Context & Dependency Injection

CDI

JSR 346

Contexts & Dependency Injection for the Java EE platform

- A rich dependency management model
- Designed for stateful and contextual objects
- Integrates the different tiers
- Includes a SPI for third-party framework integration

History

Contexts & Dependency Injection for the Java EE platform

Java EE 6

JSR-299
Contexts & Dependency Injection for the Java EE platform

Uses

Java SE

JSR-330
Dependency Injection for Java

Java EE 7

JSR-346
Contexts & Dependency Injection for the Java EE platform

JSR 346 influences

- JSR 299
  - But updated
- Seam
  - But more typesafe
- Guice
  - But more enterprise capable
- Spring
  - But more stateful and less XML

JSR-330

- Standardized dependency injection for Java SE
  - maximize reusability, testability and maintainability
- Specifies annotations and use of annotations
- Does NOT specify a container

JSR-330 annotations

- @Inject
- @Qualifier
- @Named
  - String based qualifier
- @Scope
- @Singleton
  - singleton scope
- Provider<T>
  - provides instances of T

JSR-346

- Type safe Dependency Injection for Java EE components
- Lifecycle aware
  - EE components often have a scope/context
  - i.e. request scoped JSF managed bean
- Integration with the Unified Expression Language
- Additional services
  - i.e. event notification, decoration and interceptors

```
<h:inputText value="#{hello.name}"/>
<h:commandButton value="Greet"/> <br/><br/>

#{hello.greeting}
```

Exposes bean to EL

```
@Named("hello")
@RequestScoped
public class HelloBean {
    @Inject Greeter greeter;

    private String name;

    public String getGreeting() {
        return greeter.greet(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
<h:inputText value="#{hello.name}"/>
<h:commandButton value="Greet"/> <br/><br/>

#{hello.greeting}
```

Bean valid for this request

```
@Named("hello")
@RequestScoped
public class HelloBean {
    @Inject Greeter greeter;

    private String name;

    public String getGreeting() {
        return greeter.greet(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
<h:inputText value="#{hello.name}"/>
<h:commandButton value="Greet"/> <br/><br/>

#{hello.greeting}
```

```
@Named("hello")
@RequestScoped
public class HelloBean {
    @Inject Greeter greeter;

    private String name;

    public String getGreeting() {
        return greeter.greet(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Inject instance of Greeter

An instance will be created by the container, and bound to request scope

The Greeter class

- The Greeter is a simple POJO

```
public class Greeter {
    public String greet(String name) {
        if(name != null) {
            return "Hello, " + name;
        } else {
         return "Hello, stranger";
        }
    }
}
```

The Greeter class

- But could have been a EJB Session Bean…

```
@Stateless
public class Greeter implements Greet {
    @Resource
    SessionContext context;

    public String greet(String name) {
        System.out.println("Principal: " +
                context.getCallerPrincipal().getName());

        if(name != null) {
            return "Hello, " + name;
        } else {
         return "Hello, stranger";
        }
    }
}
```

Types of injection

```
@Inject
private Greet greeter;
```

```
@Inject
public HelloBean(Greet greeter) {
    this.greeter = greeter;
}
```

```
@Inject
public void setGreeter(Greet greeter) {
    this.greeter = greeter;
}
```

Where can I use CDI?

- Inject into:
  - every POJO
  - every EJB Session Bean
  - every Servlet
- Injectable types:
  - every POJO
  - every EJB Session Bean
  - JEE Resources (e.g. a DataSource)

How do I enable CDI?

- By default all class annotated beans are discovered
- Finer grained control can be set through beans.xml

Bean discovery mode

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       http://www.w3.org/2001/XMLSchema-instance"
       ocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

- 'annotated' - Only components with a class-level annotation are processed (default).
- 'all' - all components are processed (as in Java EE 6 with the explicit beans.xml).
- 'none' - CDI is effectively disabled.

Upgrading a Java EE6 application

org.jboss.weld.exceptions.DeploymentException: WELD-001408 Unsatisfied dependencies for type…

The bean-discovery-mode attribute is new and not available in CDI 1.0

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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"
       bean-discovery-mode="all">
</beans>
```

To mimic CDI 1.0/JAVA EE6 behavior, set bean-discovery-mode to ‘all’

Qualifiers

- An interface could have multiple implementations
  - the container must choose an implementation
- Add a qualifier (another annotation) to each implementation
  - the container chooses the correct implementation

Qualifiers

Self written qualifier annotation

```
@Synchronous
public class SynchronousPaymentProcessor
        implements PaymentProcessor{
    public void process() {
        System.out.println("Synchronous processing");
    }
}
```

Self written qualifier annotation

```
@Asynchronous
public class AsynchronousPaymentProcessor
        implements PaymentProcessor {
    public void process() {
        System.out.println("Async processing");
    }
}
```

Qualifiers

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

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

Qualifiers

No compile-time coupling

```
@Model
public class PaymentBean {

    @Inject @Asynchronous
    PaymentProcessor processor;

    public String pay() {
        processor.process();

        return null;
    }
}
```

Contract

- The API type
  - user defined class or interface
  - @Local interface for session beans
- The qualifier type
  - User defined annotations

Constructors

- Container calls constructor
- No-arg constructor by default
- Argument constructor identified by @Inject
  - Each argument will be injected

```
@Inject
public MyCoolServiceImpl(@CreditCard PaymentProcessor creditcard) {
    creditcard.pay();
}
```

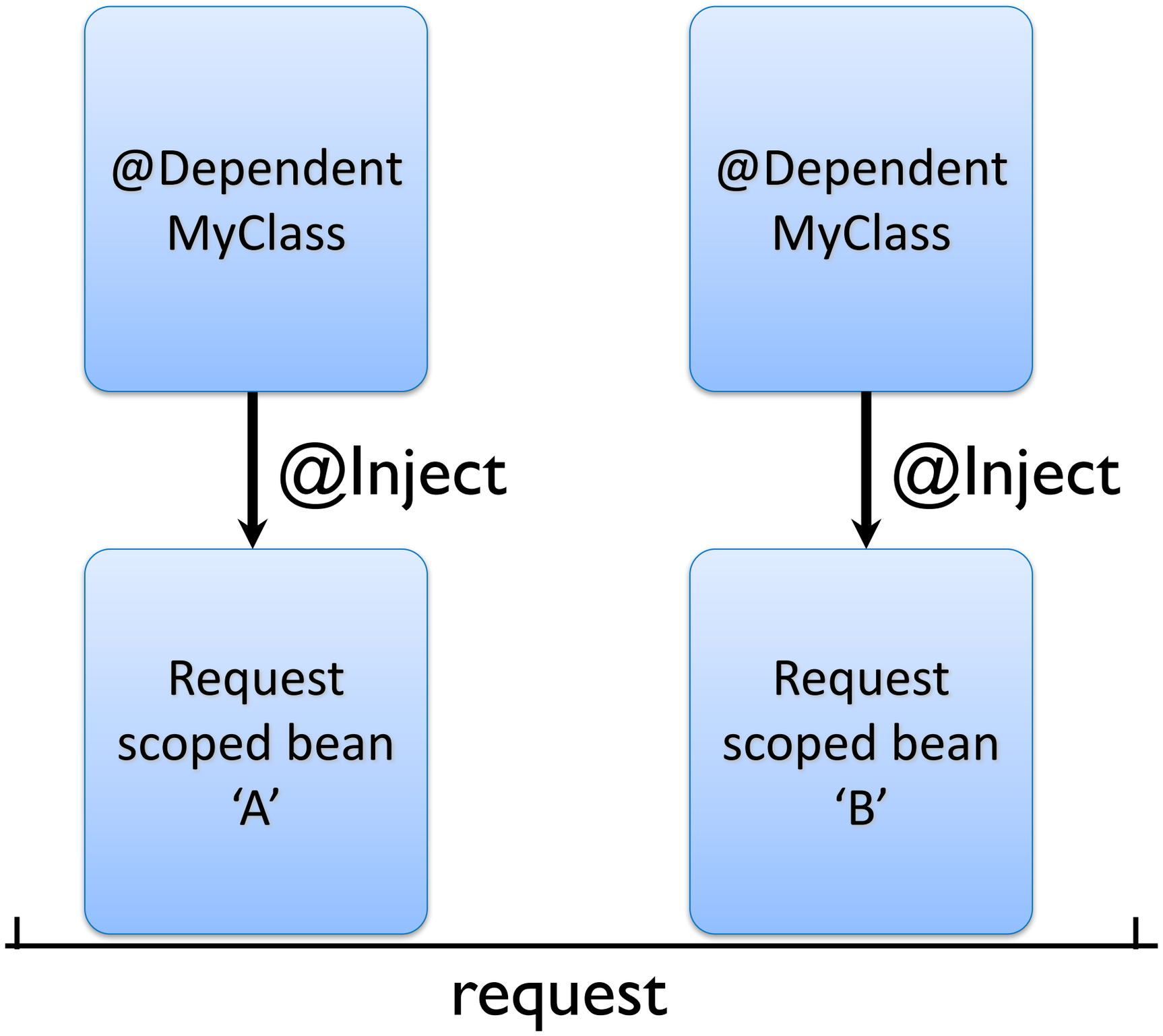
Initializer example

```
@Initializer
public MyCoolServiceImpl(@CreditCard PaymentProcessor creditcard) {
    creditcard.pay();
}
```

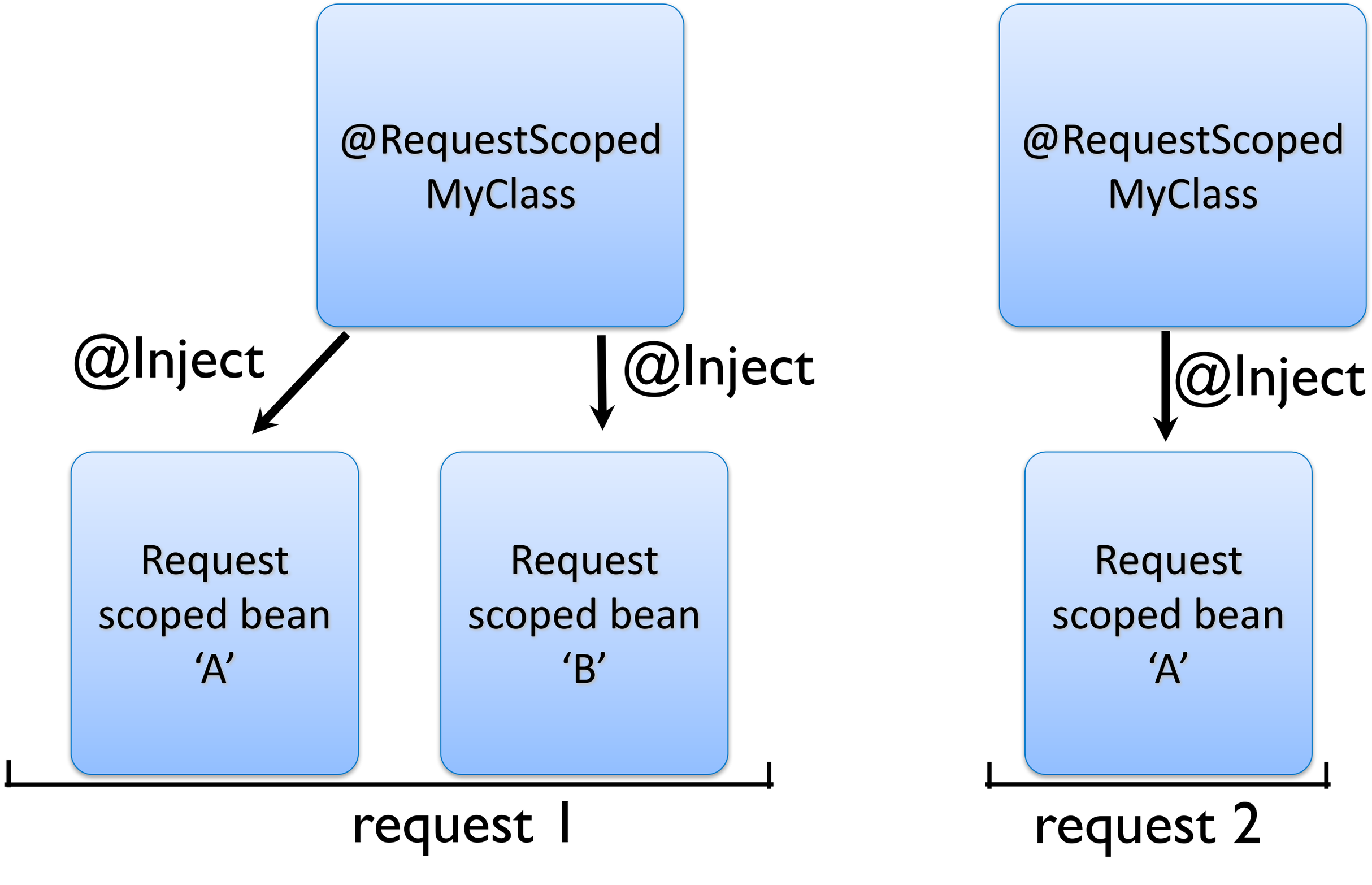
Scopes

- In a web application, an object always has a scope (e.g. Request or Session scope)
- CDI is context/scope aware
  - injection is scope sensitive
  - instances may be shared by injection points
  - container cleans up
- Scopes: Request, Session, Conversation, Application
- Default scope: @Dependent
  - never shared, bound to lifecycle of injection point

Default Scope



Request Scope



Session Scope

@SessionScoped
MyClass

@Inject

@Inject

@Inject

Request scoped bean 'A'

Request scoped bean 'B'

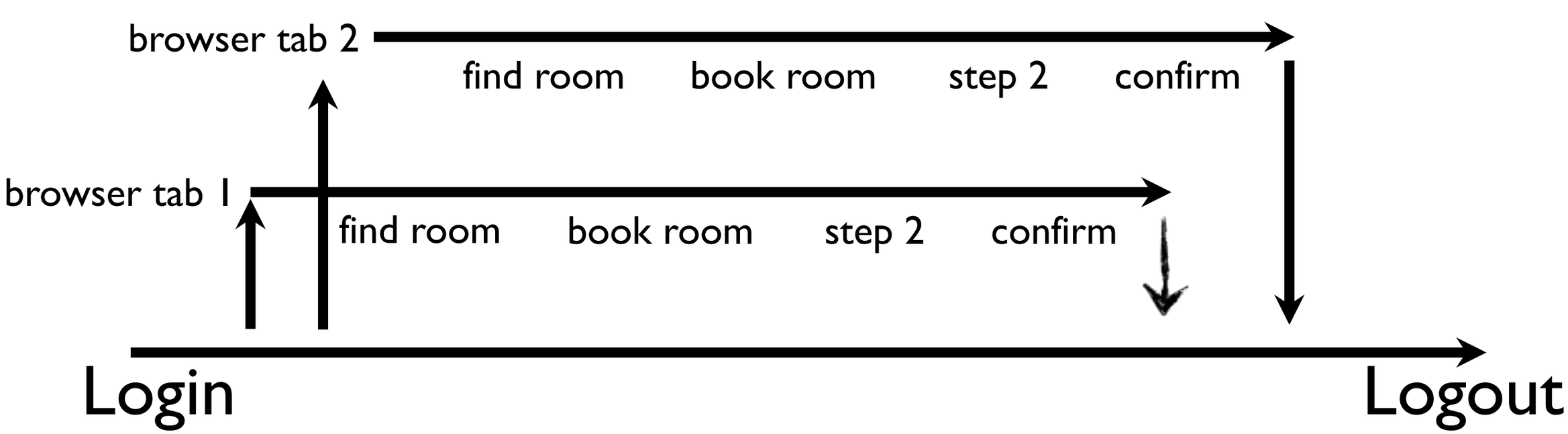
Request scoped bean 'A'

request 1

request 2

Conversation scope

- Multi-request scope within a session
- Isolated from other conversations
- Defined begin and end point



Conversations example

Conversational bean

```
@ConversationScoped
public class Basket implements Serializable {

    @Inject
    Conversation conversation;
```

Upgrade to long-running

```
if (conversation.isTransient()) {
    conversation.begin();
}
```

Schedule end of conversation at end of request

```
conversation.end();
```

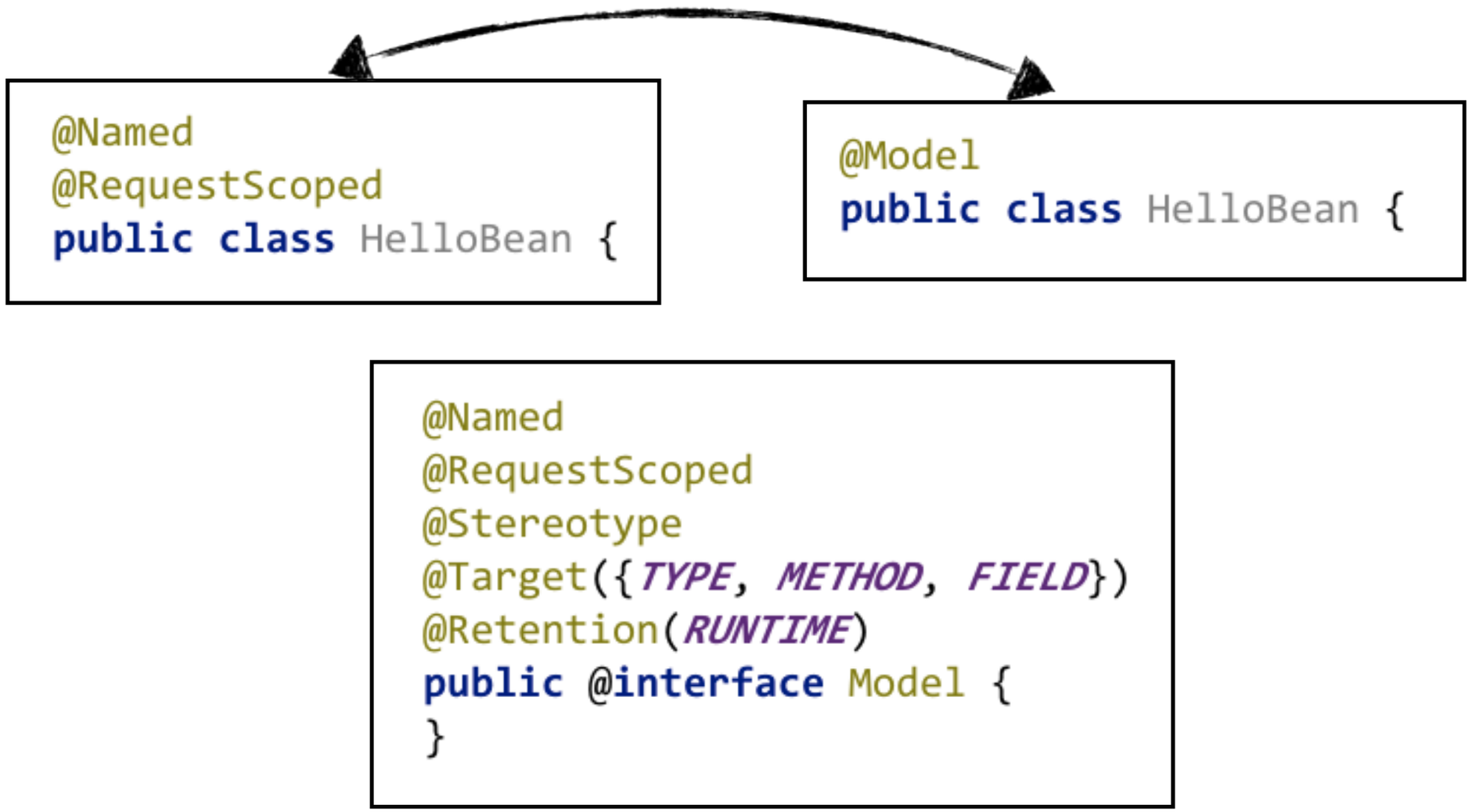
JSF Conversations

- Every JSF request has a conversation
- Associated conversation determined at restore view phase
  - Associate with existing long-running conversation

Stereotypes

- Declare common meta data in a new annotation
  - @Model is a standard stereotype

Stereotypes



```
@Named
@RequestScoped
public class HelloBean {
```

```
@Model
public class HelloBean {
```

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {
}
```

Producer methods

- A source of injectable objects
  - objects are not required to be beans (e.g. a List)
  - concrete type varies at runtime
  - objects require extra initialization
- Most similar to a traditional Factory Method
  - but more loosely coupled

Producer methods – example

```java
public class Inventory {
    @Produces
    public List<Product> getProducts() {
        //Go to db, and return products
        return Arrays.asList(
                new Product("Product 1", 50),
                new Product("Product 2", 200));
    }
}
```

```java
@Model
public class ProductsBean {
    @Inject
    List<Product> products;

    public List<Product> getProducts() {
        return products;
    }

}
```

Producer methods

Custom qualifier

Exposed to EL

Put in Application Scope

```
public class Inventory {

    @Produces @Named("products") @ApplicationScoped @Popular
    public List<Product> getProducts() {
        //Go to db, and return products
        return Arrays.asList(
                new Product("Product 1", 50),
                new Product("Product 2", 200));
    }
}
```

Producer fields

- Same semantics as producer methods

```
@Produces @SessionScoped
private List<Product> products;
```

Alternatives

- An alternative implementation of a bean may be configured
- Useful for mocking
- Annotate bean @Alternative and configure in beans.xml

Example

```
public class ProductDaoImpl implements ProductDao {
    public List<String> getProductNames() {
        throw new RuntimeException("I need my database");
    }
}
```

```
@Inject ProductDao dao;
```

A mocked service

```
<beans>
    <alternatives>
        <class>demo.beans.dao.ProductDaoMock</class>
    </alternatives>
</beans>
```

```
@Alternative
public class ProductDaoMock implements ProductDao{

    public List<String> getProductNames() {
        return Arrays.asList("Product 1", "Product 2", "Product 3");
    }

}
```

Or as stereo type

```
@Stereotype
@Alternative
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {

}
```

```
<beans>
    <alternatives>
        <stereotype>demo.beans.dao.Mock</stereotype>
    </alternatives>
</beans>
```

```
@Mock
public class ProductDaoMock implements ProductDao{

    public List<String> getProductNames() {
        return Arrays.asList("Product 1", "Product 2", "Product 3");
    }

}
```

Interceptors

- Improves the EJB interceptor mechanism
- Type-safe mechanism for associating intercepters with beans
- Can be bound to a stereo type

Interceptor example

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

```
@Secure @Interceptor
public class SecureInterceptor {
    @AroundInvoke
    public Object checkCredentials(InvocationContext ctx)
            throws Exception{
        System.out.println("Checking security");

        return ctx.proceed();
    }
}
```

Using Interceptors

```
@Secure
public class ExampleBean {
```

```
<interceptors>
    <class>demo.interceptors.SecureInterceptor</class>
</interceptors>
```

Decorators

- Apply the decorator pattern
  - In a loosely coupled way

```
public class SimpleLogger implements Logger {
    public void log(String message) {
        System.out.println(message);
    }
}
```

```
public class ExampleBean {
    @Inject Logger logger;
```

Decorator Example

```
@Decorator
public class TimestampLogger implements Logger{
    @Inject @Any @Delegate Logger logger;

    public void log(String message) {

        logger.log(new Date() + " - " + message);
    }
}
```

```
<decorators>
    <class>demo.delegates.TimestampLogger</class>
</decorators>
```

@New

- Injects a new instance of the requested class
- Not bound to a scope, but injection is performed
- Scope @dependent
- No EL name or stereotype

```
public  @Produces @Popular
Product getSpecialProduct(@New Product product) {
    product.setName("New product");
    return product;
}
```

Events

- Completely decoupled producers and observers

```
public class HelloEvent {
    private final Person person;

    public HelloEvent(Person person) {
        this.person = person;
    }

    public Person getPerson() {
        return person;
    }
}
```

Fire an event

```
@Inject @Any Event<HelloEvent> helloEvent;

public String sayHello() {
    System.out.println(hello.getGreeting());
    helloEvent.fire(new HelloEvent(new Person("Paul")));
    return null;
}
```

Observe an event

```
public class HelloListener {

    public void onHello(@Observes HelloEvent event) {
        System.out.println("Hello: " + event.getPerson().getName());
    }
}
```

Resources

- A bean that represents a reference to a JEE resource
  - DataSource, EntityManager, EntityManagerFactory, Remote EJB or WebService

Resource example

- Declare the resource

```
public class ResourceProducer {
    @PersistenceContext(unitName="productDB")
    EntityManager em;

    @Produces
    @ProductDB
    public EntityManager createProductDB() {
        return em;
    }

}
```

Resource example

- Use the resource

```
@Stateful
public class ExampleBean implements Serializable {
    @Inject @ProductDB EntityManager em;
```