



JSR-299: Contexts and Dependency Injection for Java EE

Dan Allen

Senior Software Engineer, RedHat

August 19, 2009

What JSR-299 provides

- A powerful set of new *services* for Java EE components
 - Life-cycle management of stateful components bound to well-defined contexts
 - A type-safe approach to dependency injection
 - Bean names to support Unified EL integration
 - A web conversation context
 - Interceptors decoupled from bean class
 - An event notification model
 - A complete SPI that allows portable extensions to integrate cleanly with the Java EE environment



The big picture

- Fills a major hole in the Java EE platform
- A catalyst for emerging Java EE specs
- Excels at solving stated goal



Stated goal



Transactional tier
(EJB)

Web tier
(JSF)



Going beyond Seam

- JSF-EJB integration problem still needed to be solved
 - Solve at platform level
- Get an EG involved
 - Buy-in from broader Java EE community
 - Formulate a better, more robust design



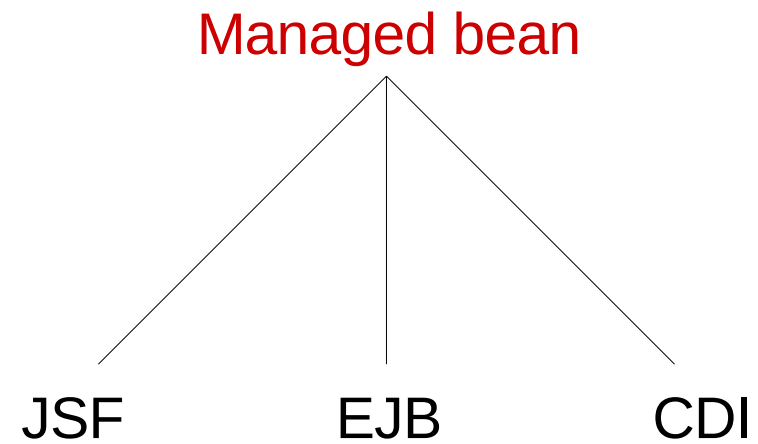
Your bean is my bean

- Everyone trying to solve the same problem
 - JSF, EJB, CDI (JSR-299), Seam, Spring, Guice, etc.
- Need a “unified bean definition”
- Can build from there



Managed bean

- Common bean definition
- Life cycle of instance managed by container
- Basic set of services
 - Resource injection
 - Life-cycle callbacks
 - Interceptors
- Foundation on which other specs can build



Why injection?

- Injection is the weakest aspect of Java EE
- Existing annotations pertain to specific components
 - `@EJB`
 - `@PersistenceContext` / `@PersistenceUnit`
 - `@Resource` (e.g., `DataSource`, `UserTransaction`)
- Third-party solutions rely on name-based injection
 - Not type-safe
 - Fragile
 - Requires special tooling to validate



Leverage and extend Java's type system

- JSR-299 introduces creative use of annotations
- Annotations considered part of type
- Comprehensive generics support
- Why augment type?
 - Can't always rely on class extension (e.g., primitives)
 - Avoid creating hard dependency between client and implementation
 - Don't rely on weak association of field => bean name
 - Validation can be done at startup



JSR-299 theme

Loose coupling...

@Inject
@InterceptorBinding
@Observes
@Qualifier

```
@Produces @WishList  
List<Product> getWishList()
```

```
Event<Order>
```

```
@UserDatabase EntityManager
```

...with strong typing



Loose **coupling**

- Decouple server and client
 - Using well-defined types and “qualifiers”
 - Allows server implementation to vary
- Decouple life cycle of collaborating components
 - Automatic contextual life cycle management
 - Stateful components interact like services
- Decouple orthogonal concerns (AOP)
 - Interceptors
 - Decorators
- Decouple message producer from message consumer
 - Events



Strong typing

- Eliminate reliance on string-based names
- Compiler can detect typing errors
 - No special authoring tools required for code completion
 - Casting virtually eliminated
- Semantic code errors detected at application startup
 - *Tooling can detect ambiguous dependencies*



What can be injected?

- Defined by the specification
 - Almost any plain Java class (managed beans)
 - EJB session beans
 - Objects returned by producer methods or fields
 - Java EE resources (e.g., Datasource, UserTransaction)
 - Persistence units and persistence contexts
 - Web service references
 - Remote EJB references
- Open book
 - SPI allows third-party frameworks to introduce additional injectable objects



CDI bean

- Set of bean types (non-empty)
- Set of qualifiers (non-empty)
- Scope
- Bean EL name (optional)
- Set of interceptor bindings
- An implementation



Bean services with CDI

- @ManagedBean annotation not required (implicit)
- Transparent create/destroy and scoping of instance
- Type-safe resolution at injection or lookup
- Name-based resolution when used in EL expression
- Life cycle callbacks
- Method interception and decoration
- Event notification



Welcome to CDI (managed bean version)

```
public class Welcome {  
    public String buildPhrase(String city) {  
        return "Welcome to " + city + "!";  
    }  
}
```

- When is a bean recognized?

/META-INF/beans.xml must be in same classpath entry



Welcome to CDI (session bean version)

```
public
@Stateless
class WelcomeBean implements Welcome {
    public String buildPhrase(String city) {
        return "Welcome to " + city + "!";
    }
}
```



A simple client: field injection

```
public class Greeter {  
    @Inject Welcome welcome;  
  
    public void welcome() {  
        System.out.println(  
            welcome.buildPhrase("Mountain View"));  
    }  
}
```

@Current annotation implied



A simple client: constructor injection

```
public class Greeter {  
    Welcome welcome;  
  
    @Inject  
    public Greeter(Welcome welcome) {  
        this.welcome = welcome  
    }  
  
    public void welcomeVisitors() {  
        System.out.println(  
            welcome.buildPhrase("Mountain View"));  
    }  
}
```

Designates the constructor
CDI should invoke



A simple client: initializer injection

```
public class Greeter {
    Welcome welcome;

    @Inject
    void init(Welcome welcome) {
        this.welcome = welcome
    }

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase("Mountain View"));
    }
}
```

Designates the initializer method CDI should invoke



Multiple implementations

- Two scenarios:
 - Multiple implementations of same interface
 - One implementation extends another

```
public class TranslatingWelcome extends Welcome {  
  
    @Inject GoogleTranslator translator;  
  
    public String buildPhrase(String city) {  
        return translator.translate(  
            "Welcome to " + city + "!");  
    }  
}
```

- Which implementation should be selected for injection?



Qualifier

An annotation associated with a type that is satisfied by some implementations of the type, but not necessarily by others.

Used to resolve a implementation variant of an API at an injection or lookup point.



Defining a qualifier

- A qualifier is an annotation

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



Qualifying an implementation

- Add qualifier annotation to make type more specific

```
public
@Translating
class TranslatingWelcome extends Welcome {

    @Inject GoogleTranslator translator;

    public String buildPhrase(String city) {
        return translator.translate(
            "Welcome to " + city + "!");
    }
}
```

- Resolves ambiguity at injection point
 - *There can never been an ambiguity when resolving!*



Using a specific implementation

- Must request to use qualified implementation explicitly
 - Otherwise you get unqualified implementation

```
public class Greeter {
    Welcome welcome;

    @Inject
    void init(@Translating Welcome welcome) {
        this.welcome = welcome
    }

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase("Mountain View"));
    }
}
```

No reference to implementation class!



Alternative bean

- Swap replacement implementation per deployment
- Replaces bean and its producer methods and fields
- Disabled by default
 - Must be activated in /META-INF/beans.xml

Put simply: **an override**



Defining an alternative

```
public
@Alternative
@Specializes
class TranslatingWelcome extends Welcome {

    @Inject GoogleTranslator translator;

    public String buildPhrase(String city) {
        return translator.translate(
            "Welcome to " + city + "!");
    }
}
```



Substituting the alternative

- Implementation activated using deployment-specific /META-INF/beans.xml resource

```
<beans>  
  <alternatives>  
    <class>com.acme.TranslatingWelcome</class>  
  </alternatives>  
</beans>
```

- Could also enable alternative by introducing and activating an intermediate annotation



Assigning a bean name

```
public
@Named("greeter")
class Greeter {
    Welcome welcome;

    @Inject
    public Greeter(Welcome welcome) {
        this.welcome = welcome
    }

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase("Mountain View"));
    }
}
```

Same as default name when
no annotation value specified



Assigning a bean name

```
public
@Named
class Greeter {
    Welcome welcome;

    @Inject
    public Greeter(Welcome welcome) {
        this.welcome = welcome
    }

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase("Mountain View"));
    }
}
```



Collapsing layers

- Use the bean directly in the JSF view

```
<h:form>  
  <h:commandButton value="Welcome visitors"  
    action="#{greeter.welcomeVisitors}"/>  
</h:form>
```

- But we still need the bean to be stored in a scope



A stateful bean

- Declare bean to be saved for duration of request

```
public
@RequestScoped
@Named("greeter")
class Greeter {
    Welcome welcome;
    private String city;

    @Inject public Greeter(Welcome welcome) {
        this.welcome = welcome
    }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase(city));
    }
}
```



Collapsing layers with state management

- Now it's possible for bean to hold state

```
<h:form>  
  <h:inputText value="#{greeter.city}"/>  
  <h:commandButton value="Welcome visitors"  
    action="#{greeter.welcomeVisitors}"/>  
</h:form>
```

- Satisfies initial goal of integrating JSF and EJB
 - Except in this case, it extends to plain managed beans



Scope types and contexts

- Absence of scope - `@Dependent`
 - Bound to life cycle of bean holding reference to it
- Servlet scopes
 - `@ApplicationScoped`
 - `@RequestScoped`
 - `@SessionScoped`
- JSF-specific scope
 - `@ConversationScoped`
- Custom scopes
 - Define scope type annotation
 - Implement context API



Scope transparency

- Scopes are not visible to client
 - No coupling between scope and use of type
 - Scoped beans are proxied for thread safety



Scoping a collaborating bean

```
public
@SessionScoped
class Profile {
    private Identity identity;

    public void register() {
        identity = ...;
    }

    public Identity getIdentity() {
        return identity;
    }
}
```



Collaboration between stateful beans


```
public
@RequestScoped @Named
class Greeter {
    Welcome welcome;
    private String city;

    @Inject
    public Greeter(Welcome welcome, Profile profile) {
        this.welcome = welcome
    }

    ...

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase(profile.getIdentity(), city));
    }
}
```

No awareness of scope



Conversation context

- Request \leq Conversation \ll Session



- Boundaries demarcated by application

- Optimistic transaction 👍
 - Conversation-scoped persistence context
 - No fear of exceptions on lazy fetch operations



Controlling the conversation

```
public
@ConversationScoped
class BookingAgent {

    @Inject @BookingDatabase EntityManager em;
    @Inject Conversation conversation;

    private Hotel selectedHotel;
    private Booking booking;

    public void select(Hotel hotel) {
        selectedHotel = em.find(Hotel.class, hotel.getId());
        conversation.begin();
    }

    ...
}
```



Controlling the conversation

...

```
public boolean confirm() {  
    if (!isValid()) {  
        return false;  
    }  
  
    em.persist(booking);  
    conversation.end();  
    return true;  
}  
}
```



Producer method

A method whose return value is a source of injectable objects.

Used for:

- Types which you cannot modify
- Runtime selection of a bean instance
- When you need to do extra and/or conditional setup of a bean instance
- Roughly equivalent to Seam's @Factory annotation



Producer method examples

@Produces

```
public PaymentProcessor getPaymentProcessor(  
    @Synchronous PaymentProcessor sync,  
    @Asynchronous PaymentProcessor async) {  
    return isSynchronous() ? sync : async;  
}
```

@Produces @SessionScoped @WishList

```
public List<Product> getWishList() { ... }
```



Disposal method

- Used for cleaning up after a producer method
 - Matched using type-safe resolution algorithm
- Called when produced bean goes out of scope

```
public class UserRepositoryManager {  
  
    @Produces @UserRepository  
    EntityManager create(EntityManagerFactory emf) {  
        return emf.createEntityManager();  
    }  
  
    void close(@Disposes @UserRepository EntityManager em) {  
        em.close();  
    }  
}
```



Bridging Java EE resources

- Use producer field to set up Java EE resource for type-safe resolution

```
public
@Stateless
class UserEntityManagerFactory {
    @Produces @UserDatabase
    @PersistenceUnit(unitName = "userDatabase")
    EntityManagerFactory emf;
}
```

```
public
@Stateless
class PricesTopic {
    @Produces @Prices
    @Resource(name = "java:global/env/jms/Prices")
    Topic pricesTopic;
}
```

Java EE resource annotations

Java EE 6 global JNDI name



Injecting resource in type-safe way

- String-based resource names are hidden

```
public class UserManager {  
    @Inject @UserDatabase EntityManagerFactory emf;  
    ...  
}
```

```
public class StockDisplay {  
    @Inject @Prices Topic pricesTopic;  
    ...  
}
```



Promoting state

- Producer methods can be used to promote state of a bean as an injectable object

```
public
@RequestScoped
class Profile {
    private Identity identity;

    public void register() {
        identity = ...;
    }

    @Produces @SessionScoped
    public Identity getIdentity() {
        return identity;
    }
}
```

Could also declare
qualifiers and/or EL name



Using promoted state

```
public
@RequestScoped @Named
class Greeter {
    Welcome welcome;
    private String city;

    @Inject
    public Greeter(Welcome welcome, Identity identity) {
        this.welcome = welcome
    }

    ...

    public void welcomeVisitors() {
        System.out.println(
            welcome.buildPhrase(identity, city));
    }
}
```

No awareness of scope



Rethinking interceptors

- Interceptors bound directly to component in Java EE 5
 - `@Interceptors` annotation on bean type
- What's the problem?
 - Should not be coupled to implementation
 - Requires level of indirection
 - Should be deployment-specific
 - Tests vs production
 - Opt-in best strategy for enabling
 - Ordering should be defined centrally



Interceptor wiring in JSR-299 (1)

- Define an interceptor binding type

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



Interceptor wiring in JSR-299 (2)

- Marking the interceptor implementation

```
public
@Secure
@Interceptor
class SecurityInterceptor {

    @AroundInvoke
    public Object aroundInvoke(InvocationContext ctx)
        throws Exception {
        // enforce security
        ctx.proceed();
    }
}
```



Interceptor wiring in JSR-299 (3)

- Applying interceptor to class with proper semantics

```
public
@Secure
class AccountManager {
    public boolean transferFunds(Account a, Account b) {
        ...
    }
}
```



Interceptor wiring in JSR-299 (4)

- Applying interceptor to method with proper semantics

```
public class AccountManager {  
  
    public  
    @Secure  
    boolean transferFunds(Account a, Account b) {  
        ...  
    }  
  
}
```



Multiple interceptors

- Application developer only worries about relevance

```
public
@Transactional
class AccountManager {

    public
    @Secure
    boolean transferFunds(Account a, Account b) {
        ...
    }
}
```



Enabling and ordering interceptors

- Interceptors referenced by binding type
- Specify binding type in /META-INF/beans.xml to activate

```
<beans>  
  <interceptors>  
    <class>com.acme.SecurityInterceptor</class>  
    <class>com.acme.TransactionInterceptor</class>  
  </interceptors>  
</beans>
```

Interceptors applied in order listed



Composite interceptor bindings

- Interceptor binding types can be meta-annotations

```
public  
@Secure  
@Transactional  
@InterceptorBinding  
@Retention(RUNTIME)  
@Target(TYPE)  
@interface BusinessOperation {}
```

Order does not matter



Multiple interceptors (but you won't know it)

- Interceptors inherited from composite binding types

```
public
@BusinessOperation
class AccountManager {
    public boolean transferFunds(Account a, Account b) {
        ...
    }
}
```



Wrap up annotations using *stereotypes*

- Common architectural patterns – recurring roles
- A stereotype packages:
 - A default scope
 - A set of interceptor bindings
 - The ability to that beans are named
 - The ability to specify that beans are alternatives



Annotation jam

- Without stereotypes, annotations pile up

```
public
@Secure
@Transactional
@RequestScoped
@Named
class AccountManager {

    public boolean transferFunds(Account a, Account b) {
        ...
    }
}
```



Defining a stereotype

- Stereotypes are annotations that group annotations

```
public
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
@interface BusinessComponent {}
```



Using a stereotype

- Stereotypes give a clear picture, keep things simple

```
public
@BusinessComponent
class AccountManager {

    public boolean transferFunds(Account a, Account b) {
        ...
    }
}
```



Events

- Completely decouples action and reactions
- Observers can use selectors to tune which event notifications are received
- Events can be observed immediately, at end of transaction or asynchronously



Firing an event

Event instance with
type-safe payload

```
public class GroundController {  
    @Inject @Landing Event<Flight> flightLanding;  
  
    public void clearForLanding(String flightNum) {  
        flightLanding.fire(new Flight(flightNum));  
    }  
}
```



An event observer

```
public class GateServices {  
    public void onIncomingFlight(  
        @Observes @Landing Flight flight,  
        Greeter greeter,  
        CateringService cateringService) {  
        Gate gate = ...;  
        flight.setGate(gate);  
        cateringService.dispatch(gate);  
        greeter.welcomeVisitors();  
    }  
}
```

Takes event API type with additional binding type

Additional parameters are injected by the container



Summary

- JSR-299 satisfies original goal to integrate JSF and EJB
- Managed bean specification emerged from JSR-299
- More problems needed to be solved
 - Robust dependency injection model
 - Further loose-coupling with events
 - Extensive SPI to integrate third-party with Java EE
- JSR-299 offers loose coupling with **strong typing**



JSR-299 status

- Conflict with JSR-330 resolved
- Proposed final draft published
- TCK nearly complete
- Send feedback to jsr-299-comments@jcp.org
- <http://jcp.org/en/jsr/detail?id=299>



Web Beans

- JSR-299 reference implementation
- Developed by Red Hat and community
- Feature complete (for second public draft)
 - Look for CR1 ~ JBoss World 2009
- <http://seamframework.org/Download>





Q & A

<http://in.relation.to/Bloggers/Dan>
<http://seamframework.org/WebBeans>

Dan Allen
Senior Software Engineer, RedHat
August 18, 2009