Why Context and Dependency Injection (CDI) is So Cool?

Sang Shin
JRebel Evangelist from ZeroTurnaround.com
Founder and Chief Instructor from JPassion.com
“Learn with Passion!”
Topics

- CDI Basics (20 minutes)
- CDI Advanced (25 minutes)
- CDI Extension
- JRebel demo (10 minutes)

This is usually 2-hour talk but is reduced to 1 hour.

Complete slides and the demo's are available as ready-to-run Maven projects from http://www.javapassion.com/jugtalks
Topics of “CDI Basics”

- What is and Why Dependency Injection?
- What is and Why CDI (JSR 299)?
- CDI theme – Loose coupling with string typing
- Bean definition (in the context of CDI)
- Basic dependency injection
- Qualifier
- @Named built-in qualifier
- Stateful objects (scoped objects)
- CDI for Java SE application
What is & Why CDI (JSR 299)?
What is CDI (JSR 299)? (Basic)

• Provides a **unifying Dependency Injection** and contextual life-cycle model for Java EE
  > Unified existing Dependency Injection schemes – Spring, Guice, Seam
  > A completely new, richer dependency management model
  > Type-safe dependency injection
  > Designed for use with stateful objects (scoped objects)

• Makes it much easier to build applications using JSF and EJB together
  > Let you use EJBs directly as JSF managed beans
What is CDI (JSR 299)? (Extensibility)

- Includes a SPI extending Java EE platform
  - Java EE is now flexible, portable, and extensible architecture
  - You can change the characteristics of existing platform services
  - You can provide new platform-level portable services through new annotations at the time of deployment
- You can now build your own next generation Java EE platform
  - You don't have to wait Java EE 7 for the platform features you need
Why CDI (JSR 299) for Java EE?

- **Reason #1: We need general-purpose dependency injection scheme**
  - Java EE 5 provides resource injection of only known resources to the container (@EJB, @PersistenceContext, @PersistenceUnit, @Resource)
  - In other words, Java EE 5 does not provide general-purpose dependency injection scheme

- **Reason #2: We need type-based injection**
  - String name or XML based injection is fragile
  - Type-based injection enables better tooling in general
Bean Definition
(in the context of CDI)
What is a Bean anyway?

• Many forms of a “bean” already exist. So which bean are we talking about?
  > JSF bean
  > EJB bean
  > Spring bean
  > Seam bean
  > Guice bean
  > CDI bean

• Java EE needs a unified bean definition
  > Managed Bean 1.0 specification in Java EE 6 provides it
Managed Bean 1.0: What is it?

- Managed Beans are container-managed POJOs
  - Lightweight component model
  - Instances are managed by the container
What about EJB, REST, CDI. etc Bean?

- You could see everything as a Managed Bean with extra services
- An EJB is a Managed Bean with
  - Transaction support
  - Security
  - Thread safety
  - Persistence
- A REST service is a Managed Bean with
  - HTTP support
- A CDI bean is a Managed Bean with
  - CDI services (explained in the next slide)
CDI Bean Services

- Auto-discovered – by the container
- Set of qualifiers – solves ambiguity
- Scope – context of a bean
- Bean EL name – support non-type based invocation
- Set of interceptor bindings
- Alternative – replace bean at deployment time
CDI Bean Example

- No annotation required
- No bean declaration in XML file required

// This is a CDI bean
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name;
    }
}

Automatic Bean Discovery

• How does container discover beans?
  > By scanning the classpath that contains both application and container archives

• How can container scan only the relevant application archives for bean discovery?
  > By detecting the presence of “beans.xml” in application archive
  > For WAR file, the “beans.xml” is under WEB-INF directory
  > For JAR file, the “beans.xml” is under META-INF directory

• “beans.xml”
  > It is not for declaring beans (like in Spring)
  > It can be empty
  > Used for some other purposes (like declaring an alternative)
Basic Injection
How do you inject a Bean?

- Use `@Inject <Java-Type> <variable>` for field injection
- `<Java-Type>` can be Java interface

```java
class MyGreeter {

    // Inject Greeting object for field injection
    @Inject Greeting greeting;

    public sayGreeting(String name){
        System.out.println(greeting.greet(name));
    }
}
```
Where can you inject a bean?

- Bean can be injected at “Injection points”
  - Field
  - Method parameter
- Method can be
  - Constructor (useful for created immutable object)
  - Initializer
  - Setter method
  - Producer (will be covered in “CDI Advanced”)
  - Observer (will be covered in “CDI Advanced”)

Demo:

inject_bean_*,
inject_ejb-to_field
4531_javaee6_cdi_basics.zip
Qualifier
What is a Qualifier?

• For a given bean type (class or interface), there may be multiple beans which implement the type (in the classpath)
  > For an interface, there could be multiple implementations
  > For a class, there could be multiple child types
  > Ambiguity error will result

• A qualifier is an annotation that lets a client choose one between multiple candidates of a certain type
  > Make type more specific
  > Assigns semantic meaning

• Injected type is identified by
  > `Qualifier(s) + Java type`
  > `e.g. @Inject @LoggedIn User user;`
How to build and use Qualifier? – 3 steps

• Step #1: Define a qualifier (type)
• Step #2: Qualify an implementation class
• Step #3: Select a qualified implementation
Step #1: Define a Qualifier (Type)

// Define “Informal” qualifier (type)
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}

Part #2: Qualify an Implementation

// Bind the “@Informal” qualifier with “InformalGreeting” // implementation class. (Think of @Informal as an // extended type of the Greeting implementation class.)

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "hi " + name;
    }
}

Part #3: Select Qualified Impl. Class

- Injected type is identified by
  > Qualifier(s) + Java type

```java
public class MyGreeter {

  // Injected type is identified @Informal qualifier and Greeting type.
  // So InformalGreeting class (of previous slide) will be chosen.
  @Inject @Informal Greeting greeting;

  public void greet() {
    System.out.println(greeting.greet("Hello") );
  }
}
```
Qualifier and Type Safety (Strong Typing)

- Qualifier + Java type makes a composite type (extended type)
  > Again, think of a Qualifier as a type
- Qualifiers make type safe injection possible
  > Qualifiers replace “look-up via string-based names”
Demo:

inject_qualifier_ambiguitycase, inject_qualifier
4531_javae6_cdi_basics.zip
Qualifier with Attributes

- Qualifier definition with attributes

```java
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface VariousGreetings {
    GreetingType type() default GreetingType.FORMAL;
}
```

- Qualify an implementation

```java
@VariousGreetings(type=GreetingType.FORMAL)
public class FormalGreeting implements GreetingInterface {
    public String greet(String name) {
        return "Formal Hello " + name;
    }
}
```
Qualifier with Attributes

- Select qualified implementation

// Inject GreetingInterface object with qualifier with attribute
@Inject
@VariousGreetings(type=GreetingType.INFORMAL)
GreetingInterface greeting;
Demo:

inject_qualifier_attributes
4531_javaee6_cdi_basics.zip
@Named
Built-in Qualifer
Why do we need @Named Annotation?

- In Java code, injected type is identified by
  > Qualifier(s) + Java type
- How do we identify a bean outside of type-safe Java code, for example in Unified EL expressions (in facelet or JSP), in which we cannot use Java type?
  > We should able to identify a bean via a name (not Java type)

```xml
<h:commandButton value="Say Hello"
  action="#{printer.greet}"/>
```
Why do we need @Named Annotation?

- Give it a name using @Named annotation

```java
public @Named("myprinter")
class Printer {

    @Inject Greeting greeting;

    public void greet() {
        System.out.println( greeting.greet("world") );
    }
}
```
Topics of “CDI Advanced”

- Producer
- Events
- Alternatives
- Interceptor (not covered in this presentation)
- Decorator
- Stereotype
Producer
Producer Methods

• Producer methods allow control over the production of the dependency objects
  > Runtime polymorphism – Producer can return different object depending on runtime condition (business logic, time of the day, etc)
  > Creation of Non-bean instance - Objects to be injected are not required to be instances of beans
  > Custom setup - Objects require some custom initialization that is not performed by the bean constructor
• Enables decoupling of a “producer” from the “consumer”
// Inject Coder object
@Inject
@RequestScoped
Coder coder;

// Producer method returns a different object depending on a runtime condition
@Produces
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci, @New CoderImpl ci) {

    switch (coderType) {
    case TEST:
        return tci;

    case SHIFT:
        return ci;

    default:
        return null;
    }
}
// Produce FacesContext object that we don't control
@Produces
@RequestScope
public FacesContext getFacesContext() {
    return FacesContext.getInstance();
}
Demo:

`cdi_producer_methods_example1`
`cdi_producer_methods_example2`
`4533_javaee6_cdi_advanced.zip`
Type-safe Use of Java EE Resources (1)

- Use Producer field to expose Java EE resources

```java
// Define a qualifier
@Qualifier
@Retention(RUNTIME)
@Target({
    METHOD, FIELD, PARAMETER, TYPE
})
public @interface UserDatabase {
}
```

```java
// Expose Java EE resource (EntityManager) via Producer
@Singleton
public class UserDatabaseEntityManager {
    @Produces
    @UserDatabase
    @PersistenceContext(unitName="producerfieldsPU")
    private EntityManager em;
}
```

```java
// Expose Java EE resource (EntityManager) via Producer
@Singleton
public class UserDatabaseEntityManager {
    @Produces
    @UserDatabase
    @PersistenceContext(unitName="producerfieldsPU")
    private EntityManager em;
}
```
@ConversationScoped
@Stateful
public class RequestBean {

  // Inject Java EE resource in type-safe way, no more string
  @Inject @UserDatabase
  EntityManager em;

  public ToDo createToDo(String inputString) {
    ...
    try {
      ...
      em.persist(toDo);

      return toDo;
    } catch (Exception e) {
      throw new EJBException(e.getMessage());
    }
  }
}
Demo:

cdi_producer_UserDatabase_db
4533_javaee6_cdi_advanced.zip
Events
CDI Event Observer Pattern

• Completely decouple action (event producer) and reactions (event consumers)
• Qualifiers tune which event notifications are received
CDI Event Observer Pattern

- Define Event Class
- Event producer fires an event
- Event consumer observes event through @Observes
**Event Class**

- Event class can be any POJO class

```java
public class LoggedInEvent {
    private String user;

    public LoggedInEvent(String user) {
        this.user = user;
    }
}
```
Event Producers

- An event is fired by an injected javax.enterprise.event.Event object

```java
public class Login {

    @Inject
    Event<LoggedInEvent> loggedInEvent;

    public void login() {
        loggedInEvent.fire(
            new LoggedInEvent(credentials.getUsername()));
    }
}
```
Event Consumer (Event Observer)

- The only thing event consumer has to do is to use `@Observes` `<Event-class>` annotation

```java
// This method gets invoked when LoggedInEvent is fired
public void afterLogin(@Observes LoggedInEvent event) {
    System.out.println("afterLogin() method is called, event = " + event);
}
```
Demo:

weld-servlet-event

4533_javaee6_cdi_advanced.zip
Event with Qualifier

- Event can be selectively fired and received through qualifier

```java
public class Login {

    @Inject
    @Admin
    Event<LoggedInEvent> loggedInEvent;

    // Event producer
    public void login() {
        loggedInEvent.fire(
            new LoggedInEvent(credentials.getUsername()));
    }

    // Event consumer
    public void afterAdminLogin(@Observes @Admin LoggedInEvent event) {
        System.out.println("----afterAdminLogin() method is called, event = " + event);'
    }
}
```
Demo:

weld-servlet-event-qualifier
4533_javaee6_cdi_advanced.zip
Alternative
What is Alternative Bean?

- Any bean with `@Alternative` is not considered for injection
  - Lets you package multiple beans that match injection type without ambiguity errors
  - In order to be considered for injection, it has to be activated in “beans.xml”

- Provide a replacement implementation during deployment
  - You can apply the `@Alternative` annotation to two or more beans, then, based on your deployment, specify the bean you want to use in the “beans.xml” configuration file
  - Useful for providing mock objects for testing – mock objects are annotated with `@Alternative`
    - In normal operation, alternative bean will not be considered for injection
    - In testing operation, activate it via “beans.xml”
/ Annotate alternative implementation with @Alternative annotation
@Alternative
public class TestCoderImpl implements Coder {

    public String codeString(
        String s,
        int tval) {
        return ("input string is " + s + ", shift value is " + tval);
    }
}

// Activate it in "beans.xml"
<beans>
    <alternatives>
        <class>encoder.TestCoderImpl</class>
    </alternatives>
</beans>
Demo: 

`cdi_alternative_encoder`

`4533_javaee6_cdi_advanced.zip`
Decorator
What is a Decorator?

- Decorators implement the Decorator design pattern
  > Allows implementation of an additional business logic for a bean
- A Decorator decorates interfaces they implement
- @Delegate is used to inject the original object
  > Original object business logic can be be invoked within the decorator
- Decorators must be activated through “beans.xml”
Define a Decorator

@Decorator
public abstract class CoderDecorator implements Coder {

@Inject
@Delegate
@Any
Coder coder;

public String codeString(
    String s,
    int tval) {
    int len = s.length();

    // The decorator's codeString method calls the delegate
    // object's codeString method to perform the actual encoding.
    return "\"" + s + "\" becomes " + "\" + coder.codeString(s, tval)
    + "\", " + len + " characters in length";
}
}
Enable Decorator

- Decorator has to be activated through “beans.xml”

```xml
<beans>
  <decorators>
    <class>decorators.CoderDecorator</class>
  </decorators>
</beans>
```
Demo:

`cdi_decorators`

`4533_javaee6_cdi_advanced.zip`
JRebel Demo
Quick Survey on JRebel

• How many of you ever heard of JRebel?
• How many of you know what JRebel does?
• How many of you ever used JRebel?

• Please fill up feedback form – 1 year free license will be given to a winner
• You can get JRebel 1-month free license

> http://zeroturnaround.com
Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps