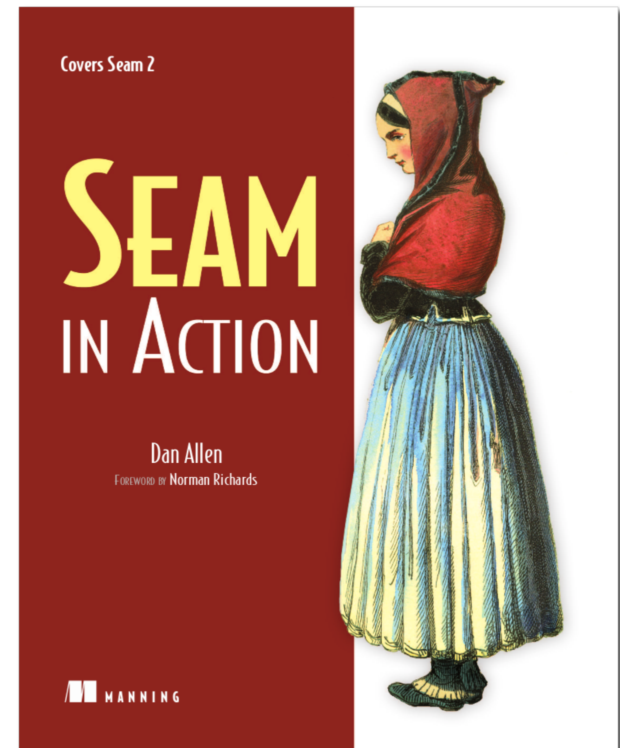
CDI (JSR-299), Weld and the future of Seam

Dan Allen
Senior Software Engineer
JBoss, by Red Hat

Who am I?

- Author of Seam in Action, Manning 2008
- Seam and Weld project member
- JSR-314 (JSF 2.0) EG member
- Champion for openness



Project terminology

- CDI (JSR-299)
  - Contexts & dependency injection for the Java EE platform
- Weld
  - JSR-299 Reference Implementation
  - Bootstrap outside of Java EE (Servlet, Java SE)
- Seam
  - Portable extensions for the Java EE platform
  - Integrations with non-Java EE technologies
  - Akin to ecosystem of JSF component libraries

What JSR-299 (CDI) provides

- A powerful new set of *services* for Java EE components
  - Lifecycle management for stateful components bound to well-defined contexts (+ new conversation context)
  - A type-safe approach to dependency injection
  - Interaction via an event notification facility
  - Reduced coupling between interceptors and beans
  - Decorators—interceptors better suited for solving business concerns
  - Unified EL integration (named beans)
  - An SPI for developing portable extensions for the Java EE platform

JSR-299: The big picture

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

Stated goal of JSR-299

Web tier
(JSF)

Transactional tier
(EJB)

Going beyond Seam

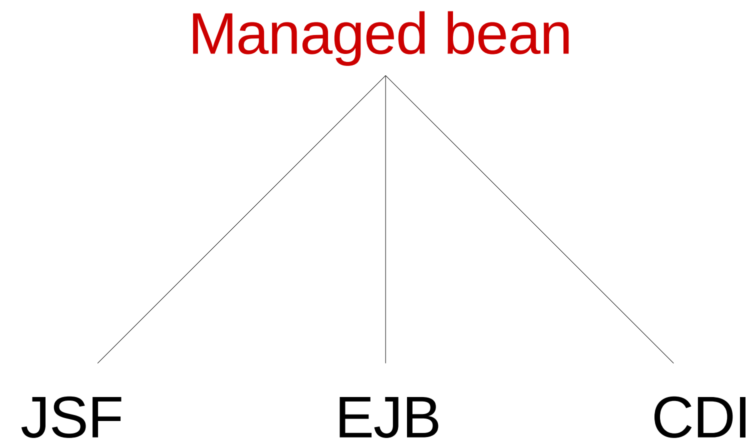
- Solve JSF-EJB integration problem at platform level
- Get an expert group (EG) involved
  - Buy-in from broader Java EE community
  - Formulate a more robust design
  - Establish foundation for an ecosystem of extensions

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"

Managed bean

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

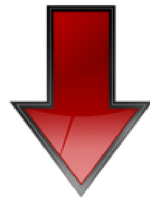


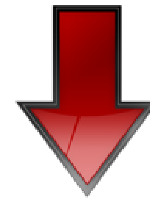
Read about how managed beans evolved: http://www.infoq.com/news/2009/11/weld10

CDI *replaces* JSF managed beans

JSF managed beans

JSP





CDI

Facelets

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 hard dependency between client and impl
  - Don’t rely on weak association of field ⇨ bean name
  - Validation can be done at startup

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 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
- SPI allows third-party frameworks to introduce additional injectable objects
- Annotations aligned with JSR-330

CDI bean

- Set of bean types (non-empty)
- Set of qualifiers (non-empty)
- Scope
- Bean EL name (optional)
- Alternatives
- Set of interceptor bindings
- Bean 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
- Lifecycle 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 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("Orlando"));
   }
}
```

@Default qualifier 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("Orlando"));
   }
}
```

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("Orlando"));
   }
}
```

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 used to resolve a implementation variant of an API at an injection 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("Orlando"));
   }
}
```

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("Orlando"));
   }
}
```

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; // getter and setter hidden

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

   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 lifecycle of bean holding reference to it
- Servlet scopes
  - @ApplicationScoped
  - @RequestScoped
  - @SessionScoped
- JSF conversation scope - @ConversationScoped
- Custom scopes
  - Define scope type annotation (i.e., @FlashScoped)
  - 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
@Named @RequestScoped
class Greeter {
   Welcome welcome;
   Profile profile;
   private String city;

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

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

No awareness of scope

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

```
public
@ConversationScoped
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

A method whose return value is an injectable object

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() { ... }
```

Bridging Java EE resources

- Use producer field to expose Java EE resource

```
public
@Stateless
class UserEntityManagerFactory {
   @Produces @UserRepo
   @PersistenceUnit(unitName = "userPU")
   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 @UserRepo EntityManagerFactory emf;
   ...
}

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

Promoting state

- Producer methods can promote state as 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;
   Identity identity;
   private String city;

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

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

No awareness of scope

Rethinking interceptors

- Interceptors handle orthogonal concerns
- Java EE 5 interceptors bound directly to component
  - `@Interceptors` annotation on bean type
- What's the problem?
  - Shouldn't 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 transfer(Account a, Account b) {
      ...
   }

}
```

Interceptor wiring in JSR-299 (4)

- Applying interceptor to method with proper semantics

```
public class AccountManager {

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

}
```

Multiple interceptors

- Application developer only worries about semantics

```
public
@Transactional
class AccountManager {

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

}
```

Decorators

- Intercept invocations for a particular Java interface
- Aware of semantics
- Complement interceptors
- Enabled in same way as interceptors

Decorator example

```
public
@Decorator
abstract class LargeTxDecorator implements Account {
   @Inject @Delegate @Any Account account;
   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      account.withdraw(amount);
      if (amount.compareTo(LARGE_AMOUNT) > 0) {
         em.persist(new LoggedWithdrawl(amount));
      }
   }

}
```

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

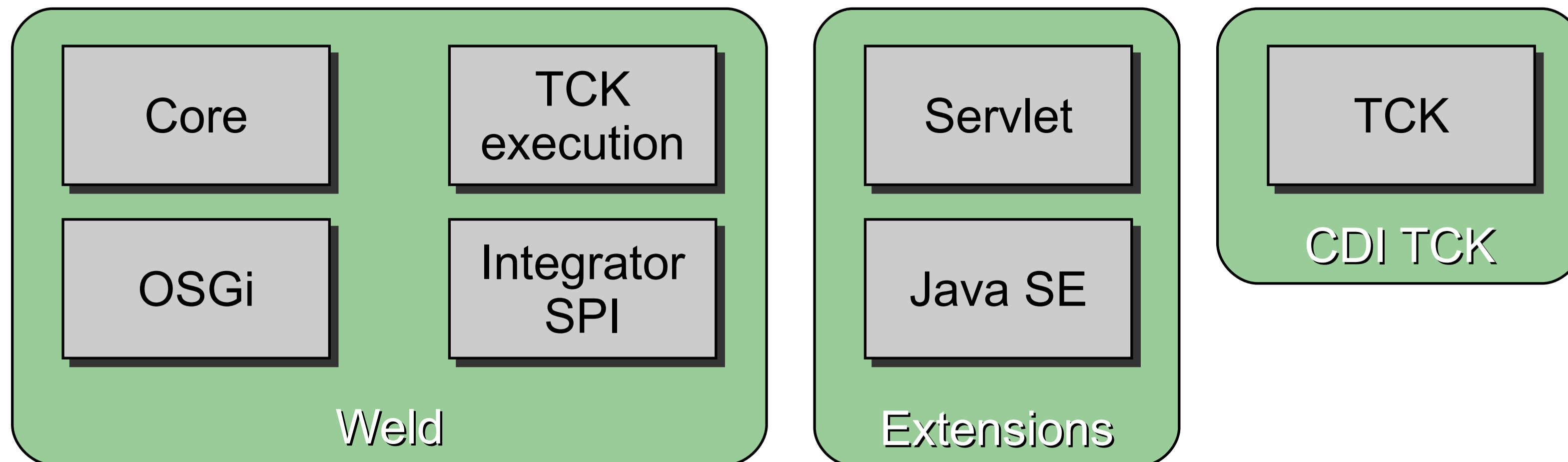
```
public class GateServices {
   public void onIncomingFlight(
         @Observes @Landing Flight flight,
         Greeter greeter,
         CateringService cateringService) {
      Gate gate = ...;
      flight.setGate(gate);
      cateringService.dispatch(gate);
      greeter.welcomeVisitors();
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

Takes event API type with additional binding type

Additional parameters are injected by the container

Weld

- JSR-299 reference implementation
- Developed under the Seam project umbrella
- Version 1.0.0 available, including Maven archetypes!
- Bundled in JBoss AS 6 and GlassFish V3
- Runs on Tomcat, Jetty and Java SE



Seam’s mission statement

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

Seam is our future

“The future for all of our projects and platform is Seam.”

“[Developers] won't have to worry about learning a new component model when they move between platforms.”

- Mark Little, JBoss CTO

http://blogs.jboss.org/blog/mlittle/2009/11/11/The_future_of_component_models.txt

Seam framework stack

- CDI foundation
- Enhanced, declarative security
- Support for multiple view layers (JSF 2, Wicket, Flex)
- JavaScript remoting (a la DWR)
- RESTeasy integration
- Bridges to Seam 2, Spring and Guice
- Email, graphics, PDF and XLS
- Pageflows and business processes
- JBoss Tools

http://in.relation.to/Bloggers/HowToStartLearningJavaEE6

Ecosystem architecture

Seam Security

Seam Mail

Wicket support

Drools 5 support

RESTEasy integration

...

Seam Framework

CDI 1.0

Integration

JBoss AS > 5.2

Core

CDI TCK

Integration

GlassFish V3

OSGi

Servlet

Tomcat/Jetty

Weld

Open Web Beans

Seam 3: Key themes

- Modularity
  - Seam à la carte
- Portability
  - Run on any CDI implementation
- Full stack
  - Similar to Eclipse’s coordinated release

Drawing the line

Unportable extension (UE)

- Integrates with proprietary SPIs in Weld

Portable extension (PE) - Weld

- Simple or general purpose
- Doesn’t pull in extra dependencies

Portable extension (PE) - Seam

- Everything else

End-to-end testing

- SeamTest modularized
- ShrinkWrap
  - Declarative creation of archives, made simple

```
JavaArchive archive =
   Archives.create("archive.jar", JavaArchive.class)
      .addClasses(MyClass.class,MyOtherClass.class)
      .addResource("mystuff.properties");
```

- Arquillian
  - Pluggable unit test
  - Standalone and in-container POJO tests
  - @RunWith(Arquillian.class)

Summary

- JSR-299 provides a set of services for Java EE
  - Satisfies original goal to bridge JSF and EJB
  - Offers loose coupling with **strong typing**
  - Catalyzed the managed bean specification
- Other problems needed to be solved
  - Robust dependency injection and context model
  - Event notification facility, furthering the loose coupling
  - Extensive SPI for third-parties to integrate with Java EE
- Weld: JSR-299 Reference Implementation
- Seam: Portable extensions for Java EE

Q & A

Dan Allen
Senior Software Engineer
JBoss, by Red Hat

http://in.relation.to
http://seamframework.org