01. MVC with Slick(ORM)

ORM JPA .

JAVA .

ORM Slick .

MVC .

Dependency

build.sbt

```
libraryDependencies += "com.h2database" % "h2" % "1.4.197"

libraryDependencies ++= Seq(
  "com.typesafe.play" %% "play-slick" % "3.0.0",
  "com.typesafe.play" %% "play-slick-evolutions" % "3.0.0"
)
```

App Setting

conf/application.conf

```
## Evolutions
# https://www.playframework.com/documentation/latest/Evolutions
# ~~~~~
# Evolutions allows database scripts to be automatically run on startup in dev mode
# for database migrations. You must enable this by adding to build.sbt:
#
# libraryDependencies += evolutions
#
play.evolutions {
  # You can disable evolutions for a specific datasource if necessary
  #db.default.enabled = false
}

## Database Connection Pool
# https://www.playframework.com/documentation/latest/SettingsJDBC
# ~~~~~
# Play doesn't require a JDBC database to run, but you can easily enable one.
#
# libraryDependencies += jdbc
#
play.db {
  # The combination of these two settings results in "db.default" as the
  # default JDBC pool:
  #config = "db"
  #default = "default"

  # Play uses HikariCP as the default connection pool.  You can override
  # settings by changing the prototype:
  prototype {
    # Sets a fixed JDBC connection pool size of 50
    #hikaricp.minimumIdle = 50
    #hikaricp.maximumPoolSize = 50
  }
}

## JDBC Datasource
# https://www.playframework.com/documentation/latest/JavaDatabase
# https://www.playframework.com/documentation/latest/ScalaDatabase
# ~~~~~
# Once JDBC datasource is set up, you can work with several different
# database options:
#
# Slick (Scala preferred option): https://www.playframework.com/documentation/latest/PlaySlick
# JPA (Java preferred option): https://playframework.com/documentation/latest/JavaJPA
# EBean: https://playframework.com/documentation/latest/JavaEbean
# Anorm: https://www.playframework.com/documentation/latest/ScalaAnorm
#
db {
  # You can declare as many datasources as you want.
  # By convention, the default datasource is named `default`

  # https://www.playframework.com/documentation/latest/Developing-with-the-H2-Database
  #default.driver = org.h2.Driver
  #default.url = "jdbc:h2:mem:play"
  #default.username = sa
  #default.password = ""

  # You can turn on SQL logging for any datasource
  # https://www.playframework.com/documentation/latest/Highlights25#Logging-SQL-statements
  #default.logSql=true
}

slick.dbs.default.driver="slick.driver.H2Driver$"
slick.dbs.default.db.profile="org.h2.Driver"
slick.dbs.default.db.url="jdbc:h2:mem:play;DB_CLOSE_DELAY=-1"
```

Router Setting

conf/routes

```
GET     /orm/                           controllers.PersonController.index
POST    /orm/person                     controllers.PersonController.addPerson
GET     /orm/persons                    controllers.PersonController.getPersons
```

View

views

```
// main.scala.html
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
<head>
    <title>@title</title>
    <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")"/>
</head>
<body>
@content
</body>
</html>

// person.scala.html
@(person: Form[CreatePersonForm])(implicit request: MessagesRequestHeader)

@import helper._

@request.flash.get("success").map { key =>
@request.messages(key)
}

@main("Welcome to Play") {
@form(routes.PersonController.addPerson()) {
@inputText(person("name"))
@inputText(person("age"))
@CSRF.formField

<div class="buttons">
    <input type="submit" value="Add Person"/>
</div>
}
}
```

Model

ORM SQL . SQL

. ORM .

models

```
//person.scala
```

```
//
package models

import play.api.libs.json._

case class Person(id: Long, name: String, age: Int)

object Person {
  implicit val personFormat = Json.format[Person]
}


// personRepository.scala
//
package models

import javax.inject.{ Inject, Singleton }
import play.api.db.slick.DatabaseConfigProvider
import slick.jdbc.JdbcProfile

import scala.concurrent.{ Future, ExecutionContext }

/**
  * A repository for people.
  *
  * @param dbConfigProvider The Play db config provider. Play will inject this for you.
  */
@Singleton
class PersonRepository @Inject() (dbConfigProvider: DatabaseConfigProvider)(implicit ec: ExecutionContext) {
  // We want the JdbcProfile for this provider
  private val dbConfig = dbConfigProvider.get[JdbcProfile]

  // These imports are important, the first one brings db into scope, which will let you do the actual db
operations.
  // The second one brings the Slick DSL into scope, which lets you define the table and other queries.
  import dbConfig._
  import profile.api._

  /**
    * Here we define the table. It will have a name of people
    */
  private class PeopleTable(tag: Tag) extends Table[Person](tag, "people") {

    /** The ID column, which is the primary key, and auto incremented */
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)

    /** The name column */
    def name = column[String]("name")

    /** The age column */
    def age = column[Int]("age")

    /**
      * This is the tables default "projection".
      *
      * It defines how the columns are converted to and from the Person object.
      *
      * In this case, we are simply passing the id, name and page parameters to the Person case classes
      * apply and unapply methods.
      */
    def * = (id, name, age) <> ((Person.apply _).tupled, Person.unapply)
  }

  /**
    * The starting point for all queries on the people table.
    */
  private val people = TableQuery[PeopleTable]

  /**
    * Create a person with the given name and age.
    *
```

```
    * This is an asynchronous operation, it will return a future of the created person, which can be used to
obtain the
    * id for that person.
    */
  def create(name: String, age: Int): Future[Person] = db.run {
    // We create a projection of just the name and age columns, since we're not inserting a value for the id
column
    (people.map(p => (p.name, p.age))
      // Now define it to return the id, because we want to know what id was generated for the person
      returning people.map(_.id)
      // And we define a transformation for the returned value, which combines our original parameters with the
      // returned id
      into ((nameAge, id) => Person(id, nameAge._1, nameAge._2))
      // And finally, insert the person into the database
      ) += (name, age)
  }

  /**
    * List all the people in the database.
    */
  def list(): Future[Seq[Person]] = db.run {
    people.result
  }
}
```

Controller

controllers/personcontroller.scala

```
package controllers

import javax.inject._

import models._
import play.api.data.Form
import play.api.data.Forms._
import play.api.data.validation.Constraints._
import play.api.i18n._
import play.api.libs.json.Json
import play.api.mvc._

import scala.concurrent.{ExecutionContext, Future}

class PersonController @Inject()(repo: PersonRepository,
                                 cc: MessagesControllerComponents
                                )(implicit ec: ExecutionContext)
  extends MessagesAbstractController(cc) {

  /**
    * The mapping for the person form.
    */
  val personForm: Form[CreatePersonForm] = Form {
    mapping(
      "name" -> nonEmptyText,
      "age" -> number.verifying(min(0), max(140))
    )(CreatePersonForm.apply)(CreatePersonForm.unapply)
  }

  /**
    * The index action.
    */
  def index = Action { implicit request =>
    Ok(views.html.person(personForm))
  }

  /**
```

```
    * The add person action.
    *
    * This is asynchronous, since we're invoking the asynchronous methods on PersonRepository.
    */
  def addPerson = Action.async { implicit request =>
    // Bind the form first, then fold the result, passing a function to handle errors, and a function to handle
succes.
    personForm.bindFromRequest.fold(
      // The error function. We return the index page with the error form, which will render the errors.
      // We also wrap the result in a successful future, since this action is synchronous, but we're required
to return
      // a future because the person creation function returns a future.
      errorForm => {
        Future.successful(Ok(views.html.person(errorForm)))
      },
      // There were no errors in the from, so create the person.
      person => {
        repo.create(person.name, person.age).map { _ =>
          // If successful, we simply redirect to the index page.
          Redirect(routes.PersonController.index).flashing("success" -> "user.created")
        }
      }
    )
  }

  /**
    * A REST endpoint that gets all the people as JSON.
    */
  def getPersons = Action.async { implicit request =>
    repo.list().map { people =>
      Ok(Json.toJson(people))
    }
  }
}

/**
  * The create person form.
  *
  * Generally for forms, you should define separate objects to your models, since forms very often need to
present data
  * in a different way to your models.  In this case, it doesn't make sense to have an id parameter in the
form, since
  * that is generated once it's created.
  */
case class CreatePersonForm(name: String, age: Int)
```

evolution

DB ,

, .

conf/evolutuins/default/1.sql

```
# --- !Ups

create table "people" (
  "id" bigint generated by default as identity(start with 1) not null primary key,
  "name" varchar not null,
  "age" int not null
);

# --- !Downs

drop table "people" if exists;
```

Database 'default' needs evolution!

An SQL script will be run on your database - Apply this script now!

This SQL script must be run:

```
1 # --- Rev:1,Ups - 50e5242
2 create table "people" (
3 "id" bigint generated by default as identity(start with 1) not null primary key,
4 "name" varchar not null,
5 "age" int not null
6 );
```