Behavioural Abstractions in Play

Tobias Neef
Consultant | innoQ

ping
Play! Edition

innoQ

What's to abstract over in a Play app?

Good News

The same stuff you find everywhere

Bad News

The same stuff you find everywhere

"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

– Benjamin C. Pierce in *Types and Programming Languages* (2002)

“The typical Play app”

> app

> controllers

> models

> services

> app

> lifecycle | sub-apps

> controllers

> HTTP-workflow | error-handling | authentication | streams

> models

> services

  > code-binding | configuration

Behaviour

Structure

Abstracting over Behaviour

Actions

The base unit of abstraction in Play Controllers

a simple use case

```
def hb = Action {
  Ok("Hello Budapest")
}
```

a real use case

```
def lookupWeather =
  Action.async(parse.json) { request =>
    (request.body \\ "city")
      .headOption
      .fold
        (Future.successful(BadRequest))
        ( city => … )
  }
```

Actions abstract over

- Workflows
  - Body Parsing
    - Authentification
- Types
  - Request | _ => Result | Future[Result]
  - Actions

How to create reusable Action?

$$A \Rightarrow B \Rightarrow C\,?$$

```
def doIt = Action{ request =>
    (stuff)
    AnotherAction
    (stuff)
}
```

The best way to prepare an Action for reuse is to create an Action of Action

```
case class ReuseAction[A](action: Action[A])
 extends Action[A] {

    def apply(request: Request[A]) = {
        …
        val res = action(request)
        …
        res
    }

    lazy val parser = action.parser
}
```

```
def stuff = ReuseAction {
    Action {
      …
    }
}
```

THE action anti pattern

```
def authAction = TimerAction {
    Authenticated(loadUser, onUnauthorized)
    { user =>
       Action {
          …
       }
    }
}

def authAction2 = TimerAction
    Authenticated(loadUser, onUnauthorized)
    { user =>
       Action {
          …
       }
    }
}
```

Remember this Guy

… where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts

Created specific instead of nested Actions

Example

- Call Time Logging
- Session Parameter Extraction => Custom Request Type
- User Action

```
case class TimerAction[A](action: Action[A])
 extends Action[A] {

   def apply(request: Request[A]) = {
      val path = request.path
      val start = System.currentTimeMillis()
      val res = action(request)
      res.onComplete { a =>
        val time = System.currentTimeMillis()
                   - start
        Logger.debug(s"call to $path took
                       $time ms")
      }
      res
    }
    lazy val parser = action.parser
}
```

```
def authAction = TimerAction {
    Authenticated(loadUser, onUnauthorized)
    { user =>
        Action { request =>
            …
        }
    }
}
```

- A way to …
  - compose existing Actions to a specific Action
  - prepare a specific Request type for the user code

```
class AuthRequest[T]
  (val username: String,
   request: Request[T])
  extends WrappedRequest[T](request)
```

```
package play.api.mvc
trait Action[A] extends EssentialAction {
    …
    def apply(request : Request[A]) :
      Future[SimpleResult]
    …
}
```

How to make the types work?

> Define your own way to build an Action

> Use Play's ActionBuilder

```
object SessionUserAction extends
    ActionBuilder[AuthRequest] {

    def invokeBlock[A](request: Request[A],
          block: (AuthRequest[A]) =>
          Future[SimpleResult]) = {
        request.session.get("mail").map {
          user => block(new AuthRequest(user,
            request))
        } getOrElse
          Future.successful(Forbidden)
    }
}
```

```
def auth = SessionUserAction { request =>
    val name = request.username
    ...
    ...
}
```

How to compose Actions

```
object SessionUserAction extends
    ActionBuilder[AuthRequest] {

    …

    override def composeAction[A](action:
Action[A]) = TimerAction(action)
}
```

Everything's good now,
right?

```
def auth = SessionUserAction { request =>
    val name = request.username
    …
}

def auth2 = SessionUserAction { request =>
    val name = request.username
    …
}
```

If 80% of your Controller methods need the same behaviour you might consider a Filter

"The filter API is intended for cross cutting concerns that are applied indiscriminately to all routes."

– Play 2.2 Doc

Global login feature |
Path based security

- /admin: requires admin role
- /*: works for all users

The Filter definition

```
class RoleBasedPathAccessFilter(
    val resRole: String,
    restrictedAreas: String*)
     extends EssentialFilter {

  def this(resRole: String,
           restricedAreas: {def url:String}*) =
  {
     this(resRole, restricedAreas.map(_.url))
  }

  val restricted = restrictedAreas.toSet
  …
}
```

```
def apply(next: EssentialAction) = new
 EssentialAction {
  def apply(request: RequestHeader) = {
    val area = request.path
    if (restricted.contains(request.path) &&
        isAuthorized(area, request))
      next(request)
    else Iteratee.ignore[Array[Byte]]
           .map(_ => Results.Forbidden(…))
  }
}
```

The Filter Configuration

```
object Global extends WithFilters(
  new OpenIdAuthFilter,
  new RoleBasedPathAccessFilter
          ("ADMIN", routes.Admin.index())
)
```

Actions + Filters = Behavioural Abstraction Toolbox?

Sometimes Actions are not the best tool for the job

Content Negotiation

```
request match {
    case Accepts.Html() => Ok(views.html …)
    case Accepts.Json() => Ok(Json.toJson(…))
}
```

Parameterized

Behavioural Abstractions

> Error Translation

> Logging

Observation

The happy path varies more than the error path

Goal

Prevent duplication of error handling and logging

Tools

HOF + Error Handling Data Types

- scala.Option
- scala.Either
- scala.util.Try
- scala.concurrent.Future
- scalaz.Validation
- play.api.libs.json.JsResult

```
def upload = Action(multipartFormData)
{ r =>
  val files = request.body.files
  val loadRef: Try[…] = fileLoaderRef(files)
  fileAccess("load file", loadRef) {
    upload =>
      dbAccess(“persist new data") {
        persist(upload, request)
      }
  }
}
```

```
def fileAccess[T](task: String, try:
 Try[T])(f: T => SimpleResult) = {
    Logger.info(s"START -> $task")
    try.map(f).recover {
        case e: FileEmptyException => …
        case e: DomainException => …
        case e => error(e, task)
    }
    .transform(e => logSuccess(task, e),
                    r => logError(task, r))
    .getOrElse(InternalServerError)
}
```

ThxMuch