Play Part 1

# Overview

In this section, we’re going to create a Play app from scratch. The play app will demo a few things, including how to set up and run your app, how to create a simple controller, and how to integrate a real rest.li service.

# Create a new demo application

play new --with-li-plugins demo1

. . .

Create project as Mint multiproduct? false

application context: demo1

Kind of project: 1 (Play frontend empty project)

# Generate an IDE project (in new terminal)

cd demo1

play

gen-idea

Open IntelliJ, select Open Project, navigate to demo1-frontend/app/controllers

# Run it (in original terminal)

play run

# Hit it

In browser go to [http://localhost:9000/demo1](http://localhost:9000/demo1/scala)

We get an "Action not found" error. Too bad, but at least something is running.

# Add a new controller

Controllers are the parts of the system which process requests. Controllers

* Receive requests
* Retrieve/update models from/on the backend
* Optionally populate view templates
* Return a result to a client

In demo1-frontend/app/controllers/ProfileStrengthController.scala

package controllers

import play.api.mvc.{Action, Controller}

object ProfileStrengthController extends Controller {

def index = Action {

Ok("it works")

}

}

Let’s take a minute to have a closer look at the code and see what’s going on. Play controllers are implemented as functions which return actions.

An action is an object which wraps a function which takes a request and returns a result.

Here we’re defining a function named index which returns an action. We are calling the apply method on the Action object to return an action object. The apply method takes a single parmeter which is a function with input type Request and output type Result.

In this case, we don’t care about the request, so we don’t specify it as a parameter in Action’s function.

The body of the action consists of a single line saying OK(“it works”). Play provides a Domain Specific Language (aka DSL) for compactly specifying Results. In this case, the body of the Http response represented by the result is the string Ok, and the status code is 200. Play will also make a guess as to the content type of the HTTP response.

Action functions are wrapped inside a controller. We define an object that inherits play.api.mvc.Controller. Note that we are using an object instead of class, which emphasizes the stateless nature of controllers. Also note that the package of the controller is prefixed with play.api. Package with this prefix indicate that we’re using the Scala, rather than the Java API.

# Add a route to it in the routes file

The next step is to add a route to the routes file. The routes file maps incoming HTTP requests to controllers.

# @noPageKeyNeeded

GET / controllers.ProfileStrengthController.index

The # @noPageKeyNeeded is a hint to the system that the route is not associated with a known page key.

GET is the http method to match

/ is the path to match, after the demo1 path segment

controllers.ProfileStrengthController.index is the controller method to call.

# Visit the endpoint

Point your browser to <http://localhost:9000/demo1/>. You should see a friendly welcome message.

# Create a pegasus model

Let’s make the application more interesting by creating a model class. We’re going to model the results of the profile strength service. A member’s profile strength is a score between 0 and 100 which represents the level of the member’s profile completeness.

In data-template/src/main/pegasus/com/linkedin/profilestrength/ProfileStrength.pdsc:

{

"type": "record",

"name": "ProfileStrength",

"namespace": "com.linkedin.profilestrength",

"fields": [

{

"name": "score",

"type": "long"

}

]

}

# Force a rebuild

In your browser, refresh <http://localhost:9000/demo1/>

Note the flurry of activity in the console... it's generating/compiling the .pdsc

# View the generated file

In intellij, mark data-template/src/main/codegen as a Generated Source Root (right click the file in the project browser then select Mark As…)

Open data-template/src/main/codegen/com/linkedin/profilestrength/ProfileStrength.java

Setup the SDK if it asks

Among other things, it has getScore() and setScore() methods

# Create a new action returning a mock profileStrength

In demo1-frontend/app/controllers/ProfileStrengthController.scala, add:

def mockProfileStrength = Action {

val profileStrength = new ProfileStrength().setScore(42)

Ok(JsonUtil.toJsValue(profileStrength))

}

# Add a route in conf/routes

# @noPageKeyNeeded

GET /mockprofilestrength controllers.ProfileStrengthController.mockProfileStrength

# Hit the endpoint to get the mock JSON

visit <http://localhost:9000/demo/mockprofilestrength>.

# Add Rest.li service to build

Quit out of the running app by entering in the play console

<Ctrl-D>

exit

Add the following to your product-spec.json under in product section

"guided-edit": {

"libraries": [

"profile-strength-api",

{

"configuration": "restClient",

"key": "profile-strength-api-restClient",

"name": "profile-strength-api"

}

],

"version": "0.0.130"

},

Upgrade some out-of-date versions in product-spec.json

models -> 2.0.187

Add the following to your project/Build.scala in the appDependencies Seq

"product.guided-edit.profile-strength-api",

# Rebuild the project

play gen-idea

Back in IntelliJ, reload the project when prompted

Explore External Libraries

Search for com.linkedin.guided-edit:profile-strength-api

# Create an action that calls a Rest.li service

In app/controllers/ProfileStrength.scala, call real rest.li service to get the profile strength for member 30

At the top of the file in the imports section, add

import scala.concurrent.\_

import scala.concurrent.duration.\_

Below the index method, add

def restliProfileStrengthForMember30 = Action { implicit request =>

val restliRequest =

new ProfileStrengthsBuilders().get().id(new Urn("urn:li:member:30")).build()

val futureResponse = RestliPlugin.getInstance.sendRequest(restliRequest)

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

val profileStrength = new ProfileStrength().setScore(profileStrengthInfo.getScore())

Ok(JsonUtil.toJsValue(profileStrength))

}

# Add route

# @noPageKeyNeeded

GET /restliprofilestrength30 controllers.Foo.restliProfileStrength

Hit the rest.lified endpoint

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

# Add an action to fetch an arbitrary member

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

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()

val profileStrength = new ProfileStrength().setScore(profileStrengthInfo.getScore())

Ok(JsonUtil.toJsValue(profileStrength))

}

Add a route

# @noPageKeyNeeded

GET /profilestrength/:memberId controllers.Foo.profileStrength(memberId: Long)

We are parameterizing the path with the :memberId. Play will pick out the memberId from the URL and pass it to the profileStrength method. Since we are saying the memberId is a long, Play will enforce that type constraint.

# Retrieve profile strengths of various members

Visit <http://localhost:9000/demo1/profilestrength/30>

Visit <http://localhost:9000/demo1/profilestrength/3>

Visit <http://localhost:9000/demo1/profilestrength/1>

Visit <http://localhost:9000/demo1/profilestrength/blah>

Notice in the last instance we get a "Bad Request" error. This is Play enforcing type constraints on the path parameters.