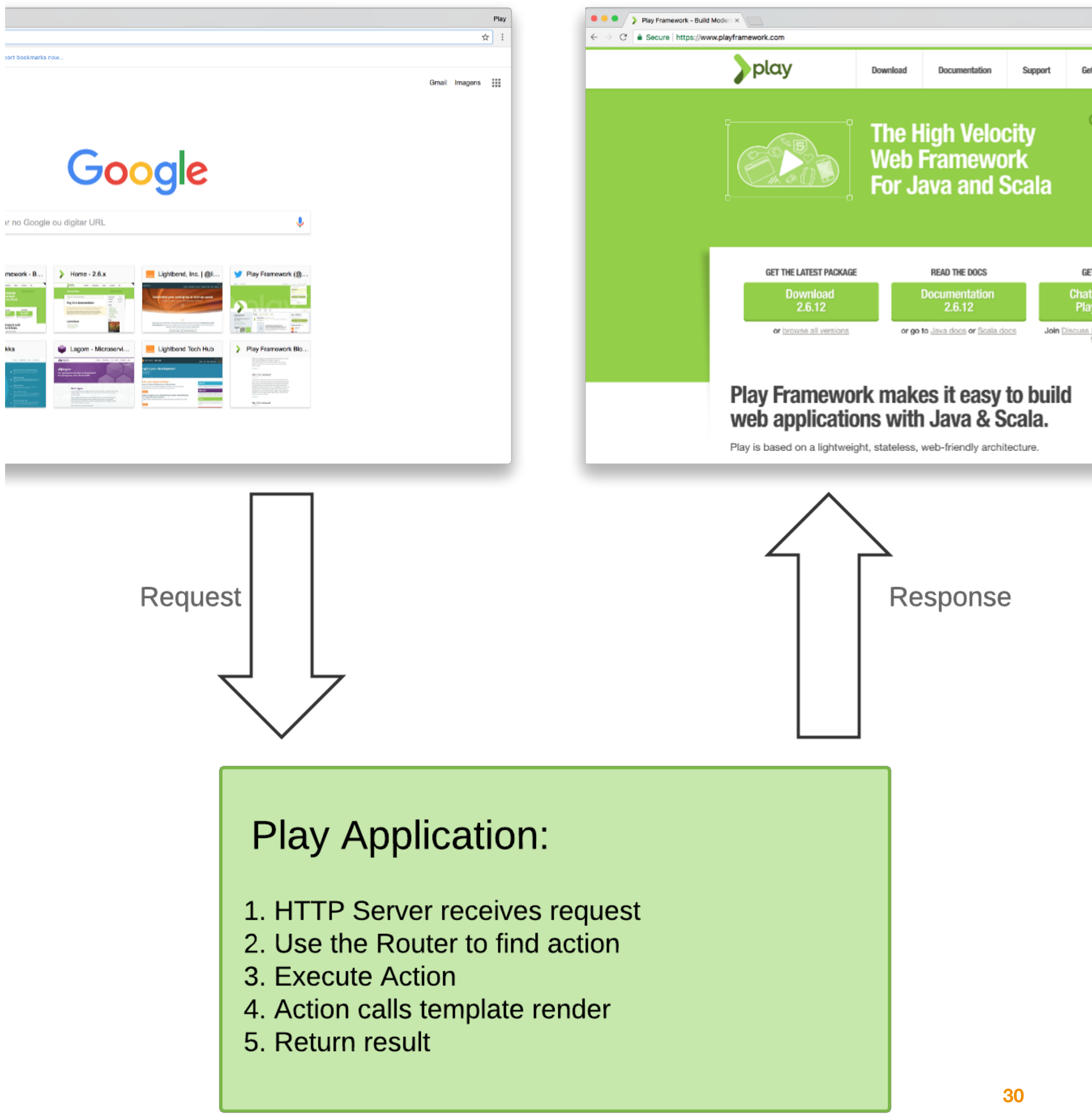
```
app                      → Application sources
 └ assets                → Compiled asset sources
    └ stylesheets        → Typically LESS CSS sources
    └ javascripts        → Typically CoffeeScript sources
 └ controllers           → Application controllers
 └ models                → Application business layer
 └ views                 → Templates
build.sbt                → Application build script
conf                     → Configurations files and other non-compiled resources (on classpath)
 └ application.conf      → Main configuration file
 └ routes                → Routes definition
dist                     → Arbitrary files to be included in your projects distribution
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                     → Logs folder
 └ application.log       → Default log file
target                   → Generated stuff
 └ resolution-cache      → Info about dependencies
 └ scala-2.11
    └ api                → Generated API docs
    └ classes            → Compiled class files
    └ routes             → Sources generated from routes
    └ twirl              → Sources generated from templates
 └ universal             → Application packaging
 └ web                   → Compiled web assets
test                     → source folder for unit or functional tests
```

Common Commands

- `sbt run` **starts your server in development mode on port 9000 - In dev mode, Play shows error messages in your browser**
- `sbt compile` **just compile your sources**
- `sbt test` **run your written tests**
- `sbt testOnly de.innfactory.play.MyClass` **run just one test**
- **You can run** `sbt shell` **to get an interactive sbt mode**
- **there is a** `help`
- `sbt dist` **create your release**

Basic Overview

1. The browser requests the root / URI from the HTTP server using the GET method.
2. The Play internal HTTP Server receives the request.
3. Play resolves the request using the routes file, which maps URIs to controller action methods.
4. The action method renders the index page, using Twirl templates.
5. The HTTP server returns the response as an HTML page.


Google
play
The High Velocity Web Framework For Java and Scala
Play Framework makes it easy to build web applications with Java & Scala.
Request
Response

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

Play Terminolgy

- Action: A Action is basically a function that handles a request and generates a result to be sent to the client.
- Controllers: A controller in Play is nothing more than an object that generates Action values. Controllers are typically defined as classes to take advantage of Dependency Injection.
- Modules Play uses public modules to augment built-in functionality

```
def echo = Action { request =>
  Ok("Got request [" + request + "]")
}
```

HTTP Routing

- The built-in HTTP router is the component in charge of translating each incoming HTTP request to an Action.
- An HTTP request is seen as an event by the MVC framework. This events contains two major pieces of information:
  - The request path (e.g. /devices/1, /actions?lastHour=true)
  - The HTTP method (e.g. GET, POST, PUT)
- URI Pattern
- Router Features also regex validation in route file

```
 //Static Path
GET /greetings            controllers.Greeter.list()
//Dynamic Parts
GET /greetings/:name          controllers.Greeter.greet(name: String)
```

Configuration

- Play uses the Typesafe config library, but Play also provides a nice Scala wrapper called Configuration with more advaed Scala features
- You can do a lot of things just with the right config (Very Good for your CI Pipeline)
- The Router mentioned is also configured by a config file

```
# Override default error handler
play.http.errorHandler = "common.errorHandling.ErrorHandler"
play.modules.enabled += "play.modules.swagger.SwaggerModule"
api.version="alpha"
api.version=${?API_VERSION}
swagger.version=2.0
swagger.api.info.title="Play2-Bootstrap"
swagger.api.info.description="Play2-Bootstrap"
```

Exercise 2

Your first steps with Play Framework[2]

[2] https://www.playframework.com/documentation/2.6.x/ScalaHome
https://www.playframework.com/documentation/2.6.x/JavaHome