Context and Dependency Injection (CDI) (JSR 299) Basics

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- What is and Why Dependency Injection?
- What is and Why CDI (JSR 299)? What about JSR 330?
- CDI theme – Loose coupling with strong 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 Dependency Injection (DI)?

What is and Why Dependency Injection?

- Classes specify what their dependencies are NOT how to obtain them
  - "How to obtain" is now handled by the container
- Why Dependency injection?
  - Dependency injection makes unit-testing and mocking easier
  - Dependency injection allows container to do the bean-discovery and bean-wiring
- Dependency Injection frameworks (containers)
  - Spring
  - Guice
  - Seam
  - EJB 3.x
  - CDI (the latest, the best, and standard-based)

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 (this was the original goal of JSR 299)
  - Let you use EJBs directly as JSF managed beans

Why CDI (JSR 299) for Java EE 6?

- 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, unlike Spring framework, the Java EE 5 does not provide general-purpose dependency injection scheme
- Reason #2: We need type-based injection
  - Non-type-based injection (String name or XML based injection) is fragile
  - Type-based injection enables better tooling in general

Terminology

- CDI (JSR 299)
  - Context & Dependency Injection for Java EE
- Weld
  - JSR 299 reference implementation
  - Provides extended CDI support for Servlet container (Tomcat, Jetty, etc) and Java SE
  - CDI enhancements for extension writers
  - Maven archetypes for CDI and Java EE

JSR 299 vs JSR 330

- JSR 299 (CDI) is built upon JSR 330 (Dependency Injection for Java)
  - The relationship between JSR 299 and JSR 299 is like the one between JPA and JDBC
- JSR 330 defines
  - Inject, Qualifier, Scope, Singleton, Named, and Provider
- JSR 299 enhances JSR 330 significantly with
  - Modularization (loose-coupling), cross cutting aspects (decorators, interceptors), custom scopes, or type safe injection capabilities

CDI Theme:
“Loose Coupling with Strong Typing”

CDI Theme: Loose Coupling

- Decouple dependency provider and dependency user
  - Using well-defined types and qualifiers – dependency is defined as a Type and user of the dependency searches the dependency using Type
  - Allows provider implementations to vary without affecting user
- Decouple lifecycle of collaborating components (dependencies) from application (dependency user)
  - Automatic contextual life-cycle management by the CDI runtime
- Decouple orthogonal concerns (AOP) from business logic
  - Interceptors & Decorators
- Decouple message producer from consumer
  - Events

CDI Theme: Strong Typing

- Type-based injection has advantages of
  - No more reliance on string-based names
  - Compiler can detect type errors at compile time
  - Casting mostly eliminated
  - Strong tooling possible
- Semantic code errors (errors that cannot be detected by the compiler) can be detected at application start-up
  - Tools can detect ambiguous dependencies
- Leverages Java type system
  - @Annotation
  - <TypeParam>

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
- Support a small set of common basic services
  - Life-cycle management (@PostConstruct, @PreDestroy)
  - Injection of a resource (@Resource...)
  - Interceptor (@Interceptors, @AroundInvoke)

Managed Beans 1.0: Example

```
@javax.annotation.ManagedBean
public class MyPojo {

  @Resource    // Resource injection
  private Datasource ds;

  @PostConstruct  // Life-cycle
  private void init() {
    ....
  }

  @Interceptors(LoggingInterceptor.class)
  public void myMethod() {...}
}
```

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

We will cover these in this and other CDI presentations in detail.

CDI Bean Example

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

```
// This is a valid CDI bean – a CDI client that is looking for
// Greeting type CDI bean will be injected with instance of
// this bean.
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name;
    }
}
```

CDI + EJB Bean Example

- Now EJB services – transaction, security, etc are added to the bean

```
// A CDI client that is looking for Greeting2 type CDI bean
// will be injected with instance of this bean, which has
// EJB bean services built-in.
@Stateless
public class Greeting2 {
    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 application archives for bean discovery?
  - > By detecting the presence of “beans.xml” in application archive
- “beans.xml”
  - > Unlike Spring, it is NOT for declaring beans - It can be empty
  - > Used for some other purposes (like declaring an alternative)
  - > Optional in Java EE 7

Basic Injection

How do you inject a Bean?

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

```
public class MyGreeter {

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

   public sayGreeting(String name){
      // You can then used the injected Greeting object
      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 method (useful for creating immutable object)
  - Initializer method
  - Setter method
  - Producer (will be covered in “CDI Advanced”)
  - Observer (will be covered in “CDI Advanced”)

Example: Constructor Injection Point

```
public class MyGreeter {

    private Greeting greeting;

     // Use constructor method injection
     @Inject
     public MyGreeter(Greeting greeting) {
         this.greeting = greeting;
     }

    public sayGreeting(String name){
        System.out.println(greeting.greet(name));
    }
 }
```

Lab:

Exercise 1: @Inject Simple Cases
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
- A qualifier is an annotation that lets a client choose one among multiple candidates of a certain type
  - Make type more specific
  - A qualifier could have more semantically meaningful name
- Injected type is identified by
  - *Qualifier(s)* + *Java type*
  - *e.g. @Inject @LoggedIn User user;*

Qualifier

Java type

How to build and use Qualifier? – 3 steps

- Define a qualifier (type)
- Qualify a class
- Select a qualified class

Define a Qualifier (Type)

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

Qualify a Class

```
// 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;
   }
}
```

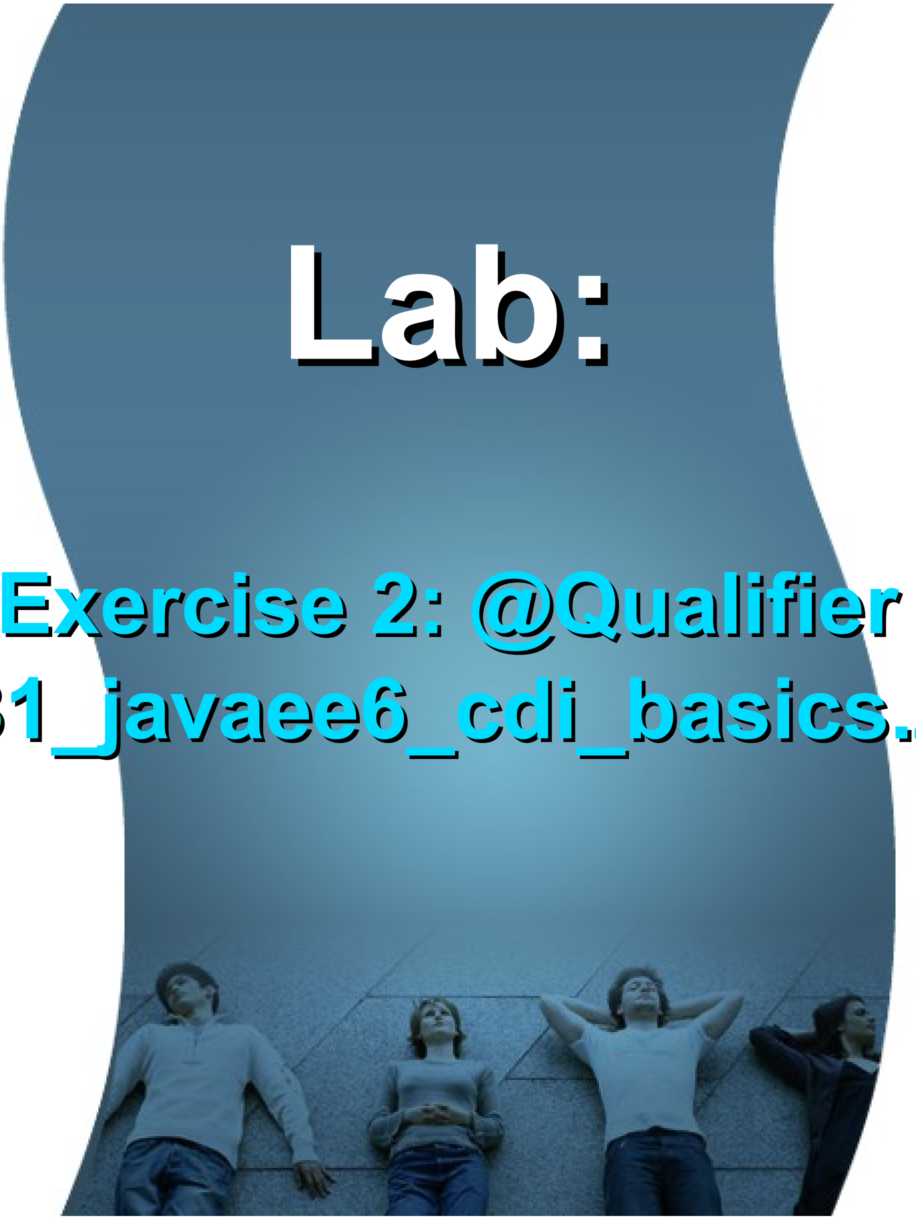
Select Qualified Class

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

```
public class MyGreeter {

  //  Injected type is identified by @Informal qualifier and Greeting type.
  //  So InformalGreeting class (of previous slide) will be chosen
  //  instead of just any Greeting type class for dependency injection.
  @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
  - Again, think of a Qualifier as a type
- Qualifiers make type safe injection possible
  - Qualifiers replace “look-up via string-based names”



Lab:

Exercise 2: @Qualifier
4531_javaee6_cdi_basics.zip

Qualifier with Attributes

- Define a Qualifier with attributes

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

- Qualify a class with qualifier

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

Qualifier with Attributes

- Select qualified class

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

Lab:

Exercise 3: Qualifier with Attributes
4531_javaee6_cdi_basics.zip

@Named
Built-in Qualifer

When do we use @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) – the reason why we need @*Named* annotation

```
<h:commandButton value="Say Hello"
                 action="#{myprinter.greet}"/>
```

- The use of @Named as an injection point qualifier is not recommended, except in the case of integration with legacy code that uses string-based names to identify beans

Why do we need @Named Annotation?

- Give it a name using *@Named* annotation

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

    @Inject Greeting greeting;

    public void greet() {
        System.out.println( greeting.greet("world") );
    }
}
```

Default name

- If the value is not specified, the default name is the name of the class (starting with lower case)

```
public
@Named    // Default name is “printer”
class Printer {

    @Inject Greeting greeting;

    public void greet() {
        System.out.println( greeting.greet("world") );
    }
}
```

Stateful Objects (Scoped Objects)

Why Do We need Stateful Objects?

- For Web applications, we need our beans to hold state over the duration of the user’s interaction with the application, for example, across multiple requests to the server

```
<!-- If you don't specify Request scope for the “printer” bean class,
      which is shown in the next slide, then the 2nd #{printer.greet}
      will result in null because bean object gets created in
      Dependent scope, which is default.  In Dependent scope,
      the “printer” bean of inputText object is different from
      the “printer” bean of the commandButton object.  -->
<h:form>
   <h:inputText value="#{printer.name}"/>
   <h:commandButton value="Say Hello"
                            action="#{printer.greet}"/>
</h:form>
```

Why do we need Scope?

- If we want our object to hold state, we need to declare the scope of that state

```
public
@RequestScoped  // Printer object is in Request scope
@Named
class Printer {

    @Inject Greeting greeting;
    private String name;

    public void setName(String name) { this.name=name; }
    public String getName() { return name; }
    public void greet() {
        System.out.println( greeting.greet(name) );
    }
}
```

A Scoped Object in CDI

- A scope gives an object a well-defined lifecycle context
  - A scoped object can be automatically created “by the container” when it is needed and automatically destroyed “by the container” when the context in which it was created ends (instead of developer is responsible for creating/destroying it in the correct scope)
  - The client of the scoped object doesn’t know anything about the lifecycle of the scoped object
  - Moreover, its state is automatically shared by any clients that execute in the same context – an application-scoped object can be shared among all clients talking to the application

Injecting Scoped Objects

- The client does not need to know (does not want to know) anything about the lifecycle of the scoped objects

```
public @Named
class Printer {

    // The client does not know anything about the lifecycle of the
    // scoped objects
    @Inject Greeting greeting;
    @Inject Login login;

    public void greet() {
        System.out.println(
                greeting.greet(login.getUser().getName()));
    }
}
```

Built-in Scopes

- @Dependent (default)
  - The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
- @RequestScoped
  - State of an object is maintained during a user's interaction with a web application in a single HTTP request.
- @SessionScoped
  - State of an object is maintained during user's interaction with a web application across multiple HTTP requests.
- @ApplicationScoped
  - Shared state across all users' interactions with a web application.
- @ConversationScoped

@ConversationScoped

- Conversation context is demarcated explicitly by the application
  - Spans multiple requests
  - But “Smaller” than session
- Used when you want to have explicit boundaries of multiple request/response conversations within a single session

Demarcation of Conversation Context

```
public @ConversationScoped
class NumberGuess {

  @Inject Conversation conversation;
  private int number;
  private int min;
  private int max;

  @Inject
  void start(@Random int random) {
    conversation.begin();
    number = random;
    min = 1;
    max = 100;
  }

// Continued to the next slide
```

Demarcation of Conversation Context

```
  // Continued from previous page

  public boolean guess(int guess) {
      if (guess == number) {
          conversation.end();
          return true;
      } else {
          if (guess < number && guess > min) {
              min = guess;
          } else if (guess > number && guess < max) {
              max = guess;
          }
          return false;
      }
   }
}
```

@New

- Built-in qualifier
- The @New qualifier allows the application to obtain a new instance of a bean which is not bound to the declared scope, but has had dependency injection performed

```
@Produces @ConversationScoped
@Special Order getSpecialOrder(@New Order order) {
    ...
    return order;
}
```

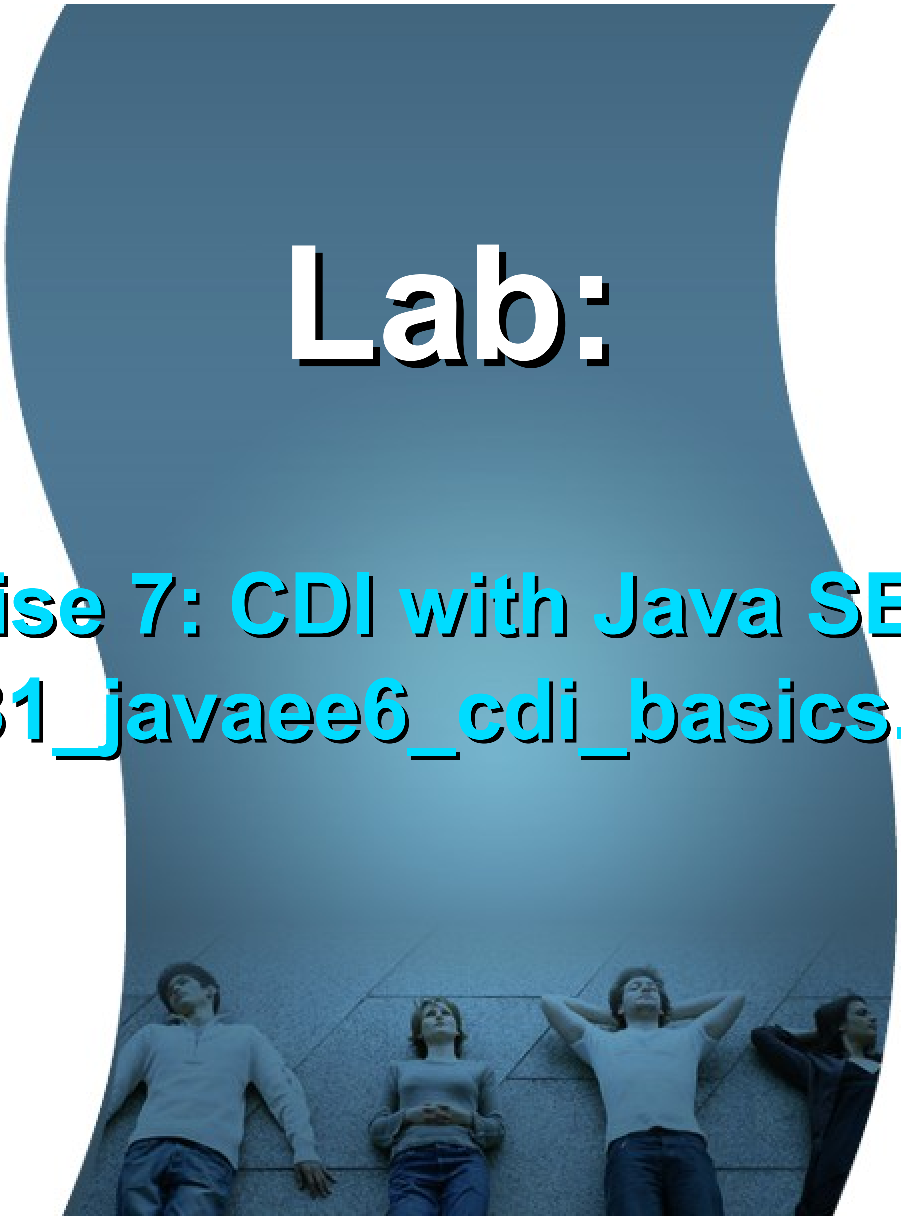
Lab:

Exercise 4: Scope
4531_javaee6_cdi_basics.zip

CDI for Java SE Application

CDI Can be use for Java SE app

```
public class Main {

    static BeanContainer beanContainer =
                                BeanContainerManager.getInstance();

    public static void main(String[] args) throws Exception {
            Greeting greeting = (Greeting) beanContainer
.                                              .getBeanByType(Greeting.class);

            System.out.println(greeting.greet("Sang Shin"));

     }

}
```



Lab:

Exercise 7: CDI with Java SE apps
4531_javaee6_cdi_basics.zip

Learn with Passion!
JPassion.com