07 Testing 1

# Concepts

In this lecture we’re going to be talking about component architectures, dependency injection, and the Cake pattern, which is how we do Dependency Injection in Scala

**Seams**

First, let’s talk about the concept of seams.

A seams is a conceptual line that can be drawn between components in a system.

There are a couple benefits of seams.

First the components on the “other” side of a seam should be swappable. So think of a circuit board containing components such as transistors. If a transisitor goes bad, it should be possible to replace the faulty component. In software terms, we can try different versions of software components that serve different purposes, such as a production component or a test component.

Code that has explicit seams is loosely coupled, meaning a component can be changed with relative ease.

Loosely coupled code tends to be testable, precisely because we can isolate various pieces of the system, and substitute test components where appropriate.

## Dependency Injection Terms

Let’s talk about some terms that are relevant to the idea of Dependency Injection.

First, a component can be loosely defined as a glob of software in a larger system

A dependency is a relationship between one component and another

Dependency resolution is the way components find other dependent components

Wiring is a specific selection of components and a configuration of dependency relationships.

**Dependency Injection**

Dependency resolution is useful for actualizing seams.

Dependency injection is often referred to as inversion of control. It decouples the logic of the system from the selection of the components and resolution of dependencies.

# Cake

There are a variety of techniques for dependency injection. For example, in Java, a commmon dependency injection framework is Spring, and at LinkedIn we have traditionally used a flavor of Spring called lispring. More recently, we are using an inhouse framework known as Offspring.

In Scala, there are various language features that enable a unique way of doing dependency injection, using what’s known as the Cake pattern. Why is it called Cake? I believe it’s meant to convey that you can mix in ingredients into your cake. Also cakes consist of layers, which is a nice metaphor for constructing software.

Some of the key language features used in the Cake pattern are

Abstract members

Self-type annotations, and

Mixin Composition.

When we see the Cake pattern in action we’ll see how these language features are used.

# Non-Cake Example

To motivate the idea of dependency injection, let’s look at an example which hardcodes dependencies.

Let’s define a hypothetical client which fetches profile data from a repository. The client doesn’t actually connect to a database or do any network calls, but let’s suspend disbelief and pretend like it it does.

// Data structure for Profile

case class Profile(name: String)

// Client of Profile service

class ProfileClient {

def fetchProfile(id: Int): Profile = {

println(s”Fetching profile $id from database…”)

Profile(s”name\_$id”)

}

}

Now let’s write an application which uses the client class.

object ProfileApp {

// The dependency is "wired in" here

def profileClient = new ProfileClient()

// Application business logic which delegates to client object.

def retrieveProfile(id: Int) = profileClient.fetchProfile(id)

}

And let’s run the application:

ProfileApp.retrieveProfile(42)

That has the intended effect, calling the (make-believe) backend Profile Service. Now what if we wanted to write a unit test for the Profile App and substitute a fake Profile Client in place of the production version? The answer is it is not possible with this setup. This is because we are instantiating the profile client inside the application and hardwiring the instance directly into the application object. This code is tightly coupled, and thus difficult to test.

# Cake Example

To decouple the main application from the component it depends on, we need to do a few things. First, we need to abstract the components that are being used, so that we can provide alternative implementations. Second, we need to separate the logic which assembles the component from the business logic.

First, let’s abstract the profile client into an abstract method and wrap it into a trait called ProfileClientComponent. This will allow us to define specific implementations for different purposes. We also define a method called profileClient which fetches a specific implementation of the ProfileClient.

trait ProfileClientComponent {

def profileClient: ProfileClient

trait ProfileClient {

def fetchProfile(id: Int): Profile

}

}

Now, let’s rework the consumer of the component to accept varying implementations of the component. To do this, we use what’s known as a *self-type* which is a type constraint on the the concrete implementation of the trait.

trait ProfileAppComponent {

self: ProfileClientComponent =>

// Application business logic which delegates to client object.

def retrieveProfile(id: Int) = profileClient.fetchProfile(id)

}

Here we’re saying that any object which uses this trait also must mixin a subtype of the ProfileClientComponent trait. Since we know that we’ll be mixing in a ProfileClientComponent trait, we can safely refer to the profileClient method to retrieve the concrete implementation of the dependent api.

Now, let’s define a concrete implementation of the ProfileClientComponent called ProdProfileClientComponent. Here we provide a concrete definition of the api method, and also provide a concrete definition of the method which fetches the api object.

trait ProdProfileClientComponent extends ProfileClientComponent {

override def profileClient = new ProfileClient {

override def fetchProfile(id: Int) = {

println(s”Fetching real profile $id from database…”)

Profile(s”name\_$id”)

}

}

}

Now let’s assemble the entire object graph, mixing in the Production Client

object ProdProfileApp extends ProfileAppComponent with ProdProfileClientComponent

and we can use it

ProdProfileApp.retrieveProfile(42)

So far it seems as if we’ve created a lot of extra code with little benefit. The real benefit is that we can now create alternative component implementations and assemble object graphs for tailored scenarios. For example, if we want to use a fake Profile Client, we can create a subtype of ProfileClientComponent for testing:

trait FakeProfileClientComponent extends ProfileClientComponent {

override def profileClient = new ProfileClient {

override def fetchProfile(id: Int) = {

println(s”Fetching fake profile $id from database…”)

Profile(s”name\_$id”)

}

}

}

And we can assemble an object graph which uses the new fake component

object FakeProfileApp extends ProfileAppComponent with FakeProfileClientComponent

and we can use it

FakeProfileApp.retrieveProfile(42)

# Componentizing ProfileStrength Controller Clients

Let’s revisit our ProfileStrengthController and refactor it to make it more componentized and testable. There are two services used by the controller: the DustFizzy service and the ProfileStrength service. Let’s turn each one of those into a Cake component.

## DustFizzyComponent

Open up the demo1 project in IntelliJ and navigate to demo1-frontend/app/controllers/ProfileStrengthController.scala. Find the helloDust action method and look at the code pertaining to the use of the DustFizzyPlugin.

def helloDust = Action { implicit Request =>

val basePage = DustFizzyPlugin.defaultBasePage

val data = DefaultBasePageData(

clientTemplate = "templates/hellodust",

clientData = Map("name" -> "eric")

)

val javascriptRoutes: Seq[JavascriptReverseRoute] =

Seq(routes.javascript.ProfileStrengthController.index)

val options = DustOptionsScala(

javascriptRoutes = javascriptRoutes

)

DustFizzyPlugin.getInstance.result(basePage, data, options)

}

To turn this code into a component, we should abstract out the important variables. Additionally, we should try to ensure that our new component does not use types that rely on external libraries. We should strive to use native Scala types such as Seqs, Strings, Ints, etc. This will allow the component to be used independently of the external libraries, and will greatly facilitate testing and reuse.

Create a new package under demo1-frontend/app named components. Within that directory, create a file named DustFizzyComponent.scala. Now create the component which defines the client API.

package components

import com.linkedin.dust.{DustFizzyPlugin, DustOptionsScala, DefaultBasePageData}

import play.api.mvc.{RequestHeader, SimpleResult}

import play.core.Router.JavascriptReverseRoute

trait DustFizzyComponent {

def dustFizzyClient: DustFizzyClient

trait DustFizzyClient {

def renderResult(clientTemplate: String,

clientData: Map[String, Any],

javascriptRoutes: Seq[JavascriptReverseRoute])

(implicit request: RequestHeader): SimpleResult

}

}

Next, in the same file, create a “prod” implementation of the component:

trait DefaultDustFizzyComponent extends DustFizzyComponent {

override def dustFizzyClient = new DefaultDustFizzyClient() {}

lazy val dustPlugin = DustFizzyPlugin.getInstance

trait DefaultDustFizzyClient extends DustFizzyClient {

override def renderResult(clientTemplate:String, clientData: Map[String, Any],

javascriptRoutes: Seq[JavascriptReverseRoute])

(implicit request: RequestHeader): SimpleResult = {

val data = DefaultBasePageData(

clientTemplate = clientTemplate,

clientData = clientData)

val basePage = DustFizzyPlugin.defaultBasePage

val options = DustOptionsScala(javascriptRoutes = javascriptRoutes)

DustFizzyPlugin.getInstance.result(basePage, data, options)

}

}

}

Now, we can rework the ProfileStrengthController to use our DustFizzyComponent. First, let’s make the controller a trait so that we can defer dependency resolution:

trait ProfileStrengthControllerComponent extends Controller {

Next, declare the self type of the controller to be a DustFizzyComponent:

this: ProfileStrengthComponent with DustFizzyComponent =>

Now, we can rework the actions which use the DustFizzy plugin to use the component. First, let’s change helloDust:

def helloDust = Action { implicit Request =>

val clientTemplate = "templates/hellodust"

val clientData = Map("name" -> "eric")

val javascriptRoutes: Seq[JavascriptReverseRoute] =

Seq(routes.javascript.ProfileStrengthController.index)

dustFizzyClient.renderResult(clientTemplate, clientData, javascriptRoutes)

}

and renderForm:

def renderForm(templateData: Map[String, Any])

(implicit request: RequestHeader) = {

val clientTemplate = "templates/profilestrengthform"

val clientData = templateData

val javascriptRoutes: Seq[JavascriptReverseRoute] =

Seq(routes.javascript.ProfileStrengthController.processForm)

dustFizzyClient.renderResult(clientTemplate, clientData, javascriptRoutes)

}

Finally, wire up the dependencies by creating an object, at the bottom of the file:

object ProfileStrengthController extends ProfileStrengthControllerComponent

with DefaultDustFizzyComponent

Test your controller by starting play and hitting

<http://localhost:9000/demo1/hellodust>

and

<http://localhost:9000/demo1/showprofilestrengthform>

## ProfileStrengthComponent

Let’s do a similar treatment to the ProfileStrengthClient.

Open up the demo1 project in IntelliJ and create a new file in demo1-frontend/app/components/ProfileStrengthComponent.scala.

Create the ProfileStrength component:

trait ProfileStrengthComponent {

def profileStrengthClient: ProfileStrengthClient

trait ProfileStrengthClient {

def fetchProfileStrength(memberId: Long)(implicit request: RequestHeader): Long

}

}

and create a concrete implementation:

trait RestliProfileStrengthComponent extends ProfileStrengthComponent {

override def profileStrengthClient = new RestliProfileStrengthClient() {}

lazy val restliPlugin = RestliPlugin.getInstance

trait RestliProfileStrengthClient extends ProfileStrengthClient {

override def fetchProfileStrength(memberId: Long)(implicit request: RequestHeader) :Long = {

val memberUrnString = s"urn:li:member:$memberId"

val restliRequest = new ProfileStrengthsBuilders().get().id(new Urn(memberUrnString)).build()

val futureResponse = RestliPlugin.getInstance.sendRequest(restliRequest)

val profileStrengthInfo = Await.result(futureResponse, 5 seconds).getEntity()

profileStrengthInfo.getScore()

}

}

}

now modify the controller, adding a self type to enforce the use of the component:

this: ProfileStrengthComponent with DustFizzyComponent =>

and modify restliProfileStrengthForMember:

def restliProfileStrengthForMember(memberId: Long) = Action { implicit request =>

val score = profileStrengthClient.fetchProfileStrength(memberId)

val profileStrength = new ProfileStrength().setScore(score)

Ok(JsonUtil.toJsValue(profileStrength))

}

and wire in the prod version of the component (at the bottom of the controller file):

object ProfileStrengthController extends ProfileStrengthControllerComponent

with RestliProfileStrengthComponent

with DefaultDustFizzyComponent

Test that the system works by hitting

<http://localhost:9000/demo1/restliprofilestrength30>

# Specs2

Update your ~/Documents/projects/playScala/play-scala-materials directory by cding into it and executing

git pull

The code in the section is in 07\_testing\_01. Make a project

cd 07\_testing\_01/specs2

sbt gen-idea

and open the project.

In the ~/Documents/projects/playScala/play-scala-materials/07\_testing\_1/specs2 dir, continually execute tests by typing

sbt

~test

Modify the first example in src/test/scala/com/linkedin/Ex1.scala:

Change

"Hello world" must have size(11)

to

"Hello world" must have size(10)

and back.

Add an example

After the first "in" clause, add

"end with 'world'" in {

"Hello world" must endWith("world")

}

# Specs2 Concepts

## Example groups

The "should" clause defines an example group, e.g.,

"The 'hello world' string" should

## Example

Each "in" clause defines an example with a description (left) and body (right), e.g.,

"end with 'world'" in {

"Hello world" must endWith("world")

}

## Expectations

An expression which returns a Result

## Result

### Standard

### success

"something is success" in {

success

}

### failure

non-met expectation

"a complete failure" in {

failure

}

"a failure with a message" in {

failure("What just happened??")

}

### skipped

"skip this one" in {

skipped

}

"end of the 'world' (skipped)" in {

"Hello world" must endWith("world")

skipped

}

### anError

unexpected exception

"causes an error" in {

anError

}

### pending

"haven't got to this one yet" in {

pending

}

## Matcher

A matcher is an operator used in an expectation

### beEqualTo

"be the same as itself" in {

"Hello world" must be equalTo("Hello world")

}

### contain

"match 'wo'" in {

"Hello world" must contain("wo")

}

### beMatching

"match '.\*w.r.\*'" in {

"Hello world" must beMatching(".\*w.r.\*")

}

### beLike

case class Person(name:String)

val joe = Person("joe")

"Joe" should {

"Have name joe" in {

joe must beLike { case Person("joe") => ok }

}

}

### combining matchers

"Hello world must start with 'He' and end with 'orld'" in {

"Hello world" must startWith("He") and endWith("orld")

}

### Multiple expectations/matchers

"Hello world must start with 'He' 1 and must be 1" in {

"Hello world" must startWith("He")

1 must\_== 1

}

# ScalaCheck

The ScalaCheck testing framework allows you to specify properties that must be satisfied.

import org.specs2.ScalaCheck

object ScalaCheckEx extends Specification with ScalaCheck {

"addition and multiplication are related" in prop { (a: Int) => a + a == 2 \* a }

}

# Mockito

## Mock a list

In your demo1 Play project, create a file in demo1-frontend/test/com/linkedinMockitoEx1

import org.specs2.mock.Mockito

object MockEx1 extends Specification with Mockito {

"mock examples" should {

"Java list can be mocked" in {

val m = mock[java.util.List[String]]

success

}

}

}

In a terminal, cd to the project directory and execute

play

~test

Return a stubbed value

"Return a stubbed value" in {

val m = mock[java.util.List[String]]

m.get(0) returns "one"

m.get(0) must\_== "one"

}

Try violating the expectation by modifying the test to

m.get(0) must\_== "two"

Change it back to

m.get(0) must\_== "one"

Verify a method was called

"Verify a method was called" in {

val m = mock[java.util.List[String]]

m.get(0) returns "one"

m.get(0)

there was one(m).get(0)

}

Experiment directly in play console:

play test:console

import org.specs2.mutable.\_

import org.specs2.mock.\_

object m extends Specification with Mockito

import m.\_

val l = mock[java.util.List[String]]

l.get(0) returns "one"

l.get(0) must be equalTo "one"

# 

# ScalaCheck

The ScalaCheck testing framework allows you to specify properties that must be satisfied.

Add dependencies for ScalaCheck.

In products-spec.json, in external, add

"scalaCheck": "org.scalacheck:scalacheck\_2.10:1.10.1"

In project/Build.scala, in appDependencies = Seq(..., add

"external.scalaCheck",

Quit out of the Play console and execute

play

~test

In demo1-frontend/test/com/linkedinMockitoEx1 add

import org.specs2.ScalaCheck

object ScalaCheckEx extends Specification with ScalaCheck {

"addition and multiplication are related" in prop { (a: Int) => a + a == 2 \* a }

}

ScalaCheck generates many example inputs for your property to verify that it is valid. To see how it handles an invalid case, modify the code to

"addition and multiplication are related" in prop { (a: Int) => a + a == 3 \* a }

It generates a counter example. Restore the code to it’s original correct logic.

# 

# 