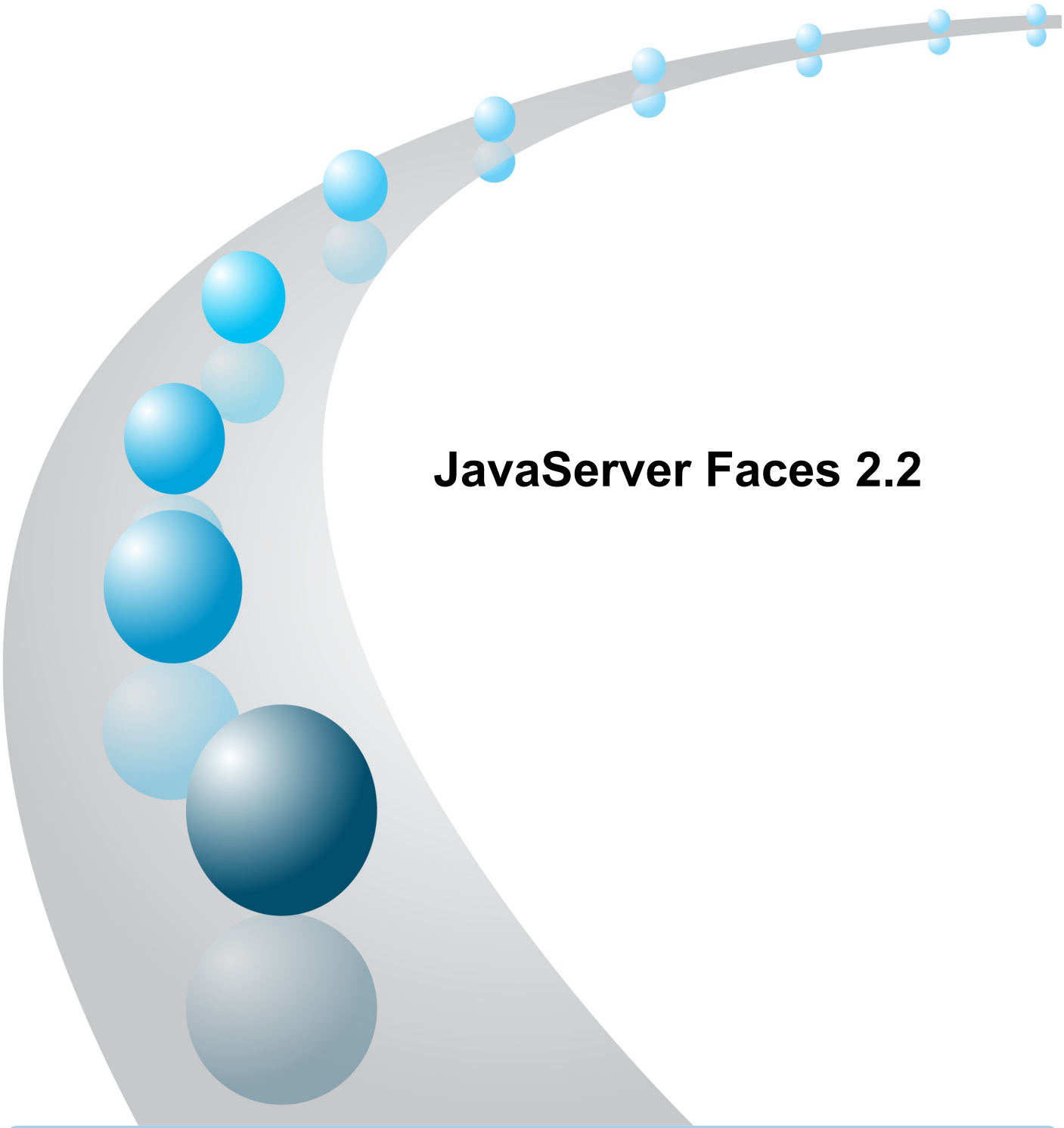
Noël Vaes

Java Trainer & Consultant



JavaServer Faces 2.2

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

20/09/2018

Inhoudsopgave

Hoofdstuk 1. Inleiding

1.1. Design patterns

Java Servlets en *Java Server Pages* (JSP) hebben het mogelijk gemaakt dynamische websites te ontwikkelen op basis van de Java-technologie. Dergelijke webapplicaties beantwoorden aan het WORA-principe: *Write Once Run Anywhere*.

Servlets bieden de mogelijkheid om programmatisch pagina's in *realtime* te genereren. *Servlets* hebben echter het nadeel dat de volledige HTML-code programmatisch gegenereerd moet worden. HTML-pagina's bestaan echter voor het grootste gedeelte uit statische gegevens. Het gebruik van *servlets* is daarom vrij omslachtig. Een wijziging van de website vergt een wijziging van de Java-code Bovendien moet de webdesigner de Java-programmeertaal machtig zijn. Hij moet zowel een programmeur als een grafisch ontwerper zijn.

Sommige van deze problemen werden opgelost met *Java Server Pages (JSP).* Dit was een makkelijke manier om een *servlet* te genereren. De rollen werden hierbij omgekeerd. JSP-pagina's bestaan gewoon uit HTML-syntax waardoor het makkelijk is de statische inhoud van de pagina te genereren en te onderhouden. De dynamische gegevens worden hieraan toegevoegd door middel van *scriptlets* en *custom tags*. De webcontainer zorgt voor de omzetting van de JSP-pagina naar een *servlet.*
Een JSP-pagina is daarom ook niet meer dan een makkelijke manier om een *servlet* te genereren.

Met de introductie van JSP werden niet alle problemen opgelost. De JSP pagina's verworden al snel tot een onleesbare en moeilijk onderhoudbare mengeling van statische HTML-code en *scriptlets*. Als er bovendien ook nog (*clientside*-)*JavaScript*-code aan wordt toegevoegd, wordt de verwarring nog groter. Ook hier moet de webontwikkelaar zowel een ontwerper als een programmeur zijn.

Dit soort problemen heeft geleid tot het ontstaan van een aantal *design patterns* voor de ontwikkeling van webapplicaties. Een *design pattern* is een beproefde manier of patroon om een bepaald soort problemen adequaat op te lossen. *Design patterns* ontstaan in de gemeenschap van (web)ontwikkelaars doordat ze allen dezelfde problemen tegenkomen en hiervoor oplossingen zoeken. Deze oplossingen worden uitgewisseld, getest en bijgestuurd tot ze een zekere maturiteit bekomen.

Voor het ontwikkelen van grafische applicaties is het *design pattern* ***Model View Controller*** zeer in trek. We kunnen dit toepassen bij dynamische webapplicaties en het zal ons helpen een goede scheiding te maken tussen de presentatielogica en applicatielogica.

1.2. Model View Controller

Het *design pattern* ***Model View Controller (MVC)*** vindt zijn oorsprong in de programmeertaal *SmallTalk*.

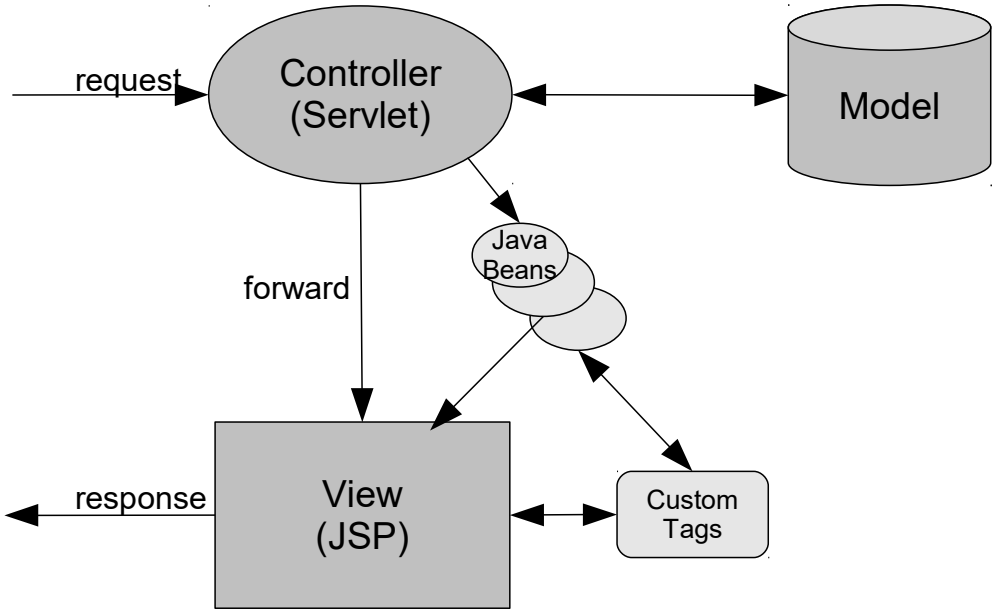
Bij het ontwikkelen van een toepassing maakt men een onderscheid tussen drie verschillende componenten:

1. **Model:** Dit is doorgaans het datamodel van de applicatie. Bij eenvoudige applicaties is dit vaak een databank waarin de gegevens bewaard worden. In meer geavanceerde applicaties omvat dit ook de componenten die de *business*-logica omvatten (*multitier*-applicaties).
2. **View:** Dit is het zicht dat de eindgebruiker heeft op het *model*. Zodra het *model* om een of andere reden verandert, wordt de *view* hiervan op de hoogte gebracht.

3. **Controller**: De gebruiker kan via de *view* ook ingrijpen op de applicatie. De acties die de gebruiker onderneemt worden afgehandeld door de controller. Deze zal de nodige wijzigingen aanbrengen in het *model* en er tevens voor zorgen dat de *view* wordt aangepast.

Door een dergelijke opsplitsing van verantwoordelijkheden, kan men op een gestructureerde wijze software ontwikkelen.

Dit *design pattern* kan concreet toegepast worden voor webapplicaties. Het plaatje ziet er dan als volgt uit:



Hier is een woordje uitleg op zijn plaats.

De pagina's die de gebruiker te zien krijgt, worden gegenereerd door middel van JSP. Dit is eerder het domein van de webdesigner die zorg moet dragen voor een mooie en consequente lay-out. De gebruiker kan echter acties ondernemen door te klikken op koppelingen of door formulieren te versturen. Al deze acties worden gestuurd naar een *servlet*. Dit is de *controller* die het verloop van de applicatie controleert. Op basis van de acties die de gebruiker doet, zal de *controller* aanpassingen doen aan het *model*. In eenvoudige applicaties is dit doorgaans het aanpassen van gegevens in een databank. Als het *model* wijzigt, moet ook de *view* aangepast worden. De nieuwe gegevens worden daarom door de controller bij het *model* opgevraagd en ondergebracht in *JavaBeans*. Deze zijn gewone Java-objecten voorzien van de nodige *getters* en *setters*. Deze *JavaBeans* worden nadien beschikbaar gesteld in een of andere *scope,* zodat ze nadien voor de JSP-pagina toegankelijk zijn.

In een volgende stap stuurt de *controller* het verzoek door naar een JSP-pagina. Deze is namelijk verantwoordelijk voor het genereren van de nieuwe *view* voor de gebruiker. In de JSP-pagina worden de gegevens uit de *JavaBeans* getoond. Om deze gegevens makkelijk te integreren, maakt men gebruik van *Expression Language* en *Custom Tags.*

De MVC-architectuur maakt het beheer en ontwikkelen van complexe applicaties makkelijker. Zo kan de beveiliging, het bijhouden van de applicatiestatus en de *flow* van de applicatie centraal beheerd worden.

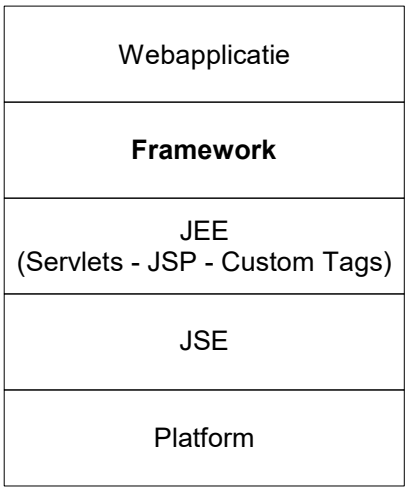
Strikt genomen is het in de MVC-architectuur mogelijk meerdere *controllers* te hebben die elk verantwoordelijk zijn voor een deel van de applicatie. Het *design pattern Front Controller* plaatst echter een centrale *servlet* aan de ingang van de webapplicatie. Van daaruit worden alle verzoeken doorgestuurd naar de verschillende componenten. De *Model View Controller* en *Front Controller* zijn twee *design patterns* die makkelijk samen gebruikt en geïntegreerd kunnen worden. Dit is onder andere het geval bij *JavaServer Faces.*

1.3. Frameworks

De MVC-architectuur is in de loop der jaren een veel toegepast *design pattern* geworden. Als men gebruik wil maken van dit *pattern* heeft men wel een goede manier om een webapplicatie op te bouwen, maar men is nog steeds verplicht de nodige componenten volgens dit *pattern* te ontwikkelen.

Gelukkig zijn er intussen allerlei *frameworks* gemaakt die ons helpen een webapplicatie te ontwikkelen. Een *framework* is niet meer dan een reeks klassen en interfaces die op een abstracte en generieke wijze bepaalde functionaliteit aanbieden waarmee we onze eigen (web-)applicatie gestalte kunnen geven. Applicaties worden als het ware boven een *framework* gebouwd waarbij het *framework* zorgt voor de onderliggende functionaliteit die gemeenschappelijk is voor alle applicaties.

Zo zijn er inmiddels ook *frameworks* die het MVC-*pattern* voor webapplicaties implementeren en waarboven een concrete webapplicaties gebouwd kan worden. De fundamentele code die zorgt voor het MVC-mechanisme wordt daarbij door het *framework* geleverd zodat de programmeur/webdesigner zich kan toeleggen op datgene wat specifiek is voor zijn applicatie.



Populaire *frameworks* zijn *Struts, Cocoon, Tapestry, Tiles, Spring MVC* enzovoort. Sommige zijn *open source,* andere daarentegen *closed source*.

1.4. JavaServer Faces

Al deze *frameworks* hebben hun eigen manier van werken en zijn niet compatibel. De keuze voor zo'n *framework* impliceert doorgaans ook dat men helemaal gebonden is aan dit

product. In het verleden zijn al meermaals allerlei *frameworks* ontstaan die na een tijdje een stille dood stierven en niet verder ontwikkeld werden. De keuze voor dergelijke *frameworks* houdt daarom ook altijd een risico in.

Vanuit de noodzaak aan een meer gestandaardiseerd *framework* is *JavaServer Faces* (JSF) ontstaan. JSF is evenwel geen concrete implementatie van een *framework* maar een specificatie die deel uitmaakt van de omvattende JEE-specificatie. Voor de concrete implementatie van deze specificatie moeten we beroep doen op de producten die door leveranciers aangeleverd worden. Producenten van JEE-webservers of applicatieservers voorzien deze concrete implementatie in hun product. Door het feit dat ze allemaal aan dezelfde specificatie moeten voldoen, bestaat de garantie dat de toepassingen die ontwikkeld worden met JSF, tevens compatibel zijn met verschillende concrete applicatieservers. Hierdoor is men niet gebonden aan een concrete leverancier, hetgeen het risico op lange termijn vermindert.

Van JSF zijn er intussen verschillende versies die gebruikt worden in de JEE-specificatie:

JEE 5: JSF 1.2
JEE 6: JSF 2.0
JEE 7: JSF 2.2

In deze cursus behandelen we JSF 2.2 en maken we gebruik van een JEE-applicatieserver (*WildFly 13)* die de JEE7-specificaties ondersteunt en in de nodige implementaties voorziet.

Hoofdstuk 2. Installatie & Configuratie

2.1. Wildfly

WildFly is beschikbaar op volgende website :*http://www.wildfly.org*.

Voor de installatie van *WildFly* moet de JDK geïnstalleerd zijn op het systeem. In deze cursus maken we gebruik van ***JDK 1.8*** in combinatie met ***WildFly 13.0.0.Final.***
Andere versies van *WildFly* kunnen een andere configuratie hebben.

2.1.1. Installatie

WildFly kan van de site afgehaald worden in de vorm van een ZIP-bestand dat we gewoon kunnen uitpakken in een of andere lokale map. Deze map noemen we de ***WildFly home directory***. Deze *home directory* kan via een omgevingsvariabele ***JBOSS_HOME*** ingesteld worden zodat hij gebruikt kan worden in andere applicaties (bijvoorbeeld *Maven*). Aangezien *WildFly* vroeger *JBoss* heette, wordt doorgaans ***JBOSS_HOME*** gebruikt.

Opdracht 1: WildFly installeren

In deze opdracht gaan we de *WildFly*-server installeren.

- Haal de laatste versie van ***WildFly*** af van de website http://wildfly.org/downloads/ (***WildFly-13.0.0-Final.zip***).
- Pak het bestand uit in een lokale map, bijvoorbeeld ***C:\WildFly***. Het is aangewezen geen map te gebruiken waarvan de naam een spatie bevat, zoals ***C:\Program Files***
- Stel de omgevingsvariabele ***JBOSS_HOME*** in met deze folder:

  JBOSS_HOME=C:\WildFly\WildFly-13.0.0.Final

2.1.2. Configuratie

Na de installatie vinden we de volgende mappenstructuur:

```
appclient
bin
docs
domain
modules
standalone
   +-- configuration
   +-- data
   +-- deployments
   +-- lib
```

Folder	***Omschrijving***
`bin`	Hierin bevinden zich de *batch*- en *shell*-bestanden voor het opstarten en afsluiten van *WildFly*.
`docs`	Hierin bevindt zich documentatie, waaronder de DTD's en schema's van bepaalde XML-bestanden.
`domain`	Bevat de configuratie voor *WildFly* in *domain*-mode.
`modules`	Bevat extra JAR-bestanden voor modules.
`standalone`	Bevat de configuratie voor *WildFly* in *standalone*-mode.
`configuration`	Hierin bevinden zich de configuratiebestanden van deze *WildFly*-configuratie. Dit zijn hoofdzakelijk XML-bestanden.
`data`	Bevat gegevensbestanden.
`deployments`	In deze folder kunnen de webapplicaties, EJB's en volledige *enterprise*-applicaties geplaatst worden. Deze worden door de server opgepikt en in werking gesteld.
`lib`	Hierin bevinden zich de extra JAR-bestanden die nodig zijn voor deze configuratie.

De server kan opgestart worden met het *batch*-bestand ***standalone.bat*** in de folder ***bin***.

Standaard is *WildFly* om veiligheidsredenen enkel toegankelijk via de *localhost* (127.0.0.1) netwerkinterface. Andere IP-adressen kunnen opgegeven worden via de optie `-b w.x.y.z` Bij 0.0.0.0 worden alle IP-adressen gebruikt.

```
standalone.bat -b 0.0.0.0
```

Om *WildFly* terug af te sluiten, drukken we CTRL-C.

De openingspagina kan bereikt worden via deze URL: ***http://localhost:8080***