Framework

Het Play framework is een action based webapplicatie framework waarbij snelheid van ontwikkeling voorop staat. Dit wordt gerealiseerd door een eigen classloader systeem dat elke aanpassing aan Java sources en resources meteen oppikt en opnieuw inlaadt zonder enige tussenkomst van de ontwikkelaar. Dit artikel geeft een globaal overzicht van wat er allemaal mogelijk is en hoe simpel het is om er mee te werken. Voor een uitgebreider voorbeeld is de documentatie op de website zelf ook erg goed.

Onbeperkt schaalbare applicaties bouwen

Play framework biedt de mogelijkheden

Ronald Haring
is Senior ontwikkelaar bij iPROFS.

Dankzij het classloader systeem zijn in het Play framework bijna geen restarts van de server meer nodig. Het Play framework houdt zelf geen data op de server vast, zodat applicaties in principe onbeperkt schaalbaar zijn. Elke actie die op de applicatie kan worden uitgevoerd, kan aan dynamische urls verbonden worden, zodat er nette REST urls kunnen worden gevormd.

Het Play framework kan complexe datastructuren opbouwen aan de hand van http parameters door middel van databinding zonder dat je daar zelf veel voor hoeft te doen. Kortom het Play framework maakt het bouwen van webapplicaties weer een stuk leuker.

Het Play framework is een MVC systeem waarbij op elke laag (dus zowel op de Model, als de View als de Controller) aanpassingen zijn doorgevoerd die het de ontwikkelaars makkelijker maakt om webapplicaties te bouwen. Om een nieuwe applicatie bouwen is het voldoende om 'play create' in te vullen. Hierna zal Play de defaultstructuur van de applicatie bouwen. Via 'play run' of 'play test' kan de server worden gestart om de applicatie te draaien en om de tests uit te voeren.

Voordat je iets met de database kunt doen, dien je deze wel eerst te enablen. Dat kun je doen door in de conf directory het bestand application.conf te openen. Hierin worden allerlei algemene instellingen bewaard waaronder de database. Zoek hierin de commentaar regel # db = men en verwijder de # zodat de in memory database wordt gebruikt.

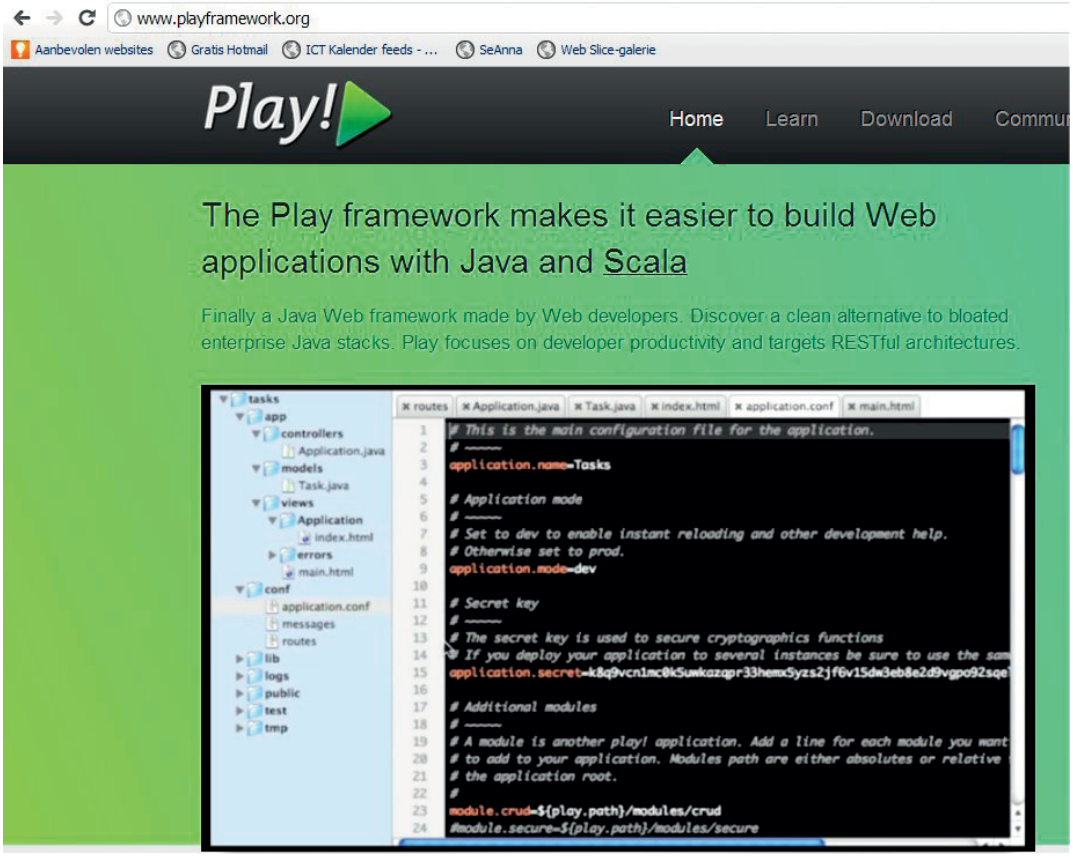
Model

Indien het Model van het Play framework wordt geextend, dan heb je active records tot je beschikking. Dit houdt in dat je voor elk model meteen een aantal standaard CRUD acties tot je beschikking hebt, zoals save, update, delete, find, count.

Het model kan alle standaard JPA annotaties bevatten die door hibernate worden uitgevoerd. De sessie wordt door middel van een Play framework plugin geopend op elke request (en ook binnen scheduler jobs en bij het uitvoeren van tests). Dit gebeurt door middel van bytecode enhancement die door de Play framework plugin wordt uitgevoerd.

Een onconventionele aanpak voor de modellen is dat de properties public zijn. Elke Java-developer zal bij het lezen van deze regel wel even achter zijn oren krabben, omdat daarmee het encapsulatie principe van OO-programmeren overboord wordt gezet, maar als je er even over nadenkt is dit weer niet zo erg. Standaard POJO's bestaan uit een aantal private properties die allemaal een public getter en setter hebben, waarmee de properties dus eigenlijk weer public zijn. Deze properties nu meteen public maken scheelt dat een boel onnodige code.

Onder water maakt het Play framework wel gebruik van getters en setters. Deze worden ook door middel van bytecode enhancement toegevoegd als deze niet bestaan. Als je dus niet wilt dat een proper-



Screenshot van de website van Play framework.

ty kan worden aangepast, kun je zelf een getter methode opnemen die het veranderen van de property niet toestaat.

Een voorbeeld van een model voor een Boek:

```
@Entity
public class Book extends Model {
   @Required
   public String name;
   @Required
   public String author;
   @Required
   public String isbn;
   public Date publishedAt;

 public static Book findBook(String name, String author)
 {
return Book.find("byNameAndAuthor", name, author).
first();
   }
}
```

Deze class vertegenwoordigt een book met wat properties. Er zijn geen getters en setters aanwezig, dus de class blijft klein. Doordat de book class het Play model extend, wordt er automatisch een id toegevoegd van het type long en zijn de active record methodes beschikbaar.

Het findBook commando gebruikt een find commando van het Play framework. Hierin kan een standaard JPA query worden opgenomen maar Play kan deze query ook zelf interpreteren, zoals hier is gebeurd door het keyword 'byNameAndAuthor' op te geven. Hierdoor wordt er een JPA query gedefinieerd waarbij de name en author worden ingevuld. Ook dat maakt de code weer wat beter leesbaar en ook nog korter. De @Required en andere validatie-annotaties worden door Play uitgevoerd op het moment dat de validaties aangeroepen worden. Dit kan o.a. door book.validateAndCreate() aan te roepen, of door de controller een @Valid notatie te geven.

Views

De standaard template engine die gebruikt wordt is gebaseerd op Groovy en kent al een aantal standaard tags die je kunt gebruiken om if-else en loopstructuren te maken. Als je zelf tags wilt maken of wilt uitbreiden is dit ook eenvoudig zelf te realiseren. Dit kan door een Java class te extenden of door Groovy snippets te maken. De templates worden vervolgens gerendered en aan de gebruiker getoond. Dit kan ook worden gebruikt om emails op te stellen. Dezelfde template die de webpagina kan tonen, kan dan worden verstuurd als email. De standaard template kan vervangen worden door een Java- of zelfs een Scala-variant. Bij de Java-variant worden de templates gegenereerd wat een performance winst oplevert die tot tien keer zo snel kan zijn als de Groovy-variant. Als de Scala-variant wordt gebruikt, kan zelfs een compiler type check afgedwongen worden voor de templates, wat bij de Groovy-variant niet kan.

Onder water maakt het Play framework wel gebruik van getters en setters.

Controllers

De controllers zorgen onder andere voor het binden van de http parameters naar objecten. Een hele simpele flow voor book management kan er zo uitzien:

- Een binnenkomstpagina waarbij maximaal 10 books worden getoond
- Elke klik op de naam van een book zal de details van een book tonen
- Een klik op een edit button zal de pagina in edit mode tonen
- Waarbij de save het book zal opslaan.

Een binnenkomstpagina waarbij alle (max 10) books getoond worden, is te vinden op localhost:9000/books/index. Books is de naam van de controller en index is de naam van de actie binnen deze controller. Deze methode ziet er als volgt uit:

```
public static void index() {
  List<Book> books = Book.all().fetch(10);
  render(books);
}
```

De template zal default opgezocht worden in de app/views/ControllerName directory met de naam van de actie die uitgevoerd is als filename. Doordat de controller een renderactie doet met alleen books als parameter, wordt de inhoud van de books als parameter met de naam books toegevoegd aan de context die door de view template kan worden uitgevraagd. Dit is tevens de reden dat de list met books expliciet benoemd wordt, aangezien anders het Play framework niet weet onder welke naam de parameters bewaard hadden moeten worden. Dit kun je wel op andere manieren oplossen binnen het Play framework, maar als ik alle opties van het Play framework ga benoemen, dan zou dit een serie artikelen worden.

He lijkt of elke link hard gecodeerd staat, maar dat is niet zo.

De template

```
#{extends 'main.html' /}
<ul>
#{list items:books, as:'book'}
  <li>#{a @Books.show(book.id)}${book.name}#{/a}
#{/list}
</ul>
#{a @Books.add()}Add#{/a}
```

De template begint met een extend statement. Dit houdt in dat er een main.html pagina is die deze pagina extend. Deze pagina heeft een tag in zich die doLayout heet, die de inhoud van deze template zal toevoegen. Dit is dus een decorator zoals tiles of sitemesh.

Vervolgens loopt de template over de books en toont alle books in een li tag. Hierin staat een aantal andere Play tags die veel gebruikt worden. De #{a} tag creëert een a href, waarbij de clickable content de naam van het book is. Wat opvalt binnen de #{a} tag is nog de @Books.show(book.id) notatie. Dit is een Play notatie die ervoor zorgt dat het href attribuut van de a tag dusdanig wordt opgebouwd dat bij het klikken op de link een actie wordt gedaan die resulteert in een methode show van de controller Books, waarbij als parameter het id van het book wordt meegegeven. De url ziet er dan als volgt uit: http://localhost:9000/books/show?id=1. Deze url wordt opgebouwd door in een speciaal bestand, het routes bestand, te zoeken naar de controller met actie. Als de actie niet gevonden wordt, of de pagina bestaat niet dan krijg je hiervan een duidelijke foutmelding in je browser te zien.

De show actie zelf is net zo simpel als het overzicht:

```
public static void show(Long id) {
   Book book = Book.findById(id);
   render(book);
}
```

De show methode zal vervolgens het book object uit de database halen (dmv een andere static methode die playframework heeft toegevoegd).

De simpele show template:

```
#{extends 'main.html' /}
<table>
   <tr><td>&{'book.name'}</td><td>${book.name}</td></tr>
   <tr><td>&{'book.author'}</td><td>${book.author}</td>
    </tr>
   <tr><td>&{'book.isbn'}</td><td>${book.isbn}</td></tr>
 <tr><td>&{'book.published'}</td><td>${book.published?.
format('dd-MM-yyyy')}</td></tr>
</table>
#{a @Books.edit(book.id)}&{'edit'}#{/a}
#{a @Books.index()}&{'cancel'}#{/a}
```

De notatie &{'key'} is een notatie voor het Play framework om messages te gaan inlezen. Uiteraard kunnen voor de messages per taal resourcebundles opgegeven worden. Een voordeel van de Play framework-manier is dat de key gewoon afgebeeld wordt indien deze niet gevonden wordt. Doordat een Groovy-template is kun je de Groovy null check gebruiken op objecten. Dit zie je in de book.published?.format('dd-MM-yyyy'). Dit houdt in dat de format methode alleen uitgevoerd wordt indien de book.published property niet null is.

Ook hier wordt weer een link opgenomen. Deze moet resulteren in een edit actie. Nu lijkt het of elke link hard gecodeerd staat, maar gelukkig is dat niet zo. Om de acties en controllers aan elkaar te praten maakt het Play framework gebruik van een routes file. In dit bestand kun je de url's opnemen die moeten resulteren in een bepaalde actie. Voorbeeld:

```
GET     /show/{<[0-9]+>id}  Books.show
GET     /add                  Books.add
POST    /save/{<[0-9]+>id}  Books.save
GET     /edit/{<[0-9]+>id}  Books.edit
```

Als vervolgens een @Books.show(book.id) gevonden wordt in de template dan wordt de routes file

bekeken om te zien welke url daarmee overeen komt. In dit geval dus de /show/<id> url. Deze wordt vervolgens gezet als de url, en als hierop geklikt wordt zal dezelfde file bekeken worden om te zien welke action dan uitgevoerd dient te worden. De regular expression in de url wordt in de url vervangen door het id veld en als de action methode aangeroepen wordt dient er wel een methode te zijn die deze parameter (met precies die naam) heeft.

Template edit.html

```
#{extends 'main.html' /}
#{form @Books.save()}<input type="hidden"
name="book.id" value="${book.id}"/>
<table>
    #{tableField book.name, fieldName:'book.name' /}
    #{tableField book.author, fieldName:'book.author' /}
    #{tableField book.isbn, fieldName:'book.isbn' /}
        <tr><td colspan="2">
        <input type="submit" value="&{'save'}"/>
</td></tr>
</table>
#{/form}
#{a @Books.show(book.id)}&{'cancel'}#{/a}
```

#{form} creëert het formulier met de correcte url om book op te slaan. Door een verborgen book.id parameter toe te voegen en een controller methode te creëren die een book als parameter gebruikt, zal het Play framework:

- Het book voor je uit de database halen
- De ingevulde waardes invullen in het book dat uit de database werd gehaald.

De tablefield tag is een eigen tag. Deze zal een tr met td maken met daarin een input voor het opgegeven veld. Een eigen tag is een Groovy template waaraan je echter nog parameters mee kunt geven. Dit is hier gedaan als impliciete parameter (zijnde de property van het book zoals de name) en als expliciete parameter, zijnde de fieldName parameter.

Controller voor save:

```
public static void save(@Valid Book book) {
   if (validation.hasErrors()) {
      params.flash();
      validation.keep();
      edit(book.id);
   }
   book.save();
   index();
}
```

De @Valid annotatie zorgt ervoor dat in het opgegeven book alle veld annotaties uitgevoerd worden. In dit geval zijn er alleen required velden, dus als een veld leeg wordt gemaakt zullen er fouten staan in de validation. Dit wordt dan vervolgens gecontroleerd.
Als er fouten zijn geconstateerd dan:

- worden de params en validatie errors bewaard in een cookie
- wordt een browser redirect naar de edit actie uitgevoerd.
- zal het playframework vervolgens de flow verlaten (dus niet meer bij de book.save methode uitkomen).

De parameters die zijn opgegeven worden als cookie bewaard, omdat het Play framework niets op de server bewaart. De opgegeven waardes worden dan weer aan de gebruiker getoond doordat er een redirect naar de edit wordt uitgevoerd, waarbij de opgegeven waardes uit het cookie worden gehaald en weer als parameters worden doorgegeven. Op deze manier houdt de server geen enkele state vast, maar worden de opgegeven waardes toch weer getoond. Een nadeel hiervan is dat een cookie maar max 4k aan tekens mag bevatten. Dit kan worden omzeild door geen redirect uit te voeren (dus geen aanroep naar edit(book.id)) maar een render actie uit te voeren (dmv render("Books/edit.html", book) waarbij dan de parameters niet geflashed hoeven te worden, aangezien de cookie dan niet wordt gebruikt.

Indien er geen fouten zijn, kan het book bewaard worden en wordt een redirect naar de list van booken uitgevoerd.

Het enige dat nog ontbreekt is de add methode:

```
public static void add() {
   Book book = new Book();
   render("Books/edit.html", book);
}
```

Hier wordt de edit pagina opnieuw gebruikt en wordt er een leeg book object toegevoegd. De rest van de flow is precies hetzelfde.

Nadeel

Een van de weinige nadelen die ik tot nu toe ben tegengekomen met het Play framework is dat er geen ondersteuning is voor Maven. Indien je dus applicaties ontwikkelt binnen een complete 'mavenized' ontwikkelstraat, inclusief deployments, dan zal het Play framework niet meteen kunnen worden ingezet. Het is wel mogelijk om continuos integration te doen met het Play framework, maar zonder de standaard Maven plugins. Alhoewel het Play framework zelf een goede webserver heeft (daarvoor wordt Netty gebruikt) is het ook mogelijk om een war te bouwen die kan worden gedeployed. Op deze manier kan een complete deployment worden neergezet, maar dan wel met enkele scripts die dit aan elkaar kunnen praten. Dit nadeel weegt echter niet op tegen de vele voordelen. «

Het enige nadeel is dat Maven niet wordt ondersteund door het Play framework.

Meer informatie
www.playframework.org
Voorbeeldcode:
https://github.com/rharing/play_article_Java_magazine