JSR-299 (CDI), Weld and the Future of Seam

Dan Allen
Principal Software Engineer
JBoss by Red Hat

Agenda

- Java EE today
- Where JSR-299 fits in
- JSR-299 themes
- CDI programming model tour
- CDI extensions
- Weld
- Seam 3

Technology terminology

- JSR-299 (CDI)
  - **C**ontexts & **D**ependency **I**njection for the Java EE Platform
- Weld
  - JSR-299 Reference Implementation & TCK
  - Extended CDI support (Servlets, Java SE)
  - *Portable* CDI enhancements for extension writers
- Seam 3
  - *Portable* extensions for Java EE
  - *Portable* integrations with non-Java EE technologies

What is Java EE?

- Standard platform comprised of managed components & services

$\downarrow$

- Business logic as components
  1. Less code
  2. Higher signal-to-noise ratio
  3. Powerful mechanisms for free
  4. Portable knowledge

Why reinvest?

Stated goal of JSR-299

Web tier
(JSF)

Transactional tier
(EJB)

What CDI provides

- *Services* for Java EE components
  - Lifecycle management of stateful beans bound to well-defined contexts (including conversation context)
  - A type-safe approach to dependency injection
  - Interaction via an *event notification* facility
  - Reduced coupling between interceptors and beans
  - Decorators, which intercept specific bean instances
  - Unified EL integration (bean names)
- *SPI* for developing extensions for the Java EE platform
  - Java EE architecture → flexible, portable, extensible

What CDI provides

contexts

dependency injection

event notification

for the Java EE platform

CDI: The big picture

- Fill in
- Catalyze
- Evolve

Why dependency injection?

- Weakest aspect of Java EE 5
- Closed set of injectable resources
  - @EJB
  - @PersistenceContext, @PersistenceUnit
  - @Resource (e.g., DataSource, UserTransaction)
- Name-based injection is fragile
- Lacked rules

Leverage and extend Java’s type system

`@Annotation` `<TypeParam>`

This information is pretty useful!

`Type`

JSR-299 theme

Loose coupling...

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

`Event<Order>`

`@InterceptorBinding`

`@Inject`

`@UserDatabase EntityManager`

`@Observes`

`@Qualifier`

...with **strong typing**

Loose coupling

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

Strong typing

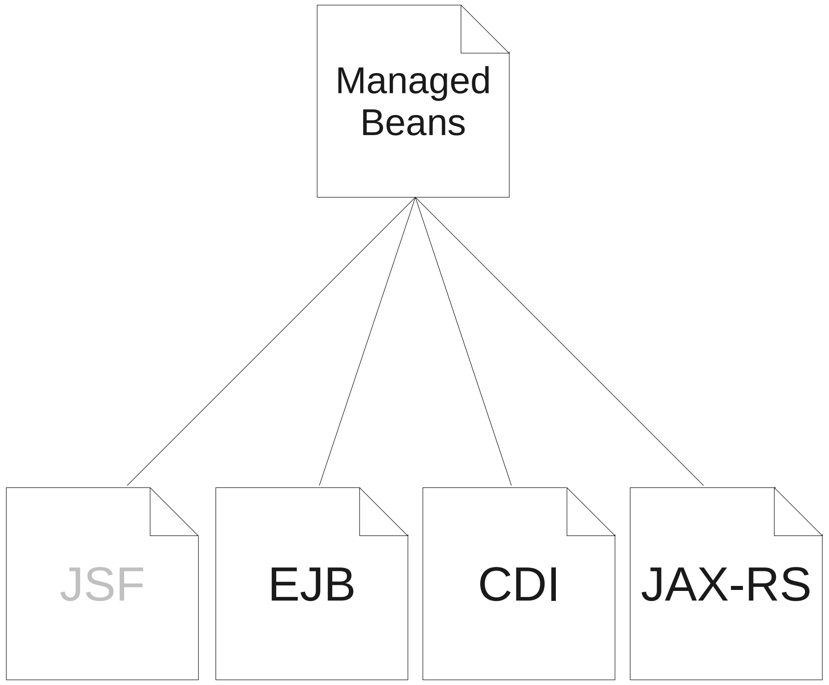
- Type-based injection
  - Eliminate reliance on string-based names
  - Refactor friendly
- Compiler can detect typing errors
  - No special authoring tools required
  - Casting mostly eliminated
- Semantic code errors detected at application startup
- Tooling can detect ambiguous dependencies (optional)

Who's bean is it anyway?

- Everyone throwing around this term “bean”
  - JSF
  - EJB
  - Seam
  - Spring
  - Guice
  - Web Beans
- Need a “unified bean definition”

Managed bean specification

- Common bean definition
- Instances managed by the container
- Common services
    - Lifecycle callbacks
    - Resource injections
    - Interceptors
- Foundation spec



How managed beans evolved: http://www.infoq.com/news/2009/11/weld10

CDI bean ingredients

- Set of bean types
- Set of qualifiers
- Scope
- Bean EL name (optional)
- Set of interceptor bindings
- Alternative classification
- Bean implementation class

Auto-discovered!

Welcome to CDI, managed beans!

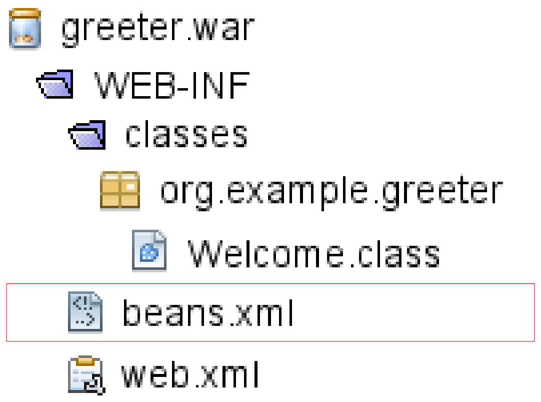
```
public class Welcome {
   public String buildPhrase(String city) {
      return "Welcome to " + city + "!";
   }
}
```

Welcome to CDI, EJB 3.1 session beans!

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

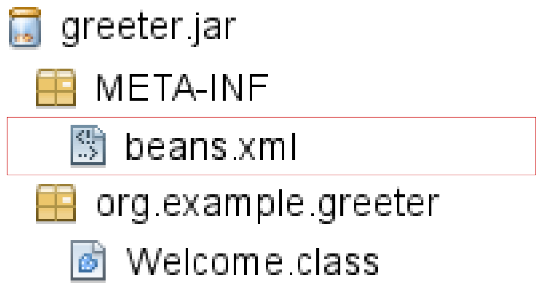
When is a bean recognized?

- Bean archive (WAR)



```
greeter.war
  WEB-INF
    classes
      org.example.greeter
        Welcome.class
    beans.xml
    web.xml
```

- Bean archive (JAR)



```
greeter.jar
  META-INF
    beans.xml
  org.example.greeter
    Welcome.class
```

beans.xml can be empty!

Injection 101

```
public class Greeter {
   @Inject Welcome w;

   public void welcome() {
      System.out.println(
         w.buildPhrase("San Francisco"));
   }
}
```

Where can it be injected?

- Field
- Method parameter
  - Constructor*
  - Initializer
  - Producer
  - Observer

What can be injected?

<u>Managed bean</u>	✓
Object returned by producer	✓
EJB session bean (local or remote)	✓
Java EE resource (DataSource, JMS destination, etc)	✓
JTA UserTransaction	✓
Persistence unit or context	✓
Security principle	✓
Bean Validation factory	✓
Web service reference	✓
Additional resources introduced through SPI	✓

The bean vs “the other implementation”

- Multiple implementations of same interface
- One implementation extends another

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

```
public class TranslatingWelcome extends Welcome {

   @Inject GoogleTranslator translator;

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

Quiz: Which implementation gets injected?

```
public class Greeter {
   private Welcome welcome;

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

   ...
}
```

It's ambiguous!

Working out an ambiguous resolution

- Qualifier
- Alternative
- Producer
- Veto (or hide)

qualifier

n. *an annotation used to resolve an API implementation variant at an injection point*

Defining a qualifier

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

@interface means annotation

Qualifying an implementation

```
@Translating
public class TranslatingWelcome extends Welcome {

   @Inject GoogleTranslator translator;

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

- makes type more specific
- assigns semantic meaning

Qualifier as a “binding type”

```
@Inject @Translating Welcome welcome;
```

```
@Translating
public class TranslatingWelcome extends Welcome {

   @Inject GoogleTranslator translator;

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

Explicitly request qualified interface

```
public class Greeter {

   private Welcome welcome;

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

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

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

In other words, **an override**

Defining an alternative

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

   @Inject GoogleTranslator translator;

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

Substituting the alternative

- Activated using beans.xml

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

Assigning a bean (EL) name

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

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

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

Assigning a bean (EL) name by convention

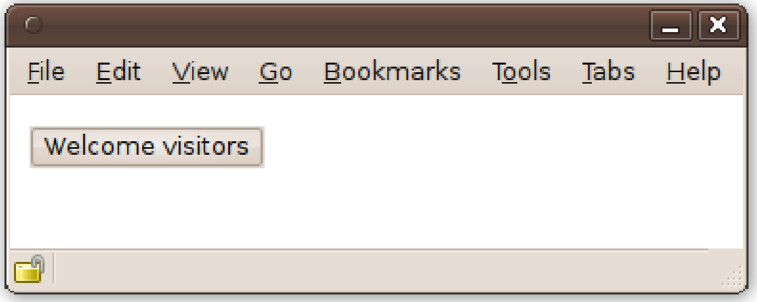
```
@Named
public class Greeter {
   private Welcome welcome;

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

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

Bean name is decapitalized simple class name

Welcome to CDI, JSF!

- Use the bean directly in the JSF view

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



~~JSF managed beans~~

↓

CDI

Stashing the bean in a context

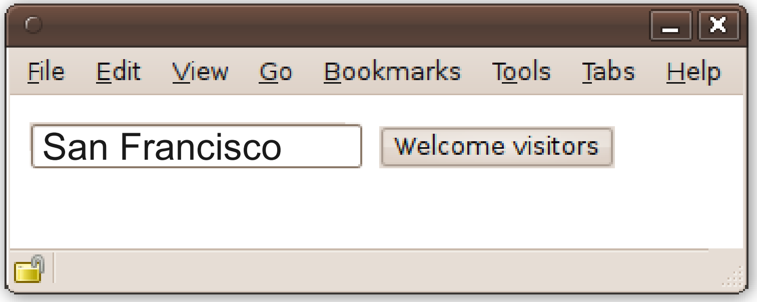
- Bean saved for the duration of a request

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

   public String getCity() { return city; }

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

   public void welcomeVisitors() {
      System.out.println(w.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>
```



Prints:

Welcome to San Francisco!

Mission accomplished: We have a deal!

Web tier
(JSF)

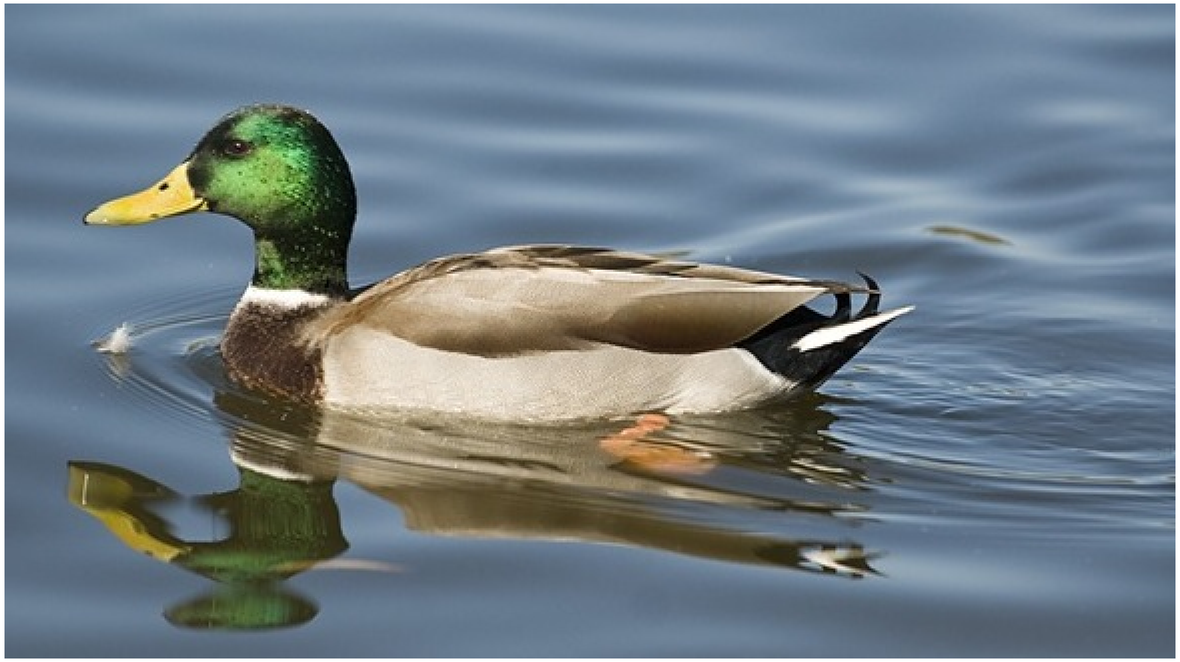
Business tier
(managed bean)

Scope types and contexts

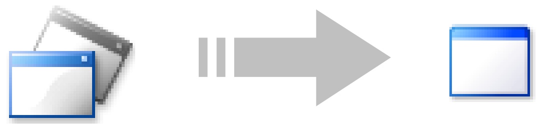
- Default scope - @Dependent
  - Bound to lifecycle of bean holding reference
- Servlet scopes
  - @ApplicationScoped
  - @RequestScoped
  - @SessionScoped
- JSF conversation scope - @ConversationScoped
- Custom scopes
  - Define scope type annotation (e.g., @FlashScoped)
  - Implement the context API in an extension

Scope transparency

- Scopes not visible to client (no coupling)
- Scoped beans are *proxied* for thread safety



Conversation context

- Request ≤ Conversation ≪ Session
- 
- Boundaries demarcated by application
- Optimistic transaction
  - Conversation-scoped persistence context
  - No fear of exceptions on lazy fetch operations

Controlling the conversation

```
@ConversationScoped
public class BookingAgent {

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

   private Hotel selected;
   private Booking booking;

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

   ...
```

Controlling the conversation

```
   ...

   public boolean confirm() {
      if (!isValid()) {
         return false;
      }

      em.persist(booking);
      conversation.end();
      return true;
   }
}
```

producer method

n. *a method whose return value produces an injectable object*

Producer method examples

```
@Produces @RequestScoped
public FacesContext getFacesContext() {
   return FacesContext.getInstance();
}

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

@Produces @SessionScoped @WishList
public List<Product> getWishList() {
   return em.createQuery("...").getResultList();
}
```

From non-bean

Runtime selection

Dynamic result set

Injecting producer return values

```
@Inject FacesContext ctx;

@Inject PaymentProcessor pp;

@Inject @WishList List<Product> wishlist;
```

Origin of product is hidden at injection point

Bridging Java EE resources

- Use producer field to expose Java EE resource

```
@Stateless
public class UserEntityManagerProducer {
   @Produces @UserRepository
   @PersistenceContext(unitName = "users")
   EntityManager em;
}

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

Injecting resources in type-safe way

- String-based resource names are hidden

```
public class UserManager {
   @Inject @UserRepository EntityManager userEm;
   ...
}

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

Rethinking interceptors

```
@Interceptors(
   SecurityInterceptor.class,
   TransactionInterceptor.class,
   LoggingInterceptor.class
)
@Stateful public class BusinessComponent {
   ...
}
```

Um, what's the point?

Define an interceptor binding type

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

Mark the interceptor implementation

```
@Secure
@Interceptor
public class SecurityInterceptor {

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

}
```

Interceptor wiring with proper semantics

```
@Secure
public class AccountManager {

   public boolean transfer(Account a, Account b) {
      ...
   }

}
```

Enabling and ordering interceptors

- Bean archive has *no* enabled interceptors by default
- Interceptors activated in beans.xml of bean archive
  - Referenced by binding type
  - Ordering is per-module
  - Declared in module in which the interceptor is used

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

Interceptors applied in order listed

Annotation jam!

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

   public boolean transfer(Account a, Account b) {
      ...
   }

}
```

stereotype

n. *an annotation used to group common architectural patterns (recurring roles)*

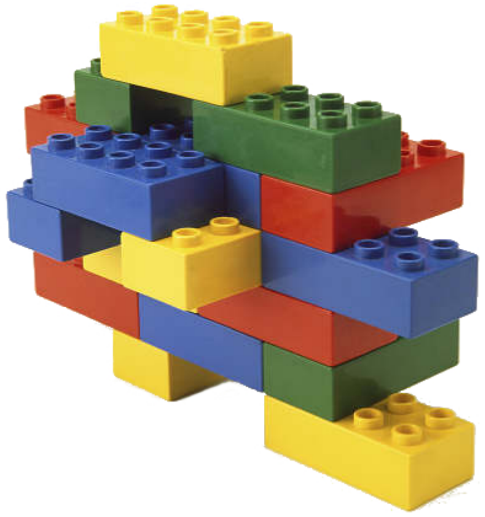
Define a stereotype to bundle annotations

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

Using a stereotype

```
@BusinessComponent
public class AccountManager {

   public boolean transfer(Account a, Account b) {
      ...
   }

}
```

Portable extensions

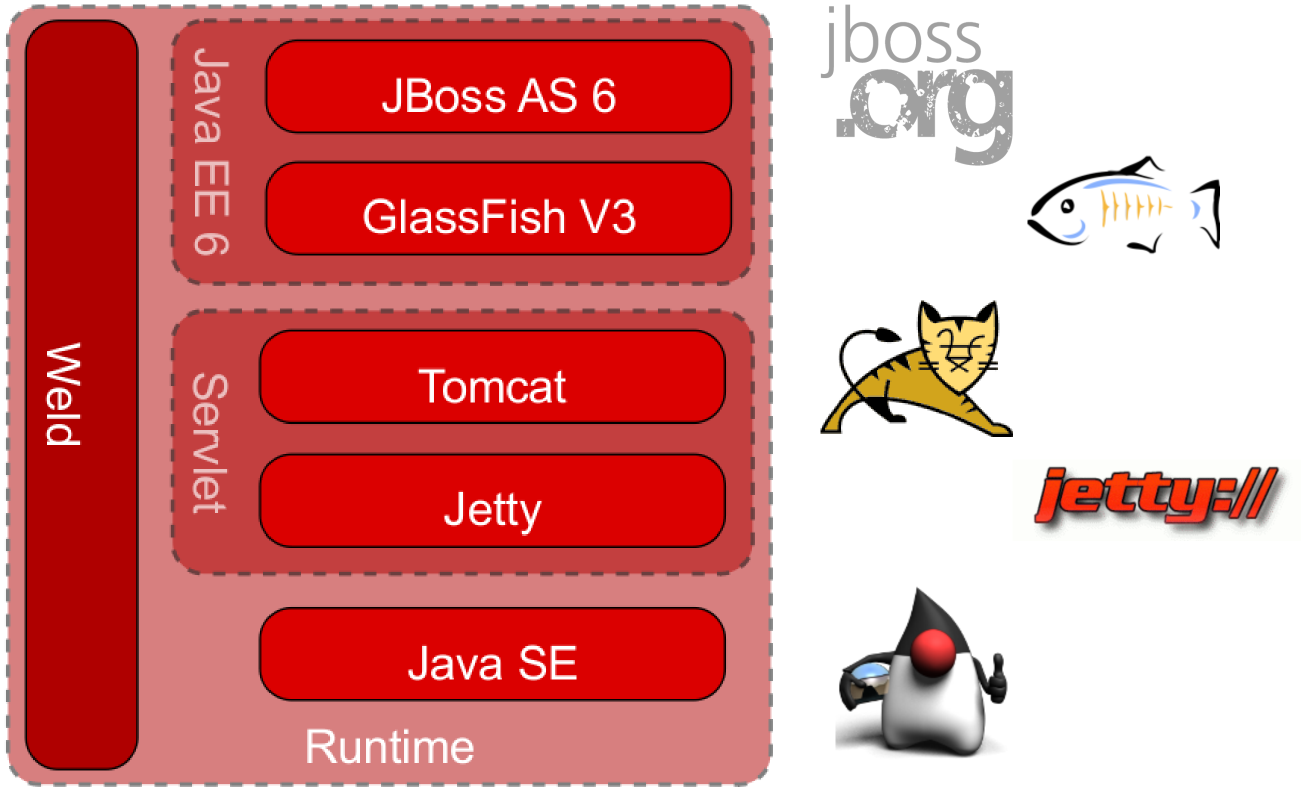


- SPI – Service Provider Interface
- Automatically discovered
- Application-scoped instance
- Observes events from CDI event bus
  - Before/after bean discovery
  - After deployment validation
  - etc...
- Can override, augment, replace or veto beans

Weld

Weld: JSR-299 Reference Implementation

- Implementation & TCK
- Weld (portable) extensions
- Apache software licensed (version 2.0)

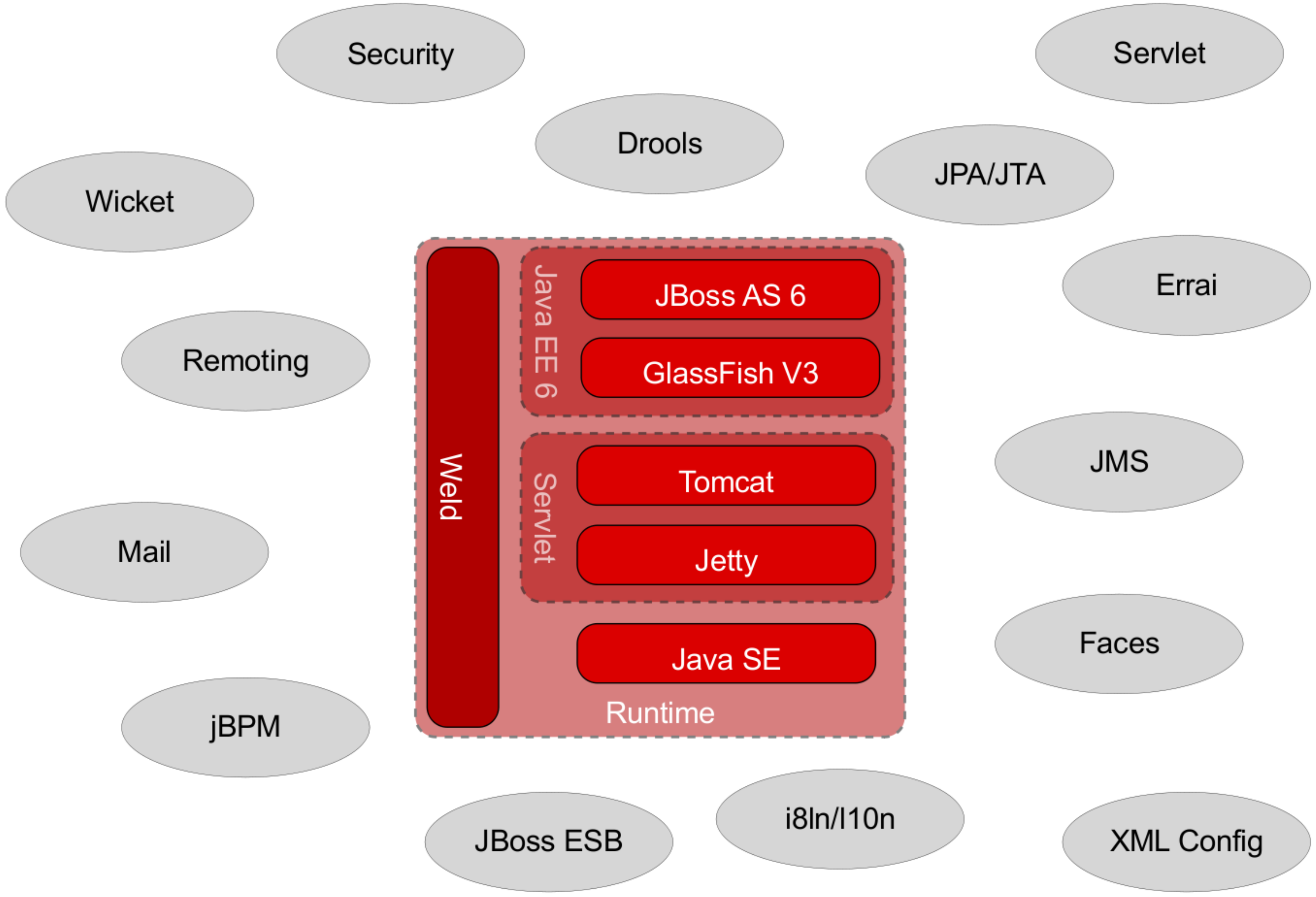


Seam

Seam’s mission statement

To provide a *fully integrated* development platform for building rich Internet applications based upon the Java EE environment.

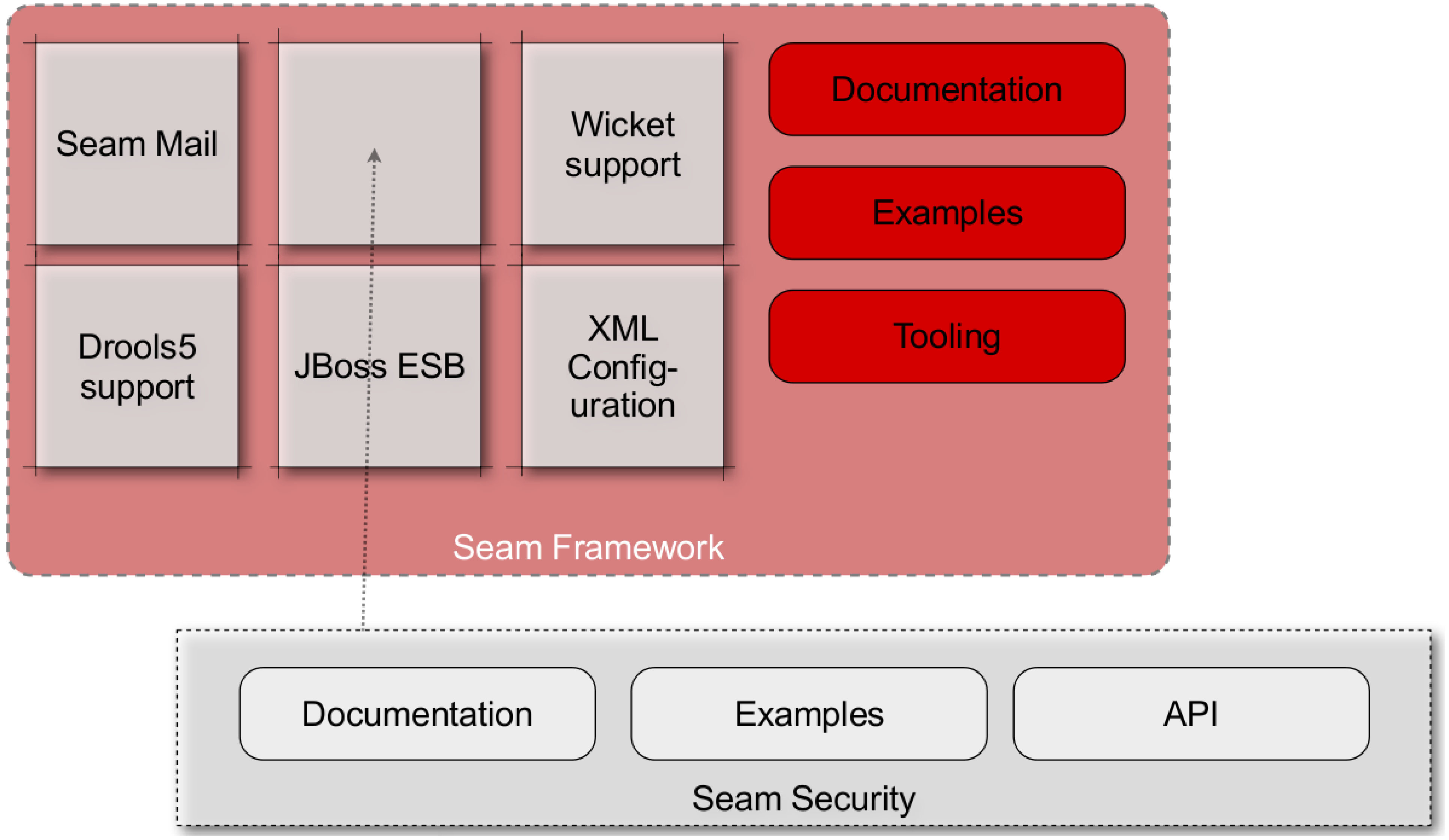
Seam’s new modular ecosystem



Portable modules

- Module per domain or integration
- Independently...
  - lead
  - versioned
  - released
- Per-module structure
  - Based on CDI
  - API & implementation
  - Reference documentation & examples

Stack releases



What's on the menu so far?

- Drools
- jBPM
- JMS
- Faces
- International
- Persistence
- JavaScript remoting
- Security
- Servlet
- Wicket
- XML configuration
- Exception handling

...and more

➥ http://github.com/seam

XML-based configuration

```
<beans ...
   xmlns:app="java:urn:com.acme">
   <app:TranslatingWelcome>
      <app:Translating/>
      <app:defaultLocale>en-US</app:defaultLocale>
   </app:TranslatingWelcome>
</beans>
```

- Define, specialize or override beans
- Add annotations (qualifiers, interceptor bindings, ...)
- Assign initial property values

Cross-field validator in Seam Faces

```
@FacesValidator("addressValidator")
public class AddressValidator implements Validator {

   @Inject Directory directory;
   @Inject @InputField String city;
   @Inject @InputField String state;
   @Inject @InputField ZipCode zip;

   public void validate(FacesContext ctx, UIComponent c,
         Object v) throws ValidatorException {
      if (!directory.exists(city, state, zip) {
         throw new ValidatorException("Bad address");
      }
   }
}
```

Wiring the validator to the inputs

```
<h:form id="address">
   City: <h:inputText id="city" value="#{bean.city}"/>
   State: <h:inputText id="state" value="#{bean.state}"/>
   Zip: <h:inputText id="zipCode" value="#{bean.zip}"/>
   <h:commandButton value="Update"
      action="#{addressController.update}"/>
   <s:validateForm validatorId="addressValidator"
      fields="zip=zipCode">
</h:form>
```

Arquillian: Container-oriented testing for Java EE

```
@RunWith(Arquillian.class)
public class GreeterTestCase {

  @Deployment
  public static Archive<?> createDeployment() {
    return ShrinkWrap.create(JavaArchive.class)
      .addClasses(Greeter.class, GreeterBean.class);
  }

  @EJB private Greeter greeter;

  @Test
  public void shouldBeAbleToInvokeEJB() throws Exception {
    assertEquals("Hello, Earthlings", greeter.greet("Earthlings"));
  }
}
```

Throwing complexity over the wall
Wed @ 4:45
Hilton, Golden Gate 4/5

Summary

- Java EE 6 is leaner and more productive
- JSR-299 (CDI) provides a set of services for Java EE
  - Bridges JSF and EJB
  - Offers loose coupling with **strong typing**
  - Provides a type-based event bus
  - Catalyzed managed bean & interceptor specifications
  - Extensive SPI for third-party integration with Java EE
- Weld: JSR-299 reference implementation & add-ons
- Seam 3: Portable extensions for Java EE

Fork me on GitHub

How do I get started?

- Download a Java EE 6 container
  - JBoss AS 6 – http://jboss.org/jbossas
  - GlassFish V3 – http://glassfish.org
- Generate a Java EE project using a Maven archetype
  - http://tinyurl.com/goweld
- Read the Weld reference guide
  - http://tinyurl.com/weld-reference-101
- Browse the CDI JavaDoc
  - http://docs.jboss.org/cdi/api/latest/
- Check out the Seam 3 project
  - http://seamframework.org/Seam3

Q & A

Dan Allen
Principal Software Engineer
JBoss by Red Hat

http://seamframework.org/Weld
http://seamframework.org/Seam3