

備忘録アプリケーション

備忘録アプリケーション

- 簡単な備忘録アプリケーションを作成することで、Play frameworkを使った開発を見てみます。

モデル

DB設定

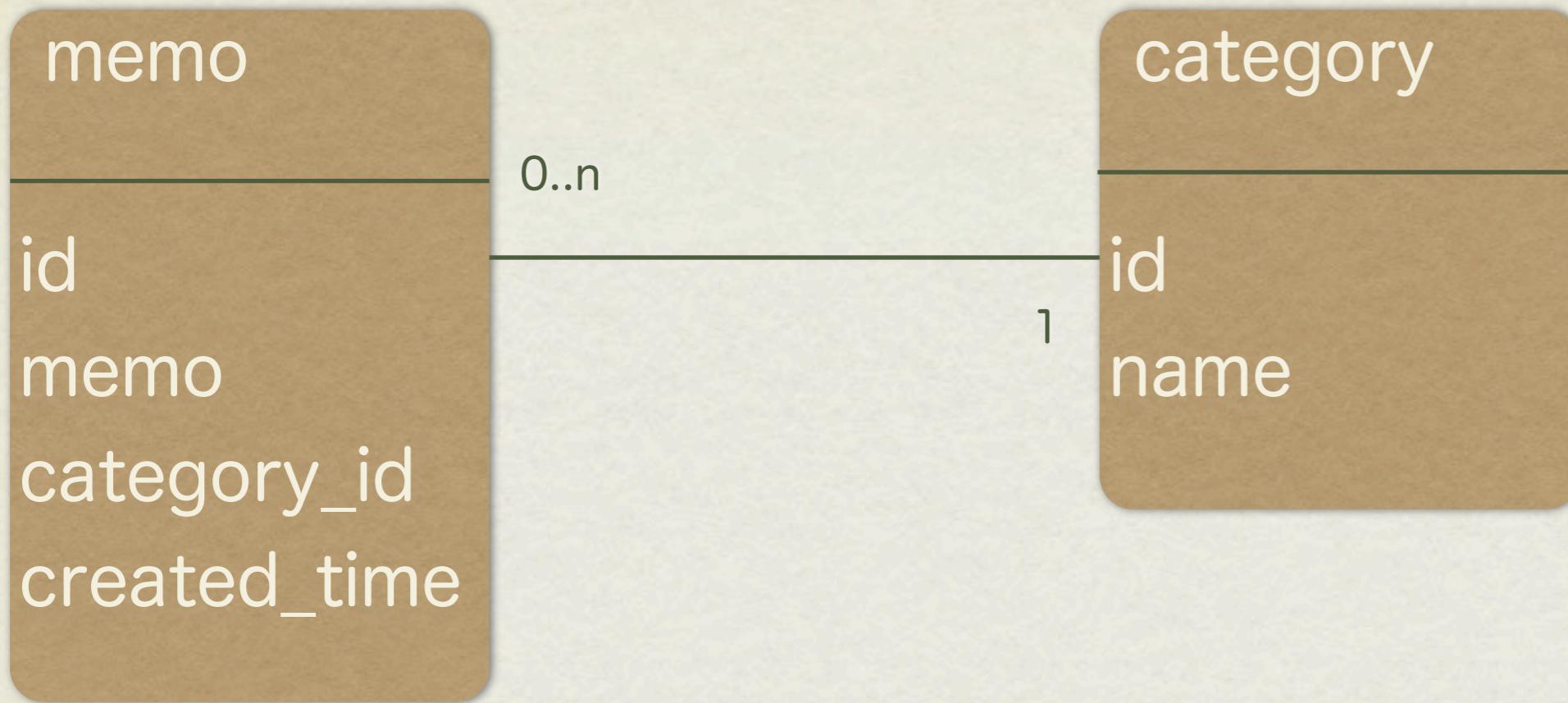
conf/application.conf

```
db.default.driver=org.h2.Driver  
db.default.url="jdbc:h2:~/h2db"  
db.default.user=sa  
db.default.password=""
```

homeディレクトリ下のh2db
ディレクトリをデータ保管場所
として使用

h2はデフォルトで存在しないDBは自動
生成するので、これ以外に設定は不要。

データモデル



DDL定義

- conf/evolutions/db名/n.sql というファイルを作成する(nは1から始まる整数)。
- DBのテーブルにどのスキーマまで適用されたかが記憶されていて、古い場合は最新のスキーマに更新される。

今回使用するDDLは、conf/evolutions/default/1.sqlです。

Anorm

- 簡単なDBアクセス用のライブラリ
- ORマッパではない
- ORマッパに対する反省
 - SQLを隠蔽しようとすると、複雑になり、DB間の違いを吸収するために最大公約数機能に制限されて、DB固有の機能の使用が使えなくなる
 - SQLを隠蔽してタイプセーフなDSLを作っても実行時エラーは無くならない
 - 既存テーブルの複雑な関連はORマッパで対処が困難
 - DBAの反抗

モデル

app/models/Message.scala

```
case class Message(  
    id: Pk[Long] = NotAssigned,  
    message: String,  
    createdTime: long,  
    categoryId: Option[Long]  
)
```

app/models/Category.scala

```
case class Category(  
    id: Pk[Long] = NotAssigned,  
    name: String  
)
```

DAO

app/models/Category.scala

```
object Category {  
    val simple = {  
        get[Pk[Long]]("category.id") ~  
        get[String]("category.name") map {  
            case id~name => Category(id, name)  
        }  
    }  
}
```

ResultSetから、
Categoryに変換する
パーサ

```
def findById(id: Long): Option[Category] =  
    DB.withConnection { implicit conn =>  
        SQL("select * from category where id = {id}")  
            .on('id -> id)  
            .as(simple.singleOpt)  
    }
```

idでレコード検索
するメソッド

DAO

app/models/Category.scala

```
object Category {  
  ...  
  def insert(category: Category) =  
    DB.withConnection { implicit conn =>  
      SQL("""  
        insert into category values (  
          select next value for category_seq),  
          {name}  
        )""")  
        .on('name -> category.name)  
        .executeInsert()  
    }  
}
```

レコードを追加するメソッド

シーケンスを使ってidをアサイン

executeInsert()はアサインされたidをOptionで返す

DAOのテスト

test/models/CategorySpec.scala

```
class CategorySpec extends Specification {  
    "Category" should {  
        "挿入したレコードを照会できる" in {  
            running(FakeApplication(additionalConfiguration = inMemoryDatabase())) {  
                val optId = Category.insert(Category(name = "買い物"))  
                val optCat = Category.findById(optId.get)  
                optCat.get.name === "買い物"  
            }  
        }  
    }  
}
```

テスト用のアプリケーション
実行環境。DBはインメモリ

DAO

app/models/Memo.scala

```
object Memo {  
    val simple = {  
        get[Pk[Long]]("memo.id") ~  
        get[String]("memo.memo") ~  
        get[Date]("memo.created_time") ~  
        get[Long]("memo.category_id") map {  
            case id~memo~createdTime~categoryId =>  
                Memo(id, memo, createdTime.getTime, categoryId)  
        }  
    }  
}
```

ResultSetから、
Memoに変換するパーサ

DAO

app/models/Memo.scala

```
val withCategory = Memo.simple ~ Category.simple map {  
  case memo~category => (memo, category)  
}
```

CategoryとMemo
の両方をパースする
パーサ

```
def listWithCategory(from: Int = 0, rows: Int = 20): Seq[(Memo, Category)] =  
  DB.withConnection { implicit conn =>  
    SQL("""  
      select * from memo  
      inner join category on memo.category_id = category.id  
      order by id  
    """")  
    .on(  
      'rows -> rows,  
      'from -> from  
    ).as(withCategory *)  
  }
```

MemoとCategoryの両方を取得

*を付けると複数レコードを処理

DAO

app/models/Category.scala

```
def listCategory(from: Int = 0, rows: Int = 20): Seq[Category] =  
  DB.withConnection { implicit conn =>  
    SQL("""  
      select * from category order by id  
      limit {rows} offset {from}  
      """"  
    ).on(  
      'rows -> rows,  
      'from -> from  
    ) .as(simple *)  
  }
```

Seqは集合(JavaのCollectionのようなもの)

DAO

app/models/Category.scala

```
def categoryOptions: Seq[(String, String)] = DB.withConnection { implicit connection =>
  SQL("select * from category order by name").as(Category.simple *).map {
    c => c.id.toString -> c.name
  }
}
```

ドロップダウンの内容を取得するSQL。
idの値を文字列にしたものと、カテゴリ名のタプルを返している。

コントローラと ビュー

カテゴリ作成画面表示

conf/routes

```
GET /startCreateCategory controllers.Application.startCreateCategory
```

app/controllers/Application.scala

```
object Application extends Controller {  
    val categoryForm = Form(  
        mapping(  
            "id" -> ignored(NotAssigned:Pk[Long]),  
            "name" -> text.verifying(nonEmpty)  
        )(Category.apply)(Category.unapply)  
    )  
    ...
```

入力フォームの定義

制約：空を許さない

requestを受け取ってResult
を返す関数

```
def startCreateCategory = Action { implicit request =>  
    Ok(views.html.createCategory(categoryForm, Category.listCategory()))  
}
```

カテゴリ作成画面表示

views/createCategory.scala.html

```
@(myForm: Form[Category], list: Seq[Category])(implicit flash: Flash)
```

```
...
```

```
@form(action = routes.Application.createCategory()) {
```

```
  <fieldset>
```

```
    @inputText(myForm("name"), '_label -> "カテゴリ名")
```

```
  </fieldset>
```

フォームの描画

```
  <input type="submit" value="作成">
```

```
}
```

```
...
```

カテゴリ作成画面表示

views/createCategory.scala.html(続き)

```
@if(list.isEmpty) {  
    <div>レコードがありません。</div>  
} else {  
    <table>  
        <thead>  
            <tr>  
                <th>カテゴリ名</th>  
            </tr>  
        </thead>  
  
        <tbody>  
            @list.map { c => <tr><td>@c.name</td></tr> }  
        </tbody>  
    </table>  
}
```

カテゴリのリスト表示

カテゴリ作成処理

controllers/Application.scala

```
def createCategory = Action { implicit request =>
  categoryForm.bindFromRequest.fold(
    formWithErrors =>
      BadRequest(views.html.createCategory(formWithErrors, Category.listCategory())),
      value => {
        Category.insert(value)
        Redirect(routes.Application.startCreateCategory)
          .flashing("message" -> ("カテゴリ " + value.name + " が追加されました。"))
      }
  )
}
```

フォームにリクエストの内容を設定

失敗した時

成功した時

フラッシュ記憶域。リダイレクトする場合は、別のリクエストになるので、フラッシュ記憶域を使用してメッセージを渡す。

カテゴリ作成処理

views/createCategory.scala.html

```
@(myForm: Form[Category], list: Seq[Category])(implicit flash: Flash)
```

```
...
```

```
@flash.get("message").map { msg =>
  <div class="globalMessage">
    @msg
  </div>
}
```

flash.get()はOptionで返るので、 mapを使ってhtmlに変換すると、
メッセージが無い時には何もhtmlが生成されずに済む

メモ作成処理

メモ作成画面表示

controllers/Application.scala

```
val memoForm = Form(  
  mapping(  
    "id" -> ignored(NotAssigned:Pk[Long]),  
    "memo" -> text.verifying(nonEmpty),  
    "created" -> ignored(0: Long),  
    "category" -> longNumber  
)  
(Memo.apply)(Memo.unapply)  
)
```

idとカテゴリ名のペアを返すDAO

```
def startCreateMemo = Action { implicit request =>  
  Ok(views.html.createMemo(memoForm, Memo.listWithCategory(), Category.categoryOptions))  
}
```

メモ作成画面表示

views/createMemo.scala.html

```
@(myForm: Form[Memo], list: Seq[(Memo, Category)], categories: Seq[(String, String)])(implicit flash: Flash)
```

```
@form(action = routes.Application.createMemo()) {
```

```
  <fieldset>
```

```
    @select(
```

```
      myForm("category"),
```

```
      categories,
```

```
      '_label -> "カテゴリ", '_showConstraints -> false
```

```
    )
```

```
    @inputText(myForm("memo"), '_label -> "メモ")
```

```
  </fieldset>
```

```
  <input type="submit" value="作成">
```

```
}
```

ドロップダウン表示

メモ作成画面表示

controllers/Application.scala

```
def createCategory = Action { implicit request =>
    categoryForm.bindFromRequest.fold(
        formWithErrors =>
            BadRequest(views.html.createCategory(formWithErrors, Category.listCategory())),
        value => {
            Category.insert(value)
            Redirect(routes.Application.startCreateCategory)
                .flashing("message" -> ("カテゴリ " + value.name + " が追加されました。"))
        }
    )
}
```

機能テスト

機能テスト

test/functional/CreateCategorySpec.scala

```
class CreateCategorySpec extends Specification {
  "Create Category" should {
    "Empty name results in bad request" in {
      running(FakeApplication()) {
        val page = route(FakeRequest(GET, "/createCategory")).get
        status(page) must equalTo(BAD_REQUEST)
        contentType(page) must beSome.which(_ == "text/html")
      }
    }
  }
}
```

テスト用にアプリケーションを
起動してGETリクエスト

カテゴリ名が入力されていないので、
BAD_REQUESTが返ることを確認

機能テスト

test/functional/CreateCategorySpec.scala

```
"Create category success" in {
```

```
  running(FakeApplication(additionalConfiguration = inMemoryDatabase())) {
```

```
    models.Category.listCategory().isEmpty === true
```

インメモリDBを作成

最初は空

```
    val page = route(FakeRequest(GET, "/createCategory?name=TestCategory")).get
```

```
    status(page) must equalTo(SEE_OTHER)
```

```
    redirectLocation(page) must beSome.which(_ == "/startCreateCategory")
```

```
    val recs = models.Category.listCategory()
```

```
    recs.size === 1
```

```
    recs.head.name === "TestCategory"
```

```
}
```

```
}
```

成功したので、/startCreateCategory
へリダイレクト

DBにカテゴリが作成さ
れていることを確認

ブラウザテスト

ブラウザテスト

- Seleniumが同梱されている
- サーバ起動、DB初期化、Selenium起動がテストクラス内で全て完結する。

ブラウザテスト

```
class CreateCategorySpec extends Specification {  
    "カテゴリの作成" should {  
        "何も入力せずにsubmitすると必須エラーが表示されること" in {  
            running(new TestServer(  
                port = 3333,  
                application = FakeApplication(additionalConfiguration = inMemoryDatabase())  
, classOf[FirefoxDriver])  
            { browser =>  
                browser.goTo("http://localhost:3333/startCreateCategory")  
                browser.title === "カテゴリの作成"  
  
                browser.submit("form")  
                browser.$("div.error div.input span.help-inline").getText == "Required"  
            }  
        }  
    }  
}
```

テストサーバをポート
3333で起動し、インメ
モリDBを使用する

firefoxでテスト

タイトルの確認

カテゴリ作成画面で何も入力せずにsubmitす
るとRequiredエラーが表示されるか検証

ブラウザテスト

```
"カテゴリが登録されること" in {
    running(new TestServer(
        port = 3333,
        application = FakeApplication(additionalConfiguration = inMemoryDatabase())
    ), classOf[FirefoxDriver])
    { browser =>
        browser.goTo("http://localhost:3333/startCreateCategory")
        browser.find("#name").text("My category")
        browser.submit("form")
        browser.$("#categoryList tr td").getText === "My category"
    }
}
```

カテゴリ名を入れて
submit

リストに登録されたカテゴリが表示されるこ
とを検証

その他

- IDE対応
 - playのコンソールで、eclipseと入れると、Eclipse用のプロジェクトファイルが生成される。
 - playのコンソールでideaと入れると、IntelliJ IDEA用のプロジェクトファイルが生成される。
- デバッガ
 - play debugでコンソールを開いてから、runすればデバッガをアタッチしてデバッグできる。

まとめ

- 関数の組み合わせで処理を書く
- Immutable(変更できないオブジェクト)を用いる
- ORマッパを使わずに直接SQLを使用
- 機能テスト、ブラウザテストが簡単に自動化できる