



Noël Vaes

Java Trainer & Consultant



Spring 5.1

Roode Roosstraat 5
3500 Hasselt
België

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

Vrijwel alle namen van software- en hardwareproducten die in deze cursus worden genoemd, zijn tegelijkertijd ook handelsmerken en dienen dienovereenkomstig te worden behandeld.

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand of openbaar worden gemaakt in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen of op enige andere manier, zonder voorafgaande schriftelijke toestemming van de auteur. De enige uitzondering die hierop bestaat, is dat eventuele programma's en door de gebruiker te typen voorbeelden mogen worden ingevoerd opgeslagen en uitgevoerd op een computersysteem, zolang deze voor privé-doeleinden worden gebruikt, en niet bestemd zijn voor reproductie of publicatie.

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

Ondanks alle aan de samenstelling van deze tekst bestede zorg, kan de auteur geen aansprakelijkheid aanvaarden voor eventuele schade die zou kunnen voortvloeien uit enige fout, die in deze uitgave zou kunnen voorkomen.

01/12/2020

Copyright© 2020 Noël Vaes



Inhoudsopgave

Hoofdstuk 1: Core Spring.....	6
1.1 Inleiding.....	6
1.1.1 <i>Multitier</i> gedistribueerde applicaties.....	6
1.1.1.1 <i>One-tier</i> -applicaties.....	6
1.1.1.2 <i>Two-tier</i> -applicaties.....	6
1.1.1.3 <i>Three-tier-applicaties</i>	7
1.1.2 <i>Layers</i> of logische lagen.....	8
1.1.3 Java Enterprise Edition.....	10
1.1.4 Spring.....	10
1.1.4.1 Kenmerken van Spring.....	10
1.1.4.2 Spring modules.....	11
1.2 Installatie van Spring.....	11
1.3 <i>Dependency Injection</i>	12
1.3.1 <i>Inversion Of Control (IOC)</i>	12
1.3.2 <i>BeanFactory</i> en <i>ApplicationContext</i>	18
1.3.3 De configuratieklasse.....	25
1.3.4 Bean-methoden.....	26
1.3.5 De container.....	27
1.3.6 De <i>scope</i> van <i>beans</i>	28
1.3.7 Parameters van een <i>bean</i> -methode.....	33
1.3.8 De levenscyclus van een <i>bean</i>	35
1.4 Annotaties.....	38
1.4.1 Spring-annotaties.....	38
1.4.1.1 <code>@ComponentScan</code>	38
1.4.1.2 <code>@Component</code>	39
1.4.1.3 <code>@Scope</code>	40
1.4.1.4 <code>@Lazy</code>	40
1.4.1.5 <code>@AutoWired</code>	40
1.4.1.6 <code>@Primary</code>	43
1.4.1.7 <code>@Qualifier</code>	44
1.4.1.8 <code>@Required</code>	45
1.4.1.9 <code>@Nullable</code> en <code>@NonNull</code>	45
1.4.1.10 <code>@Value</code>	46
1.4.1.11 <code>@Order</code>	46
1.4.1.12 <code>@Bean</code>	48
1.4.2 JEE-annotaties (JSR-250).....	50
1.5 Geavanceerde configuratie.....	53
1.5.1 Annotaties in de configuratieklasse.....	53
1.5.2 Importeren van andere (configuratie)klassen.....	54
1.5.3 Profiles.....	56
1.5.4 Environment.....	59
1.5.5 <i>Property</i> -bestanden.....	61
1.5.6 XML-configuratie.....	62
1.6 Spring Expression Language.....	65
1.6.1 Inleiding.....	65
1.6.2 Literals.....	66
1.6.3 Bean referenties.....	67
1.6.4 Operatoren.....	67
1.6.5 Methoden en constructors.....	69
1.6.6 Verzamelingen.....	69
1.7 Event handling.....	71



1.8	Internationalization.....	73
1.9	Aspect Oriented Programming.....	77
1.9.1	Inleiding in AOP.....	77
1.9.2	AOP met Spring.....	79
1.9.2.1	Proxies.....	80
1.9.2.2	Pointcuts.....	81
1.9.2.3	Aspecten definiëren met annotaties.....	83
1.10	Unit testing.....	90
1.10.1	Inleiding.....	90
1.10.2	Testen met <i>JUnit 5</i>	90
1.10.3	<i>Spring</i> en <i>JUnit 5</i>	92
1.11	Spring Boot.....	96
1.11.1	Inleiding.....	96
1.11.2	Parent POM.....	97
1.11.3	Starter POM.....	97
1.11.4	Java-versie en <i>source encoding</i>	98
1.11.5	<i>Maven plugin</i>	98
1.11.6	Automatische configuratie.....	99
1.11.7	@SpringBootApplication.....	99
1.11.8	De hoofdklasse.....	100
1.11.9	Applicatie-argumenten.....	101
1.11.10	Properties.....	101
1.11.11	Internationalization (I18N).....	104
1.11.12	Profielen.....	104
1.11.13	Aspecten.....	104
1.11.14	Testen.....	105
1.12	Samenvatting.....	108
Hoofdstuk 2: Enterprise Spring.....		110
2.1	Inleiding.....	110
2.2	Database-toegang.....	111
2.2.1	DataSources.....	111
2.2.2	JDBC Templates.....	117
2.2.3	JPA/Hibernate.....	121
2.3	Transactiebeheer.....	131
2.3.1	Inleiding.....	131
2.3.2	Transaction Managers.....	132
2.3.3	Declaratief transactiebeheer met AOP.....	133
2.3.3.1	Propagatie van de transactie.....	137
2.3.3.2	Isolatie van de transactie.....	137
2.3.3.3	Exceptions.....	138
2.3.3.4	Read-only.....	138
2.3.3.5	Timeouts.....	138
2.3.4	<i>Unit</i> -testen met transacties.....	139
2.4	Spring Data.....	142
2.4.1	Inleiding.....	142
2.4.2	Definitie van een <i>Repository</i>	142
2.4.3	CRUD <i>Repositories</i>	143
2.4.4	Queries.....	144
2.4.4.1	Named Queries.....	144
2.4.4.2	Methodenamen.....	146
2.4.4.3	Query by example.....	148
2.4.5	Transactiebeheer.....	151
2.4.6	Locking.....	152
2.5	Beveiliging.....	154
2.5.1	Inleiding.....	154



2.5.2	Authenticatie.....	155
2.5.2.1	Gebruikers bewaren in het geheugen.....	155
2.5.2.2	Gebruikers uit een databank.....	156
2.5.2.3	Aanmelden.....	157
2.5.3	Autorisatie.....	158
2.5.4	Testen met beveiliging.....	158
2.6	Remoting.....	161
2.6.1	Inleiding.....	161
2.6.2	SOAP Web Services.....	161
2.6.2.1	Inleiding.....	161
2.6.2.2	WSDL.....	162
2.6.2.3	UDDI.....	162
2.6.2.4	Ontwikkelstrategieën.....	163
2.6.2.5	Spring beans voor gebruik van Web Services.....	163
2.6.2.6	<i>Web Service Provider</i> met <i>Spring</i>	164
2.6.2.7	Web Services Requester met <i>Spring</i>	169
2.6.3	RESTful Web Services.....	175
2.6.3.1	Inleiding: <i>Web API</i>	175
2.6.3.2	REST.....	176
2.6.3.3	<i>Web Services</i> volgens de REST-architectuur.....	178
2.6.3.3.1	HTTP-protocol.....	178
2.6.3.3.2	URL's.....	179
2.6.3.3.3	URL templates.....	180
2.6.3.3.4	Methoden.....	180
2.6.3.3.5	Representaties van resources.....	183
2.6.3.3.6	Status-codes.....	184
2.6.3.3.7	Verwijzingen.....	186
2.6.3.3.8	Request parameters.....	187
2.6.3.3.9	Documentatie van een REST API.....	188
2.6.3.4	<i>RESTful Web Services</i> met <i>Spring</i>	188
2.6.3.4.1	Configuratie van de <i>Spring</i> -webapplicatie.....	188
2.6.3.4.2	Domeinmodel en repository.....	190
2.6.3.4.3	<i>RestController</i>	192
2.6.3.4.4	Client-toepassingen.....	199
2.6.3.4.4.1	Browser.....	199
2.6.3.4.4.2	Java.....	200
2.6.3.4.4.3	HTML en Ajax.....	206
2.6.3.4.5	URL's.....	208
2.6.3.4.5.1	URL-patronen.....	209
2.6.3.4.5.2	URL-templates en padvariabelen.....	209
2.6.3.4.6	HTTP-methoden.....	211
2.6.3.4.6.1	GET.....	211
2.6.3.4.6.2	POST.....	214
2.6.3.4.6.3	PUT.....	215
2.6.3.4.6.4	PATCH.....	216
2.6.3.4.6.5	DELETE.....	217
2.6.3.4.7	Validatie.....	218
2.6.3.4.8	Mime types en Data binding.....	220
2.6.3.4.8.1	XML.....	220
2.6.3.4.8.2	JSON.....	224
2.6.3.4.9	Foutafhandeling.....	226
2.6.3.5	Beveiliging van RESTful Web Services.....	229
2.6.3.5.1	Inleiding.....	229
2.6.3.5.2	Authenticatie.....	229
2.6.3.5.3	Autorisatie.....	231
2.6.3.5.4	Authenticatie via de REST-template.....	232



2.6.3.5.5	Authenticatie via Ajax.....	232
2.6.3.6	Deployen als WAR-bestand.....	233
2.6.4	Messaging (JMS).....	234
2.6.4.1	Inleiding.....	234
2.6.4.2	<i>Messaging</i> -architectuur.....	235
2.6.4.2.1	Point-to-point-domein.....	235
2.6.4.2.2	Publish-subscribe-domein.....	236
2.6.4.2.3	Synchrone - asynchrone verwerking.....	236
2.6.4.2.4	De Naming Service.....	236
2.6.4.3	Java Messaging Service API.....	237
2.6.4.4	Message Oriented Middleware.....	241
2.6.4.5	JMS clients zonder Spring.....	241
2.6.4.6	JMS <i>clients</i> met Spring.....	243
2.6.4.7	JMS servers met Spring.....	246
2.7	Spring Batch.....	250
2.7.1	Inleiding.....	250
2.7.2	Begrippenkader.....	250
2.7.3	Mijn eerste batch.....	252
2.7.4	Jobs.....	257
2.7.4.1	JobParameters.....	257
2.7.4.2	Herstarten van een <i>JobInstance</i>	259
2.7.5	Repository.....	260
2.7.6	Steps.....	261
2.7.6.1	Blokverwerking.....	261
2.7.6.2	ItemStreams.....	262
2.7.6.3	Exceptions.....	264
2.7.6.4	TaskletStep.....	265
2.7.6.5	Meerdere <i>Steps</i>	266
2.7.6.5.1	Sequentiële uitvoering.....	266
2.7.6.5.2	Conditionele uitvoering.....	267
2.7.6.5.3	Herstarten van een <i>Step</i>	268
2.7.6.6	De <i>Job</i> beëindigen.....	269
2.7.7	Parallele verwerking.....	270
2.7.7.1	Multithreading in een <i>Step</i>	270
2.7.7.2	Multithreading tussen <i>Steps</i>	271
2.7.7.3	Asynchroon opstarten van een <i>Job</i>	272
2.7.8	Listeners.....	273
2.7.9	ItemReaders en ItemWriters.....	276
2.8	Spring Integration.....	279
2.8.1	Inleiding.....	279
2.8.2	Installatie en configuratie.....	280
2.8.3	Boodschappen en kanalen.....	283
2.8.4	Adapters.....	286
2.8.5	<i>Transformers</i> , <i>Enrichers</i> en <i>Content Filters</i>	292
2.8.6	Filters.....	302
2.8.7	Routers.....	303
2.8.8	Activators.....	305
2.8.9	Gateways.....	307
2.9	Samenvatting.....	310
Hoofdstuk 3: Spring MVC.....	311	
3.1	Inleiding.....	311
3.1.1	Model View Controller.....	311
3.1.2	Frameworks.....	312
3.1.3	Spring MVC-architectuur.....	312
3.2	Configuratie van Spring MVC.....	313



3.2.1	Configuratie van de webapplicatie.....	313
3.2.2	Configuratie van <i>HandlerMappings</i>	315
3.2.3	Configuratie van <i>Controllers</i>	315
3.2.4	Configuratie van <i>Service Beans</i>	315
3.2.5	Configuratie van <i>View Resolvers</i>	316
3.3	Mijn eerste webpagina met Spring MVC.....	318
3.4	Thymeleaf templates.....	320
3.5	De <i>WebApplicationContext</i> en <i>scope</i> van <i>beans</i>	322
3.6	Internationalization.....	324
3.7	<i>ViewControllers</i>	325
3.8	Het <i>Model</i> in <i>Spring MVC</i>	327
3.9	Return-waarden van afhandelingsmethoden.....	328
3.10	<i>URI Mapping</i> met <i>@RequestMapping</i>	329
3.10.1	Methode-mapping.....	329
3.10.2	Klassenmapping met vernauwing.....	330
3.10.3	<i>URI-Patronen</i>	333
3.10.4	<i>URI templates</i> en padvariabelen.....	334
3.11	Parameters van afhandelingsmethoden.....	335
3.11.1	<i>Request</i> -parameters gebruiken met <i>@RequestParam</i>	336
3.11.2	Het <i>model-object</i>	338
3.11.3	Het <i>command object</i>	341
3.11.4	Gegevens bewaren in een sessie.....	344
3.12	Conversie.....	347
3.12.1	Conversie en formattering van getallen.....	347
3.12.2	Conversie en formattering van datums en tijden.....	348
3.12.3	Conversiefouten.....	349
3.12.4	Programmatische conversie.....	350
3.13	Validatie.....	351
3.14	Formulieren.....	353
3.14.1	Koppeling met het <i>command object</i>	353
3.14.2	Invoervelden met tekst.....	354
3.14.3	Selectievakjes (checkbox).....	355
3.14.4	Radioknoppen.....	359
3.14.5	Keuzelijsten.....	361
3.14.6	Foutmeldingen.....	362
3.15	Exception Handling.....	366
3.16	Interceptors.....	369
3.17	Spring Web Security.....	371
3.17.1	Inleiding.....	371
3.17.2	Authenticatie.....	372
3.17.3	Autorisatie.....	373



Hoofdstuk 1: Core Spring

1.1 Inleiding

Spring is een *open source framework* voor de ontwikkeling van Java-toepassingen met behulp van herbruikbare en configureerbare componenten in de vorm van *plain old Java objects (POJO's)*. Wat dat precies betekent zullen we in de loop van deze cursus stap voor stap duidelijk maken. In feite kunnen we allerlei soorten toepassingen met *Spring* ontwikkelen maar vaak zijn het *multitier* gedistribueerde applicaties.

Deze cursus is gericht op het ontwikkelen van dergelijke *multitier* gedistribueerde applicaties met *Spring*. Daarom zullen we eerst wat meer uitleg geven over de architectuur van dit soort applicaties.

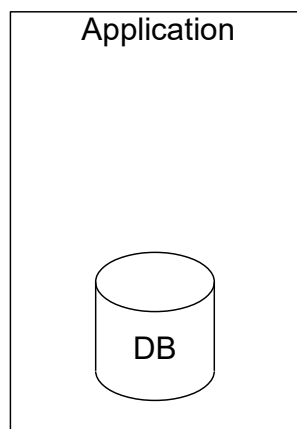
1.1.1 Multitier gedistribueerde applicaties

Multitier gedistribueerde applicaties zijn toepassingen waarbij de functionaliteit verspreid ligt over meerdere systemen die door middel van een netwerk met elkaar verbonden zijn. De software wordt onderverdeeld in verschillende fysieke lagen met elk hun eigen verantwoordelijkheid.

Om de noodzaak van dat soort applicaties aan te tonen, geven we even een overzicht van de verschillende soorten applicaties.

1.1.1.1 One-tier-applicaties

De meest eenvoudige applicaties zijn de *one-tier*-applicaties. Heel de functionaliteit is vervat in de applicatie en deze kan bijgevolg volledig zelfstandig uitgevoerd worden.



Afbeelding 1: One-tier-applicatie

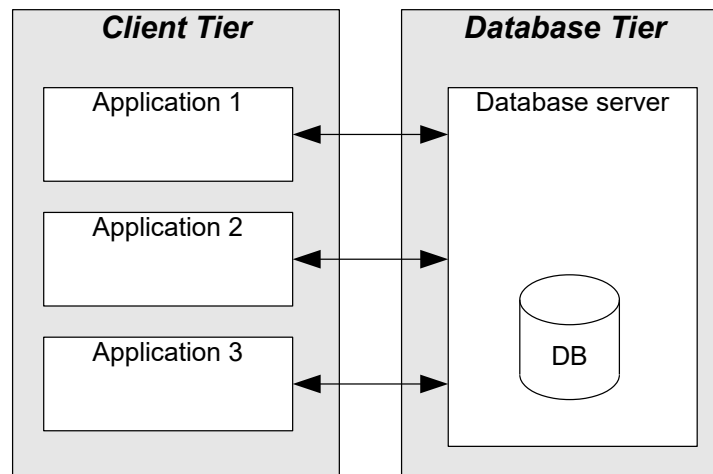
Vaak wordt er in dat soort applicaties gebruikgemaakt van een database. Deze is dan vervat in de applicatie zelf. Men spreekt dan van een *embedded database*. Dit soort applicaties is goed indien er geen informatie gedeeld moet worden met andere applicaties, eventueel met andere instanties van dezelfde applicatie. Iedere applicatie staat volledig op zichzelf en is niet verbonden met andere applicaties. Alle functionaliteit wordt uitgevoerd op het lokale systeem.

1.1.1.2 Two-tier-applicaties

Meestal is het echter nodig dat een applicatie informatie deelt met andere applicaties,



eventueel met andere instanties van dezelfde applicatie. Hierbij wordt het opslaan van de gegevens afgezonderd uit de applicatie en toevertrouwd aan een tweede applicatie die voor meerdere toepassingen toegankelijk is. Indien de data bewaard wordt in een database, wordt gebruikgemaakt van een databaseserver.



Afbeelding 2: Two-tier-applicatie

De software wordt daarbij verspreid over twee *tiers*. Vooreerst is er de *client tier* die de applicatie bevat waarmee de gebruiker werkt. Verder is er de *database tier* die de databaseserver bevat. Deze is doorgaans geplaatst op een andere machine in het netwerk. De communicatie tussen de applicatie en de database server verloopt dan via het netwerk (meestal op basis van het TCP/IP-protocol).

De gegevens die bewaard worden door de *database server* zijn toegankelijk vanuit verschillende applicaties. Dit maakt het mogelijk dat deze applicaties hun gegevens delen.

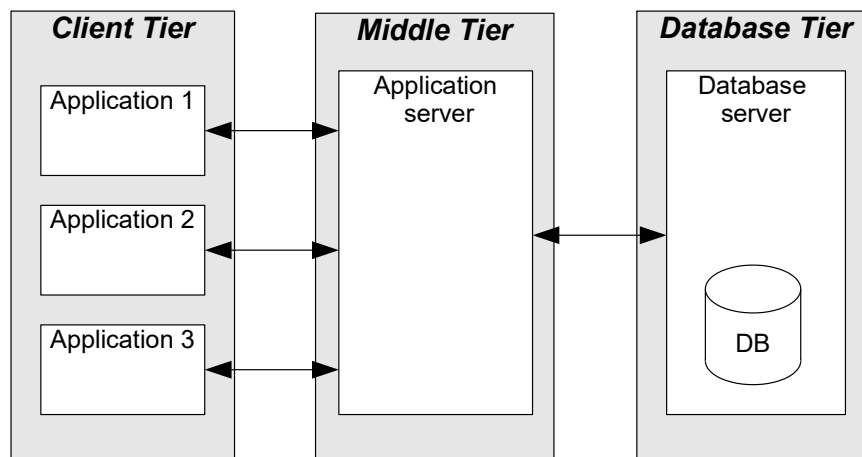
1.1.1.3 Three-tier-applicaties

Bij *two tier*-applicaties bevindt zich heel de applicatielogica in de *client tier*. Tevens bevat deze *tier* ook alles om de gebruikersinterface te presenteren. We noemen dit ook wel de presentatielogica.

Het is echter mogelijk dat dezelfde applicatie verschillende soorten gebruikersinterfaces heeft. Denk maar aan een applicatie met zowel een webinterface als een *Swing*-interface.

In het *two-tier*-model moeten we voor iedere gebruikersinterface een afzonderlijke applicatie maken met de eigen presentatielogica. Aangezien ook de applicatielogica vervat is in de applicatie moeten we deze daarin telkens opnieuw voorzien.

Het zou echter beter zijn de applicatielogica af te zonderen van de presentatielogica. Dit komt de herbruikbaarheid van de softwarecomponenten ten goede. Dit resulteert in een *three-tier*-applicatie.



Afbeelding 3: Three-tier-applicatie

De middelste *tier* bevat de *application server* die de applicatieloga bevat. Hiermee bedoelen we uiteindelijk alle functionaliteit die niet onmiddellijk gerelateerd is aan de presentatie van de applicatie aan de gebruiker. We spreken in het algemeen van *middleware*; dit is software die zich in het midden bevindt.

Deze *application server* bevindt zich doorgaans ook op een afzonderlijke machine in het netwerk. Verschillende applicaties kunnen simultaan gebruikmaken van de *application server*. De eindapplicaties moeten nu enkel nog zorgen voor de aangepaste presentatie van de applicatie naar de gebruiker toe. Zo kunnen verschillende applicaties met totaal verschillende gebruikersinterfaces toch samen gebruikmaken van dezelfde applicatieloga. Aangezien de applicaties enkel nog de presentatieloga bevatten, zijn ze dus vaak erg afgeslankt. Men spreekt in dat geval ook wel van *thin clients*.

1.1.2 Layers of logische lagen

Tiers zijn de fysieke lagen waarover de applicatie verspreid is. Daarnaast spreekt men vaak ook over de logische lagen of *layers*. Dit zijn de lagen waarin de code georganiseerd is. Deze logische lagen zijn niet noodzakelijk gebonden aan een fysieke laag. Soms bevinden meerdere logische lagen zich op dezelfde fysieke laag en in andere gevallen kan een logische laag verspreid zijn over meerdere fysieke lagen.

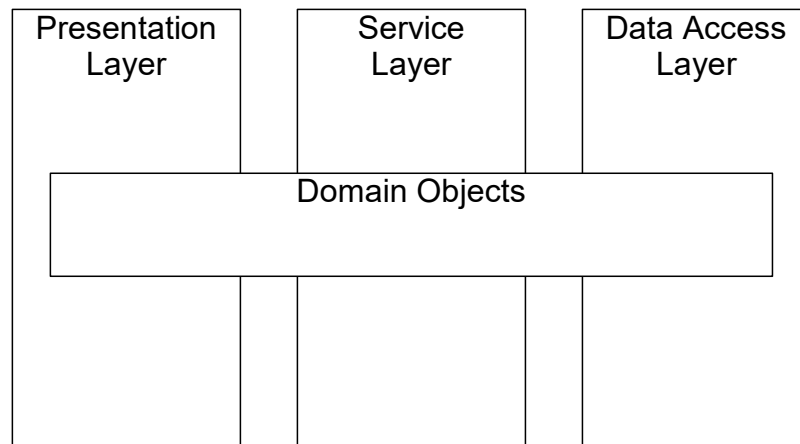
Doorgaans onderscheidt men de volgende logische lagen in een applicatie¹:

1. **Data Access Layer:** Deze laag is verantwoordelijk voor de communicatie met de databank. De toegang tot de databank is hier gecentraliseerd en afgescheiden van de rest.
2. **Service Layer:** In deze laag wordt de *business logic* uitgevoerd. Deze bestaat uit allerlei diensten (*services*) ten behoeve van onder andere de *Presentation Layer*.
3. **Presentation Layer:** Deze laag voorziet de presentatie van de applicatie naar de eindgebruiker toe. Tenminste in het geval er een grafische gebruikersinterface nodig is. Bij een B2B (*business to business*) toepassing is dit niet noodzakelijk het geval.
4. **Domain Objects:** In de gehele applicatie zijn er meestal data-objecten nodig. Deze worden in de *Data Access Layer* gesynchroniseerd met de databank. In de *Service Layer* worden deze objecten gemanipuleerd en in de *Presentation Layer* worden ze gebruikt om gegevens te tonen en nieuwe invoer van de gebruiker over te brengen

¹ De hier voorgestelde architectuur is gebruikelijk binnen *Spring*-toepassingen maar dit ontwerp is niet zonder kritiek. Het wordt ook wel *anemic domain model* genoemd omdat de data-objecten vaak ontdaan zijn van iedere *business logic*. Deze *business logic* is ondergebracht in de *service*-lagen. Een meer geavanceerde benadering tracht zoveel mogelijk *business logic* toch onder te brengen in de data-objecten zodat ze voldoen aan de basisconcepten van het objectgeoriënteerd programmeren: namelijk dat objecten een samenballing zijn van gegevens en gedrag.

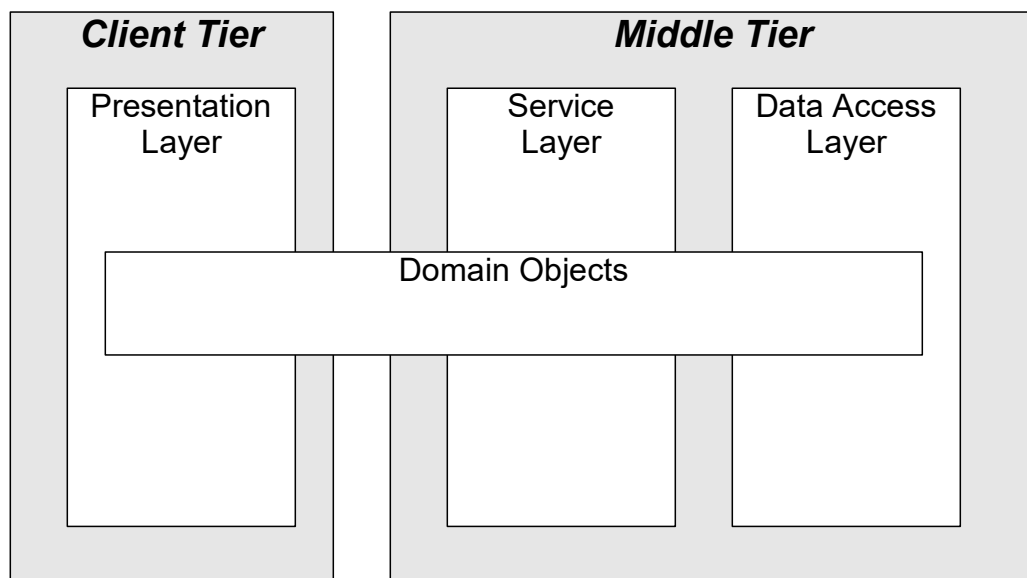


naar de *Service Layer*. Deze objecten worden dus gebruikt in de drie andere lagen en om die reden wordt in de tekening deze blok dwars weergegeven.



Afbeelding 4: De logische lagen in een applicatie.

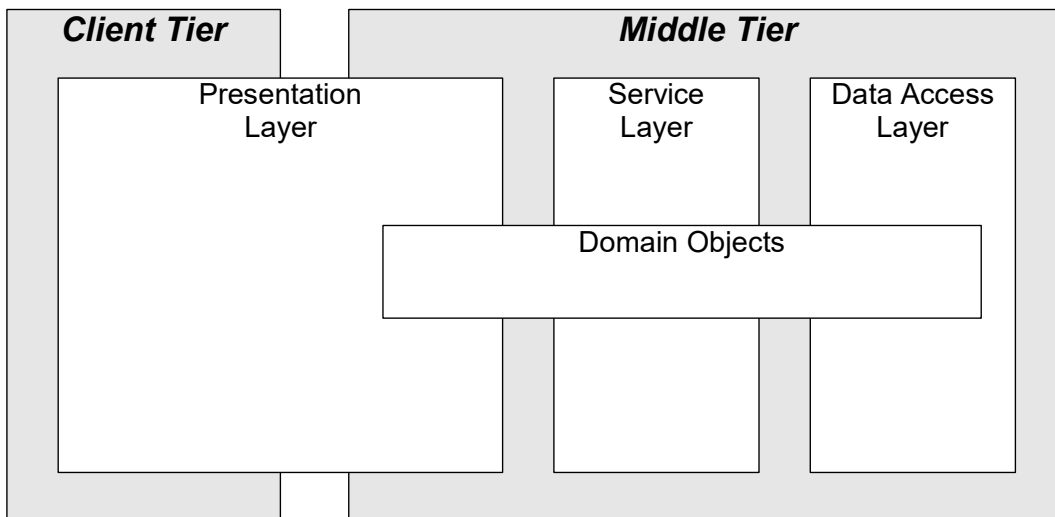
In de onderstaande afbeelding geven we de verschillende lagen weer: Deze lagen kunnen zich nu op verschillende fysieke lagen bevinden. In de onderstaande afbeelding bevindt de *Presentation Layer* zich volledig op de *client tier*. Dit zou het geval kunnen zijn indien de grafische gebruikersinterface ontworpen is als een toepassing in Swing of Java-FX.



Afbeelding 5: Presentation Layer op de client tier

Ook toepassingen op basis van *JavaScript frameworks* zoals bijvoorbeeld *JQuery* of *AngularJS* komen overeen met deze architectuur. De presentatielaag is daarbij volledig geschreven met behulp van HTML, CSS en *JavaScript*.

Bij een klassieke dynamische webtoepassing daarentegen is de *Presentation Layer* meestal gespreid over de *client tier* en de *middle tier*. De HTML-pagina's worden dynamisch aangemaakt op de webserver en getoond in de browser.



Afbeelding 6: Presentation Layer op de client tier en middle tier

1.1.3 Java Enterprise Edition

Om dergelijke *multitier* gedistribueerde applicaties met meerdere logische lagen te maken kan men best beroep doen op een bestaande software-infrastructuur die een aantal functionaliteiten kant en klaar aanbiedt.

Om ervoor te zorgen dat de verschillende aanbieders van dergelijke infrastructuur hun diensten op dezelfde manier aanbieden, is in de Java-wereld de *Java Enterprise Edition (JEE)* ontwikkeld. Dit is een uitbreiding op de *Java Standard Edition (JSE)* en bevat hoofdzakelijk specificaties waaraan de aanbieders moeten voldoen. Technisch gaat het hierbij onder andere om allerlei interfaces waarvoor men een implementatie moet voorzien.

JEE bevat de nodige technologieën om een *multitier* gedistribueerde applicatie te maken waarbij de ontwikkelaar zich kan concentreren op zijn specifieke logica en visuele voorstelling en waarbij de algemene dingen worden afgehandeld door een platform dat de JEE-specificaties implementeert.

Het JEE-verhaal met vooral het onderdeel EJB (*Enterprise JavaBeans*) heeft echter een bewogen geschiedenis. De complexiteit van EJB 2 was erg hoog en dat maakt dat dit onderdeel door de gemeenschap van ontwikkelaars nauwelijks werd aanvaard. Intussen is men met EJB 3 een nieuwe richting ingeslagen die opnieuw beter aanslaat. Intussen hadden veel ontwikkelaars wel reeds afgehaakt en waren op zoek gegaan naar betere alternatieven: onder andere *Spring* en *Hibernate*.

1.1.4 Spring

Spring is ontstaan als alternatief voor de complexe EJB's (vooral EJB 2.x). Men is hierbij teruggeslagen naar de eenvoud van gewone Java-objecten (*plain old Java objects*) die in uiteenlopende omstandigheden gebruikt kunnen worden; dit kan zowel binnen een *standalone client*-applicatie, binnen een webapplicatie of binnen om het even welke laag van een *multitier*-applicatie.

1.1.4.1 Kenmerken van Spring

We sommen even een aantal belangrijke kenmerken van *Spring* op:

1. *Spring* is **lightweight**: het gebruik van *Spring* vergt geen lijvige JAR-bestanden en ook de *overhead* tijdens de uitvoering is beperkt. Er is dus geen zware



- applicatieserver nodig die veel systeembronnen nodig heeft, zoals dat wel het geval is bij EJB.
2. **Spring** is **nonintrusive**: de componenten zijn niet afhankelijk van *Spring* of deze afhankelijkheid wordt zo klein mogelijk gehouden zodat dergelijke objecten ook elders gebruikt kunnen worden.
 3. **Dependency injection**: *Spring* promoot de losse koppeling tussen componenten waarbij de afhankelijkheden van buitenaf geïnjecteerd worden in plaats van intern opgezocht of gecreëerd worden. We zullen hier later nog uitvoerig op terugkomen.
 4. **Aspect oriented**: *Spring* maakt gebruik van *aspect oriented* technieken om de objecten te voorzien van een functionaliteit die niet tot hun kerntaak behoort, de zogenaamde *cross cutting concerns*. Ook dit zullen we uitvoerig behandelen.
 5. **Container**: *Spring* is een soort mini-container aangezien hij instaat voor de levenscyclus van Java-objecten.
 6. **Framework**: *Spring* is tevens een *framework* dat een aantal belangrijke taken voor zijn rekening kan nemen zodat de ontwikkelaar zich enkel hoeft bezig te houden met de specifieke logica van zijn toepassing.

1.1.4.2 Spring modules

Spring is opgebouwd rond een eenvoudige kern (*Spring Core*) die verder uitgebreid kan worden met allerlei modules. In deze cursus beginnen we met *Spring Core*. Deze kern kan nadien uitgebreid worden met allerlei modules die ons extra mogelijkheden bieden. Het aantal extra modules is zeer groot en in deze cursus maken we een selectie van enkele modules die het meest gangbaar zijn.

Zo breiden we in het hoofdstuk *Spring Enterprise* de kern uit met extra modules voor de integratie van JPA/Hibernate en transactiebeheer. Tevens voegen we modules toe om onze software vanop afstand ter beschikking te stellen via verschillende protocollen. Dit noemt men ook wel *remoting*.

In het hoofdstuk *Web Spring* voegen we dan weer andere modules toe voor het maken van een webapplicatie gebaseerd op de MVC-architectuur.

Dit alles zal duidelijk worden naarmate we voortschrijden in de cursus. Laten we alvast beginnen met de installatie van het *Spring framework*.

1.2 Installatie van Spring

Alle informatie over het *Spring framework* is te vinden op de volgende locatie: <http://spring.io>.

De nodige JAR-bestanden kan men eventueel afhalen op de volgende locatie: <http://repo.spring.io/release/org/springframework/spring/>.

Spring is geen applicatieserver met een specifieke installatie- en opstartprocedure. *Spring* bestaat gewoon uit een aantal JAR-bestanden die opgenomen moeten worden in het *classpath* van de applicatie. Zelf maakt *Spring* ook gebruik van andere *open-source*-projecten. De JAR-bestanden van deze afhankelijke projecten zijn evenwel niet opgenomen in de *Spring*-distributie en moeten daarom ook afzonderlijk gedownload worden en toegevoegd worden aan het *module path*.

Gezien de veelheid van afhankelijkheden en de complexiteit van de configuratie is het daarom aangewezen gebruik te maken van hulpsystemen zoals *Maven*, *Gradle* of *Ivy*. Deze zijn in staat de afhankelijkheden automatisch van het internet te plukken en te integreren in het project. In deze cursus maken we gebruik van *Maven*.

In de POM moeten we de volgende *dependency* toevoegen:



```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.19.RELEASE</version>
  </dependency>
</dependencies>
```

Later zullen we nog extra afhankelijkheden toevoegen voor bijkomende modules. Om ervoor te zorgen dat alle versies consequent hetzelfde zijn en kunnen samenwerken, kunnen we gebruikmaken van een *Bill of Materials (BOM)*. Deze wordt als volgt in de POM geconfigureerd:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.1.19.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
</dependencies>
```

Het is dan niet meer nodig het versienummer voor iedere afhankelijkheid op te geven. Dit wordt overgenomen uit de BOM.

1.3 Dependency Injection

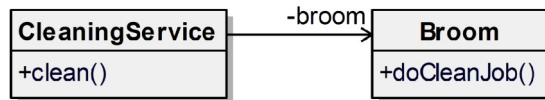
Zoals we in onze inleiding al vertelden is *Spring* een *framework* om toepassingen te maken met herbruikbare en configureerbare componenten op basis van *plain old Java objects (POJO's)*. Met *Spring* kunnen we dergelijke objecten, of ook wel *beans* genoemd, aaneenrijgen tot een werkend geheel. Dit aaneenrijgen doen we door middel van *dependency injection*.

Voor we beginnen met het gebruik van *Spring* willen we daarom even stilstaan bij deze belangrijke techniek van *dependency injection* en het concept van *inversion of control*. Beide vormen namelijk de grondslag van *Spring*.

1.3.1 Inversion Of Control (IOC)

Spring wordt vaak omschreven als een container voor *Inversion Of Control*. Dit principe komt erop neer dat Java-objecten (*beans*) hun afhankelijkheid van andere objecten van buitenaf krijgen toebedeeld in plaats dat ze zelf intern deze afhankelijkheden moeten creëren.

We zullen dit onmiddellijk illustreren met een dagdagelijks voorbeeld. Stel dat we in ons huishouden beroep willen doen op een poetsdienst die gebruikmaakt van een borstel om het huis schoon te houden. De poetsdienst wordt voorgesteld door een object van de klasse `CleaningService` die op zijn beurt gebruikmaakt van een object van de klasse `Broom`.



Afbeelding 7: Klassendiagram CleaningService - Broom

De code ziet er dan als volgt uit:

```

public class Broom {
    public void doCleanJob() {
        System.out.println("Scrub scrub");
    }
}
  
```

```

public class CleaningService {
    private Broom broom = new Broom();

    public void clean() {
        System.out.println("Cleaning the house");
        broom.doCleanJob();
    }
}
  
```

De hoofdapplicatie zou als volgt kunnen zijn:

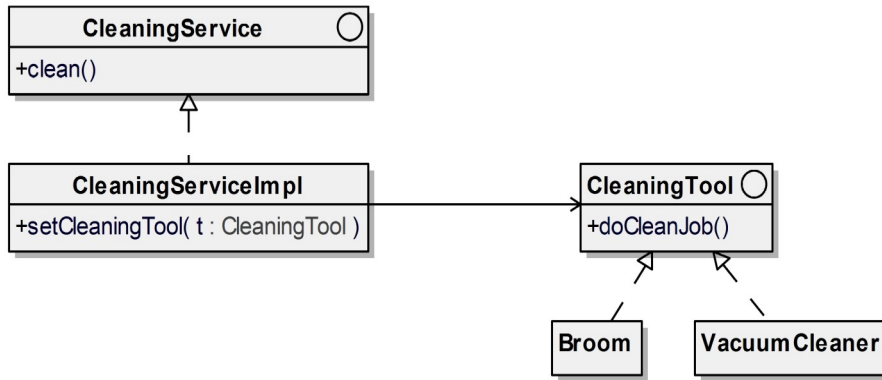
```

public class HouseApp {
    public static void main(String[] args) {
        CleaningService cleaningService = new CleaningService();
        cleaningService.clean();
    }
}
  
```

In dit voorbeeld hoeven we enkel een nieuw object van de klasse `CleaningService` te instantiëren en vervolgens te vragen zijn job te doen. De poetsdienst maakt voor zijn job gebruik van een bezem waar hij zelf voor zorgt; het object van de klasse `Broom` wordt door de `CleaningService` zelf geïnstantieerd.

We zouden hier kunnen zeggen dat de controle over het poetsgerei bij de poetsdienst ligt. Het gebruik van een borstel is als het ware ingebakken in de poetsdienst. In het huishouden wordt er echter toch iets meer flexibiliteit verwacht en zou een poetsdienst ook overweg moeten kunnen met ander poetsgerei zoals bijvoorbeeld een stofzuiger.

Een beter concept is daarom een poetsdienst die je om het even welk poetsgerei kan geven en die vervolgens zijn werk daarmee doet. Dit wordt weergegeven in het onderstaande klassendiagram:



Afbeelding 8: Klassendiagram met Inversion Of Control

We introduceren hier de interface `CleaningTool` waarin we het gedrag van poetsgerei beschrijven. Tevens voorzien we in de klassen `Broom` en `VacuumCleaner` die beide deze interface implementeren.

Ook van de `CleaningService` maken we een interface met implementerende klasse `CleaningServiceImpl`.

De `CleaningService` maakt nu gebruik van een object dat de interface `CleaningTool` implementeert om zijn job te kunnen doen. Welk soort poetsgerei gebruikt wordt, is niet meer van belang, zolang het maar aan de voorwaarden voldoet van poetsgerei; de interface dus. Zo zie je hoe polymorfisme ook in het huishouden heel wat flexibiliteit creëert.

Dergelijke poetsdienst kan je om het even welk poetsgerei in de handen duwen en die gaat daar vervolgens mee aan de slag. Het is dus niet meer de poetsdienst die beslist dat hij alleen maar met een borstel kan werken. De controle over het gereedschap ligt dus niet meer bij de poetsdienst, maar buiten hem; doorgaans de man of vrouw des huizes. Dit is dus *inversion of control*. Door dit principe is de poetsdienst dus veel flexibeler en kan voor meerdere taken ingezet worden. In softwaretermen kunnen we zeggen dat de klasse `CleaningService` beter herbruikbaar is; en dit is een belangrijk principe bij het ontwikkelen van software.

In feite hebben we hier voldaan aan het principe '*high cohesion and loose coupling*'. *High cohesion* wil zeggen dat ieder object verantwoordelijk is voor één kerntaak en *loose coupling* wil zeggen dat de objecten onderling losgekoppeld zijn en onder andere makkelijk uitwisselbaar zijn.

De opdrachtgever levert aan de poetsdienst dus de werkmiddelen. Dit kan op twee manieren gebeuren: ofwel door middel van een argument van de constructor ofwel door middel van een *setter*. Dit doorgeven van afhankelijke objecten noemt met *dependency injection*. Met andere woorden *inversion of control* wordt gerealiseerd door middel van *dependency injection*.

We nemen even de code erbij:

```
public interface CleaningTool {
    public void doCleanJob();
}
```

```
public class Broom implements CleaningTool {
    public void doCleanJob() {
```




```

        System.out.println("Scrub scrub");
    }
}

```

```

public class VacuumCleaner implements CleaningTool {
    public void doCleanJob() {
        System.out.println("Zuuuuuuuuuuuu");
    }
}

```

```

public interface CleaningService {
    public void clean();
}

```

```

public class CleaningServiceImpl implements CleaningService {
    private CleaningTool tool;

    public void setCleaningTool(CleaningTool tool) {
        this.tool = tool;
    }

    public void clean() {
        System.out.println("Cleaning the house");
        tool.doCleanJob();
    }
}

```

Het hoofdprogramma:

```

public class HouseApp {
    public static void main(String[] args) {
        CleaningTool broom = new Broom();
        CleaningTool vacuum = new VacuumCleaner();

        CleaningServiceImpl jill = new CleaningServiceImpl();
        jill.setCleaningTool(broom);
        CleaningServiceImpl bob = new CleaningServiceImpl();
        bob.setCleaningTool(vacuum);

        jill.clean();
        bob.clean();
    }
}

```

We hebben meteen een team ingehuurd waarbij de ene met de borstel en de andere met de stofzuiger aan de slag kan.

Opdracht 1: *Inversion Of Control*

In deze opdracht gaan we bovenstaand voorbeeld uitproberen.

- Maak een nieuw project in je IDE, bij voorkeur een *Maven*-project. Voorlopig hoeft *Spring* nog niet toegevoegd te worden aan het *classpath*, of als *dependency*. Gebruik de volgende POM:



```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>eu.noelvaes.spring</groupId>
  <artifactId>Spring01_01</artifactId>
  <version>5.1</version>
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>
      UTF-8
    </project.build.sourceEncoding>
  </properties>
  <build>
    <finalName>HouseKeeping</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
    </plugins>
  </build>
</project>
```

- Voeg een modulebeschrijving **module-info.java** toe:

```
module eu.noelvaes.spring {
}
```

- Maak de interfaces **CleaningTool** en **CleaningService** in het pakket `eu.noelvaes.housekeeping.services`.

```
package eu.noelvaes.housekeeping.services;

public interface CleaningTool {
    public void doCleanJob();
}
```

```
package eu.noelvaes.housekeeping.services;

public interface CleaningService {
    public void clean();
}
```

- Maak in hetzelfde pakket de klassen **Broom** en **VacuumCleaner** die beide de interface **CleaningTool** implementeren.

```
package eu.noelvaes.housekeeping.services;
```



```
public class Broom implements CleaningTool {
    public void doCleanJob() {
        System.out.println("Scrub scrub");
    }
}
```

```
package eu.noelvaes.housekeeping.services;

public class VacuumCleaner implements CleaningTool {
    public void doCleanJob() {
        System.out.println("Zuuuuuuuuuuuu");
    }
}
```

- **Optioneel: maak ook een klasse Sponge die de interface CleaningTool implementeert.**
- **Maak in hetzelfde pakket de klasse CleaningServiceImpl.**

```
package eu.noelvaes.housekeeping.services;

public class CleaningServiceImpl implements CleaningService {
    private CleaningTool tool;

    public void setCleaningTool(CleaningTool tool) {
        this.tool = tool;
    }

    public void clean() {
        System.out.println("Cleaning the house");
        tool.doCleanJob();
    }
}
```

- **Maak een hoofdprogramma HouseApp in het pakket eu.noelvaes.housekeeping waarin een aantal mensen aan het werk gezet worden om het huis schoon te maken, elk met hun eigen poetsgerei.**

```
package eu.noelvaes.housekeeping;

import eu.noelvaes.housekeeping.services.*;

public class HouseApp {
    public static void main(String[] args) {
        CleaningTool broom = new Broom();
        CleaningTool vacuum = new VacuumCleaner();
        CleaningTool sponge = new Sponge();

        CleaningServiceImpl jill = new CleaningServiceImpl();
        jill.setCleaningTool(broom);
        CleaningServiceImpl bob = new CleaningServiceImpl();
        bob.setCleaningTool(vacuum);
        CleaningServiceImpl jane = new CleaningServiceImpl();
    }
}
```



```

        jane.setCleaningTool(sponge);

        jill.clean();
        bob.clean();
        jane.clean();
    }
}

```

- Het resultaat zijn de volgende interfaces en klassen:

```

eu.noelvaes.housekeeping.HouseApp
eu.noelvaes.housekeeping.services.CleaningTool
eu.noelvaes.housekeeping.services.CleaningService
eu.noelvaes.housekeeping.services.Broom
eu.noelvaes.housekeeping.services.VacuumCleaner
eu.noelvaes.housekeeping.services.Sponge
eu.noelvaes.housekeeping.services.CleaningServiceImpl

```

1.3.2 BeanFactory en ApplicationContext

Door de *Inversion of Control* hebben we herbruikbare en herconfigureerbare componenten gemaakt. Bovendien hebben we gebruikgemaakt van de *JavaBean*-specificatie met zijn *getters* en *setters* om communicatie met de objecten op een gestandaardiseerde wijze te laten verlopen. Kort samengevat komt de *JavaBean*-specificatie hier op neer:

1. Objecten hebben minstens een constructor zonder argumenten zodat ze op makkelijke manier geïnstantieerd kunnen worden.
2. *Properties* (gegevens of afhankelijkheden) kunnen ingesteld en opgevraagd worden met *setters* en *getters*, ook wel *accessor methods* genoemd. Deze zien er als volgt uit:

```

public datatype getMyProperty()
public void setMyProperty(datatype prop)

```

De namen van de *properties* volgen de *camelcase*-notatie: dat wil zeggen dat ieder nieuw woord begint met een hoofdletter.

Objecten die aan deze voorwaarden voldoen, noemen we *JavaBeans* of ook gewoon *beans*.

De afhankelijkheden worden niet langer door de *bean* zelf gemaakt, maar worden van buiten aangereikt. We hebben daardoor enerzijds meer flexibiliteit bij het inzetten van de componenten maar anderzijds dienen deze afhankelijkheden wel eerst op de juiste wijze geïnjecteerd te worden alvorens we de *beans* kunnen gebruiken.

We zien dit in onze hoofdapplicatie:

```

CleaningTool broom = new Broom();
CleaningTool vacuum = new VacuumCleaner();
CleaningTool sponge = new Sponge();

CleaningServiceImpl jill = new CleaningServiceImpl();
jill.setCleaningTool(broom);
CleaningServiceImpl bob = new CleaningServiceImpl();
bob.setCleaningTool(vacuum);
CleaningServiceImpl jane = new CleaningServiceImpl();
jane.setCleaningTool(sponge);

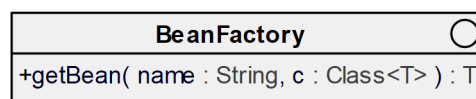
```



```
jill.clean();  
bob.clean();  
jane.clean();
```

We hebben met dit concept dus meer werk met het instantiëren en aaneenrijgen (*wiring*) van de *beans* alvorens we ze kunnen gebruiken. Deze acties moeten dus ook weer ergens in de code gebeuren. In ons voorbeeld heeft de opdrachtgever eerst heel wat werk voor hij zijn poetsdiensten aan het werk kan zetten. Dit kan uiteraard niet de bedoeling zijn.

En hier komt het *Spring framework* ons ter hulp. *Spring* neemt het instantiëren en aaneenrijgen van de *beans* voor zijn rekening en levert ons kant en klare *beans* af die onmiddellijk gebruikt kunnen worden. Dit alles wordt afgehandeld door de `BeanFactory`. Deze heeft een methode `getBean()` waarmee we zo een kant en klare *bean* kunnen opvragen.



Afbeelding 9: De *BeanFactory*

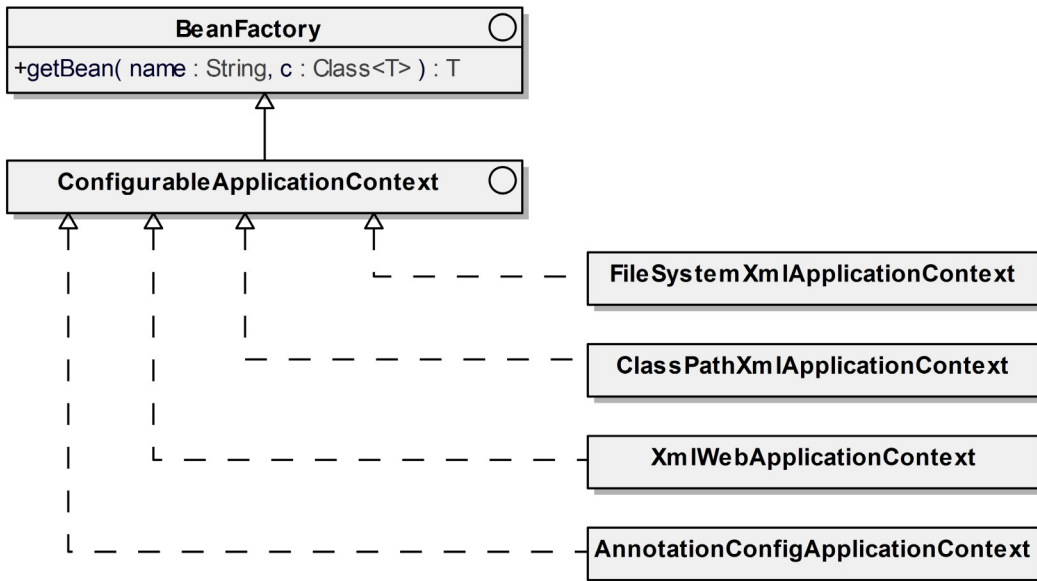
De `BeanFactory` leest de nodige configuratiegegevens uit een externe bron, instantieert de nodige *beans* en rijgt ze vervolgens op de juiste wijze aaneen. Deze `BeanFactory` is een implementatie van het **factory design pattern**.

Een speciale afgeleide interface van de `BeanFactory` is de `ConfigurableApplicationContext`. Deze biedt nog extra mogelijkheden waar we in een applicatie handig gebruik van kunnen maken:

- Het gebruik van internationalisatie (I18N).
- Het laden van allerlei *resources*.
- Het afvuren van *events* naar *beans*.

Doorgaans wordt steeds de `ConfigurableApplicationContext` in een applicatie gebruikt in plaats van de `BeanFactory`. We zullen later in de cursus de extra mogelijkheden hiervan onder de loep nemen.

De interface `ConfigurableApplicationContext` heeft veel implementaties. In het onderstaande schema geven we er enkele:



Afbeelding 10: BeanFactory en implementaties

De configuratiegegevens die de `ConfigurableApplicationContext` nodig heeft voor het instantiëren en aaneenrijgen van de *beans* kunnen afkomstig zijn van twee bronnen:

1. **XML:** De configuratiegegevens zijn beschreven in een XML-document. Dit kan eventueel uitgebreid worden met annotaties in de *bean*-klasse.
2. **Java-configuratie:** Sinds *Spring 3.0* kan de configuratie ook volledig gedaan worden in Java-objecten die voorzien zijn van annotaties.

In de bovenstaande afbeelding zien we vier implementaties van de interface `ConfigurableApplicationContext`:

Implementatie	Omschrijving
<code>FileSystemXmlApplicationContext</code>	De configuratie komt uit een XML -bestand dat zich op het bestandssysteem bevindt. We moeten hierbij het absolute of relatieve pad van het XML-bestand opgeven.
<code>ClassPathXmlApplicationContext</code>	De configuratie komt uit een XML -bestand dat zich in het <i>classpath</i> van de applicatie bevindt. We moeten hier het relatieve pad binnen het <i>classpath</i> opgeven. We kunnen hierdoor het XML-bestand in een JAR-bestand plaatsen.
<code>XmlWebApplicationContext</code>	De configuratie komt uit een XML -bestand dat zich in het WAR-bestand van een webapplicatie bevindt.
<code>AnnotationConfigApplicationContext</code>	De configuratie komt van Java-configuratie-objecten die voorzien zijn van annotaties.

Tabel 1: Implementaties van `ApplicationContext`

Aanvankelijk werd binnen *Spring* vooral gebruikgemaakt van XML. Het gebruik van XML heeft echter een aantal nadelen: fouten in de configuratie komen vaak pas bij de uitvoering (*at runtime*) aan het licht. Dat komt onder andere omdat klassennamen en namen van *properties* in de tekst van het XML-bestand gedefinieerd zijn. Indien er bijvoorbeeld een tikfout in zit, zal die niet door de compiler gedetecteerd worden maar zal die pas bij het



interpreteren van het XML-bestand ontdekt worden. Een tweede nadeel is dat we bij de XML-configuratie geen gebruik kunnen maken van bijvoorbeeld *code completion* om een *bean* te configureren. We moeten hier in de documentatie opzoeken wat de volledige naam van een klasse is en welke eigenschappen voorhanden zijn voor de configuratie.

Kortom, alle voordelen die we hebben met het louter werken binnen Java-code vallen weg: controle van de syntax door de compiler, *code completion*, koppeling met de JavaDoc enzovoort.

Om aan dit euvel tegemoet te komen is een alternatieve configuratie ontwikkeld die geen gebruikmaakt van XML maar waarbij de configuratie volledig met Java-code en annotaties gebeurt. Dit is de `AnnotationConfigApplicationContext`. Doorgaans spreekt men gewoon van *JavaConfig*.

Aangezien de laatste jaren *JavaConfig* meer gebruikt wordt dan XML, zullen we in deze cursus in eerste instantie gebruikmaken van deze op Java gebaseerde configuratie.

De instructies voor het aaneenrijgen van de *beans* worden beschreven in een afzonderlijke Java-klasse, de configuratieklasse, die voorzien is van een aantal annotaties.

We zullen dit alles illustreren aan de hand van ons huishoudvoorbeeld.

We maken een nieuwe klasse waarin we de instantiatie en configuratie van de verschillende *beans* onderbrengen:

```
@Configuration
public class AppConfig {

    @Bean
    public CleaningTool broom() {
        return new Broom();
    }

    @Bean
    public CleaningTool vacuum() {
        return new VacuumCleaner();
    }

    @Bean
    public CleaningTool sponge() {
        return new Sponge();
    }

    @Bean
    public CleaningService jill() {
        CleaningServiceImpl cs = new CleaningServiceImpl();
        cs.setCleaningTool(broom());
        return cs;
    }

    @Bean
    public CleaningService bob() {
        CleaningServiceImpl cs = new CleaningServiceImpl();
        cs.setCleaningTool(vacuum());
        return cs;
    }
}
```



```
@Bean
public CleaningService jane() {
    CleaningServiceImpl cs = new CleaningServiceImpl();
    cs.setCleaningTool(sponge());
    return cs;
}
}
```

Voor elke *bean* voorzien we in een methode die het object instantieert en de nodige afhankelijkheden injecteert.

Aan de configuratieklasse zelf voegen we de annotatie `@Configuration` toe en de methoden krijgen de annotatie `@Bean`.

We komen later nog uitvoerig terug op de syntax van deze configuratie.

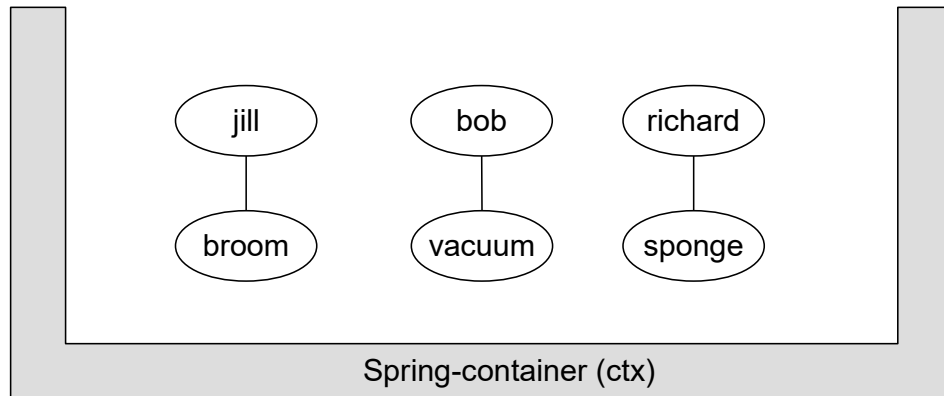
In ons hoofdprogramma maken we gebruik van de `AnnotationConfigApplicationContext`:

```
public class HouseApp {
    public static void main(String[] args) {
        ConfigurableApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppConfig.class);
        CleaningService jill =
            ctx.getBean("jill", CleaningService.class);
        CleaningService bob =
            ctx.getBean("bob", CleaningService.class);
        CleaningService jane =
            ctx.getBean("jane", CleaningService.class);
        jill.clean();
        bob.clean();
        jane.clean();
        ctx.close();
    }
}
```

Met de methode `getBean()` vragen we een *bean* op met een bepaalde naam. De *bean* die we terugkrijgen, is volledig geconfigureerd en klaar voor gebruik.

We zien hier dus dat *Spring* verantwoordelijk is voor het aanmaken en aaneenrijgen van de *beans*. Deze *beans* worden vervolgens toegevoegd aan een container. Nadien kunnen we in ons hoofdprogramma deze *beans* kunnen opvragen en gebruiken.

In de onderstaande afbeelding geven we een visuele voorstelling van de *Spring*-container met de zes *beans*:



Afbeelding 11: Spring-container met zes beans

Opdracht 2: Spring in het huishouden

In deze opdracht gaan we in het hoofdprogramma gebruikmaken van *Spring* voor het instantiëren en aaneenrijgen van de *beans*.

- Voeg *Spring* toe aan het project. Bij gebruik van *Maven* dient men de volgende *tags* toe te voegen (we maken hier gebruik van de BOM):

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.1.19.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
</dependencies>
```

- Koppel in je IDE ook de API-documentatie aan de JAR-bestanden van *Spring*. Deze documentatie is online beschikbaar op <http://docs.spring.io/spring/docs/current/javadoc-api/>. Je kan ook een koppeling maken met de lokale documentatie die onderdeel is van de distributie. Indien je gebruikmaakt van *Eclipse* kan je de documentatie ook automatisch binnenhalen en koppelen via **Maven->Download JavaDoc**. In *IntelliJ* open je het zijvenster voor *Maven* en selecteer je vervolgens **Download Documentation**.
- Voeg de volgende afhankelijkheden toe in de modulebeschrijving en open je eigen module voor reflectie:

```
open module eu.noelvaes.spring {
```