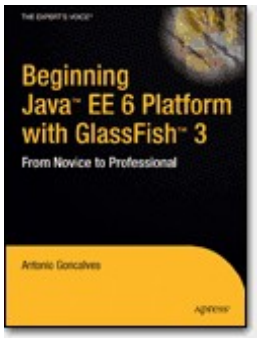
To inject or not to inject : CDI is the question

by antonio goncalves

Welcome to a type-safe injection journey

Antonio Goncalves

- Freelance software architect
- Author (Java EE 5 and Java EE 6)
- JCP expert member (Java EE 6, Java EE 7)
- Co-leader of the Paris JUG
- Les Cast Codeurs podcast
- Java Champion

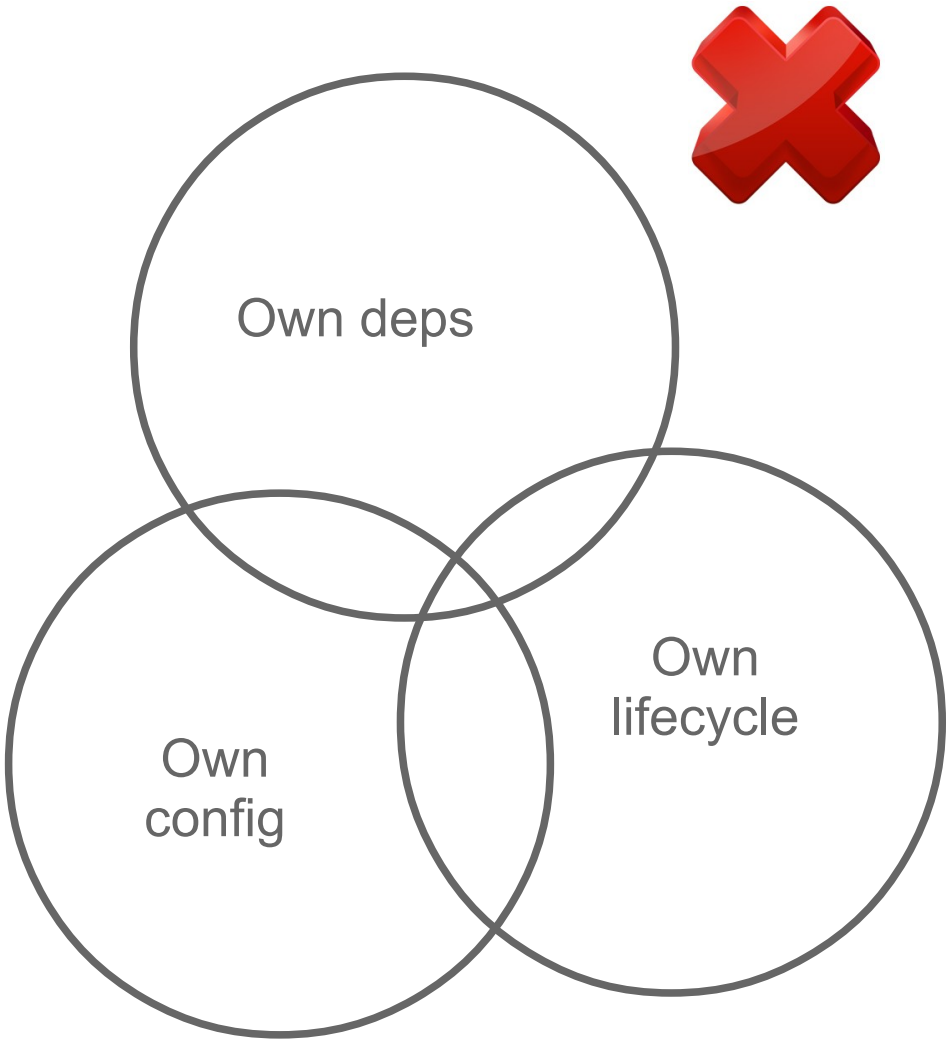


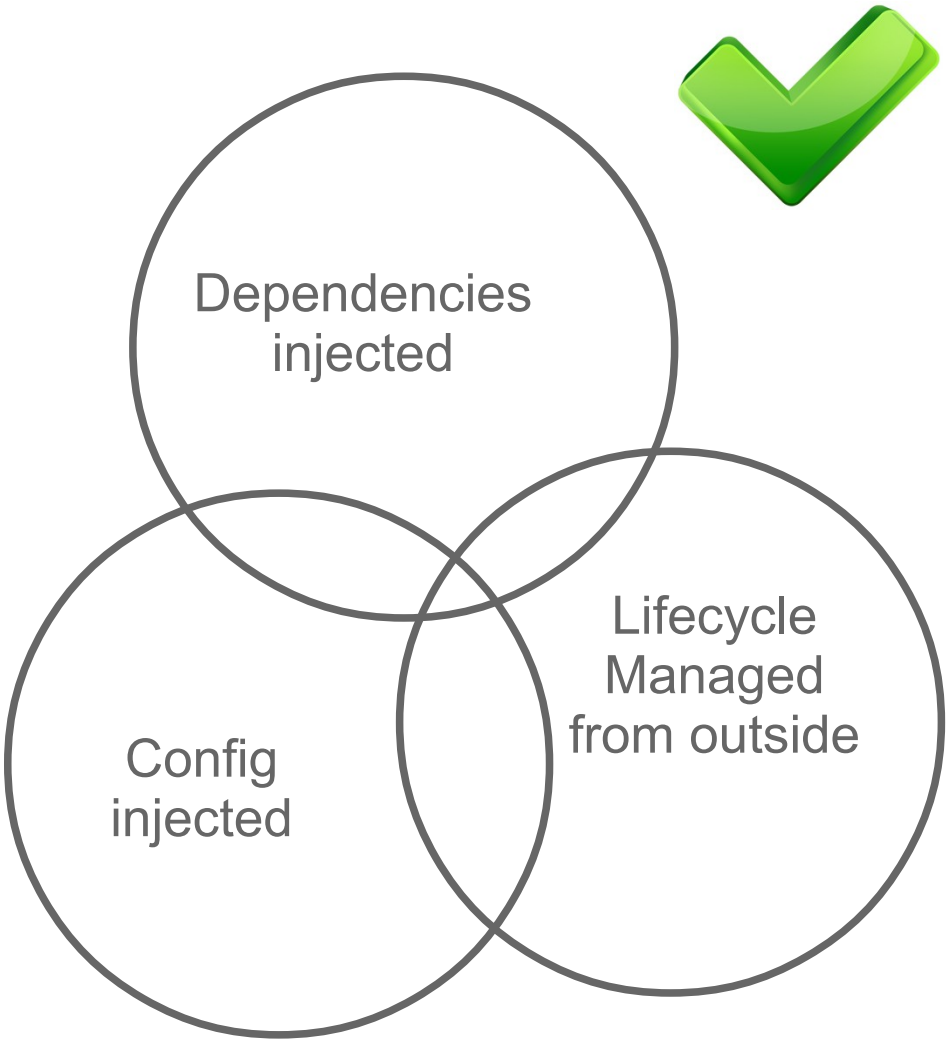
Summary

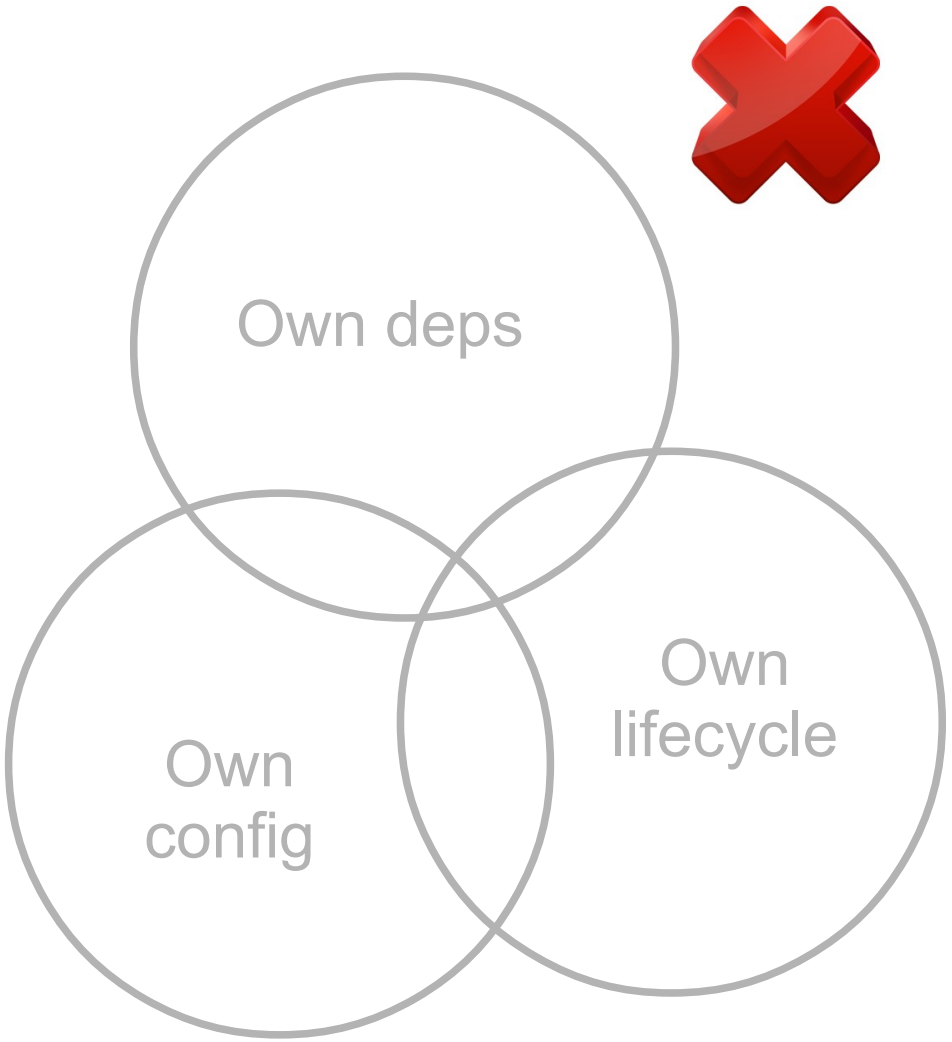
- IoC
- Dependency Injection
- @Inject & CDI
- Interceptors, Decorators, Events

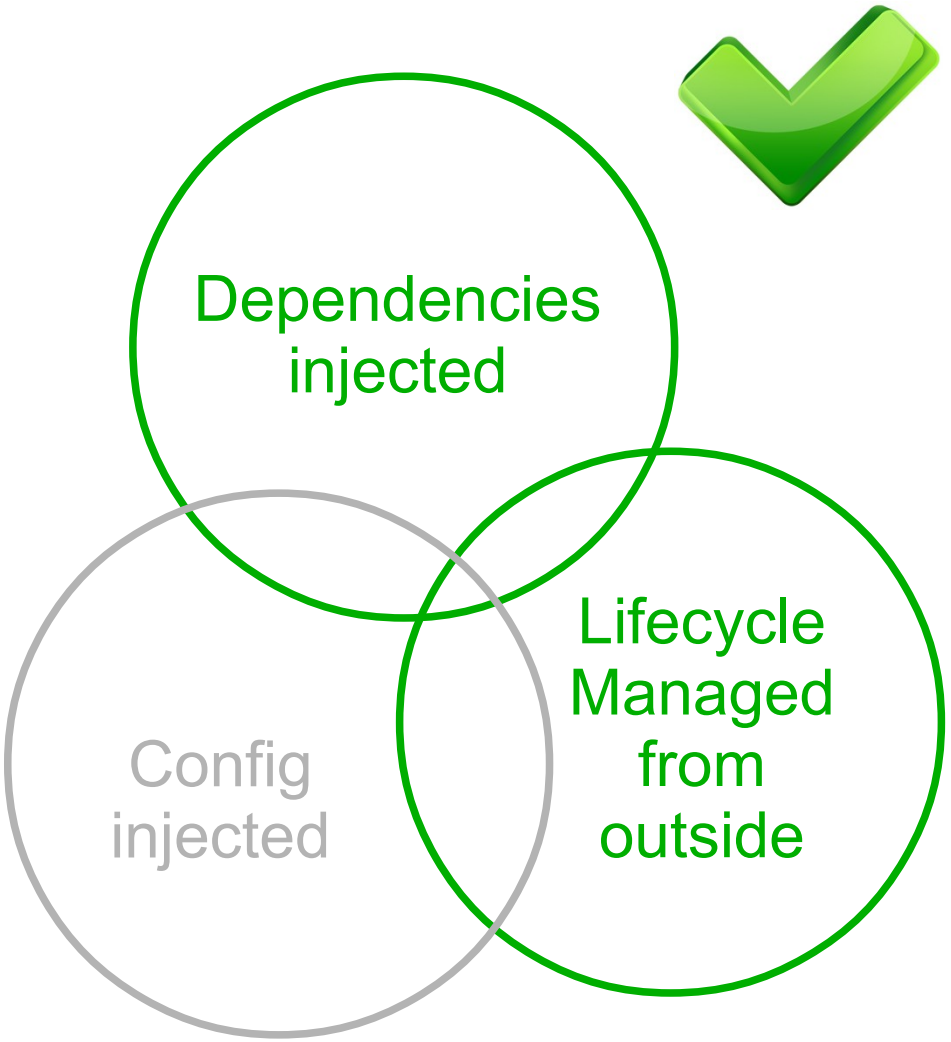
Inversion of control

- Term popilarised in 1998
- The control of :
  - dependency resolution (DI)
  - lifecycle
  - configuration
- Is given to an external component (eg. container)
- ...not the component itself
- It brings loose coupling

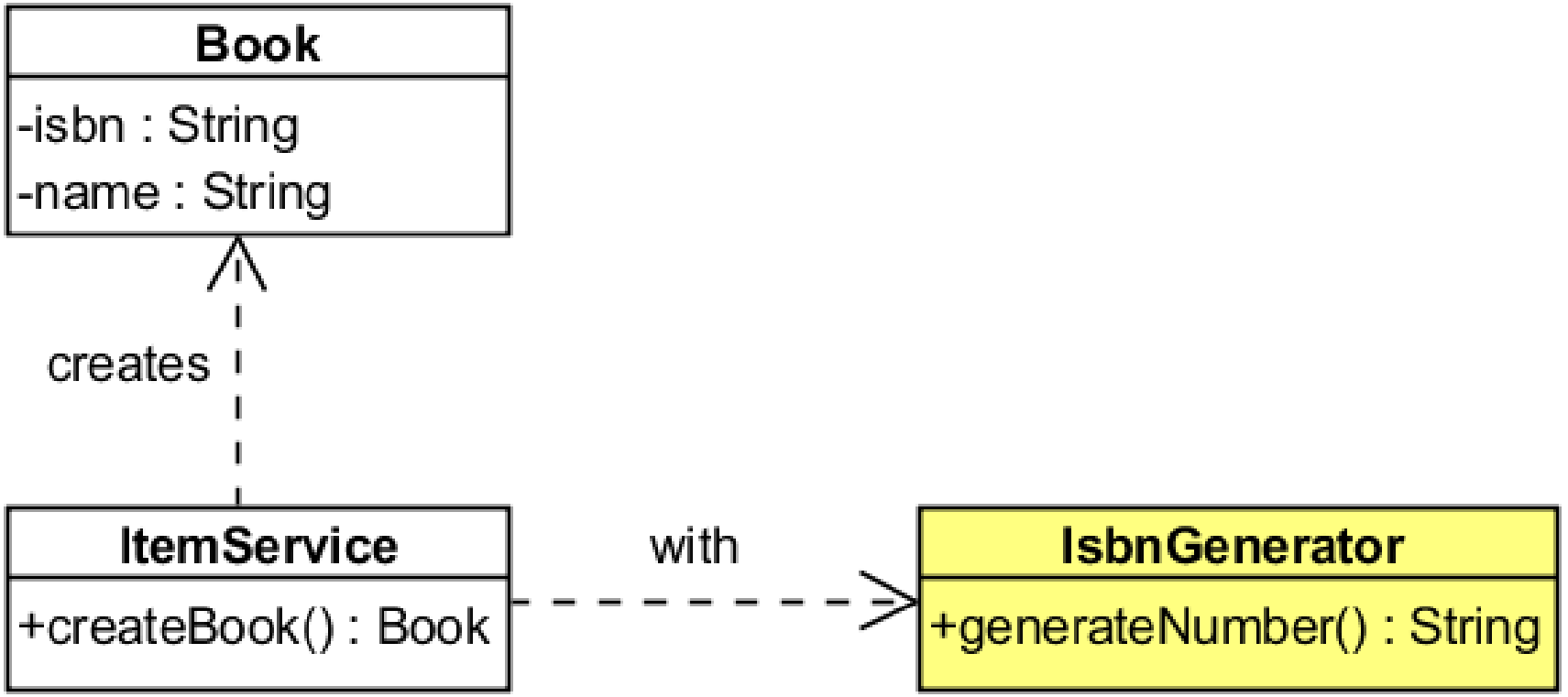

Own deps
Own config
Own lifecycle


Dependencies injected
Config injected
Lifecycle Managed from outside


Own deps
Own config
Own lifecycle


Dependencies injected
Config injected
Lifecycle Managed from outside

Example of dependency



The good old `new`

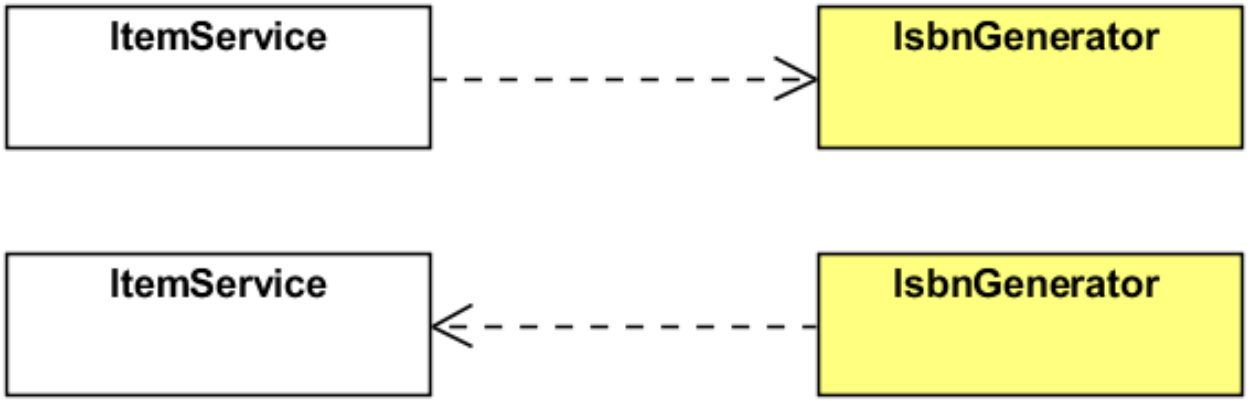
```
public class ItemService {
  private IsbnGenerator isbnGenerator;

  public ItemService() {
    this.isbnGenerator = new IsbnGenerator();
  }

  public Book createBook(Book book) {
    ...
    book.setIsbn(isbnGenerator.generateNumber());
  }
}
```

What's wrong with that ?

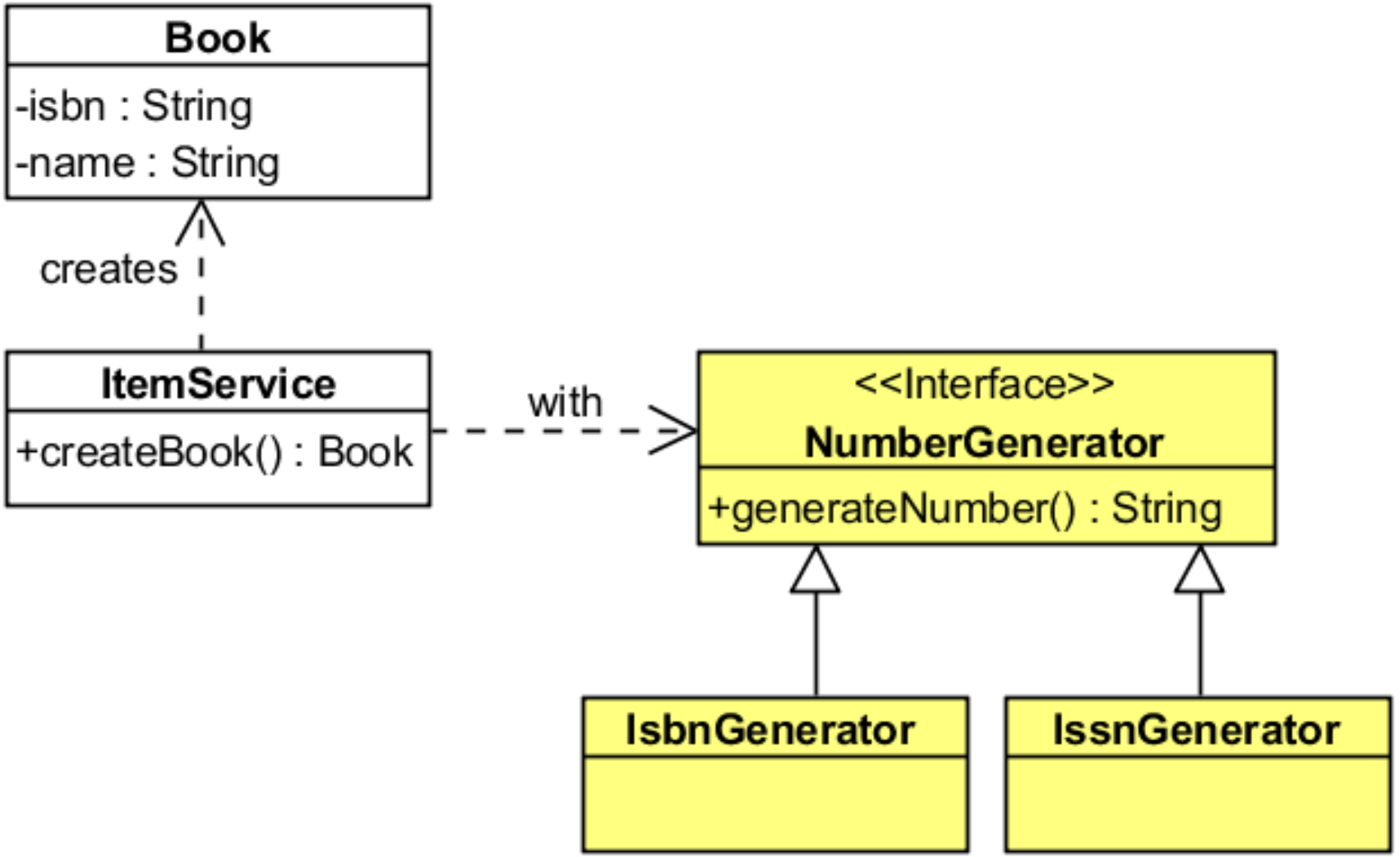
- Strong coupling
  - Impossible to change implementations
  - Impossible to MOCK if needed
- Lifecycle done by the component
  - Sometimes `new()` is not enough
  - Creating an instance, opening, closing

That's why we need DI

- Design pattern (coined by Martin Fowler)
- Decouples dependent components
  - Loose coupling
- Hollywood Principle
  - "don’t call us, we’ll call you!"



How to choose implementation?



Constructor (or setter) injection

```
public class ItemService {
  private NumberGenerator numberGenerator;

  public ItemService(NumberGenerator numberGenerator) {
    this.numberGenerator = numberGenerator;
  }

  public Book createBook(Book book) {
    ...
    book.setIsbn(numberGenerator.generateNumber());
  }
}
```

With the good old `new` again

```
// With constructor injection
ItemService itemService = new ItemService(
                                  new IsbnGenerator());
ItemService itemService = new ItemService(
                                  new IssnGenerator());


// With setter injection
ItemService itemService = new ItemService();
itemService.setNumberGenerator(new IsbnGenerator());
itemService.setNumberGenerator(new IssnGenerator());
```

Using factories

- Graph dependency can be complex
- Factory design pattern (GoF)
- Everywhere in Java
  - `java.util.Calendar#getInstance()`
  - `java.util.Arrays#asList()`
  - `java.sql.DriverManager#getConnection()`
  - `java.lang.Class#newInstance()`
  - `java.lang.Integer#valueOf()`

With a Factory

```
public class ItemServiceFactory {


  public ItemService newIsbnGenerator() {
    return new ItemService(new IsbnGenerator());
  }
  public ItemService newIssnGenerator() {
    return new ItemService(new IssnGenerator());
  }
}


// Client
ItemService itemService =
        new ItemServiceFactory().newIsbnGenerator();
```

Another pattern: Service Locator

- J2EE Design pattern
- Used to find services
- May reside in the same application, machine or network
- JNDI is a perfect service locator

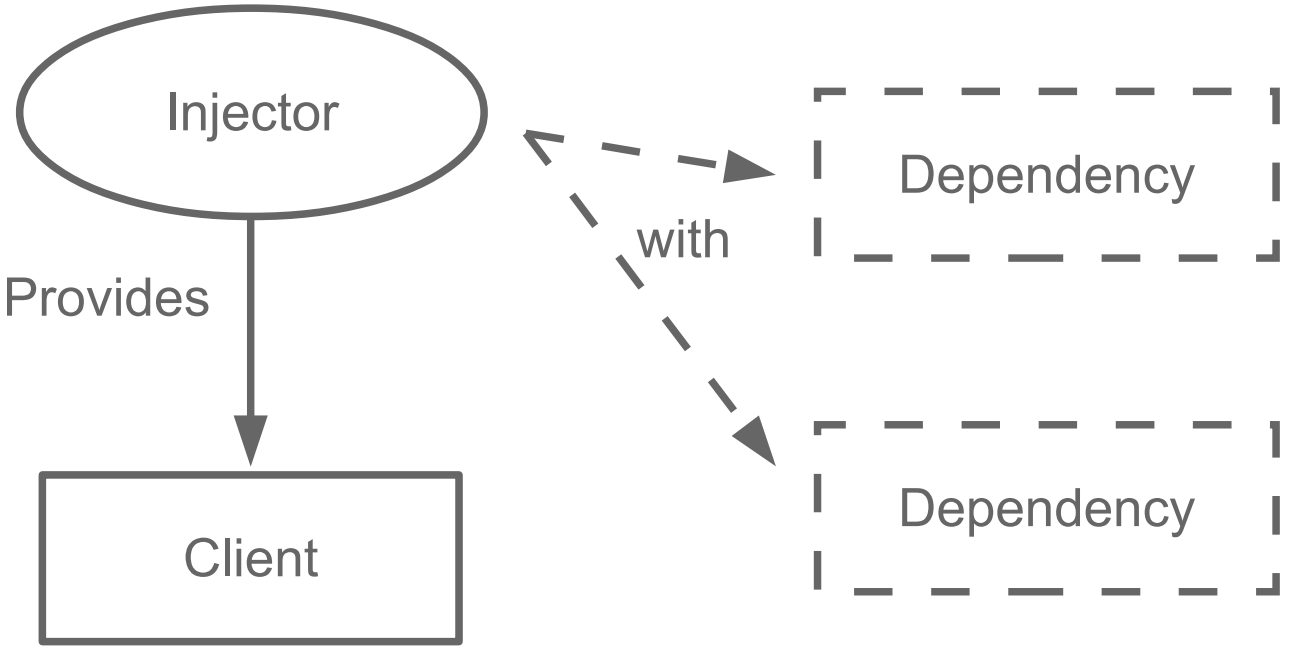
```
ItemService itemService =
        new ServiceLocator().get("IsbnGeneratorService");
```

All that is constructing by hand

```
// new()
public ItemService() {
  this.isbnGenerator = new IsbnGenerator();
}

// Factory
ItemService itemService =
        new ItemServiceFactory().newIsbnGenerator();

// Service locator
ItemService itemService =
        new ServiceLocator().get("IsbnGeneratorService");
```

Give control to an injector

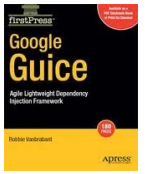
- Injector aka container aka provider
- Creating, assembling and wiring done by an external framework



Dependency injector

- *Apache Avalon*
- Spring framework
- Pico container
- Nano container
- Apache Hivemind
- Seam
- Google Guice
- Contexts and Dependency Injection (CDI)

A bit of history



Project JPE

May 1998

J2EE 1.2

Servlet/JSP
EJB
JMS
RMI/IIOP

Dec 1999

J2EE 1.3

CMP
JCA

Sept 2001

J2EE 1.4

WS
Management
Deployment

Nov 2003

Resource injection

Java EE 5

Annotations
EJB 3
JPA 1.0
WS-*
JSF

May 2006

Dependency injection

Java EE 6

EJB 3.1
JPA 2.0
Servlet 3.0
JSF 2.0
JAX-RS 1.1
CDI 1.0
@Inject
Bean Validat°

Web Profile

Managed Bean

Q4 2009

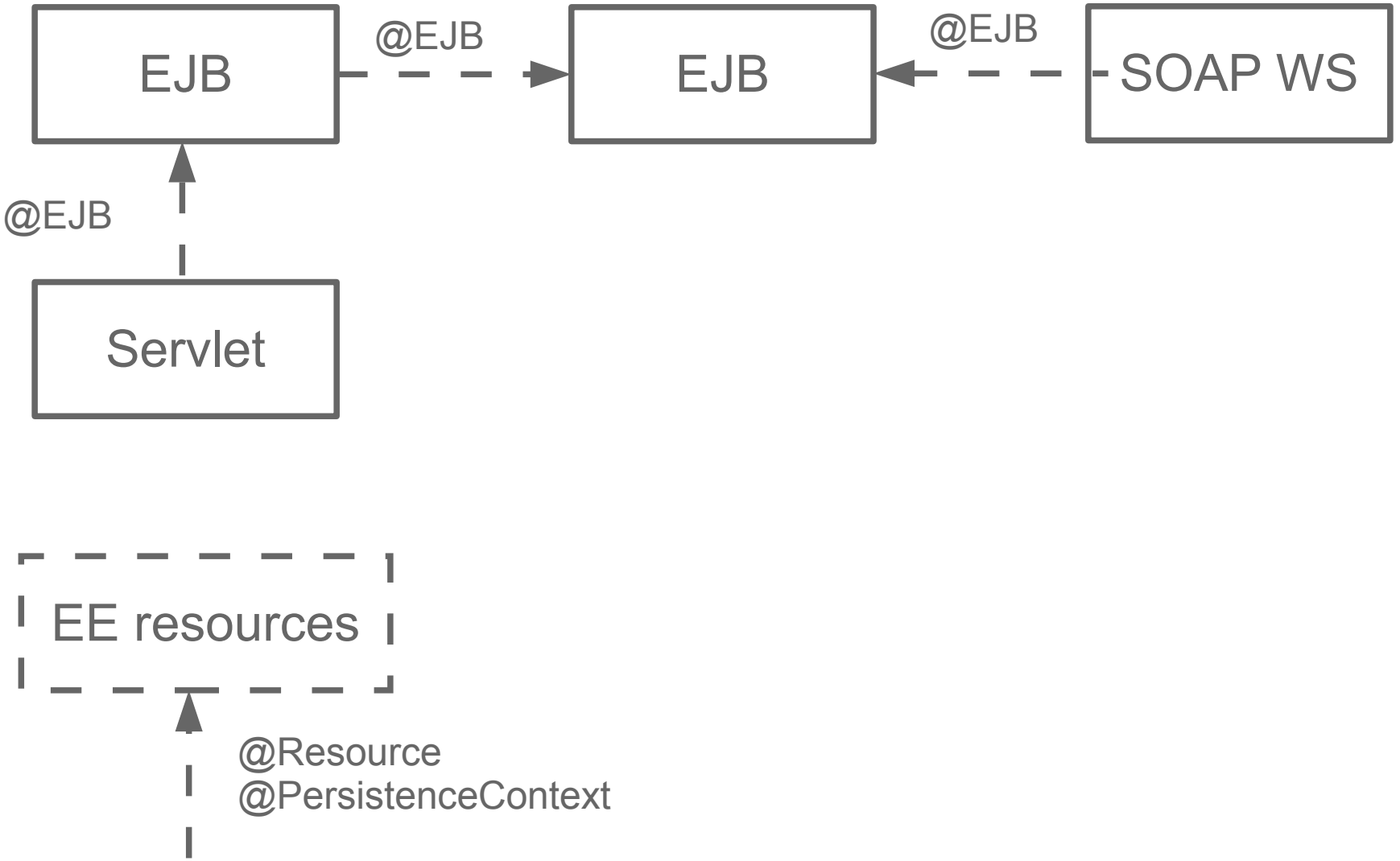
Resource injection in EE 5

- Only inject container resources
  - EJBs, entity manager, datasources, JMS factories & destinations
- To certain components
  - EJBs, servlets, JSF managed beans
- Several annotations
  - `@Resource`, `@PersistenceContext`, `@PersistenceUnit`, `@EJB` & `@WebServiceRef`
- No POJOs

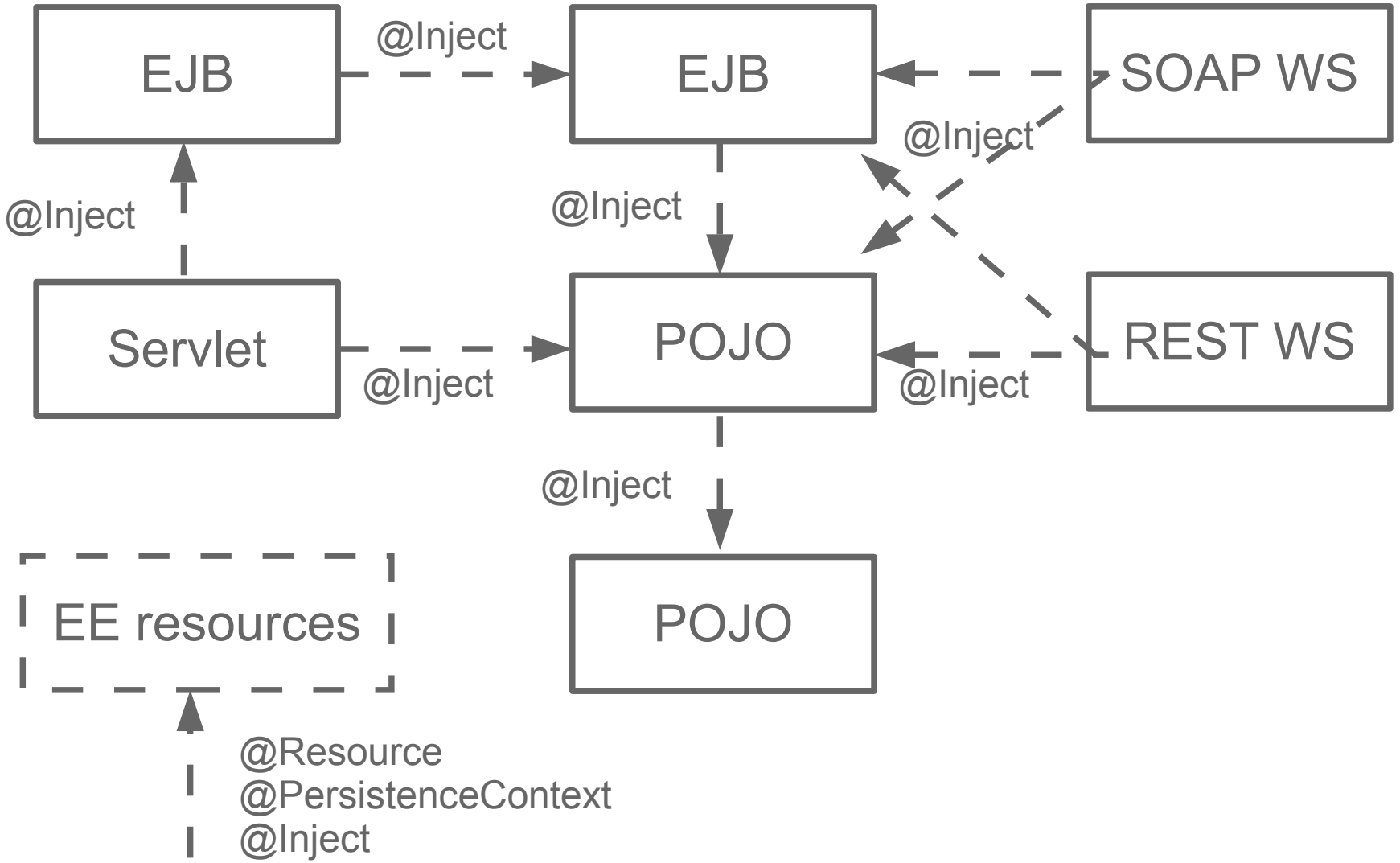
Dependency injection in EE 6

- Two seperate specifications
  - Context & Dependency Injection (CDI) JSR 299
  - DI (aka @Inject) JSR 330
- Bean container for Java EE
- … and even outside Java EE

EE 5 resource injection



EE 6 dependency injection

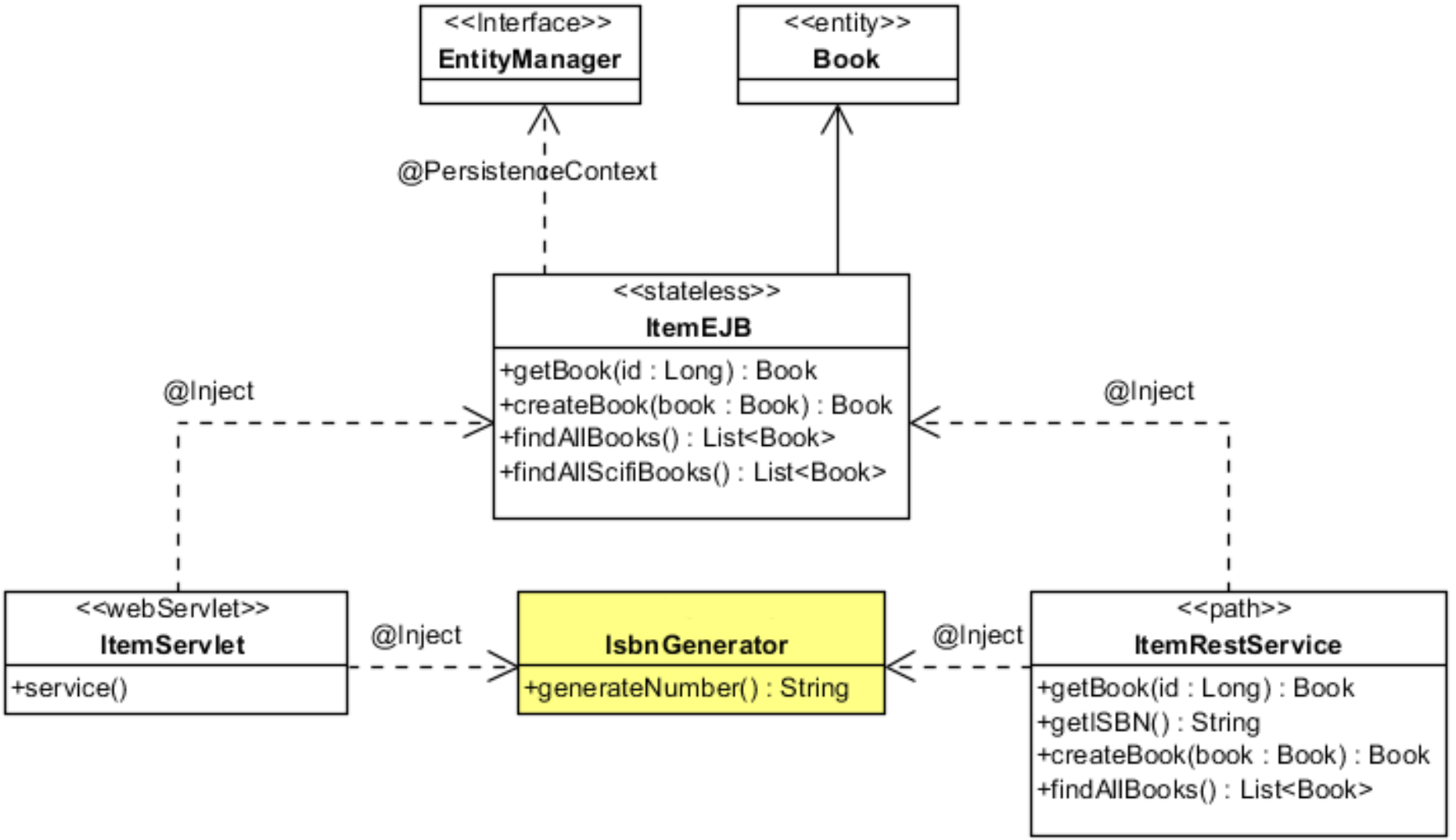


2 specs to archieve it

- DI (@Inject)
- JSR 330
- **`javax.inject`**
- `@Inject`
- `@Named`
- `@Singleton`
- `@Qualifier`
- `@Scope`

- CDI
- JSR 299
- **`javax.enterprise.context`**
- Alternatives
- Producers
- Scopes & context
- Stereotypes
- Decorators, Events
- Extensions

Injection with @Inject



@Inject

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject
    private IsbnGenerator numberGenerator;
    ...
    book.setIsbn(isbnGenerator.generateNumber());
}
```

Servlet

Injection point

```
public class IsbnGenerator {
    public String generateNumber () {
        return "13-84356-" + nextNumber());
    }
}
```

POJO

What's needed to make it work ?

- A container
- CDI
- An empty `beans.xml` file
  - `META-INF`
  - `WEB-INF`

@Inject

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject
    private IsbnGenerator numberGenerator;
    ...
    book.setIsbn(isbnGenerator.generateNumber());
}


public class IsbnGenerator {
    public String generateNumber () {
        return "13-84356-" + nextNumber());
    }
}
```

@Default @Inject

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject @Default
    private IsbnGenerator numberGenerator;
    ...
    book.setIsbn(isbnGenerator.generateNumber());
}
```

Every injection point has a `@Default` qualifier

Every bean has a `@Default` qualifier

```
@Default
public class IsbnGenerator {
    public String generateNumber () {
        return "13-84356-" + nextNumber());
    }
}
```

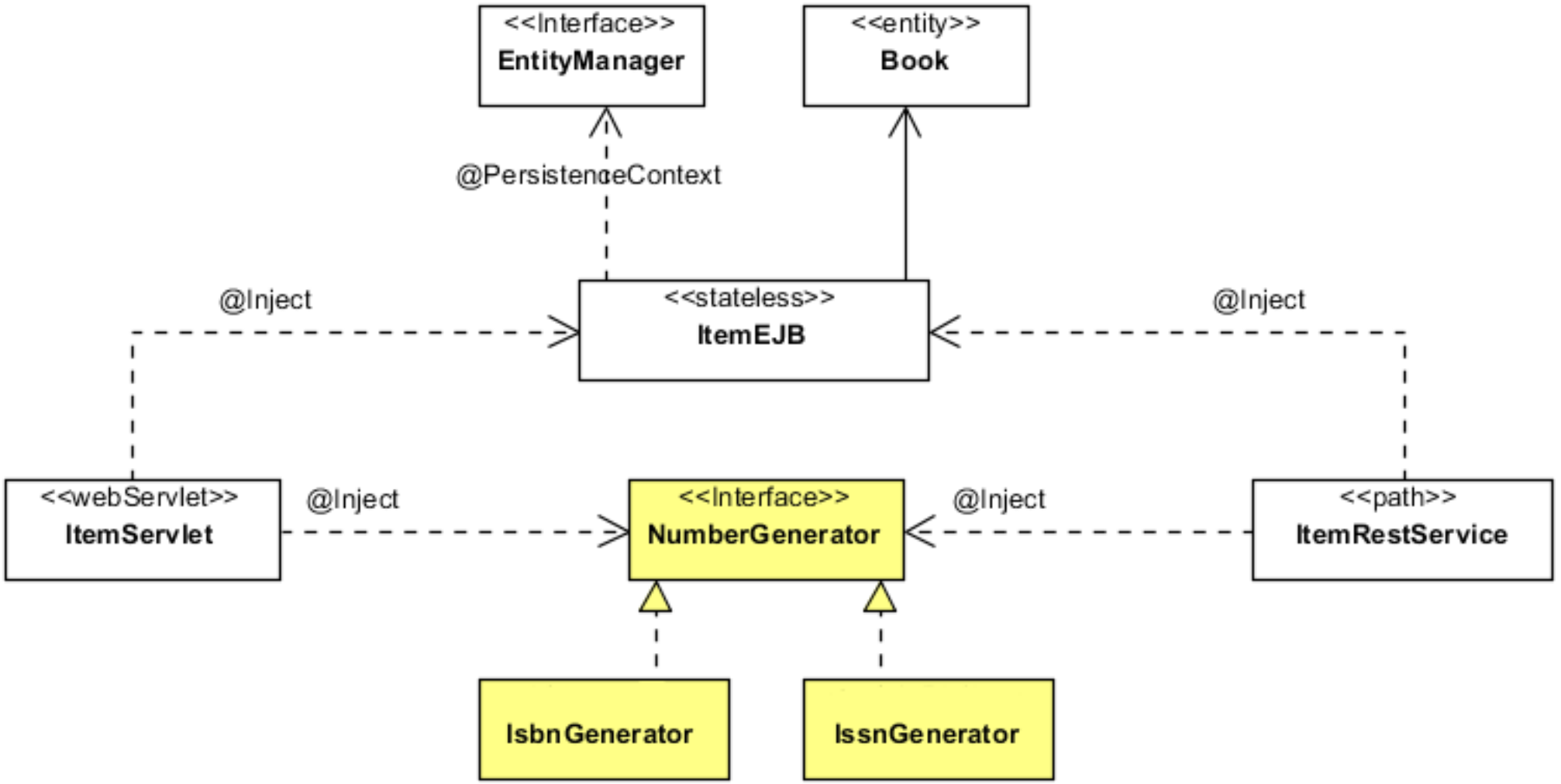
Use your own qualifier

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject @MyOwnQualifier
    private IsbnGenerator numberGenerator;
    ...
    book.setIsbn(isbnGenerator.generateNumber());
}

@MyOwnQualifier
public class IsbnGenerator {
    public String generateNumber () {
        return "13-84356-" + nextNumber());
    }
}
```

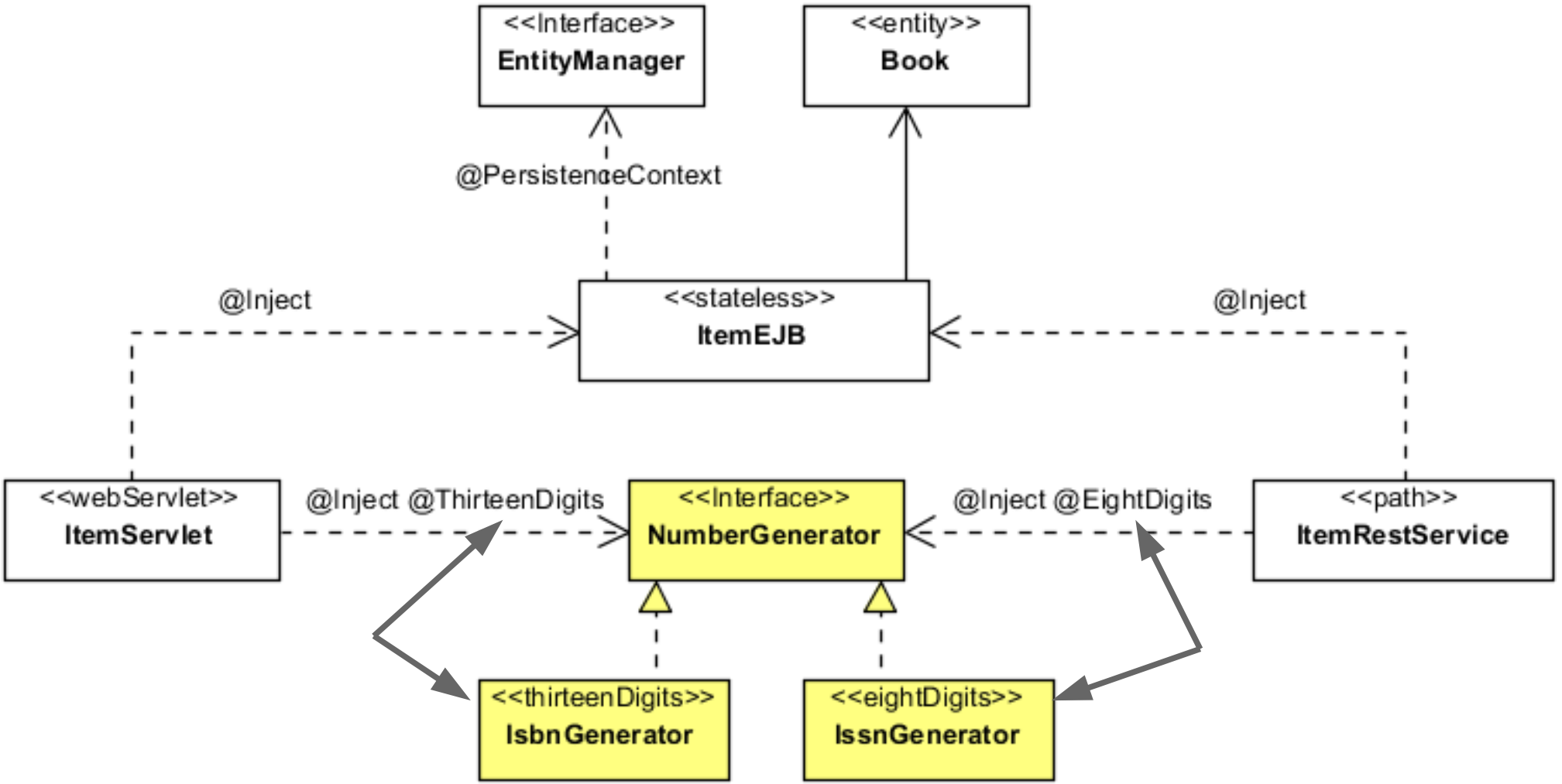
@MyOwnQualifier

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface MyOwnQualifier {
}
```

Ambiguous injection



Non ambiguous injection



Defining the qualifiers

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface EightDigits {
}

@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface ThirteenDigits {
}
```

Defining the beans

```
@EightDigits
public class IssnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "8-" + nextNumber();
    }
}

@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

Injection points

```
@Path("/items") @ManagedBean
public class ItemRestService {
    @Inject @EightDigits
    private NumberGenerator numberGenerator;
    ...
}
```

Strong typing
No strings

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject @ThirteenDigits
    private NumberGenerator numberGenerator;
    ...
}
```

Loose coupling
No reference to the implementation

From XML hell to qualifier hell

- What if you need many qualifiers ?
- 13, 8 digits but also 3, 7, 16, 19, 22, 26...
- Use qualifiers with enumerations

Defining the qualifier & enum

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface NumberOfDigits {
    Digits value();
}

public enum Digits {
    TWO,
    EIGHT,
    TEN,
    THIRTEEN
}
```

Defining the beans

```
@NumberOfDigits(Digits.EIGHT)
public class IssnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "8-" + nextNumber();
    }
}

@NumberOfDigits(Digits.THIRTEEN)
public class IsbnGenerator implements NumberGenerator {
    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

Injection points

```
@Path("/items") @ManagedBean
public class ItemRestService {
    @Inject @NumberOfDigits(Digits.EIGHT)
    private NumberGenerator numberGenerator;
    ...
}

@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject @NumberOfDigits(Digits.THIRTEEN)
    private NumberGenerator numberGenerator;
    ...
}
```

Different injection points

- Field
- Constructor
- Setter

Field injection

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    @Inject @ThirteenDigits
    private NumberGenerator numberGenerator;

    @Inject
    private ItemEJB itemEJB;
    ...
}
```

Constructor injection

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    private NumberGenerator numberGenerator;
    private ItemEJB itemEJB;

    @Inject
    public ItemServlet(@ThirteenDigits NumberGenerator
            numberGenerator, ItemEJB itemEJB) {
        this.numberGenerator = numberGenerator;
        this.itemEJB = itemEJB;
    }
    ...
}
```

Setter injection

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    private NumberGenerator numberGenerator;
    private ItemEJB itemEJB;

    @Inject
    public void setNumberGenerator(@ThirteenDigits
            NumberGenerator numberGenerator) {
        this.numberGenerator = numberGenerator;
    }
    @Inject
    public void setItemEJB(ItemEJB itemEJB) {
        this.itemEJB = itemEJB;
    }
```

Differences

- Field
- Constructor
  - only one constructor injection point
  - add logic to the constructor
- Setter
  - add logic to the constructor
- *Remember* : the container is the one calling the constructor or the setters

Have you seen any XML so far ?

Alternatives

- Define an alternative implementation
- Vary at deployment time
- By default, alternative beans are disabled
- Enable them with `beans.xml` file

Alternative to @ThirteenDigits

```
@Alternative
@ThirteenDigits
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "MOCK-" + nextNumber();
    }
}
```

Alternative to both

```
@Alternative
@ThirteenDigits @EightDigits
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "MOCK-" + nextNumber();
    }
}


@Inject @ThirteenDigits
private NumberGenerator numberGenerator;
```

beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">

    <alternatives>
        <class>org.agoncal.cdi.MockGenerator</class>
    </alternatives>

</beans>
```

Does this look too verbose ?

```
@Alternative
@ThirteenDigits @EightDigits
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "MOCK-" + nextNumber();
    }
}
```

Use stereotypes

```
@MyMock
public class MockGenerator implements NumberGenerator {

    public String generateNumber() {
        return "MOCK-" + nextNumber();
    }
}
@Stereotype
@Retention(RUNTIME) @Target(TYPE)
@Alternative @ThirteenDigits @EightDigits
public @interface MyMock {}
```

Stereotypes

- Models a common role in your application
- You can think of a pattern
  - Modeling recurring concerns
- Stereotypes may declare other stereotypes
- Built-in stereotypes in CDI
  - `@Model` as a replacement of JSF managed beans
- And what is `@Stateless` was a stereotype ?
- And what about `@WebServlet` ?
- And `@WebService` ?

So, we only inject beans ?

Producers

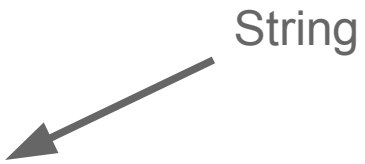
- Producers are a source for objects to be injected
  - these objects are not required to be beans
- For example, producer methods let us:
  - expose a JPA entity as a bean
  - expose any JDK class as a bean

Produce an injectable String

```
public class PrefixGenerator {

    @Produces @ThirteenDigits
    public String getIsbnPrefix() {
        return "13-84356";
    }

    @Produces @EightDigits
    public String getIssnPrefix() {
        return "8";
    }
}
```

Inject the produced String

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {

    @Inject @ThirteenDigits
    private NumberGenerator numberGenerator;

    @Inject @ThirteenDigits
    private String prefix;

    ...
    book.setIsbn(prefix + isbnGenerator.generateNumber());
}
```

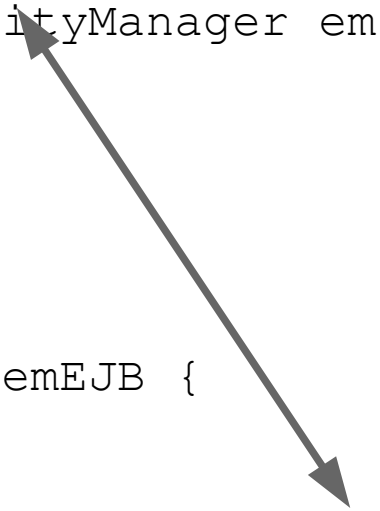


Java EE integration

- Built-in beans in EE :
  - current JTA `UserTransaction`
  - a `Principal` for the current caller identity
  - default Bean Validation `ValidationFactory`
  - a `Validator` for the `ValidationFactory`
- For other beans, use producers

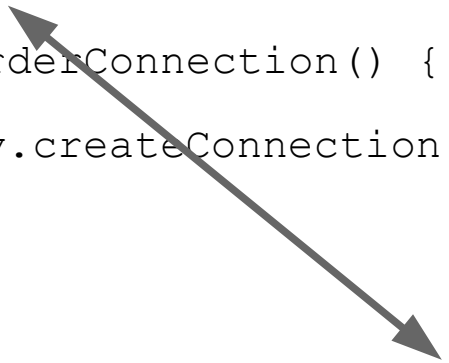
Produce an EntityManager

```
public class DatabaseProducer {

    @Produces @PersistenceContext(unitName = "cdiPU")

    @BookStoreDatabase

    private EntityManager em;

}


@Stateless

public class ItemEJB {


    @Inject @BookStoreDatabase

    private EntityManager em;

    ...

}
```



Produce JMS endpoints

```
public class OrderResources {
  @Resource(name = "jms/ConnectionFactory")
  private ConnectionFactory connectionFactory;
  @Resource(name = "jms/OrderQueue")
  private Queue orderQueue;

  @Produces @OrderConnection
  public Connection createOrderConnection() {
    return connectionFactory.createConnection();
  }
  @Produces @OrderSession
  public Session createOrderSession(@OrderConnection Connection conn){
    return conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
  }
  ...
}
```



Consume JMS endpoints

```
@Inject @OrderSession QueueSession orderSession;


public void sendMessage() {
  MapMessage msg = orderSession.createMapMessage();
  msg.setLong("orderId", order.getId());
  ...
  producer.send(msg);
}
```

Can I use all that from a JSF page ?

@Named

- Reference a bean from the Expression Language
- Gives it a name (a String)
- Most commonly from a JSF view
- Non type safe dependency injection
- Any bean can be named
- Do we still need JSF managed beans ?

Default name

```
@Named
public class IsbnGenerator {

    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

```
<h:outputLabel value="#{isbnGenerator.generateNumber}"/>
```

A different name

```
@Named("generator")
public class IsbnGenerator {

    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

```
<h:outputLabel value="#{generator.generateNumber}"/>
```

Mix a producer and a default name

```
public class IsbnGenerator {
    @Produces @Named
    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

```
<h:outputLabel value="#{generateNumber}"/>
```

Mix a producer and a name

```
public class IsbnGenerator {
    @Produces @Named("isbnNumber")
    public String generateNumber() {
        return "13-84356-" + nextNumber();
    }
}
```

```
<h:outputLabel value="#{isbnNumber}"/>
```

CDI is not just about DI

- Interceptor enhancement
- Decorators
- Events

Interceptors 1.1

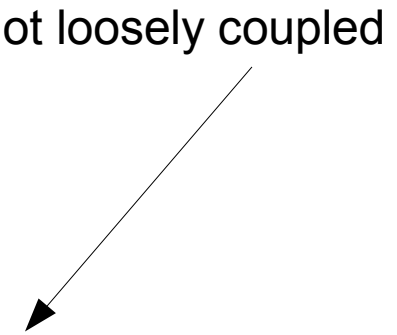
- Address cross-cutting concerns in Java EE
- Were part of the EJB 3.0 spec
- Now a seperate spec shipped with EJB 3.1
- Can be uses in EJBs...
- … as well as ManagedBeans
- `@AroundInvoke`
- `@AroundTimeout` for EJB timers

Interceptors 1.1

```
public class LoggingInterceptor {

  private Logger logger = Logger.getLogger("org.myapp");

  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }
}
```

Interceptors 1.1

```
@Stateless
public class CustomerEJB {

    @PersistenceContext
    private EntityManager em;

    @Interceptors(LoggingInterceptor.class)
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }

    public Customer findCustomerById(Long id) {
        return em.find(Customer.class, id);
    }
}
```



CDI adds interceptor binding

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Loggable {}
```

CDI adds interceptor binding

```
@Loggable @Interceptor
public class LoggingInterceptor {

  private Logger logger = Logger.getLogger("org.myapp");

  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
}
```

```
@Stateless
public class CustomerEJB {

    @Loggable
    public void createCustomer(Customer customer) {
        em.persist(customer);
    }
}
```

Decorators

- Interceptors
  - separate concerns which are orthogonal
  - intercept invocations of any Java type
  - perfect for solving technical concerns
  - unaware of the semantics of the events they intercept
  - not appropriate for separating business concerns
- Use decorators
  - intercepts invocations only for a certain Java interface
  - aware of the semantics attached to that interface

Decorators

```
public interface Account {
  public BigDecimal getBalance();
  public User getOwner();
  public void withdraw(BigDecimal amount);
  public void deposit(BigDecimal amount);
}

public class AccountImpl implements Account {
  public BigDecimal getBalance() {...}
}
```

```
@Decorator
public abstract class LargeTransactionDecorator implements Account
  public BigDecimal getBalance() {...}
}
```

```
@Inject Account account;
// Calls the decorator first
account.getBalance();
```

Events

- Event producers raise events that are delivered to event observers by the container
- Observer/observable pattern
- Typesafe approach
- An event is a `@Qualifier`
- No need to use JMS

Events

```
@Qualifier
@Target({FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Updated {}
```

```
@Inject @Updated Event<Document> docUpdatedEvent;
docUpdatedEvent.fire(document);
```

```
public void afterDocumentUpdate(@Observes @Updated Document documen
{ ... }
```

And more...

- Scopes & context
- Extensions
  - Finally makes the EE platform extendable

How can I have all these goodies ?

CDI is not just about Java EE 6

- CDI comes for free in Java EE 6 & Web Profile 1.0
- Bootstrap CDI in several environments
- Not standard (yet)
  - Java SE
  - Tomcat 6.x / 7.x
  - Jetty 6.X / 7.x
- Doesn't work ?
  - Application client container (ACC)
  - Spring 3.x

Doesn't work with Spring?

- Spring 3.x
- Only DI is supported (JSR 330)
  - `javax.inject`
  - `@Inject, @Named, @Qualifier`
- Not CDI (JSR 299)
  - `javax.enterprise.context`
  - Alternatives, Producers, Stereotypes, Decorators, Events
- You can replace `@Autowired` with `@Inject`

CDI implementations

- Weld (JBoss)
- Open WebBeans (Apache)
- CanDI (Caucho)

What makes CDI Unique ?

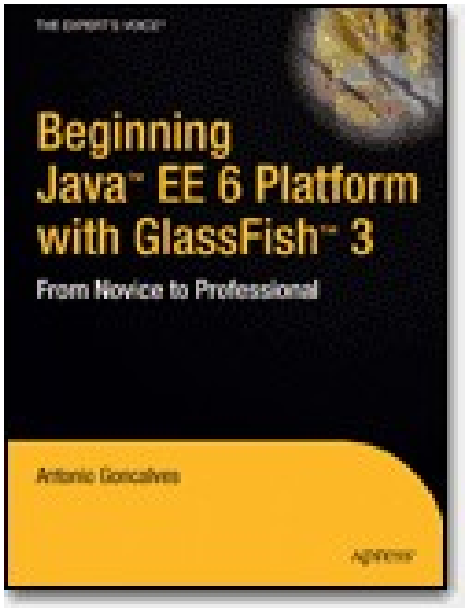
- It's standard
- It's loosely coupled
- It's strongly typed
- It's extensible
- Might even become the platform for EE

And what about the future ?

- Java EE 7
- CDI 1.1 is on the way
  - Embedded mode outside Java EE container
  - Static injection
  - Better support for CDI in certain EE components
  - Application lifecycle events
  - ...

Java EE 6

- A book
  - 450 pages about Java EE 6
  - Second edition
  - Covers most specs
- A training
  - 3 days
  - Most specs
  - Hands on labs
  - Contact me



Thanks

www.antoniogoncalves.org
antonio.goncalves@gmail.com
@agoncal
@lescastcodeurs

Creative Commons

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial — You may not use this work for commercial purposes.

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.