



Noël Vaes

Java Trainer & Consultant



CDI 1.1

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.

20/09/2018

Copyright© 2018 Noël Vaes



Inhoudsopgave

Hoofdstuk 1: Inleiding.....	3
1.1. Inversion of Control en Dependency Injection.....	3
1.2. Contextual lifecycle management.....	4
Hoofdstuk 2: Beans.....	5
2.1. Managed Beans.....	5
2.2. Containers.....	5
2.3. Mijn eerste CDI-bean.....	6
2.4. Named beans en expression language.....	9
2.5. CDI-beans en JavaServer Faces.....	10
2.6. De levenscyclus van een bean.....	11
Hoofdstuk 3: Contexten en scopes.....	13
3.1. Inleiding.....	13
3.2. Dependent Scope.....	14
3.3. Request Scope.....	16
3.4. Session Scope.....	16
3.5. Conversation Scope.....	18
3.6. Application Scope.....	21
3.7. Singleton Scope.....	22
3.8. Samenspel tussen scopes.....	22
Hoofdstuk 4: Dependency Injection.....	28
4.1. Inversion of Control.....	28
4.2. Dependency Injection met CDI.....	31
4.3. Datatypes.....	33
4.4. Qualifiers.....	34
4.4.1. Eigen qualifiers maken.....	35
4.4.2. Ingebouwde qualifiers @Default en @Any.....	36
4.4.3. Qualifiers met eigenschappen.....	37
4.5. Alternatieven.....	38
4.6. Specialization.....	39
4.7. Proxies.....	39
4.8. Programmatisch opzoeken van een bean.....	39
4.9. InjectionPoint.....	41
4.10. Stereotypes.....	41
Hoofdstuk 5: Design patterns.....	43
5.1. Inleiding.....	43
5.2. Factory: producers.....	43
5.2.1. Producer-methoden.....	43
5.2.2. Injectie in de producer-methode.....	44
5.2.3. Disposer-methoden.....	45
5.2.4. Producer-fields.....	46
5.3. Observer-observable: event handling.....	47
5.3.1. Het event-object.....	47
5.3.2. Events ontvangen.....	48
5.3.3. Events versturen.....	48
5.3.4. Events met kwalificaties.....	49
5.3.5. Transactionele events.....	51
5.4. Interceptors.....	52
5.4.1. Inleiding.....	52
5.4.2. De interceptor-annotatie.....	52
5.4.3. De interceptor-klasse.....	52



5.4.4. Interceptors koppelen aan klassen en methoden.....	53
5.4.5. Interceptors activeren.....	54
5.4.6. @Transactional.....	55
5.5. Decorators.....	55
5.5.1. Inleiding.....	55
5.5.2. De decorator-klasse.....	55
5.5.3. Decorators activeren.....	56
Hoofdstuk 6: CDI en JEE.....	58
6.1. Inleiding.....	58
6.2. Resource injections in CDI-beans.....	59
6.3. Resource injection met CDI-factories.....	59
6.4. Injectie van CDI-beans in JEE-componenten.....	60
6.5. EJB versus CDI.....	61



Hoofdstuk 1: Inleiding

CDI is de afkorting van *Contexts and Dependency Injection*. Voorheen was deze technologie ook gekend als *Web Beans*.

CDI is geen echt nieuwe technologie. Het is een standaardisatie van technologieën die buiten de JEE-standaard ontwikkeld werden. Het gaat onder andere om het concept van ***Inversion of Control (IOC)*** en ***Dependency Injection (DI)*** dat vooral in het *Spring-framework* toegepast wordt. Maar ook *frameworks* als *Google Guice* en *Seam* staan aan de wieg van CDI.

Daarnaast biedt CDI ook **contextueel beheer van de levenscyclus** van een object.

Een gedetailleerde beschrijving van beide concepten volgt later in de cursus maar we kunnen hier alvast een summiere introductie geven.

1.1. *Inversion of Control en Dependency Injection*

Bij object georiënteerde programmeertalen bestaat software uit een aantal objecten die gebruikmaken van elkaars diensten. Belangrijk hierbij is dat de objecten ontwikkeld worden volgens het principe ***Loose coupling and high cohesion***.

Objecten dienen een kerntaak te vervullen en zich niet bezig te houden met nevenactiviteiten. Dit maakt ze meer geschikt voor hergebruik in andere omstandigheden. We noemen dit *high cohesion*.

Voor de taken die niet tot de kerntaak van een object behoren dient het object beroep te doen op andere objecten. Er moet dus op de een of andere manier een koppeling zijn tussen verschillende objecten. Praktisch kan dit gerealiseerd worden doordat een object een referentie heeft naar een ander object en dit andere object zelf instantieert. Dit is echter een vaste koppeling (*tight coupling*) en biedt weinig flexibiliteit. Het is niet mogelijk het gekoppelde object later op een eenvoudige manier door iets anders te vervangen. De afhankelijkheid tussen de twee objecten is intrinsiek. We zouden dit evenwel kunnen omdraaien en deze afhankelijkheid van buiten aanreiken. Dit noemt men *Inversion of Control (IOC)*. De controle over de afhankelijkheid ligt dan niet langer binnen het object zelf maar wordt extern geregeld. Ze wordt van buitenaf toegewezen. Praktisch gebeurt dit door de referentie naar het andere object extern in te stellen. Dit noemt men *Dependency Injection (DI)*.

Indien we in de afhankelijkheid verder gebruikmaken van polymorfisme via interfaces of abstracte klassen is de koppeling nog lossier. Dit alles resulteert in de vereiste *loose coupling*.

Frameworks als *Spring* zijn op dit principe van *Inversion of Control* gebaseerd aangevuld met *Aspect Oriented Programming (AOP)* om *cross cutting concerns* te integreren.

In JEE5 werden reeds een aantal van deze technieken opgenomen. Zo was het onder andere mogelijk allerlei *resources* in EJB's en webcomponenten te injecteren door middel van annotaties. Tevens konden EJB's onderling via eenvoudige annotaties geïnjecteerd worden. Het AOP-verhaal kreeg tevens een gedeeltelijk equivalent door het gebruik van *interceptors*.

In JEE6 worden deze concepten nog verder uitgewerkt in de CDI-specificatie. Dit is eigenlijk een ecosysteem waarbij *beans* gecreëerd en aaneengeregen (*wiring*) worden door middel van *Dependency Injection*. Deze *beans* slaan bovendien de brug tussen de *managed beans*



van *Java Service Faces (JSF)* in de presentatielaag en de EJB's in de *business*-laag. Dit is een hele mond vol en daarom zullen we in deze cursus de verschillende concepten stap voor stap introduceren en illustreren met praktische voorbeelden.

1.2. Contextual lifecycle management

CDI zorgt voor het instantiëren, aaneenrijgen en nadien opruimen van objecten of *beans*. De hele levenscyclus van een *bean* wordt geregeld door de CDI-container. Maar wanneer start deze levenscyclus en wanneer eindigt deze? Dat hangt af van de context waarin een object nodig is. Sommige objecten zijn nodig gedurende de hele tijd dat de applicatie actief is, andere slechts gedurende de interactie met een gebruiker.

CDI zorgt ervoor dat de levenscyclus van een *bean* doorlopen wordt naargelang de gevraagde omstandigheden of *context*. Men noemt dit daarom *contextual lifecycle management*.

Bij een interactie met de eindgebruiker kunnen er zowel globale *beans* als gebruikersgerichte *beans* in het spel zijn. CDI zorgt er steeds voor dat deze vlekkeloos met elkaar kunnen samenwerken door de *beans* op het gepaste moment te instantiëren, te koppelen aan andere *beans* en ze nadien op te ruimen.



Hoofdstuk 2: Beans

2.1. Managed Beans

Binnen CDI wordt er steeds gesproken van *beans*. Het woord *bean* heeft binnen de Java-wereld echter al een hele geschiedenis achter de rug. Zo kennen we de oeroude *JavaBeans*, de *JSF managed beans* en de *Enterprise JavaBeans (EJB)*, maar een uniforme en eenduidige definitie bestond niet echt. CDI tracht binnen het JEE-platform de definitie van een *bean* te consolideren in een duidelijk afgebakend begrip.

Het basisbegrip is de *managed bean*. Dit zijn gewoon Java-objecten waarvan de levenscyclus door een container beheerd (*managed*) wordt. Als we spreken over 'gewoon Java-object' dan wil dat zeggen dat deze objecten niet aan allerlei speciale voorwaarden moeten voldoen. Dergelijke objecten worden wel eens aangeduid met het acroniem POJO: *Plain Old Java Object*.

Managed beans dienen de volgende mogelijkheden te hebben:

- *lifecycle callbacks*.
- *interceptors*.
- *resource injection*.

Van dit soort *managed beans* zijn er specialisaties die extra mogelijkheden toevoegen. Zo is een EJB een *managed bean* die tevens beveiliging en transactioneel gedrag toevoegt.

Een CDI-*bean* daarentegen is een *managed bean* die *contextual lifecycle management* toevoegt.

De voorwaarden voor een dergelijke CDI-*bean* zijn miniem: ieder object met een standaardconstructor wordt beschouwd als een *bean*, op een aantal uitzonderingen na dan.

Voor de volledigheid geven we hier de lijst van voorwaarden waaraan voldaan moet worden:

- De klasse moet een **constructor zonder argumenten** hebben ofwel een **constructor met geïnjecteerde argumenten**. Dit laatste zullen we later in de cursus behandelen.
- De klasse mag **geen geneste klasse** zijn, een *static inner class* mag wel.
- Het moet een **concrete klasse** zijn, met uitzondering van een *decorator* voorzien van de annotatie `@Decorator`.
- Het mag **geen EJB** zijn. Deze vormen namelijk een afzonderlijke categorie. We wijden hier overigens een afzonderlijke paragraaf aan.
- De klasse mag de interface `javax.enterprise.inject.spi.Extension` niet implementeren.

Geef toe, de voorwaarden zijn heel ruim en de meeste objecten die we kennen, komen in aanmerking.

2.2. Containers

Een *bean* is een object dat door een container beheerd wordt. Concreet wil dit zeggen dat de volledige levenscyclus door een containerapplicatie geregeld wordt: de instantiatie, initialisatie en opruiming.

Het is in dit programmeermodel niet langer de programmeur die een nieuw object maakt door de constructor op te roepen: objecten worden door de container gemaakt en ter beschikking gesteld van andere componenten.



Als we hier spreken over 'container' dan is dat doorgaans de JEE-applicatieserver: dit is namelijk de natuurlijke habitat van CDI-*beans*. Maar CDI is evenwel niet beperkt tot deze omgeving. Het is mogelijk binnen een gewone *standalone*-applicatie gebruik te maken van een CDI-container. Dit kan bijvoorbeeld door gebruik te maken van de referentie-implementatie *Weld*. We verwijzen hiervoor naar website van *Weld*: <http://weld.cdi-spec.org>.

Applicatieservers die voldoen aan de JEE6-standaard of hoger dienen een implementatie van CDI aan boord te hebben. En dit geldt zowel voor het *full profile* als het *web profile*.

In deze cursus laten we het gebruik van CDI buiten JEE buiten beschouwing. Wij zullen gebruikmaken van een JEE7-applicatieserver, meer concreet *WildFly 13*. Alle voorbeelden kunnen evenwel ook uitgevoerd worden op om het even welke andere JEE-server vanaf versie 6. Webcontainers zoals *Tomcat* voldoen niet volledig aan de JEE-standaard en hebben momenteel geen CDI aan boord. Ze kunnen evenwel wel met CDI uitgebreid worden. Hiervoor verwijzen we naar de uitgebreide documentatie van *Weld*.

Vermits CDI geïntegreerd is in een JEE-applicatieserver, dienen we hier geen extra bibliotheken toe te voegen. We moeten wel CDI activeren. Deze activatie kan op drie manieren gebeuren:

1. Door het bestand ***beans.xml*** toe te voegen aan een JAR- of WAR-bestand in de respectievelijke mappen ***META-INF*** of ***WEB-INF***. Bij de aanwezigheid van dit bestand zal CDI automatisch actief worden voor de *beans* die in het archief aanwezig zijn.
2. Door *beans* te voorzien van een ***scope-annotatie***. Dit is een specifieke CDI-annotatie waarmee we het werkingsgebied van een *bean* instellen. Het gebruik van zo'n annotatie geeft dus tevens aan dat we CDI willen gebruiken.
3. Bij de aanwezigheid van EJB's. Zodra EJB's in een archief aanwezig zijn, zal ook automatisch CDI geactiveerd worden.

Het XML-bestand ziet er als volgt uit:

beans.xml

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.1" bean-discovery-mode="all">
</beans>
```

Merk alvast het attribuut `bean-discovery-mode` op. We stellen de waarde hiervan in op `all` zodat alle objecten in aanmerking komen als *bean*. De voorkeurswaarde in CDI 1.1 (JEE7) is evenwel `annotated` zodat enkel *beans* die voorzien zijn van een annotatie in aanmerking komen. Eventueel kan men bepaalde klassen uitsluiten met de annotatie `@Vetoed`.

CDI maakt vooral gebruik van annotaties. Slechts in bepaalde gevallen zullen we extra gegevens toevoegen aan het XML-bestand. Doorgaans blijft dit evenwel leeg en dient het enkel om CDI te activeren voor een bepaald archief.

2.3. Mijn eerste CDI-bean

Tijd om de handen uit de mouwen te steken en onze eerste CDI-*bean* tot leven te wekken.

Ook hier maken we de zoveelste variant van de obligate "Hello World".



De *bean* ziet er als volgt uit:

```
public class HelloBean {
    public String sayHello() {
        return "Hello World";
    }
}
```

Niet bijzonder dus. Een gewone klasse met standaardconstructor en een publieke methode. Ook extra annotaties zijn hier niet nodig om deze klasse tot *bean* te maken.

We kunnen deze *bean* nu gebruiken in een webapplicatie met bijvoorbeeld een *servlet*:

```
@WebServlet("/Hello")
public class HelloServlet extends HttpServlet {
    @Inject
    private HelloBean bean;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        Writer out = response.getWriter();
        out.write("<html><head><title>Hello</title></head><body>");
        out.write(bean.sayHello());
        out.write("</body></html>");
    }
}
```

De *servlet* heeft een instantie van de *bean* nodig om zijn werk te kunnen doen. In plaats van deze zelf te instantiëren, wordt dit overgelaten aan de CDI-container. Deze zal een instantie maken en in het veld injecteren. We geven dit aan met de annotatie `@Inject`.

We merken op dat een *servlet* zelf omwille van zijn bijzondere functie niet als *bean* beschouwd wordt, maar het is wel mogelijk *beans* te gebruiken binnen de *servlet*. Hetzelfde geldt voor andere componenten als *stateless session beans* en *message driven beans*. We komen hier later nog op terug.

Opdracht 1: Mijn eerste CDI-bean

In deze opdracht gaan we een webproject maken met een *servlet* die gebruikmaakt van een *bean*. We gaan er hierbij van uit dat *WildFly* en *Maven* reeds geïnstalleerd zijn.

- Maak een nieuw *Maven*-project met 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</groupId>
    <artifactId>cdi</artifactId>
    <version>1.1</version>
    <packaging>war</packaging>

    <properties>
```



```

    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>
        UTF-8
    </project.build.sourceEncoding>
</properties>

<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <!-- This plugin version no longer requires web.xml -->
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.2</version>
        </plugin>
        <plugin>
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-maven-plugin</artifactId>
            <version>1.2.2.Final</version>
        </plugin>
    </plugins>
</build>
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
</project>

```

- Voeg het bestand **beans.xml** toe in de map **/webapp/WEB-INF**:

```

<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
version="1.1" bean-discovery-mode="all">

</beans>

```

- Voeg de **bean**-klasse toe in een pakket naar keuze:

```

public class HelloBean {
    public String sayHello() {
        return "Hello World";
    }
}

```

- Voeg ten slotte de **Servlet**-klasse toe:

```

@WebServlet("/Hello")
public class HelloServlet extends HttpServlet {

```