Tour Of Scala

Ian Murray

October 21, 2013

Overview

- Last month, we learnt about the langauge.

Overview

- Tools
- A few useful techniques
- Common libraries and frameworks addressing specific problems

Overview

- **Simple Build Tool**
- **Cake Pattern** for compile-time dependency injection.
- **Slick** for using relational databases.
- **Spray** for writing REST APIs
- **Play** Framework for writing websites
- **Akka** for writing highly concurrent systems using actors.

Overview

- **Simple Build Tool**
- **Cake Pattern** for compile-time dependency injection.
- **Slick** for using relational databases.
- **Spray** for writing REST APIs
- **Play** Framework for writing websites
- **Akka** for writing highly concurrent systems using actors.

That's a **lot**.

Section 1

Simple Build Tool

SBT

- Scala-based configuration
- Continuous compilation and testing
- Dependency management
- Extendable via plugins
- Package & publish jars
- Generate documentation

A HelloWorld Configuration File

Contents of `build.sbt`:

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

Scala-based configuration

Contents of `project/Build.scala`:

```
import sbt._
import Keys._

object BiddingExample extends Build {

  lazy val globalSettings = Defaults.defaultSettings ++ Seq(
    scalaVersion in ThisBuild := "2.10.3"
  )

  lazy val hello = Project(id        = "hello",
                           base      = file("."))
}
```

Managed Dependencies

```
lazy val globalSettings = Defaults.defaultSettings ++ Seq(
  scalaVersion in ThisBuild := "2.10.3",
  libraryDependencies ++= Seq(
     "joda-time"       %  "joda-time"     % "2.2",
     "org.joda"        %  "joda-convert" % "1.3.1",
     "org.scalatest"  %% "scalatest"     % "1.9.2"   % "test")
)
```

Managed Dependencies

The %% adds the scala version to the artifact, ie

```
"org.scalatest"  %% "scalatest"          % "1.9.2"   % "test")
```

becomes:

```
"org.scalatest"  % "scalatest_2.10.3"    % "1.9.2"   % "test")
```

Some dependencies are compiled for different versions of Scala, this is a convenience for picking the version that matches your project.

Different versions of Scala can be binary incompatible (but maintain source compatibility). SBT can help you publish your library against multiple versions of Scala.

Managed Dependencies

Adding repositories

SBT uses the standard Maven2 Repository by default. If your dependency is not available there, then you can add further repositories:

```
lazy val globalSettings = Defaults.defaultSettings ++ Seq(
  scalaVersion in ThisBuild := "2.10.3",
  libraryDependencies ++= Seq(
     "joda-time"      %  "joda-time"    % "2.2",
     "org.joda"       %  "joda-convert" % "1.3.1",
     "org.scalatest"  %% "scalatest"    % "1.9.2"   % "test"),
  resolvers ++= Seq(
    "Spray Repo" at "http://repo.spray.io/")

)
```

Directory Structure

Sources

Follows Maven's:

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
```

Directory Structure

Build Defintion(s)

```
build.sbt
project/
  Build.scala
```

Artificats

```
target/
```

SBT Console

- REPL-like console
- execute tasks
  - `compile`
  - `run`
  - `test`
- drop into the Scala REPL
  - `console`

SBT Console

Compile

SBT will only compile source files that have changed since the previous compilation, and those sources which depend on them.

Reduces compilation time.

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.
- `test-only` can be used to selectively run particular tests

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.
- `test-only` can be used to selectively run particular tests
  - `test-only org.example.Test1 org.example.*Slow`

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.
- `test-only` can be used to selectively run particular tests
  - `test-only org.example.Test1 org.example.*Slow`
- `test-quick` will only run tests which:

SBT Console

Testing

- specs2, ScalaCheck and ScalaTest runners can all be run within SBT, just by executing the test task.
- test-only can be used to selectively run particular tests
  - `test-only org.example.Test1 org.example.*Slow`
- test-quick will only run tests which:
  - have failed in the previous run, or

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.
- `test-only` can be used to selectively run particular tests
  - `test-only org.example.Test1 org.example.*Slow`
- `test-quick` will only run tests which:
  - have failed in the previous run, or
  - were not run previously, or

SBT Console

Testing

- `specs2`, `ScalaCheck` and `ScalaTest` runners can all be run within SBT, just by executing the `test` task.
- `test-only` can be used to selectively run particular tests
  - `test-only org.example.Test1 org.example.*Slow`
- `test-quick` will only run tests which:
  - have failed in the previous run, or
  - were not run previously, or
  - tests which have had dependencies recompiled

SBT Console

Scala REPL

- Drop into the Scala console with the `console` task
- Or, if your project isn't compiling, drop into the Scala console with `console-quick`.

SBT Console

Triggered Execution

Any task can be prefixed by the ~ character. SBT will monitor source files and re-run the task

Eg.

- `~compile` will continually and incrementally re-compile your project when a source file is saved.

SBT Console

Triggered Execution

Any task can be prefixed by the ~ character. SBT will monitor source files and re-run the task

Eg.

- `~compile` will continually and incrementally re-compile your project when a source file is saved.
- `~test-quick` will continually run the parts of your test-suite that have been affacted by your code changes.

Sub-projects

Splitting projects can be useful, let's give our hello world application a REST API and a command line interface.

- `core` project, defining models, business logic etc. Acts as a library.
- `api` project, defining the REST API, which depends on `core`.
- `cli` project which also depend on `core`.
- `root` umbrealla project

Sub-projects

1. Create a `root` project which aggregates everything together:

```
lazy val root      = Project(id        = "hello",
                             base      = file("."),
                             aggregate = Seq(core, cli, api))
```

Sub-projects

2. Create a core project which defines its own dependencies:

```
lazy val core = Project(
  id         = "hello-core",
  base       = file("core"),
  settings   = globalSettings ++ Seq(
    libraryDependencies ++= Seq(
      "joda-time"       %  "joda-time"     % "2.2",
      "org.joda"        %  "joda-convert" % "1.3.1")
  ))
```

Sub-projects

3. The `cli` project has no extra dependencies other than `core`:

```
lazy val cli = Project(
  id   = "hello-cli",
  base = file("cli")) dependsOn(core)
```

Sub-projects

4. The api project requires some extra libraries:

```
lazy val api = Project(
  id        = "hello-api",
  base      = file("api"),
  settings  = globalSettings ++ Seq(
    libraryDependencies ++= {
      val sprayVersion = "1.2-M8"
      val akkaVersion  = "2.2.0-RC1" // required for 1.2-M8
      Seq(
        "io.spray"            %   "spray-can"     % sprayVersion,
        "io.spray"            %   "spray-routing" % sprayVersion,
        "io.spray"            %   "spray-testkit" % sprayVersion,
        "com.typesafe.akka"   %%  "akka-actor"    % akkaVersion)
    }
  )) dependsOn(core)
```

Sub-projects

Using sub-projects

- Each sub-project will be packaged into its own JAR.
- In the SBT console, you can work in the context of a single sub-project *or* the `root` umbrella project.

Section 2

Cake Pattern

The Problem

- Let's say we want some component for creating users and authenticating them.

The Problem

- Let's say we want some component for creating users and authenticating them.
- We want to abstract over the mechanism for persisting users.

The Problem

- Let's say we want some component for creating users and authenticating them.
- We want to abstract over the mechanism for persisting users.
- Which in turn may abstract over some things of its own.

Cake Pattern

Setting the scene

Contents of `users.scala`:

```scala
case class User(
    id: UserId,
    name: String,
    password: HashedPassword)

case class UserId(value: Long) extends AnyVal
case class HashedPassword(value: String) extends AnyVal
case class PlaintextPassword(value: String) extends AnyVal
```

Cake Pattern

Setting the scene

Contents of `users.scala`:

```scala
trait UserService {

  def create(username: String,
             password: PlaintextPassword): Try[User]

  def authenticate(username: String,
                   password: PlaintextPassword): Option[User]

}
```

Cake Pattern

Setting the scene

Contents of `dao.scala`:

```
trait UserDAO {

  def insert(username: String,
             password: HashedPassword): Try[User]

  def byUsername(username: String): Option[User]
}
```

Cake Pattern

Setting the scene

Now we can create a concrete implementation of our UserService that uses a given `UserDAO`

Cake Pattern

Setting the scene

Contents of `users.scala`:

```scala
class PersistantUserService(dao: UserDAO) extends UserService {

  def create(username: String, plaintext: PlaintextPassword) = {
    val hashed = hashPassword(plaintext)
    dao.insert(username, hashed)
  }

  def authenticate(username: String, plaintext: PlaintextPassword) = {
    for {
      user <- dao.byUsername(username)
      if passwordsMatch(plaintext, user.password)
    } yield user
  }

  private def hashPassword(plaintext: PlaintextPassword)
                          : HashedPassword = ???

  private def passwordsMatch(plaintext: PlaintextPassword,
                             hashed: HashedPassword): Boolean = ???

}
```

The Problem

- *Somewhere*, a piece of code is responsible for creating a new `PersistantUserService`.

Cake Pattern

The Problem

- *Somewhere*, a piece of code is responsible for creating a new `PersistantUserService`.
- That means it needs to provide a concrete `UserDAO` implementation as well.

The Problem

- *Somewhere*, a piece of code is responsible for creating a new `PersistantUserService`.
- That means it needs to provide a concrete `UserDAO` implementation as well.
- Which, in turn, will have its own dependencies to pass-in to the constructor.

Cake Pattern

The Problem

```
val userDao: UserDAO = new PostgresUserDAO( ... )
val userService: UserService = new PersistantUserService(userDAO)
```

Extrapolate this across a larger code-base, and we have a reason to use Spring.

Right?

Enter The Cake Pattern...

Traditional dependency injection frameworks will, at runtime, populate your instance's dependencies based upon the configuration found in an `XML` file.

What could possibly go wrong?!

Using the Cake Pattern is one way ensure that your dependencies are fulfilled **at compile time**, and does not require any extra dependencies or injection framework.

It can however appear a little warped...

Cake Pattern

Step 1

Add a layer of indirection. . .

Cake Pattern

Step 1

```
trait UserServiceModule {

  val userService: UserService

  trait UserService {
    def create(username: String,
               password: PlaintextPassword): Try[User]

    def authenticate(username: String,
                     password: PlaintextPassword): Option[User]
  }

}
```

Cake Pattern

Step 1

```
trait UserServiceModule {

  val userService: UserService

  trait UserService {
    def create(username: String,
               password: PlaintextPassword): Try[User]

    def authenticate(username: String,
                     password: PlaintextPassword): Option[User]
  }

}
```

- The `UserServiceModule` defines **what** a `UserService` is,
- . . . and **how** we can get one.

Cake Pattern

Step 2

Repeat the same for our `UserDAOModule`...

Cake Pattern

Step 2

```
trait UserDAOModule {

  val userDAO: UserDAO

  trait UserDAO {
    def insert(username: String,
               password: HashedPassword): Try[User]

    def byUsername(username: String): Option[User]
  }
}
```

Cake Pattern

Step 3

Define a concrete implmentation of our `UserDAOModule` which uses a database backend.

Cake Pattern

Step 3

```scala
trait PostgresUserDAOModule extends UserDAOModule {

  val connectionPool: ConnectionPool
  val userDAO = new Impl()

  class Impl extends UserDAO {

    // Concrete implementations...
    def insert(username: String,
               password: HashedPassword) = ???
    def byUsername(username: String) = ???
    }
  }
}
```

Cake Pattern

Step 3

```scala
trait PostgresUserDAOModule extends UserDAOModule {

  val connectionPool: ConnectionPool
  val userDAO = new Impl()

  class Impl extends UserDAO {

    // Concrete implementations...
    def insert(username: String,
               password: HashedPassword) = ???
    def byUsername(username: String) = ???
    }
  }
}
```

- The module itself **extends** `UserDaoModule`, ie it defines **how** to access a `UserDAO` instance.

Cake Pattern

Step 3

```scala
trait PostgresUserDAOModule extends UserDAOModule {

  val connectionPool: ConnectionPool
  val userDAO = new Impl()

  class Impl extends UserDAO {

    // Concrete implementations...
    def insert(username: String,
               password: HashedPassword) = ???
    def byUsername(username: String) = ???
    }
  }
}
```

- The module itself **extends** `UserDaoModule`, ie it defines **how** to access a `UserDAO` instance.
- The `Impl` class provides the actual implementation of `UserDAO`.

Cake Pattern

Step 3

```scala
trait PostgresUserDAOModule extends UserDAOModule {

  val connectionPool: ConnectionPool
  val userDAO = new Impl()

  class Impl extends UserDAO {

    // Concrete implementations...
    def insert(username: String,
               password: HashedPassword) = ???
    def byUsername(username: String) = ???
    }
  }
}
```

- The module itself **extends** `UserDaoModule`, ie it defines **how** to access a `UserDAO` instance.
- The `Impl` class provides the actual implementation of `UserDAO`.
- The module itself is still abstract, so we can defer things we don't know about, in this case, the `connectionPool`. (Although, you'd consider creating a `ConnectionPoolModule` or similar in this case).

Cake Pattern

Step 4

Define a concrete implementation of our `UserServiceModule` which delegates to a `UserDAO`.

Cake Pattern

Step 4

```
trait PersistedUserServiceModule extends UserServiceModule {
  this: UserDAOModule =>    // declare dependency on UserDAOModule

  val userService = new Impl()

  class Impl extends UserService {
    def create(username: String,
               plaintext: PlaintextPassword) = {
      val hashed = hashPassword(plaintext)
      userDAO.insert(username, hashed)
    }

    def authenticate(username: String, plaintext: PlaintextPassword) = {
      for {
        user <- userDAO.byUsername(username)
        if passwordsMatch(plaintext, user.password)
      } yield user
    }

    // Helper methods elided
  }
}
```

Cake Pattern

Self-types

```scala
trait PersistedUserServiceModule extends UserServiceModule {
  this: UserDAOModule =>    // declare dependency on UserDAOModule

  // rest of trait ...
}
```

- `this: UserDAOModule` means "When I'm mixed in to a concrete class, that concrete class must also implement the `UserDAOModule` trait.

Cake Pattern

Self-types

```scala
trait PersistedUserServiceModule extends UserServiceModule {
  this: UserDAOModule =>    // declare dependency on UserDAOModule

  // rest of trait ...
}
```

- `this: UserDAOModule` means "When I'm mixed in to a concrete class, that concrete class must also implement the `UserDAOModule` trait.
- In the Cake Pattern context, it signals a dependency.

Cake Pattern

Self-types

```scala
trait PersistedUserServiceModule extends UserServiceModule {
  this: UserDAOModule =>    // declare dependency on UserDAOModule

  // rest of trait ...
}
```

- `this: UserDAOModule` means "When I'm mixed in to a concrete class, that concrete class must also implement the `UserDAOModule` trait.
- In the Cake Pattern context, it signals a dependency.
- It's what allows us to use `userDAO` in the `Impl` class.

Cake Pattern

Self-types

Sometime inheritance is used instead, ie:

```scala
trait PersistedUserServiceModule extends UserServiceModule
                                              with UserDAOModule {
  // rest of trait ...
}
```

But self-types:

- Separate *role* from *dependencies*.
- Allow for circular references.

```scala
trait A { this: B => }
trait B { this: A => }      // compiles

trait A extends B
trait B extends A           // won't compile
```

Cake Pattern

Step 5

Pulling it all together at the end of the world... (baking the cake)

Cake Pattern

Step 5

```scala
object Modules extends PersistedUserServiceModule
                      with PostgresUserDAOModule {

  // Probably makes more sense to declare dependency on a
  // ConnectionPoolModule, but this illustrates some of the flexibility.
  val connectionPool: ConnectionPool = ???
}

Modules.userService.create("ian", PlaintextPassword("password"))
```

Cake Pattern

Step 5

```scala
object Modules extends PersistedUserServiceModule
                  with PostgresUserDAOModule {

  // Probably makes more sense to declare dependency on a
  // ConnectionPoolModule, but this illustrates some of the flexibility.
  val connectionPool: ConnectionPool = ???
}

Modules.userService.create("ian", PlaintextPassword("password"))
```

What have we achieved?

Cake Pattern

Type safety

It's impossible to forget to mix in a required ingredient, eg. this won't compile:

```
object Modules extends PersistedUserServiceModule
                  with PostgresUserDAOModule {
  // Connection pool missing.
}
```

Cake Pattern

Type safety

It's impossible to forget to mix in a required ingredient, eg. this won't compile:

```scala
object Modules extends PersistedUserServiceModule
                  with PostgresUserDAOModule {
  // Connection pool missing.
}
```

Neither will this:

```scala
object Modules extends PersistedUserServiceModule {
  val connectionPool: ConnectionPool = ???
}
```

Cake Pattern

Loosely coupled

- There's nothing that says a `PersistantUserService` must use a **particular** `UserDAO` implementation.
- There's nothing that says a `UserService` must use a `UserDAO` **at all**.

Cake Pattern

Testable

We can wire up a mock UserDAO to a PersistedServiceModule:

```
trait MockUserDAOModule extends UserDAOModule {
  val userDAO = mock[UserDAO]
}

object TestEnv extends PersistedUserServiceModule
                  with MockUserDAOModule

// ... or even

object TestEnv extends UserDAOModule
                  with PersistedUserServiceModule {
  val userDAO = mock[UserDAO]
}
```

Cake Pattern

Cons

- Can't be used at runtime to configure components.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.
- Can run into name collisions.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.
- Can run into name collisions.
- Trait inheritance can get hairy

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.
- Can run into name collisions.
- Trait inheritance can get hairy
  - Order matters when using `abstract override defs`.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.
- Can run into name collisions.
- Trait inheritance can get hairy
  - Order matters when using `abstract override def`s.
  - Instantiation order can cause `NullPointerException`s.

Cake Pattern

Cons

- Can't be used at runtime to configure components.
- Quite verbose.
  - Not only the nested definitions, but the *naming* too.
- Can run into name collisions.
- Trait inheritance can get hairy
  - Order matters when using `abstract override def`s.
  - Instantiation order can cause `NullPointerException`s.
    - Can mostly be mitigated by using `lazy val`s.

Where Next?

- "Scalable Abstract Components" by Martin Odersky

Cake Pattern

Where Next?

- "Scalable Abstract Components" by Martin Odersky
- Popularised by blog post by Jonas Boner

Where Next?

- "Scalable Abstract Components" by Martin Odersky
- Popularised by blog post by Jonas Boner
  - jonasboner.com

Where Next?

- "Scalable Abstract Components" by Martin Odersky
- Popularised by blog post by Jonas Boner
  - jonasboner.com
- "Bakery from the Black Lagoon" Talk by Daniel Spiewak

Where Next?

- "Scalable Abstract Components" by Martin Odersky
- Popularised by blog post by Jonas Boner
  - jonasboner.com
- "Bakery from the Black Lagoon" Talk by Daniel Spiewak
- Experimental macro-based implementation which removes boiler-plate.

Where Next?

- "Scalable Abstract Components" by Martin Odersky
- Popularised by blog post by Jonas Boner
  - jonasboner.com
- "Bakery from the Black Lagoon" Talk by Daniel Spiewak
- Experimental macro-based implementation which removes boiler-plate.
  - In fact, scabl.blogspot.co.uk has a very good, and more in-depth explanation of the cake pattern

Section 3

Slick

Slick: Scala Language-Integrated Collection Kit

A modern database query and access library for Scala.

Slick

- Query for data in a similar way to how you work with scala collections.

Slick

- Query for data in a similar way to how you work with scala collections.
- Write queries in Scala instead of SQL.

Slick

- Query for data in a similar way to how you work with scala collections.
- Write queries in Scala instead of SQL.
- Static checking of queries.

Slick

- Query for data in a similar way to how you work with scala collections.
- Write queries in Scala instead of SQL.
- Static checking of queries.
- Database access is kept explicit, not hidden away.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.
- Query templates.

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.
- Query templates.
- Raw SQL

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.
- Query templates.
- Raw SQL
- User defined functions.

Slick

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.
- Query templates.
- Raw SQL
- User defined functions.
- Direct embedding API.

Overview

There's a **lot** to Slick, and I'm not going to attempt to cover all of it here. This is just a taster really, and Slick would easily warrant a talk of its own.

- Querying using Slick's *Lifted Embedding* API.
- Table definitions.
- Some practicalities

I *won't* be covering:

- Aggregation.
- Examples of different joins and zipping.
- Query templates.
- Raw SQL
- User defined functions.
- Direct embedding API.
- Anything under the hood.

Slick

Querying

- Slick has a `Query[+E, U]` type which models a SQL query.

Slick

Querying

- Slick has a `Query[+E, U]` type which models a SQL query.
- `E` represents the type of the *supplier* of the data that is being queried, eg. `(Column[String], Column[Long])`.

Querying

- Slick has a `Query[+E, U]` type which models a SQL query.
- `E` represents the type of the *supplier* of the data that is being queried, eg. `(Column[String], Column[Long])`.
- `U` represents the type of the data that you get back when the query is run, eg. `(String,Long)`.

Slick

Querying

With for-comprehensions:

```
for {
  user <- Query(Users)
  if user.username.toLowerCase like "frank"
} yield user.id
```

Slick

Querying

With for-comprehensions:

```
for {
  user <- Query(Users)
  if user.username.toLowerCase like "frank"
} yield user.id
```

Or with method calls:

```
Query(Users).filter(_.username.toLowerCase like "frank")
            .map(_.id)
```

Slick

Querying

With for-comprehensions:

```
for {
  user <- Query(Users)
  if user.username.toLowerCase like "frank"
} yield user.id
```

Or with method calls:

```
Query(Users).filter(_.username.toLowerCase like "frank")
            .map(_.id)
```

- The database is not accessed in the above. It is merely a transformation of `Query` objects.

Querying

With for-comprehensions:

```
for {
  user <- Query(Users)
  if user.username.toLowerCase like "frank"
} yield user.id
```

Or with method calls:

```
Query(Users).filter(_.username.toLowerCase like "frank")
            .map(_.id)
```

- The database is not accessed in the above. It is merely a transformation of `Query` objects.
- Things like `toLowerCase` are methods made implicitly available on `Column[String]` which result in the required SQL being constructed - ie. they are not functions that act on String instances running in Scala.

Querying

Cross-joins are created when `flatMap`-ing across multiple tables/queries:

```
for {
  user1 <- Query(Users)
  user2 <- Query(Users)
  if user1.username.toLowerCase === user2.user.toLowerCase
} yield (user1.id, user2.id)
```

- Also, note the triple = equality. This is required for checking for equality. Other operators (>, >=) work as you'd expect.

Slick

Querying

The idea is that `Query`s are composable and re-usable:

```
def nameLike[E](users: Query[UsersTable,E], pattern: Column[String]) = {
  users.filter { _.username.toLowerCase like pattern }
}
```

Slick

Querying

The idea is that Querys are composable and re-usable:

```
def nameLike[E](users: Query[UsersTable,E], pattern: Column[String]) = {
  users.filter { _.username.toLowerCase like pattern }
}
```

- nameLike accepts a Query over the Users table as its first argument:

```
val q1 = Query(Users).filter(_.id > 10)
nameLike(q, "frank")
```

Slick

Querying

The idea is that `Query`s are composable and re-usable:

```
def nameLike[E](users: Query[UsersTable,E], pattern: Column[String]) = {
  users.filter { _.username.toLowerCase like pattern }
}
```

- `nameLike` accepts a `Query` over the Users table as its first argument:

```
val q1 = Query(Users).filter(_.id > 10)
nameLike(q, "frank")
```

- In this case, `"frank"` is implicitely converted from a `String` to a `Rep[String]`, but we could use an actual database column there too:

Slick

Querying

The idea is that `Query`s are composable and re-usable:

```
def nameLike[E](users: Query[UsersTable,E], pattern: Column[String]) = {
  users.filter { _.username.toLowerCase like pattern }
}
```

- `nameLike` accepts a `Query` over the Users table as its first argument:

```
val q1 = Query(Users).filter(_.id > 10)
nameLike(q, "frank")
```

- In this case, `"frank"` is implicitely converted from a `String` to a `Rep[String]`, but we could use an actual database column there too:

```
for {
  user1 <- Query(Users)
  user2 <- nameLike(Query(Users), user1.username)
} yield (user1, user2)
```

Slick

Querying

To actually execute any of these queries, access to a Session is required. The simplest way to get hold of one is:

```
// Import the slick driver for the particular database
import scala.slick.driver.H2Driver.simple._

// Instantiate a Database
val db = Database.forURL("jdbc:h2:mem:test1", driver = "org.h2.Driver")

// Have slick manage the Session for you...
db.withSession { implicit s: Session => /* Execute Query(s) */ }

// ... alternatively, within a transaction
db.withTransaction { implicit s: Session => /* Execute Query(s) */ }
```

Slick

Querying

To actually execute any of these queries, access to a Session is required. The simplest way to get hold of one is:

```
// Import the slick driver for the particular database
import scala.slick.driver.H2Driver.simple._

// Instantiate a Database
val db = Database.forURL("jdbc:h2:mem:test1", driver = "org.h2.Driver")

// Have slick manage the Session for you...
db.withSession { implicit s: Session => /* Execute Query(s) */ }

// ... alternatively, within a transaction
db.withTransaction { implicit s: Session => /* Execute Query(s) */ }
```

There are a couple of things going on here:

1. There's the usual loading of the correct JDBC driver class for the database you want to connect to.

Slick

Querying

To actually execute any of these queries, access to a Session is required. The simplest way to get hold of one is:

```scala
// Import the slick driver for the particular database
import scala.slick.driver.H2Driver.simple._

// Instantiate a Database
val db = Database.forURL("jdbc:h2:mem:test1", driver = "org.h2.Driver")

// Have slick manage the Session for you...
db.withSession { implicit s: Session => /* Execute Query(s) */ }

// ... alternatively, within a transaction
db.withTransaction { implicit s: Session => /* Execute Query(s) */ }
```

There are a couple of things going on here:

1. There's the usual loading of the correct JDBC driver class for the database you want to connect to.
2. In order for Slick to be able to generate the SQL code specific to your database engine, it needs to load it's own drivers for a particular database.

Querying

Once you have a Query and a Session, then you can execute the Query, and access the data in a number of ways:

```
val q = nameLike(Query(Users), "frank")
val qPairs = q.map { user => (user.id, user) }

db.withSession {
  // Reading a complete result set into various collections
  val franks:   List[User]          = q.list
  val franks2:  List[User]          = q.to[List]
  val frankSet: Set[User]           = q.to[Set]
  val frankMap: Map[UserId, User]   = qPairs.toMap
  val frank:    User                = q.first
  val frankO:   Option[User]        = q.firstOption
}
```

Querying

You can also consume the resultset lazily:

```
db.withSession {
  // Folding over the result
  q.foldLeft(""){_ + _.name}: String

  // Perform side-effect for row
  q.foreach(println)

  // Manually iterating over the result
  val iter: CloseableIterator[User] = q.elements
  try {
    iter.foreach(println)
  } finally { iter.close() }
}
```

Updating

It's not just about querying for data either, Querys can be used to update data too:

```
val theOnlyFrank = Query(Users).filter(_.username === "frank")
                               .map(_.username)

db.withSession { implicit s: Session =>
  theOnlyFrank.update("frank sinatra")
}
```

Updating

It's not just about querying for data either, `Query`s can be used to update data too:

```
val theOnlyFrank = Query(Users).filter(_.username === "frank")
                               .map(_.username)

db.withSession { implicit s: Session =>
  theOnlyFrank.update("frank sinatra")
}
```

There are some limitations:

- The `Query` (`theOnlyFrank`) must return raw columns selected from a single table.
- There is (currently) no way to **transform** the existing data, ie - the update method expects a single scalar value.

Deleting

And as you'd expect, *deleting* data works too:

```
val theOnlyFrank = Query(Users).filter(_.username === "frank")

db.withSession { implicit s: Session =>
  theOnlyFrank.delete
}
```

Inserting

We'll come to this shortly after we've looked at defining table definitions.

Table Definitions

Tables are declared in Scala, and can optionally be mapped to domain objects.

Slick

Table Definitions

```
// Slick driver required for table definition.
import scala.slick.driver.H2Driver.simple._

class UsersTable extends Table[(Long, String, String)]("users") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[String]("password")

  def * = id ~ username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}

val Users = new UsersTable()
```

Slick

Table Definitions

```
// Slick driver required for table definition.
import scala.slick.driver.H2Driver.simple._

class UsersTable extends Table[(Long, String, String)]("users") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[String]("password")

  def * = id ~ username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}

val Users = new UsersTable()
```

A few things to note:

1. The types of the columns don't reflect the types used in the `User` model.

Slick

Table Definitions

```
// Slick driver required for table definition.
import scala.slick.driver.H2Driver.simple._

class UsersTable extends Table[(Long, String, String)]("users") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[String]("password")

  def * = id ~ username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}

val Users = new UsersTable()
```

A few things to note:

1. The types of the columns don't reflect the types used in the `User` model.
2. We're reading `(Long,String,String)`s from the table, not `Users`.

Slick

Table Definitions

```
// Slick driver required for table definition.
import scala.slick.driver.H2Driver.simple._

class UsersTable extends Table[(Long, String, String)]("users") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[String]("password")

  def * = id ~ username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}

val Users = new UsersTable()
```

A few things to note:

1. The types of the columns don't reflect the types used in the `User` model.
2. We're reading `(Long,String,String)`s from the table, not `Users`.
3. The table definition is tied to the Slick H2 driver

Table Definitions

Improve first definition by storing `UserId`s and `HashedPassword`s:

```
implicit def userIdTypeMapper = MappedTypeMapper.base[UserId, Long] (
  userId => userId.value,
  long   => UserId(long)
)

implicit def hashedPasswordTypeMapper = {
  MappedTypeMapper.base[HashedPassword, String] (
    _.value,
    HashedPassword.apply)
}
```

Table Definitions

These implicit definitions can then be picked up and used to improve the table definition:

```
class UsersTable extends Table[(UserId, String, HashedPassword)]("users") {
  def id       = column[UserId]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[HashedPassword]("password")

  def * = id ~ username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}
```

Table Definitions

Secondly, in this situation it'd be more convenient to map the row content to an actual `User` instance. We can do this by mapping the default projection, `*`, using the `User`'s companion object's `apply` and `unapply` methods:

```
class UsersTable extends Table[User]("users") {
  def id       = column[UserId]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[HashedPassword]("password")

  def * = id ~ username ~ password <> (User, User.unapply _)

  def usernameIdx = index("idx__users__username", username, unique = true)
}
```

Table Definitions

Thirdly, we can arrange for the table definition to be independant of database driver, using the cake pattern:

Table Definitions

```
import scala.slick.driver.ExtendedProfile

trait DatabaseModule {
  val slickProfile: ExtendedProfile
  import slickProfile.simple._
  val database: Database
}
```

Slick's `ExtendedProfile` type provides `simple: SimpleQL` which in-turn provides us with the ability to define a `Table[R]`, and the type `Database`.

The basic idea is that this module can be declared as a dependency in another module which describes the user table.

And concrete implementations of the DatabaseModule can be created for the different database you wish to use.

Slick

Table Definitions

```
trait SlickUserTableModule {
  this: DatabaseModule =>        // dependency

  // The Slick database driver is required to define the UsersTable
  import slickProfile.simple._

  val Users = new UsersTable()

  trait UserTypeMappers { /** elided */ }

  class UsersTable extends Table[User]("users")
                      with UserTypeMappers {

    def id       = column[UserId]("id", O.PrimaryKey, O.AutoInc)
    def username = column[String]("username")
    def password = column[HashedPassword]("password")

    def * = id ~ username ~ password <> (User, User.unapply _)

    def usernameIdx = index("idx__users__username", username, unique = true
  }
}
```

Table Definitions

Finally, concrete implementations of the database module can be written:

```
trait PostgresDatabaseModule extends DatabaseModule {
  import scala.slick.driver.PostgresDriver.simple._
  lazy val slickProfile = scala.slick.driver.PostgresDriver
  lazy val config = ConfigFactory.load()
  lazy val database = Database.forURL(
    config.getString("database.url"),
    driver = config.getString("database.driver"),
    user = config.getString("database.user"),
    password = config.getString("database.password"))
}
```

Slick

Table Definitions

```scala
trait InMemoryDatabaseModule extends DatabaseModule {
  import scala.slick.driver.H2Driver
  lazy val slickProfile = H2Driver
  import slickProfile.simple._

  lazy val database = Database.forURL(
      s"jdbc:h2:mem:bidding-${randomDBName};DB_CLOSE_DELAY=-1",
      driver = "org.h2.Driver"
  )

  protected def randomDBName = { /** elided */ }
}
```

Implementing UserService

Re-visit the DataService to provide an implementation backed by a Slick table.

Note that we're not going to define the `UserDAO`, but the `UserService` itself.

Slick's `Querys` lend themselves to being re-used directly, so rather than composing together a `UserDAOs` functions, we can compose `Querys` together and run them all within the same `Session` or even `Transaction`.

Slick UserService

Defining the module. . .

```
trait SlickUserServiceModule extends UserServiceModule {
  this: DatabaseModule with SlickUserTableModule =>

  val userService = new Impl()

  class Impl extends UserService { /** elided */ }

}
```

- `SlickUserTableModule` gives us access to `Users`, the instance of the table object.

Slick UserService

Defining the module...

```
trait SlickUserServiceModule extends UserServiceModule {
  this: DatabaseModule with SlickUserTableModule =>

  val userService = new Impl()

  class Impl extends UserService { /** elided */ }

}
```

- `SlickUserTableModule` gives us access to `Users`, the instance of the table object.
- `DatabaseModule` gives us access to two things:

Slick UserService

Defining the module. . .

```
trait SlickUserServiceModule extends UserServiceModule {
  this: DatabaseModule with SlickUserTableModule =>

  val userService = new Impl()

  class Impl extends UserService { /** elided */ }

}
```

- `SlickUserTableModule` gives us access to `Users`, the instance of the table object.
- `DatabaseModule` gives us access to two things:
    1. The `database: Database` which we will use to obtain `Sessions`

Slick UserService

Defining the module. . .

```
trait SlickUserServiceModule extends UserServiceModule {
  this: DatabaseModule with SlickUserTableModule =>

  val userService = new Impl()

  class Impl extends UserService { /** elided */ }

}
```

- `SlickUserTableModule` gives us access to `Users`, the instance of the table object.
- `DatabaseModule` gives us access to two things:
  1. The `database: Database` which we will use to obtain `Session`s
  2. The `ExtendedProfile` instance which allows us to import the necessary functions to construct `Query`s against the database.

Slick

Slick UserService

Defining the `Impl`:

```scala
class Impl extends UserService {
  import slickProfile.simple._

  def create(username: String,
             plaintextPassword: PlaintextPassword) = { /** elided */ }

  def authenticate(username: String,
                   plaintextPassword: PlaintextPassword) = {

    val byName = Query(Users).filter(_.username === username)

    database.withSession { implicit s: Session =>
      for {
        user <- byName.firstOption
        matches = checkPassword(plaintextPassword, user.password)
        if matches
      } yield user
    }
  }
}
```

Slick

Inserting Rows

So far I've been skipping over this

Inserting Rows

```
class UsersTable extends Table[User]("users")
                    with UserTypeMappers {

  def id       = column[UserId]("id", O.PrimaryKey, O.AutoInc)
  def username = column[String]("username")
  def password = column[HashedPassword]("password")

  def * = id ~ username ~ password <> (User, User.unapply _)
  def forInsert = username ~ password

  def usernameIdx = index("idx__users__username", username, unique = true)
}
```

Inserting Rows

```
Users.forInsert returning Users.id insert(username, hashedPassword)
```

Slick UserService

```scala
class Impl extends UserService {
  def create(username: String,
             plaintextPassword: PlaintextPassword) = {
    try {
      database.withSession { implicit s: Session =>
        val hashedPassword = hashPassword(plaintextPassword)
        val userId =
          Users.forInsert returning Users.id insert (username,
                                                     hashedPassword)
        Success(User(userId, username, hashedPassword))
      }
    } catch {
      // Should really check the SQL error code
      case e: SQLException => Failure(UniqueConstraintException)
      case e: Exception    => Failure(e)
    }
  }
}
```

Where Next?

- The documentation leaves a little to be desired.
- The source code's unit tests are very useful (www.github.com/slick)
- As is the examples project, slick-examples.

Section 4

Spray

Spray: Providing a REST API

Spray is an open-source toolkit for building REST/HTTP-based integration layers on top of Scala and Akka. Being asynchronous, actor-based, fast, lightweight, modular and testable it's a great way to connect your Scala applications to the world.

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression
  - Request building

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression
  - Request building
  - Response transformation

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression
  - Request building
  - Response transformation
- spray-routing : DSL for describing REST API

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression
  - Request building
  - Response transformation
- spray-routing : DSL for describing REST API
- spray-servlet : Adapter layer for running spray-can server on the Servlet API

Spray

Spray has many components and is highly modularized:

- spray-io : low-level network IO connecting Akka actors to asynchronous Java NIO sockets.
  - Now being migrated into Akka itself.
- spray-can : Low-level HTTP server and client built on top of spray-io
- spray-client : High-level HTTP client built on top of spray-can
- spray-http : Fully immutable, case-class based model of the major HTTP data structures
- spray-httpx : Higher level logic for working with HTTP messages
  - Marshalling/un-marshalling data
  - (De-)compression
  - Request building
  - Response transformation
- spray-routing : DSL for describing REST API
- spray-servlet : Adapter layer for running spray-can server on the Servlet API
- spray-testkit : DSL for testing routing logic

Spray

Overview

- Routing

Spray

Overview

- Routing
- Marshalling

Spray

Overview

- Routing
- Marshalling
- Running on spray-can HTTP server

Routing

Spray-can provides an actor-level interface which allows your application to respond to incoming HttpRequests by replying with a HttpResponse instance:

```
import spray.http._
import HttpMethods._

class MyHttpService extends Actor {
  def receive = {
    case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
      sender ! HttpResponse(entity = "PONG")
  }
}
```

Routing

Alternatively, `spray-routing` provides a DSL for describing the structure of the REST API using composable elements called `Directives`. The above example would be written as:

```
import spray.routing._

class MyHttpService extends HttpServiceActor {
  def receive = runRoute {
    path("ping") {
      get {
        complete("PONG")
      }
    }
  }
}
```

The `runRoute` function constructs a partial function from the structure you've described in the DSL and applies that to each `HttpRequest`.

Routing

The routing DSL is made available through the `HttpService` trait, which has one abstract member:

```
def actorRefFactory: ActorRefFactory
```

In the previous example we mixed in `HttpServiceActor` which already defines the Akka actor context required.

Spray

Routes

All structures you build with the routing DSL are subtypes of type `Route`:

```scala
type Route = RequestContext => Unit
```

Routes

All structures you build with the routing DSL are subtypes of type `Route`:

```
type Route = RequestContext => Unit
```

Note that the return type is `Unit`, **not** `HttpResponse`!

Instead, a `RequestContext` is *completed*:

```
ctx.complete(httpResponse)
```

Which the `ctx` then sends asyncronously to an Akka actor which takes care of writing the response to the socket. This allows for chunked responses that are only available incrementally.

Routing

So, since a `Route` is just an ordinary function, it's simplest form looks like this:

```
ctx => ctx.complete("Response")
```

Alternatively:

```
_.complete("Response")
```

Or using the `complete` directive:

```
complete("Response")
```

Routing

A `RequestContext` can also be *rejected*:

```
ctx => ctx.reject(...)
```

Which means that this particular route does not want to handle this particular request.

Routing

A `RequestContext` can also be *rejected*:

```
ctx => ctx.reject(...)
```

Which means that this particular route does not want to handle this particular request.

Or *ignored*:

```
ctx => { /** neither completed nor rejected */ }
```

This is generally an error. If a `RequestContext` is not acted upon (rejected or completed) then the request will eventually time out, and the user will be presented with a `500 Internal Server Error` response.

Spray

Routing

Routes are composed together to produce complex Routes from simpler building blocks:

- Route transformation: Processing is delegated to an inner-route, but in doing so, the incoming request or the outgoing response is modified in some manner.

Routing

Routes are composed together to produce complex Routes from simpler building blocks:

- Route transformation: Processing is delegated to an inner-route, but in doing so, the incoming request or the outgoing response is modified in some manner.
- Route filtering: Only letting requests satisfying some predicate pass, all others are rejected.

Routing

Routes are composed together to produce complex Routes from simpler building blocks:

- Route transformation: Processing is delegated to an inner-route, but in doing so, the incoming request or the outgoing response is modified in some manner.
- Route filtering: Only letting requests satisfying some predicate pass, all others are rejected.
- Route chaining: Tries a second `Route` if a first `Route` rejects it.

Routing

Routes are composed together to produce complex Routes from simpler building blocks:

- Route transformation: Processing is delegated to an inner-route, but in doing so, the incoming request or the outgoing response is modified in some manner.
- Route filtering: Only letting requests satisfying some predicate pass, all others are rejected.
- Route chaining: Tries a second `Route` if a first `Route` rejects it.

- Route transformation and filtering are achieved through the use of `Directives`.

Routing

Routes are composed together to produce complex Routes from simpler building blocks:

- Route transformation: Processing is delegated to an inner-route, but in doing so, the incoming request or the outgoing response is modified in some manner.
- Route filtering: Only letting requests satisfying some predicate pass, all others are rejected.
- Route chaining: Tries a second `Route` if a first `Route` rejects it.

- Route transformation and filtering are achieved through the use of `Directives`.
- Route chaining is achieved through the `~` operator.

Routing

The routing DSL essentially describes a tree:

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

A request is injected into the root of this tree, and flows down through it depth-first until some leaf of the tree completes it, *or* it is fully rejected.

Spray

Routing

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

- Route 1 will only be reached if directives a, b and c all let the request pass through.

Spray

Routing

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

- Route 1 will only be reached if directives a, b and c all let the request pass through.
- Route 2 will run if a and b pass, c rejects and d passes.

Spray

Routing

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

- Route 1 will only be reached if directives a, b and c all let the request pass through.
- Route 2 will run if a and b pass, c rejects and d passes.
- Route 3 will run if a and b pass, but c and d reject.

Routing

Back to our first example:

```
import spray.routing._

class MyHttpService extends HttpServiceActor {
  def receive = runRoute {
    path("ping") {
      get {
        complete("PONG")
      }
    }
  }
}
```

Routing

`Directives` are the building blocks of the routing tree. They do at least one of the following:

1. **Filter** the `RequestContext` according to some logic, i.e. only pass on certain requests and reject all others.
2. **Extract values** from the `RequestContext` and make them available to its inner Route as "extractions".
3. **Complete** the request.
4. **Transform** the incoming RequestContext before passing it on to its inner Route.

Routing

Filtering directives:

```
path("users") {
  get {
    ...
  } ~
  post {
    ...
  }
}
```

`path`, `get` and `post` are all examples of **filter** directives.

Routing

Exracting values:

```
path("users" / LongNumber) { id =>
  get {
    ...
  }
}
```

`path("users" / LongNumber)` **extracts** a `Long` from the URL and provides it as the value `id` in the inner route.

Routing

The `path("users")` directive is also an example of a directive which **transforms** the incoming `RequestContext` because it extracts the suffix of the path that was not matched, and passes that in as the path to the inner route.

Routing

The `path("users")` directive is also an example of a directive which **transforms** the incoming `RequestContext` because it extracts the suffix of the path that was not matched, and passes that in as the path to the inner route.

Simmilarly, the `HttpResponse` can be altered as it filters back up the matched path. Eg.

```
path("users" / LongNumber) { id =>
  respondWithHeader(httpHeader) {
    get {
      ...
    }
  }
}
```

Routing

Some useful directives:

- `get`, `put`, `post`, `delete` match only http requests with those methods.
- Can be composed together, eg:

```
val getOrPut = get | put
val route = getOrPut { ... }
```

Routing

Some useful directives:

- `get`, `put`, `post`, `delete` match only http requests with those methods.
- Can be composed together, eg:

```
val getOrPut = get | put
val route = getOrPut { ... }
```

- `path("users")` and `pathPrefix("users")` for matching paths.
- `IntNumber`, `LongNumber`, `JavaUUID` all extract values from a single segment in a url path.

And user-defined ones can also be created.

Spray

Routing

We now have the building blocks to describe the API structure for our UserService.

```
import spray.routing._

trait UserRoutes extends HttpService {
  this: UserServiceModule =>

  val userRoutes = {
    path("users") {
      post {
        userService.create(???, ???)
        ???
      }
    } ~
    path("users" / PathElement) { username =>
      get {
        userService.find(username)
        ???
      }
    }
  }
}
```

Routing

And also, how we might use the UserService to authenticate users when accessing a protected resource:

```
import spray.routing._

trait PrivateRoutes extends HttpService {
  this: UserServiceModule =>

  /** elided */

  val privateRoutes = path("protected") {
    authenticate(BasicAuth(userAthenticator, "realm name")) { user =>
      complete("Secret!")
    }
  }

}
```

Spray

Routing

Pulling these two route definitions together and serving them using spray-can:

```
class ApiActor extends Actor
                with AllRoutes
                with AllServices
                with AllSlickTables
                with InMemoryDatabaseModule

  // the HttpService trait defines only one abstract member, which
  // connects the services environment to the enclosing actor or test
  def actorRefFactory = context

  // this actor only runs our route, but you could add
  // other things here, like request stream processing,
  // timeout handling or alternative handler registration
  def receive = runRoute(allRoutes)

  lazy val allRoutes = userRoutes ~ privateRoutes
}
```

Spray

Routing

```
object Boot extends App {

  // we need an ActorSystem to host our application in
  implicit val system = ActorSystem("on-spray-can")

  // create and start our service actor
  val service = system.actorOf(Props[ApiActor], "hello-service")

  // start a new HTTP server on port 8080 with our service actor as the hand
  IO(Http) ! Http.Bind(service, "localhost", port = 8080)
}
```

Marshalling and Unmarshalling

Conversion of Scala objects to/from `Array[Byte]` for transmission.

- Just going to cover very basic marshalling to/from JSON.

Marshalling

Marshalling in Spray is the conversion of a type `T` to a `HttpEntity`. It is performed by a Marshaller:

```
trait Marshaller[T] {
  def apply(value: T, ctx: MarshallingContext): Unit
}
```

Marshalling

Marshalling in Spray is the conversion of a type `T` to a `HttpEntity`. It is performed by a Marshaller:

```
trait Marshaller[T] {
  def apply(value: T, ctx: MarshallingContext): Unit
}
```

Again, the type of the `apply` function is perhaps not expected. The Spray documentation cites 3 reasons for this:

1. Marshalling must support content negotiation. The claim is that this is easier if the `Marshaller` drives the process.
2. Using the side-effectful style, the `Marshaller` can delay their action and complete the marshalling from another thread (for example, if the result of a `Future` arrives).
3. Marshallers can produce content in chunks.

Marshalling

The marshalling in Spray uses type-classes. That is, to marshall a type `T`, the compiler must be able to find a `Marshaller[T]` in implicit scope.

Marshalling

There are a whole host of default marshallers available in the `Marshaller` companion object.

```
Array[Byte]
Array[Char]
String
NodeSeq
Throwable
spray.http.FormData
spray.http.HttpEntity

Option[T]
Either[A, B]
Try[T]
Future[T]
Stream[T]
```

(Details of the Scala implicit scope resolution mean that because they are defined on the `Marshaller` companion object, they are always available, but are still overridable by bringing your own `Marshallers` into *local* scope).

JSON Marshalling

Spray has it's own JSON library, `spray-json`. And also `Marshallers` for each of the JSON types defined in that library.

JSON Marshalling

spray-json's JSON support is also via type-classes.

Converting an instance of type T to a spray-json JsValue is achieved using a JsonWriter[T] or a RootJsonWriter[T]:

```
trait JsonWriter[T] {
  def write(obj: T): JsValue
}

trait RootJsonWriter[T] extends JsonWriter[T]
```

- RootJsonWriter is capable of writing a JSON array or a JSON object.

JSON Marshalling

Similarly an instance of type `T` *from* a `spray-json` JsValue is achieved using a `JsonReader[T]` or a `RootJsonReader[T]`:

```
trait JsonReader[T] {
  def write(js: JsValue): T
}

trait RootJsonReader[T] extends JsonReader[T]
```

JSON Marshalling

Finally, `JsonFormat[T]` combines the two reader and writer traits:

```
trait JsonFormat[T] extends JsonReader[T] with JsonWriter[T]
```

Spray

JSON Marshalling

To convert our Scala classes to/from JSON we must create an instance of the above JsonFormat[T].

In the case of case classes, this is very simple:

```
trait JSONFormats {

  implicit val userIdFormat = jsonFormat(UserId, "id")

  implicit val userWriter = new RootJsonWriter[User] {
    def write(user: User) = JsObject(
      "username" -> user.name.toJson,
      "id"       -> user.id.toJson)
  }
}
```

- jsonFormatN can be used to construct a JsonFormat[T] when T is a case class.
  - I've shown the variant which accepts alternate field names.
- We haven't constructed a full JsonFormat[User] because we cannot construct the HashedPassword. But we have provided the read-side.

JSON Marshalling

- To tie JSON conversion into the marshalling, spray provides `spray.httpx.SprayJsonSupport`.

JSON Marshalling

- To tie JSON conversion into the marshalling, spray provides `spray.httpx.SprayJsonSupport`.
- It provides a `Marshaller[T]` and `Unmarshaller[T]` for every type `T` that an implicit `RootJsonWriter[T]` and `RootJsonReader[T]` is available for.

Spray

JSON Marshalling

Returning to the route service:

```scala
import spray.httpx.SprayJsonSupport

trait UserRoutes extends HttpService
                    with JSONFormats
                    with SprayJsonSupport {
  this: UserServiceModule =>

  val userRoutes = pathPrefix("users") {
    path("") { /** elided */ }
    } ~
    path(PathElement) { username =>
      get {
        rejectEmptyResponse {
          complete(userService.find(username))
        }
      }
    }
  }
}
```

JSON Marshalling

By defining a helper case class:

```
case class UserCreateData(username: String, password: String)
```

We can use the same methods to complete the user creation endpoint.

Spray

Json Marshalling

```
val userRoutes = pathPrefix("users") {
  path("") {
    post {
      entity(as[UserCreateData]) { data =>
        complete {
          userService.create(data.username,
                             PlaintextPassword(data.password))
                      .toOption
        }
      }
    }
  } ~
  path(PathElement) { username =>
    get {
      rejectEmptyResponse {
        complete(userService.find(username))
      }
    }
  }
}
```

Where Next?

- There's a *lot* in Spray.
- They also have some good posts on their blog (spray.io/blog)
  - One highlight is the "Magnet Pattern", which is used to overcome problems of type-erasure when overloading methods.

Section 5

Play

Play Framework

Play is based on a lightweight, stateless, web-friendly architecture. Built on Akka, Play provides predictable and minimal resource consumption (CPU, memory, threads) for highly-scalable applications.

Play Overview

- Developer Friendly

Play Overview

- Developer Friendly
  - Just hit refresh in the browser

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales
  - Stateless web tier, built on top of Akka

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales
  - Stateless web tier, built on top of Akka
- Modern web & mobile

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales
  - Stateless web tier, built on top of Akka
- Modern web & mobile
  - RESTful by default

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales
  - Stateless web tier, built on top of Akka
- Modern web & mobile
  - RESTful by default
  - Asset compiler for LESS and CoffeeScript

Play Overview

- Developer Friendly
  - Just hit refresh in the browser
  - Type Safety
- Scales
  - Stateless web tier, built on top of Akka
- Modern web & mobile
  - RESTful by default
  - Asset compiler for LESS and CoffeeScript
  - Websockets, Comet etc.

Play

Overview

We'll take a look at:

- Simple controllers

Overview

We'll take a look at:

- Simple controllers
- HTTP routing

Overview

We'll take a look at:

- Simple controllers
- HTTP routing
- Simple templating

Overview

We'll take a look at:

- Simple controllers
- HTTP routing
- Simple templating
- Form submission

Play

Overview

We won't take a look at:

- Advanced controllers techniques

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads
- Caching

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads
- Caching
- Calling web services

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads
- Caching
- Calling web services
- I18n

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads
- Caching
- Calling web services
- I18n
- Plugins

Play

Overview

We won't take a look at:

- Advanced controllers techniques
  - body parsers
  - content negotiation
  - Action composition
- Asynchronous HTTP programming
  - Handling asynchronous results
  - Comet sockets / WebScokets
  - Iteratee library
- Play's JSON library
- File uploads
- Caching
- Calling web services
- I18n
- Plugins
- You get the idea...

Setting up the sub-project.

Cast your minds back to the SBT sub-projects, let's add a web-frontend sub-project.

project/Build.scala:

```
import play.Project._

object ScalaTalkExample extends Build {

  /** As before ... */

  lazy val frontend  = play.Project("hello-frontend",
                                    path = file("frontend")
                                   ) dependsOn(core) settings(
                                     templatesImport ++= Seq(
                                      "uk.co.sprily.scalaTalk._")
                                   )
}
```

Play

Setting up the sub-project

`project/play.sbt`:

```
resolvers += "Typesafe repository" at "http://repo.typesafe.com/typesafe/re

addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.2.0")
```

Note if your Play project is standalone and doesn't need this sort of modularization, then the website has details on how to setup Play projects using the Play installer.

Play

File 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
conf                     - Configurations files and other non-compiled resource
 - application.conf      - Main configuration file
 - routes                - Routes definition
public                   - Public assets
 - stylesheets           - CSS files
 - javascripts           - Javascript files
 - images                - Image files
test                     - source folder for unit or functional tests
```

Play

Dive straight in...

That's right, we're gonna hook up our trusty UserService!

Controllers

Controllers define functions which return `Actions`.

An `Action` is a function from request to response:

```
play.api.mvc.Request => play.api.mvc.Result
```

For example:

```
package controllers

import play.api.mvc._

object UserController extends Controller {
  def detail(name: String) = Action {
    Ok(s"Got the name: ${name}")
  }

  def create = TODO
  def submit = TODO
}
```

Play

Controllers

- There **is** a mechanism for dependency injection of controllers, but it involves runtime matching of the DI'd class.
- AFAIK DI using the cake pattern isn't solved properly for Play yet.

Controllers

```scala
def detail(name: String) = Action {
  userService.find(name).map { user =>
    Ok(views.html.userDetails(user))
  }.getOrElse { NotFound("User not found") }
}
```

Routing

To hook up an Action to a URL path, you add it to `conf/routes`:

```
GET     /users/:name    controllers.UserController.detail(name)
```

It uses it's own DSL, and uses the routes table to compute reverse routing functions, which given a Controller's Action method return the URL for that Action.

Templates

Play has its own template engine, and templates are text files which contain small blocks of Scala code.

Templates are compiled down to functions, using the filename as the function name. The following file, `app/views/layout.scala.html` will be compiled down to the function `views.html.layout`.

Play

Templates

'layout.html.scala':

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>@title</title>
  </head>
  <body>
  @content
  </body>
</html>
```

Templates

This particular template (function) is called by other templates to render their content within the site's standard layout. eg.

`userDetail.html.scala`:

```
@(user: User)

@layout("User Details") {
  <h1>@user.name</h1>
  <p>@user.id.value</p>
}
```

Templates

The `@` character is used to indicate the beginning of a dynamic expression.

There is no end-block character, so only simple statements are possible. Multi-token statements are possible by marking the beginning and end with parentheses or curly brackets, eg:

```
<p>@(user.name + " " + user.name)</p>
```

Templates

Control structures exist:

```
<ul>
@for(p <- products) {
  <li>@p.name ($@p.price)</li>
}
</ul>
```

and

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

Templates

You can define re-usable blocks:

```
@display(product: Product) = {
  @product.name ($@product.price)
}
```

and call it from the same template:

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

Templates

And re-usable variables:

```
@defining(user.firstName + " " + user.lastName) { fullName =>
  <div>Hello @fullName</div>
}
```

Templates

And re-usable variables:

```
@defining(user.firstName + " " + user.lastName) { fullName =>
  <div>Hello @fullName</div>
}
```

Finally, you can import other functions into scope:

```
@import utils._
```

Forms

Play's support for defining and using forms is pretty good. The best way I've found to use them is:

1. Declare case classes for the data your trying to capture in the form.
2. Define the form to construct those classes when bound to data from the client.
3. Transform the form data into the models for your backend.

Forms

```scala
case class UserCreation(username: String, password: String)

object UserController extends Controller {

  /** Rest of controller elided */

  private val userCreationForm = Form(
    mapping(
      "name" -> nonEmptyText,
      "password" -> nonEmptyText
    )(UserCreation.apply)(UserCreation.unapply))
}
```

- Note that more elaborate validation checks are possible by defining your own validation functions at either the field-level, or the form-level.

Forms

```
def create = Action {
  val emptyForm = userCreationForm    // can optionally pre-fill it with da
  Ok(views.html.userCreation(emptyForm))
}
```

Play

Forms

```scala
def submit = Action { implicit request =>
  userCreationForm.bindFromRequest.fold(
    formErrors => BadRequest(views.html.userCreation(formErrors)),
    userData   => {
      val username = userData.username
      val password = PlaintextPassword(userData.password)
      userService
        .create(username, password)
        .map(creationRedirect)
        .recover {
          case UniqueConstraintException => duplicateUserError(userData)
          case e                         =>
            InternalServerError("Something went wrong...")
        }
        .get
    }
  )
}
```

Play

Forms

```
private def duplicateUserError(userData: UserCreation) = {
  val form = userCreationForm.fill(userData)
  val uniqueError = FormError(
                    "name",
                    s"User ${userData.username} already exists.")
  BadRequest(views.html.userCreation(
              form.copy(errors = uniqueError +: form.errors)))
}
```

This demonstrates adding extra errors to a form. Global (ie, form-level) errors can be attached in the same way.

Play

Forms

```
private def duplicateUserError(userData: UserCreation) = {
  val form = userCreationForm.fill(userData)
  val uniqueError = FormError(
                      "name",
                      s"User ${userData.username} already exists.")
  BadRequest(views.html.userCreation(
              form.copy(errors = uniqueError +: form.errors)))
}
```

This demonstrates adding extra errors to a form. Global (ie, form-level) errors can be attached in the same way.

```
private def creationRedirect(user: User) = {
  Redirect(routes.UserController.detail(user.name)).flashing {
    "success" -> s"Created new user: ${user.name}"
  }
}
```

- This demonstrates redirection (again, using reverse routing).
- And the helper `.flashing` function which adds a user message to the session for the next request only.

Forms

Finally, a default rendering of the Form is easy to achieve:

```
@(form: Form[controllers.UserCreation])

@layout("Create User") {
  @helper.form(action = controllers.routes.UserController.submit) {
    @helper.inputText(form("name"))
    @helper.inputPassword(form("password"))
    <input type="submit" value="Submit"/>
  }
}
```

- There are bootstrap @helper functions for rendering form elements as bootstrap expects them to be. Or you can define your own.

Forms

Finally, a default rendering of the `Form` is easy to achieve:

```
@(form: Form[controllers.UserCreation])

@layout("Create User") {
  @helper.form(action = controllers.routes.UserController.submit) {
    @helper.inputText(form("name"))
    @helper.inputPassword(form("password"))
    <input type="submit" value="Submit"/>
  }
}
```

- There are bootstrap @helper functions for rendering form elements as bootstrap expects them to be. Or you can define your own.
- The form's sbumission URL is constructed by reversing the routes table.

Console

Play has a very useful feature for writing web applications, which is you `run` the Play sub-project, it will start a server with auto-loading enabled. Which means any change to source files will trigger a re-compilation before the response is returned to the browser.

Where Next?

- *Huge* amount of stuff online.
- Documentation is really very good.
- The “play-iteratee” library (which doesn’t depend on Play) is quite interesting and powerful.

Section 6

Akka

Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM.

Akka

Overview

Best known for it's actor system...

- **Very** lightweight actors

Akka

Overview

Best known for it's actor system...

- **Very** lightweight actors
  - 2.5 million actors / GB heap

Overview

Best known for it's actor system...

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors

Akka

Overview

Best known for it's actor system...

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors
- Typed actors

Akka

Overview

Best known for it's actor system. . .

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors
- Typed actors
- Fault tolerance / supervision

Akka

Overview

Best known for it's actor system. . .

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors
- Typed actors
- Fault tolerance / supervision
- Load balancing

Akka

Overview

Best known for it's actor system. . .

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors
- Typed actors
- Fault tolerance / supervision
- Load balancing
- Routing

Overview

Best known for it's actor system. . .

- **Very** lightweight actors
  - 2.5 million actors / GB heap
- Remote actors
- Typed actors
- Fault tolerance / supervision
- Load balancing
- Routing
- FSMs

Overview

But also encompasses other solutions to the problems associated with concurrency, resiliency and distribution.

- Dataflow concurrency

Overview

But also encompasses other solutions to the problems associated with concurrency, resiliency and distribution.

- Dataflow concurrency
- Software Transactional Memory

Overview

But also encompasses other solutions to the problems associated with concurrency, resiliency and distribution.

- Dataflow concurrency
- Software Transactional Memory
- Agents

Overview

But also encompasses other solutions to the problems associated with concurrency, resiliency and distribution.

- Dataflow concurrency
- Software Transactional Memory
- Agents
- Transactors

Overview

But also encompasses other solutions to the problems associated with concurrency, resiliency and distribution.

- Dataflow concurrency
- Software Transactional Memory
- Agents
- Transactors
- Asynchrounous IO

Akka

Overview

Again, Akka is a huge topic which would warrant many talks devoted to it. I'll cover:

- A brief overview of the actor model.

Overview

Again, Akka is a huge topic which would warrant many talks devoted to it. I'll cover:

- A brief overview of the actor model.
- What a simple actor looks like in Akka.

Overview

Again, Akka is a huge topic which would warrant many talks devoted to it. I'll cover:

- A brief overview of the actor model.
- What a simple actor looks like in Akka.
- Akka's philosophy of "let it fail".

Overview

Again, Akka is a huge topic which would warrant many talks devoted to it. I'll cover:

- A brief overview of the actor model.
- What a simple actor looks like in Akka.
- Akka's philosophy of "let it fail".
- An example of a FSM actor

Overview

Again, Akka is a huge topic which would warrant many talks devoted to it. I'll cover:

- A brief overview of the actor model.
- What a simple actor looks like in Akka.
- Akka's philosophy of "let it fail".
- An example of a FSM actor
- No `UserServices` this time, I swear...

Akka

Actor Model

- An actor encapsulates state and behaviour.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.

- Actors should not block

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.
- Actors should not block
  - waiting on a lock

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.

- Actors should not block
  - waiting on a lock
  - waiting on IO etc.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.
- Actors should not block
  - waiting on a lock
  - waiting on IO etc.
- Actors should only pass *immutable* data between one-another.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.
- Actors should not block
  - waiting on a lock
  - waiting on IO etc.
- Actors should only pass *immutable* data between one-another.
- In Akka, actors form a hierarchy.

Akka

Actor Model

- An actor encapsulates state and behaviour.
- and communicates with other actors **solely** by placing messages in other `Actor`'s mailboxes.
- An `Actor` is guaranteed to be processing only a single message at a time.
- If I send 2 messages to another `Actor`, I know she will receive them in the order I sent them (although maybe interleaved with other incoming messages).
- Actors are designed to be very lightweight:
  - it's possible to have millions on a single machine
  - they are disposable (cheap to create for small one-off tasks)
- They are run on top of configurable executor pools.

- Actors should not block
  - waiting on a lock
  - waiting on IO etc.
- Actors should only pass *immutable* data between one-another.

- In Akka, actors form a hierarchy.
- Parent actors are responsible for supervising their children.

Simple Actor

```scala
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

`receive` has type `PartialFunction[Any, Unit]`.

Simple Actor

To create a `MyActor` we first need an `ActorSystem`. This is a hierarchical group of `Actors` with common configuration.

```
import akka.actor.ActorSystem
val system = ActorSystem("name-for-my-actor-system")
```

Simple Actor

Once we have an `ActorSystem`, we can start creating `Actors` at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: `ActorRef`, *not* `Actor`.

Simple Actor

Once we have an ActorSystem, we can start creating Actors at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: ActorRef, *not* Actor.
- an ActorRef is handle to an Actor.

Simple Actor

Once we have an `ActorSystem`, we can start creating `Actors` at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: `ActorRef`, *not* `Actor`.

- an `ActorRef` is handle to an `Actor`.
- it is immutable (can be passed in messages)

Simple Actor

Once we have an `ActorSystem`, we can start creating `Actors` at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: `ActorRef`, *not* `Actor`.

- an `ActorRef` is handle to an `Actor`.
- it is immutable (can be passed in messages)
- . . . and serializable and network-aware.

Simple Actor

Once we have an `ActorSystem`, we can start creating `Actors` at the route of it:

```scala
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: `ActorRef`, *not* `Actor`.

- an `ActorRef` is handle to an `Actor`.
- it is immutable (can be passed in messages)
- . . . and serializable and network-aware.
  - it can be send across the wire, and it will still represent the same `Actor` on the *original* machine.

Simple Actor

Once we have an ActorSystem, we can start creating Actors at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: ActorRef, *not* Actor.

- an ActorRef is handle to an Actor.
- it is immutable (can be passed in messages)
- ... and serializable and network-aware.
  - it can be send across the wire, and it will still represent the same Actor on the *original* machine.

Simple Actor

Once we have an ActorSystem, we can start creating Actors at the route of it:

```
val myActor: ActorRef = system.actorOf(Props[MyActor], "myactor")
```

- Note the return type: ActorRef, *not* Actor.

- an ActorRef is handle to an Actor.
- it is immutable (can be passed in messages)
- . . . and serializable and network-aware.
  - it can be send across the wire, and it will still represent the same Actor on the *original* machine.

```
myActor ! "test"    // prints "TEST"
```

Simple Actor

Actors beget Actors. . .

```
class ParentActor extends Actor {
   val child = context.actorOf(Props[MyActor], "my-child")
   def receive = {
     case "test" => child ! "test"
     case "TEST" => println("Don't shout at my kid!")
   }
}

val parent = system.actorOf(Props[ParentActor], "parent")

parent ! "test"     // prints "TEST"
parent ! "TEST"     // prints "Don't shout at my kid!"
```

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state

Akka

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Terminate the subordinate permanently

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Terminate the subordinate permanently
4. Escalate the failure, thereby failing itself

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Terminate the subordinate permanently
4. Escalate the failure, thereby failing itself

Simple Actor

In the previous example, the parent will be notified if the child fails (ie throws an Exception).

When a child actor detects a failure, it:

1. Suspends itself and all *it's* descendants.
2. Sends a message to its parent.

The parent then has a choice of options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Terminate the subordinate permanently
4. Escalate the failure, thereby failing itself

Note - Failure messages by-pass the standard mailbox, and therefore there are no guarantees about the order of delivery of failure messages with respect to ordinary messages.

Simple Actor

Note - Akka talks in terms of "supervisors" and "sub-ordinates", so I'll stick to that now...

Simple Actor

Who polices the police?

- There are 3 special actors

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.
- /system : the system guardian actor

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.
- /system : the system guardian actor
  - This supports the system support hierarchy.

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.
- /system : the system guardian actor
  - This supports the system support hierarchy.
  - One of the things it does is it's notified of termination of the `/user` actor hierarchy, and shuts down cleanly.

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.
- /system : the system guardian actor
  - This supports the system support hierarchy.
  - One of the things it does is it's notified of termination of the `/user` actor hierarchy, and shuts down cleanly.
- / : the root actor.

Simple Actor

Who polices the police?

- There are 3 special actors
- /user : the guardian actor
  - This is root of the actors created using `system.actorOf()`.
- /system : the system guardian actor
  - This supports the system support hierarchy.
  - One of the things it does is it's notified of termination of the `/user` actor hierarchy, and shuts down cleanly.
- / : the root actor.
  - Supervisor of the above two actors.

Simple Actor

When a failed `Actor` is restarted, a *new* instance of the `Actor` is instantiated. And the reference from the *original* `ActorRef` to the orginal `Actor` is replace with a reference to the *new* `Actor`.

So holders of the `ActorRef` will not notice a difference.

Simple Actor

There are hooks at the various stages of restart for cleaning up/setting up state etc:

```
oldInstance.preRestart()
newInstance.postRestart()
```

Simple Actor

In addition to the supervision above, an Actor may monitor any other Actor and be notified of it's `Termination`. (ie - not restarts).

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.
- An auctioneer opens the bidding.

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.
- An auctioneer opens the bidding.
- Actors bid without seeing other Actor's bids.

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.
- An auctioneer opens the bidding.
- Actors bid without seeing other Actor's bids.
- After a fixed duration of no new bids, the bidding is closed.

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.
- An auctioneer opens the bidding.
- Actors bid without seeing other Actor's bids.
- After a fixed duration of no new bids, the bidding is closed.
- The winning actor (if any) gets notified that they are successful.

FSM Actor

Akka has a DSL for constructing Finite State Machine Actors.

The goal is to represent a sealed bid:

- Something is up for auction with a reserve price.
- An auctioneer opens the bidding.
- Actors bid without seeing other Actor's bids.
- After a fixed duration of no new bids, the bidding is closed.
- The winning actor (if any) gets notified that they are successful.
- The rest get notified that they lost.

FSM Actor

The Auction's states:

```
sealed trait RcvMsg
case object OpenBidding extends RcvMsg
case class  Bid(amount: Double) extends RcvMsg
case object CloseBidding extends RcvMsg
```

And it's *internal* state:

```
case class Ledger(reserve: Double, offers: Map[ActorRef, Bid])
```

FSM Actor

The Auction handles the following events:

```
sealed trait RcvMsg
case object OpenBidding extends RcvMsg
case class  Bid(amount: Double) extends RcvMsg
case object CloseBidding extends RcvMsg
```

FSM Actor

```scala
class Auction(product: Any,
              reserve: Double,
              auctioneer: ActorRef) extends Actor
                                       with ActorLogging
                                       with FSM[DealState, Ledger] {

  startWith(Draft, Ledger(reserve, Map.empty))

  /** elided */
}
```

FSM Actor

```
when(Draft) {
  case Event(OpenBidding, ledger) =>
    if (sender == auctioneer) {
      goto(AcceptingOffers) using ledger forMax(10.seconds)
    } else {
      stay
    }
}
```

FSM Actor

```
when(AcceptingOffers) {
  case Event(b: Bid, ledger) =>
    val newLedger = ledger.copy(
      offers = ledger.offers + (sender -> b))

    stay using newLedger forMax(10.seconds)

  case Event(CloseBidding, ledger) =>
    if (sender == auctioneer) {
      stop
    } else {
      stay
    }

  case Event(StateTimeout, ledger) =>
    log.warning("Timed out")
    stop
}
```

FSM Actor

```
whenUnhandled {
  case Event(e,s) =>
    log.warning("received unhandled message {} in state {}/{}",
                  e, stateName, s)
    stay
}

onTermination {
  case StopEvent(FSM.Normal, _, ledger) => {
    winner(ledger) match {
      case None    => notifyLoss(ledger.offers.keys.toSeq, product)
      case Some(w) =>
        notifyLoss(ledger.offers.keys.filter(_ != w).toSeq, product)
        notifyWin(w, product)
    }
  }
}
```

FSM Actor

```scala
sealed trait AuctioneerCommand
case class CreateAuction(product: Any, reserve: Double) extends AuctioneerC
case class BidOn(product: Any, amount: Double) extends AuctioneerCommand
case object AuctionClosed extends AuctioneerCommand
```

Akka

FSM Actor

```
class Auctioneer extends Actor {

  private var auctions: Map[Any, ActorRef] = Map.empty

  def receive = {
    case CreateAuction(product, reserve) =>
      val auction = context.actorOf(Props(new Auction(product, reserve, sel
      auctions += (product -> auction)
      auction ! FSM.SubscribeTransitionCallBack(self)
      auction ! OpenBidding

    case AuctionClosed =>
      auctions = auctions.filterNot { case (p, a) => a == sender }

    case BidOn(product, amount) =>
      auctions.get(product) match {
        case Some(auction) => auction forward Bid(amount)
        case _             => {}
      }
  }

}
```

FSM Actor

```
auctioneer ! CreateAuction("car", 100.0)
auctioneer ! BidOn("BidOn("car", 150.0)
auctioneer ! BidOn("BidOn("car", 180.0)
```

Where Next?

- Where to start?
- All the things I mentioned at the beginning...
- There's the new IO layer for asynchronous socket programming on top of NIO
- ... and persisted actors which when restarted will replay any unsent messages
- ... useful for eventsourcing architectures (for which there *is* an akka project, "eventsourced")

Section 7

Where Next?

Where Next?

- ScalaCheck

Where Next?

- ScalaCheck
- Scalaz

Where Next?

- ScalaCheck
- Scalaz
- ScalaSTM

Where Next?

- shapeless

Where Next?

- shapeless
- scalaz

Where Next?

- shapeless
- scalaz
- machines