Список літератури

1. Метод итерации [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8. – Загл. с экрана.
2. Число пі [Електронний ресурс]. – Режим доступу: http://uk.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE_ %D0%BF%D1%96. – Назва з екрана.
3. Hewitt C. Actor Model of Computation for Scalable Robust Information Systems [Electronic resource] / Carl Hewitt. – Inconsistency Robustness, 2015. – 978-1-84890-159-9. – <hal-01163534v3>. – Mode of access: https://hal.archives-ouvertes.fr/hal-01163534/document. – Title from the screen.
4. Introduction to Parallel Computing. Simple Heat Equation [Electronic resource]. – Mode of access: https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesHeat. – Title from the screen.
5. Karmani R. Actor Frameworks for the JVM Platform: A Comparative Analysis / Rajesh Karmani, Amin Shali, Gul Agha. – Department of Computer Science University of Illinois, Urbana-Champaign, 2009. – 10 p.
6. Karmani R. Actors / Rajesh K. Karmani and Gul Agha // Encyclopedia of Parallel Computing. – Springer, 2011. – P. 1–11.
7. Munish K. Gupta. Akka Essentials / K. Munish. – Packt Publishing, 2012. – 234 p.
8. Quasar. User Manual [Electronic resource]. – Mode of access: http://docs.paralleluniverse.co/quasar/. – Title from the screen.

O. Pyechkrova, P. Akhmedzyanov

ANALYSIS IMPLEMENTATION MODEL ACTORS FRAMEWORKS

In the article a question is considered in relation to modern development of frameworks, realization of model of actors status, they are considered functional possibilities, and also features of work, with them. The results of comparison and testing of select frameworks are analysed.

Keywords: framework, model of actors, Akka, Quasar, Gpars.

Матеріал надійшов 30.09.2015

УДК 004.42

Кирієнко О. В., Малькевич Б. І.

ВИКОРИСТАННЯ ФРЕЙМВОРКУ JAVA PLAY ДЛЯ РОЗРОБКИ ВЕБ-ЗАСТОСУНКІВ

У роботі розглянуто основні особливості Java фреймворку Play. Описано середовище та основні переваги цього фреймворку.

Ключові слова: Play Framework, Java фреймворк Play.

Вступ

Play – каркас розробки з відкритим кодом, написаний на Scala і Java, який імплементує патерн проектування модель-представлення-контролер (MVC).

Play 2 повністю базований на RESTful. У ньому інтегровано юніт тестування, JUnit та Selenium, вбудовані в ядро; API містить велику кількість різних компонентів. Усі контролери каркасу статичні, а архітектура модульна [4].

Основна частина

1. Оброблення HTTP запитів

Більшість запитів (requests) застосунку обробляються за допомогою Action – в основному це Java метод, який обробляє параметри запиту і повертає результат для відправлення клієнту в його браузер. Тип результату *play.mvc.Result*.

Контролер реалізується як клас, який наслідує *play.mvc.Controller*, що містить свої методи (з параметрами або без).

Різні види результатів підтримуються за допомогою *play.mvc.Result* та *play.mvc.Results.* Вони підтримують задавання різного роду хедерів і тіл запитів, також мають хелпери, які легко задають різні коди, такі як *ok*, *notFound, badRequest, internalServerError, status,* останній дозволяє створювати власний код з описом. Усіх цих помічників можна знайти в класі *play.mvc. Results*.

Переадресація браузера до нового URL – це специфічний *Result*, але, на відміну від статусів, він не має тіла. Наприклад:

```
public static Result index() {
        return redirect("/user/home");
}.
```

«Роутер» – це компонент, який переводить кожен вхідний HTTP запит у виклик методу «дії» (*static, public* метод у контролері). HTTP запит розглядається як івет до MVC фреймворку. Він складається з двох частин: шлях запиту *(/clients/1542, /photos/list*) і HTTP метод (*GET, POST, ...*).

Усі шляхи, які оголошені у файлі *conf/routes,* компілюються. Це означає, що ми бачитимемо помилки одразу в браузері (якщо працюємо в дебаг-режимі). Кожен шлях складається з HTTP методу (підтримуються всі типи HTTP методів: *GET, POST, PUT, DELETE, HEAD*) і URI патерну, за яким закріплений відповідний метод (*action*). У цьому файлі можна додавати коментарі, і вони мають починатися з #. З прикладу можна помітити, що тип параметра йде після імені (як у мові Scala):

```
# Display a client.
GET /clients/:id controllers.Clients.show(id: Long)
```

URI патерни оголошують шляхи для запитів. Наприклад, для отримання всіх клієнтів: *GET / clients/all controllers.Clients.list()*.

За потреби знаходження клієнта за його id додається динамічна частина (може бути не одна): *GET /clients/:id controllers.Clients.show(id: Long)*.

Для отримання динамічної частини, яка охоплює більше ніж один сегмент URI, розділений слешами, можна оголосити динамічну частину за допомогою *id, який використовує регулярний вираз:

*GET /files/*name controllers.Application. download(name)*.

Наприклад, для запиту: *GET /files/images/ logo.png,* ім'я – name динамічна частина, яка міститиме значення images/logo.png.

Можна оголошувати власні вирази для динамічної частини, використовуючи синтакс $id<regex>:

GET /items/$id<[0-9]+> controllers.Items. show(id: Long).

Остання частина шляху повинна мати назву методу, який викликатиметься при відповідному запиті. Якщо ж вона не має жодних параметрів – явно викличте відповідний метод: *GET / controllers.Application.homePage()*.

Якщо метод має параметри, вони шукатимуться в URI шляху або братимуться з посилання, яке було зроблене запитом. Наприклад:

GET /:page controllers.Application. show(page), або:

```
# i.e. http://myserver.com/?page=index
GET / controllers.Application.show(page).
```

Відповідний метод *show* має оголошуватися в контролері *controllers.Application*:

```
public static Result show(String page) {
String content = Page.getContentOf(page);
response().setContentType("text/html");
return ok(content);
}.
```

Зручність цього підходу полягає в зібранні всіх можливих шляхів в одному місці та використанні мінімуму простору для оголошення одного такого шляху.

Інколи нам потрібні параметри з фіксованими значеннями. Їх ми можемо використовувати за таким принципом:

```
# Extract the page parameter from the path,
or fix the value for /
GET / controllers.Application.show(page =
"home")
GET /:page controllers.Application.show(page).
```

Можна використовувати параметри зі значенням за замовчуванням. Вони використовуватимуться, коли значення не буде знайдено із вхідного запиту:

 # Pagination links, like /clients?page=3
GET /clients controllers.Clients.list(page: Int ?= 1).

Також можна вказати опціональні параметри, які можуть не бути присутніми у всіх запитах: *GET /api/list-all controllers.Api.list(version ?= null).*

Тип результату автоматично визначається зі значення, яке ви вкажете як тіло: *Result textResult = ok("Hello World!").*

У цьому випадку автоматично визначить Content-Type в хедері як *text/ plain*. Але в наступному прикладі тип буде визначено як *application/json*:

JsonNode json = Json.toJson(object);
Result jsonResult = ok(json).

Інколи нам потрібно задавати тип власноруч. Для цього слід використати *as* («тут вказуємо тип, який необхідний»). Приклад використання в коді:

Result htmlResult = ok("<h1>Hello World!</h1>").as("text/html");

Аналогічно можна задати і тип через відповідь (response):

```
public static Result index() {
response().setContentType("text/html");
return ok("<h1>Hello World!</h1>");
}.
```

Таким же чином ми можемо легко та зручно задавати будь-які інші типи заголовків:

```
public static Result index() {
response().setContentType("text/html");
response().setHeader(CACHE_CONTROL,
"max-age=3600");
response().setHeader(ETAG, "xxx");
return ok("<h1>Hello World!</h1>");
}.
```

Потрібно пам'ятати, що будь-які зміни ведуть до скидання налаштувань заголовків за замовчуванням.

JSON (JavaScript Object Notation, джейсон) – це текстовий формат обміну даними між комп'ютерами. JSON базується на тексті і може бути з легкістю прочитаним людиною. Формат дає змогу описувати об'єкти та інші структури даних. Головним чином він використовується для передачі структурованої інформації через мережу (завдяки процесу, що називають серіалізацією) [3].

У Play є хелпери, які дозволяють перетворювати тіло запиту в джейсон нод (JsonNode).

JsonNode mapSettingsNode = request().body().asJson()

Існує більш елегантний спосіб мапінгу. Можна створити з джейсон-об'єкта об'єкт, який для нас звичний і знайомий з Java.

ObjectMapper().treeToValue(mapSettingsNode, MapSettings.class);

Легко можна забрати значення за ключем:

String name = json.findPath("name").textValue().

Ми отримали джейсон нод із запиту, але цим Play не обмежується і ви можете з такою ж легкістю створити свою ноду і відправити її як респонс:

```
public static Result sayHello() {
ObjectNode result = Json.newObject();
result.put("exampleField1", "foobar");
result.put("exampleField2", "Hello world!");
return ok(result);
},
```

Або:

```
public Result getPeople() {
List<Person> people = personDao.findAll();
return ok(Json.toJson(people));
}.
```

Play підтримує «хелпери», які роблять роботу з Cookies приємнішою та легшою. Вони додаються до HTTP відповіді:

response().setCookie("theme", "blue");.

Якщо потрібно задати більше параметрів (шлях, домен, час життя, додаткова безпека...), то вони задаються через кому, перевантажуючи методи, як показано нижче:

```
response().setCookie(
"theme",              // name
"blue",               // value
3600,                 // maximum age
"/some/path",         // path
".example.com",       // domain
false,                // secure
true                  // http only
);.
```

Для видалення Cookie, попередньо збережених у браузері, слід задати:

response().discardCookie("theme");.

За замовчуванням Play встановлює UTF-8 кодування. У Play передбачено зміну кодування. Зміна зазначається при створенні *Result*:

```
public static Result index() {
        response().setContentType("text/html;
    charset=iso-8859-1");
```

```
        return ok("<h1>Hello World!</h1>",
    "iso-8859-1");
}.
```

Для збереження даних між декількома HTTP запитами можна скористатися сесією або Flash scopes. Дані, що зберігаються в сесії, доступні протягом усього сеансу користувача, а дані, що зберігаються у Flash scopes, доступні тільки для наступного запиту. Ці дані не зберігаються на сервері, але додаються до кожного наступного HTTP запиту, використовуючи Cookies. Розмір даних дуже обмежений (до 4 КБайт), і можна використовувати лише стрінгові значення.

Cookies автоматично підписані секретним ключем, так що клієнт не може їх змінити (будуть позначені як недійсні). Сесії Play не призначені для використання як кеш-пам'ять. Якщо потрібно кешувати деякі дані, пов'язані з конкретною сесією, використовують вбудований у Play механізм кешування і сесію для зберігання унікального ідентифікатора, щоб зв'язати закешовані дані з конкретним користувачем. У Play немає жодного тайм-ауту, який видаляє сесію після того, як користувач закриває веб-браузер. За бажанням можна це реалізувати власноруч, додавши мітку в сесії, і використовувати її в додатку.

Play підтримує декілька корисних методів, які спрощують роботу з сесією, яка використовує Cookie. Наприклад, збереження даних реалізується так:

```
public static Result login() {
session("connected", "user@gmail.com");
return ok("Welcome!");
},
```

а видалення сесії за іменем – таким чином:

```
public static Result logout() {
session().remove("connected");
return ok("Bye");
}.
```

Також дуже просто отримують дані з HTTP запиту:

```
public static Result index() {
String user = session("connected");
if(user != null) {
        return ok("Hello" + user);
        } else {
                return      unauthorized("Oops,
            you are not connected");
        }
}.
```

Для очищення всієї сесії є готове рішення:

```
public static Result logout() {
session().clear();
return ok("Bye");
}.
```

Body parser – це HTTP запит (для методів POST і PUT), який містить тіло, що «мапить» в об'єкт з Java. Зауважимо, що не можна використовувати нативний BodyParser із Java, адже Play BodyParser повинен обробляти вміст тіла поступово, використовуючи *Iteratee [Byte Array [], A]*, що реалізований у Scala. Однак Play забезпечує обробку *BodyParsers* за замовчуванням, які включають більшість випадків використання (JSON, XML, текст, завантаження файлів).

У Java API всі аналізатори тіла повинні генерувати значення *play.mvc.Http.RequestBody*, які можна отримати за допомогою:

```
request().body():
public static Result index() {
RequestBody body = request().body();
return ok("Got body: " + body);
}.
```

Можете вказати *BodyParser* для конкретної дії, використавши анотацію *@BodyParser.Of:*

```
@BodyParser.Of(BodyParser.Json.class)
public static Result index() {
RequestBody body = request().body();
return ok("Got json: " + body.asJson());
}
```

Методи RequestBody (*asText ()* або *asJson*()...) повернуть NULL, якщо парсер використовується для обчислення цього тіла, а тип контенту був вказаний інший.

Текстові парсери (*text, json, xml* чи *formUrlEncoded*) мають обмеженість на пам'ять (вони мають завантажувати увесь контент у пам'ять). За замовчуванням це значення рівне 100 КБайт. Його можна змінити в полі *parsers.text.maxLength* у головному файлі конфігурацій *application.conf*:

parsers.text.maxLength=128K.

Можна вказати обмеження за допомогою анотації *@BodyParser.Of*:

```
// Accept only 10KB of data.
@BodyParser.Of(value = BodyParser.Text.
class, maxLength = 10 * 1024)
public static Result index() {
```

```
        if(request().body().
    isMaxSizeExceeded()) {
                return badRequest("Too much
            data!");
        } else {
                return ok("Got body: " +
            request().body().asText());
        }
}.
```

2. Асинхронні запити

Play обробляє кожен запит в асинхронному режимі без жодного блокування. За замовчуванням він налаштований для асинхронних контролерів, тобто контролер не повинен блокуватись і чекати на виконання якоїсь операції. Типовими прикладами таких блокувань є JDBC запити, потокове API, HTTP запити і довгі обчислення.

Незважаючи на можливість збільшення кількості потоків, для збільшення кількості обробки одночасних запитів, які блокують контролери, все ж рекомендовано дотримуватись підходу зі збереженням асинхронних контролерів. Це робить їх більш гнучкими і дає змогу підтримувати систему в нормальному робочому стані за великих навантажень.

Враховуючи, що Play не має методів, які блокують роботу, ми мусимо повертати значення, яке ще навіть не було обчислене. Play дозволяє це зробити за допомогою повернення не самого результату, а обіцянки *Promise<Result>*, що цей результат буде оброблений. Веб-клієнт, звісно, буде блокований і чекатиме на оброблений результат, а сервер робитиме обчислення і під час цього не буде блокований.

Для створення *Promise<Result>* нам потрібно спочатку зробити ще одну «обіцянку»:

```
Promise<Double> promiseOfPIValue =
computePIAsynchronously();
Promise<Result> promiseOfResult =
promiseOfPIValue.map(
        new Function<Double,Result>() {
                public Result apply(Double pi) {
                        return ok("PI value
                    computed: " + pi);
                }
        }
);
```

Асинхронні Play методи повертають нам *Promise*. Ще один варіант реалізації «обіцянки» за допомогою методу *promise()*:

```
Promise<Integer> promiseOfInt = Promise.
promise(
        new Function0<Integer>() {
                public Integer apply() {
                return intensiveComputation();
                }
        }
);
```

За потреби повернути файл великого розміру і розрахувати його *Content-Length* у нашому фреймворку підтримується стандартна передача файлів за іменем. Знаючи ім'я, можна безпроблемно відіслати файл. Він також автоматично задає такі заголовки, як *Content-Length* та *Content-Disposition* (*attachment; filename=fileToServe.pdf*). Наприклад:

```
public static Result index() {
return ok(new java.io.File("/tmp/fileToServe.pdf"));
}
```

Play чудово справляється з відправленням файлів фіксованого розміру, проте бувають файли, розмір яких ми не знаємо. Наприклад, стріми, або файли, які динамічно розраховуються. У Play і для цього є готове рішення під назвою «Chunked responses». Технологія дозволяє відправляти частини даних, щойно вони стають доступними. Недоліком такої відправки є те, що клієнт не має можливості бачити конкретний прогрес-бар (індикацію процесу) – його просто неможливо створити в таких умовах.

```
public static Result index() {
InputStream is =
getDynamicStreamSomewhere();
return ok(is);
}
```

Однією з особливостей Chunked-відповідей є створення на їхній базі *Comet sockets*. Ці сокети просто повертають поблочно text/html текст, який містить тільки <script> елементи. Для кожного блоку ми пишемо тег <script>, який містить JavaScript, що відразу ж виконується в браузері. Цей метод дозволяє відправляти івенти клієнтам без жодних затримок.

```
Public static result index() {
        Comet comet = new comet("console.log") {
                Public void onconnected() {
                        Sendmessage("kiki");
                        Sendmessage("foo");
                        Sendmessage("bar");
                        Close();
                }
```

```
        };
        Return ok(comet);
    }
```

WebSockets – це сокети, які можуть бути використані у веб-браузері на основі протоколу, що дозволяє серверу спілкуватись з усіма клієнтами. Клієнт може відправляти повідомлення будь-коли і сервер отримувати їх доти, доки існує активне з'єднання WebSocket між сервером і клієнтом.

У сучасних HTML5 сумісних браузерах вбудована підтримка WebSockets через API JavaScript WebSocket. Однак WebSockets не обмежуються веб-браузером. Play підтримує два різні механізми використання сокетів. Перший полягає у використанні «акторів», другий – побудований на «колбеках» (callbacks).

Для використання «акторів» потрібно дати Play об'єкт *akka.actor.Props*. Він вказує, що потрібно створити актора, коли створиться нове підключення. Кожне повідомлення від клієнта буде надіслане актору, і кожне повідомлення, надіслане актором, буде надіслане відповідному клієнту.

У Play реалізовано ряд хелперів, які допомагають працювати із сокетами. Для визначення часу закриття веб-сокету Play має спеціальний метод *postStop*. У ньому можна викликати методи очищення системи. Зручним є й саме закриття сокету, коли актор завершив свою роботу. Для завершення роботи актора можна надіслати йому команду *PoisonPill*:

```
self().tell(PoisonPill.getInstance(), self());
```

Звичайно Play має інструменти для прийняття або відхилення запиту. Уявимо ситуацію, коли користувач не авторизований і він не має права приєднуватись через веб-сокети. Тоді нам знадобиться відхилення запиту, і робиться це за допомогою простого методу:

```
public static WebSocket<String> socket() {
        if (session().get("user") != null) {
                return
WebSocket.withActor(MyWebSocketActor::props);
        } else {
        return WebSocket.reject(forbidden());
        }
}
```

Це все можна робити асинхронно, замінивши *WebSocket<A>* обіцянкою, що результат повернеться пізніше *Promise<WebSocket<A>>*.

Якщо ми хочемо відповісти не стрінгом, а передати, наприклад, якусь картинку чи простий Json, ми повертаємо звичний для нас масив байтів byte[], якщо ж ми хочемо працювати з джейсоном, повертаємо JSONNode.

3. Шаблони

Twirl – шаблонний Scala двигун для Play Framework. Для автономного використання він надає SBT плагін, що безпечно інтегрується з додатками без будь-яких додаткових залежностей.

Шаблони – це текстові файли, що містять суміш розмітки і Scala-блоків коду. Компілятор Twirl переводить їх у справжнісінький Scala код, вони потім компілюються Scala і поєднуються разом з іншим кодом у файл *.class* [2].

Twirl дозволяє за допомогою мінімуму символів написати хороший проект без усіляких переривань коду для вставки html-коду.

4. SQL database

Play забезпечує плагін для управління JDBC з'єднань. Можна налаштувати стільки баз даних, скільки потрібно. Для включення плагіну бази даних потрібно додати *javaJdbc* в залежності:

```
libraryDependencies += javaJdbc.
```

Потім необхідно налаштувати пул з'єднань у конфігураційному файлі */application.conf*. За замовчуванням джерело даних JDBC має назву *default*:

```
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

щоб налаштувати інші:

```
# Orders database
db.orders.driver=org.h2.Driver
db.orders.url="jdbc:h2:mem:orders"
# Customers database
db.customers.driver=org.h2.Driver
db.customers.url="jdbc:h2:mem:customers"
```

Play підтримує більшість відомих баз даних. З їх підключенням детальніше можна ознайомитися з документації на офіційному сайті [1].

Висновки

Аналіз показав, що фреймворк Play є потужним і зручним для швидкого створення веб-додатків з багатою бібліотекою спеціалізованих налаштувань.

Хороша підтримка асинхронності і багатопотоковості позбавляє зависань з очікуванням обробки даних. Фреймворк має хороші можливості для подальшої модифікації зробленого проекту. Зауважимо, що він працює набагато ефективніше в разі використання Scala.

Список літератури

1. Developing with the H2 Database [Electronic resource]. – Mode of access: https://www.playframework.com/documentation/2.4.x/Developing-with-the-H2-Database. – Title from the screen.
2. Dinesh M. Using Twirl template engine in Scala [Electronic resource] / M. Dinesh. – Mode of access: http://www.patterns7tech.com/twirl-template-with-scala-spray. – Title from the screen.
3. Handling and serving JSON [Electronic resource]. – Mode of access: https://www.playframework.com/documentation/2.4.x/JavaJsonActions. – Title from the screen.
4. Play Framework – Build Modern & Scalable Web Apps with Java and Scala [Electronic resource]. – Mode of access: https://www.playframework.com. – Title from the screen.

O. Kyriienko, B. Malkevych

USING THE JAVA PLAY FRAMEWORK FOR DEVELOPING WEB-APPLICATIONS

This paper describes the main features of Java framework Play. Describes the environment and the basic advantages of this framework.

Keywords: Play Framework, Java framework Play.

Матеріал надійшов 15.09.2015