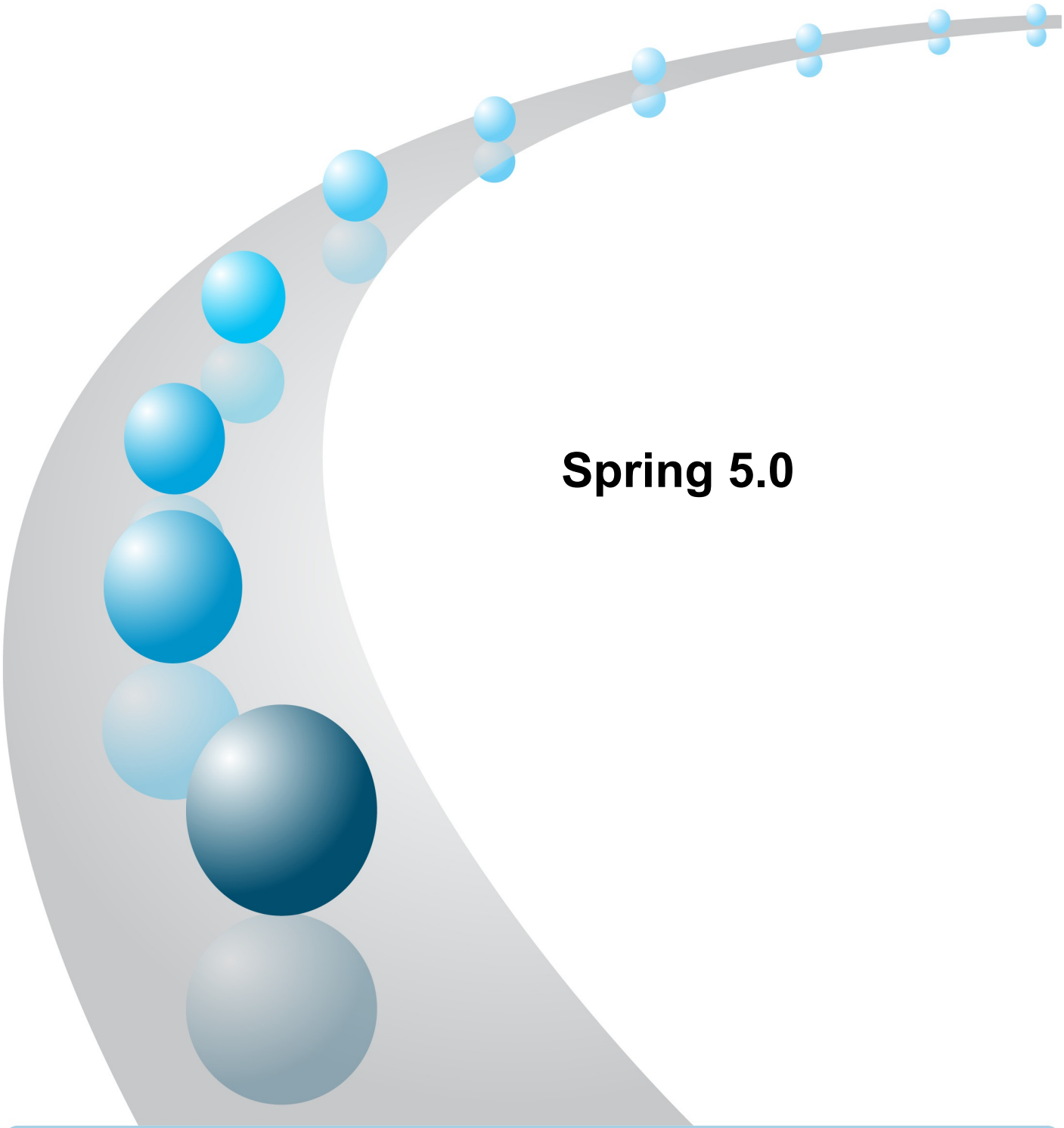
Noël Vaes

Java Trainer & Consultant



# Spring 5.0


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



14/08/2018

# Inhoudsopgave

# 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.

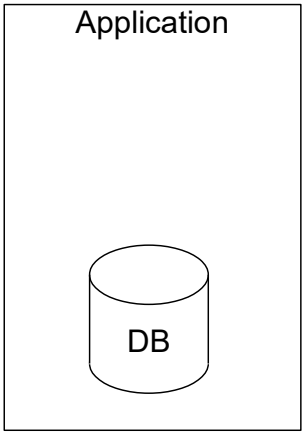
### 1.1.1 *Multitier* gedistribueerde applicaties

*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.



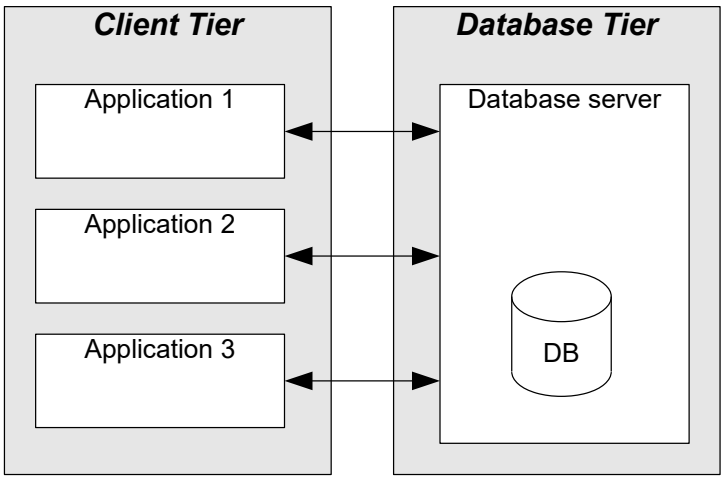

*Afbeelding 1: One-tier-applicatie*

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.

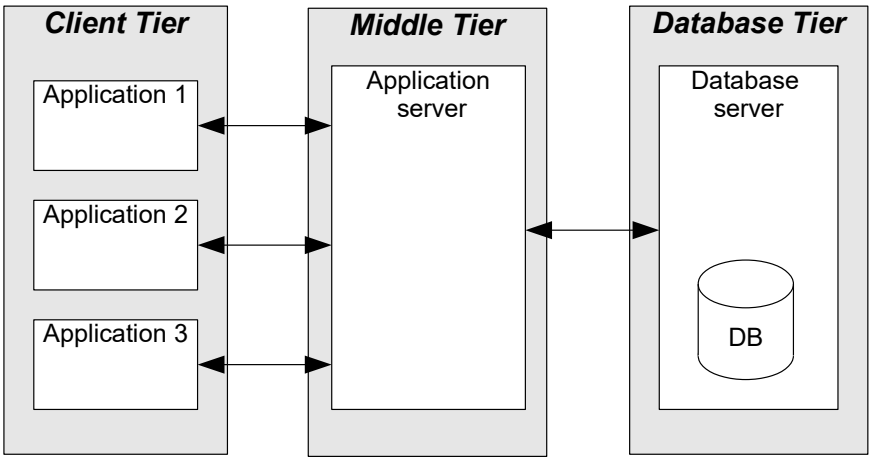
#### 1.1.1.3 *Three-tier-applicaties*

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. Vermits 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 applicatielogica 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 applicatielogica. Vermits de applicaties enkel nog de presentatielogica bevatten, zijn ze dus vaak erg afgeslankt. Men spreekt in dat geval ook wel van *thin clients*.

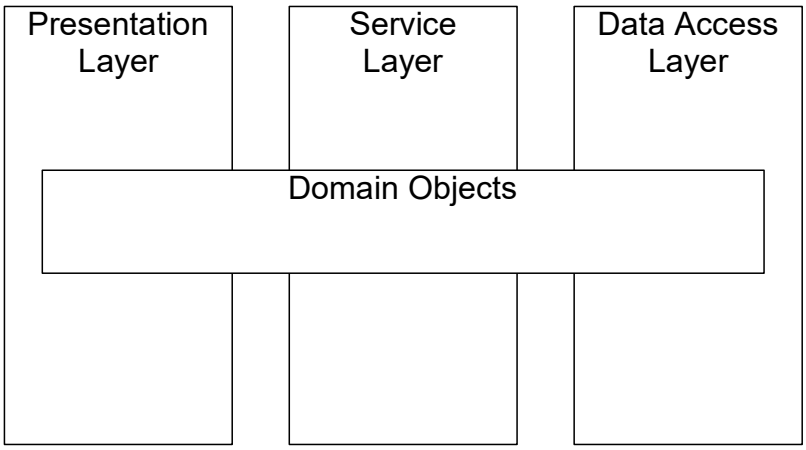
### 1.1.2 *Layers* of logische lagen

*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 applicatie:

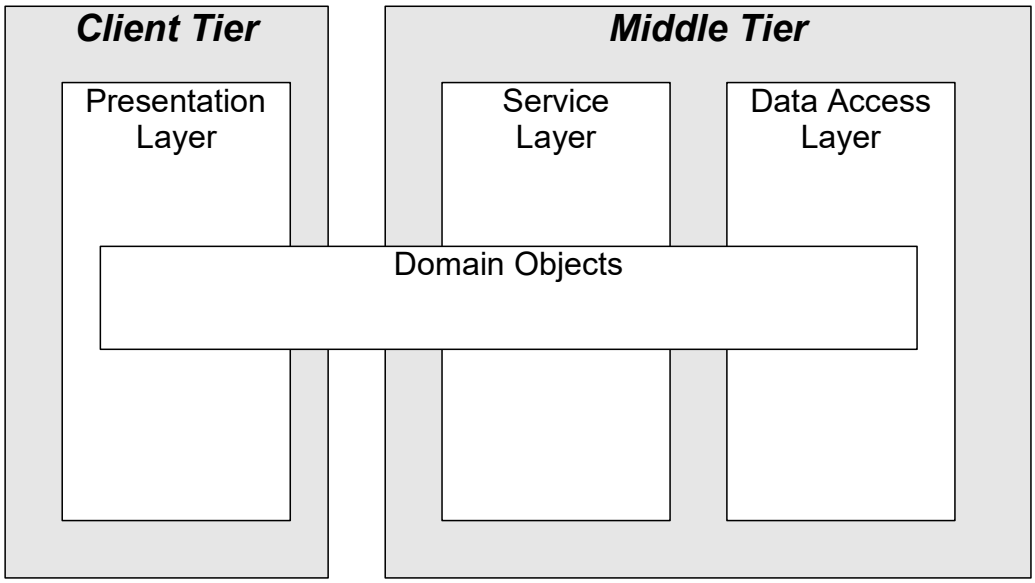
1. ***Data Access Layer***: Deze laag is verantwoordelijk voor de communicatie met de databank. De toegang tot de databank is hier gecentraliseerd en afgescheiden van de rest.
2. ***Service Layer***: In deze laag wordt de *business logic* uitgevoerd. Deze bestaat uit allerlei diensten (*services*) ten 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 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.

In de onderstaande afbeelding geven we de verschillende lagen weer:




*Afbeelding 4: De logische lagen in een applicatie.*

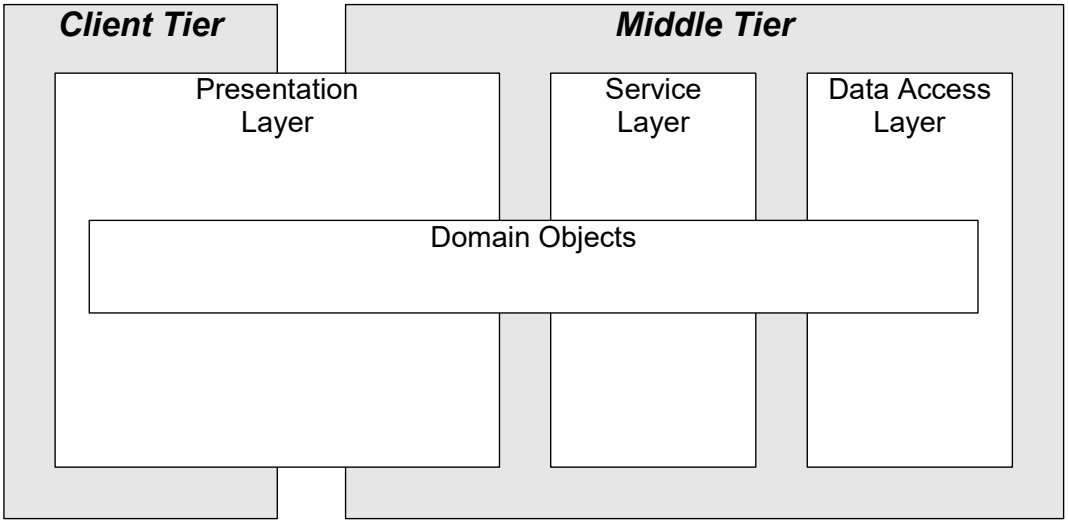
Deze lagen kunnen zich nu op verschillende fysieke lagen bevinden. In de onderstaande afbeelding bevindt de *Presentation Layer* zich volledig op de *client tier*. Dit zou het geval kunnen zijn indien de grafische gebruikersinterface ontworpen is als een toepassing in Swing of Java-FX.




*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 teruggegaan 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 de *Spring*-distributie en moeten daarom ook afzonderlijk gedownload worden en toegevoegd worden aan het *classpath.*

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>5.0.7.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>5.0.7.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*

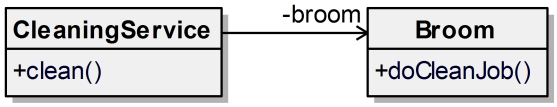
Zoals we in onze inleiding al vertelden is *Spring* een *framework* om toepassingen te maken met herbruikbare en configureerbare componenten op basis van *plain old Java objects (POJO's)*. Met *Spring* kunnen we dergelijke objecten, of ook wel *beans* genoemd, aaneenrijgen tot een werkend geheel. Dit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