Meteor: The Manual

Creating adaptable software

Ted Stockwell
Meteor Lead Developer
OpenBookz

Part I. Introduction to Meteor

Why Meteor?

Small businesses are notoriously underserved by the software industry because developing high-quality, full-featured software is very expensive due to the large amount of labor required and the high cost of the engineers required to create, maintain, and customize software. Companies that produce software systems usually target large companies that can afford the high price of software systems. Software systems for small businesses are often low-quality, slimmed down and incomplete because that's really all that small businesses can afford to pay for. Meteor was conceived with the goal of making it possible to profitably create software for small businesses by assembling high-quality, full-featured, customized software systems from modules available from many vendors.

Meteor is designed to benefit the developers of software modules by decreasing the cost to create a module and by increasing the revenue that can be generated from a module by making it possible to reuse modules in more systems.

Meteor is designed to benefit the small businesses that buy software systems by reducing the cost of buying and maintaining a system and by increasing the quality and available features of systems.

1.1. Design goals

In order to meet the above business goals, the Meteor framework is designed with the following design goals:

- All functionality is packaged into modules that are assembled into software systems.
- A global metadata repository makes it possible for a module to implement entire software aspects like logging, persistence and UI.
- All functionality may be customized without requiring vendors to build customization hooks into their modules.
- Customizations and extensions are packaged into modules and may be dynamically added or removed from a system without technical assistance.
- All modules in a system may be dynamically updated without technical assistance.
- Meteor itself implements many aspects out of the box, like user authentication and authorization, data storage, and user interface.

The details of how to create modules for Meteor-based systems is left to the rest of this manual. However, it is important to understand that Meteor was designed to meet the above business goals.

1.2. Comparison to other frameworks

Many frameworks are created with only the goal of making software development easier and less expensive, but they don't address software reuse, maintenance, or customization issues thus making software prohibitively expensive to maintain and customize.

Some existing frameworks even try to make software reuse and customization easier but don't support dynamic modularization, thus making software prohibitively expensive by requiring engineers to apply updates, customizations, and extensions.

And some frameworks support modules but don't support packaging customizations in modules or don't allow modules to be dynamically added or removed, again making any software prohibitively expensive by requiring engineers to apply updates, customizations, and extensions.

Meteor is designed to bring it all together: high development productivity, modular software reuse, and dynamic system updates, customizations, and extensions that don't require technical assistance.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

2.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

2.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, "Deployment types"*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

2.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

2.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

2.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

2.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

2.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

2.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

2.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

2.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

2.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

2.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

3.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

3.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

3.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

3.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

3.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

3.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
   ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

3.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

3.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

3.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

3.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

3.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

3.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

4.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

4.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

4.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

4.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

4.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

4.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

4.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

4.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

4.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

4.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

4.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

4.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

5.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

5.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

5.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

5.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

5.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

5.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

5.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

5.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

5.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

5.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

5.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in ???.

5.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in ???.

Getting started with Web Beans, the Reference Implementation of JSR-299

The Web Beans is being developed at *the Seam project* [http://seamframework.org/WebBeans]. You can download the latest developer release of Web Beans from the *the downloads page* [http://seamframework.org/Download].

Web Beans comes with a two deployable example applications: `webbeans-numberguess`, a war example, containing only simple beans, and `webbeans-translator` an ear example, containing enterprise beans. There are also two variations on the numberguess example, the tomcat example (suitable for deployment to Tomcat) and the jsf2 example, which you can use if you are running JSF2. To run the examples you'll need the following:

- the latest release of Web Beans,
- JBoss AS 5.0.1.GA, or
- Apache Tomcat 6.0.x, and
- Ant 1.7.0.

6.1. Using JBoss AS 5

You'll need to download JBoss AS 5.0.1.GA from *jboss.org* [http://www.jboss.org/jbossas/downloads/], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/jboss-5.0.1.GA.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Next, we need to tell Web Beans where JBoss is located. Edit `jboss-as/build.properties` and set the `jboss.home` property. For example:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

To install Web Beans, you'll need Ant 1.7.0 installed, and the `ANT_HOME` environment variable set. For example:

Note

JBoss 5.1.0 comes with Web Beans built in, so there is no need to update the server.

```
$ unzip apache-ant-1.7.0.zip
$ export ANT_HOME=~/apache-ant-1.7.0
```

Then, you can install the update. The update script will use Maven to download Web Beans automatically.

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Now, you're ready to deploy your first example!

Tip

The build scripts for the examples offer a number of targets for JBoss AS, these are:

- `ant restart` - deploy the example in exploded format
- `ant explode` - update an exploded example, without restarting the deployment
- `ant deploy` - deploy the example in compressed jar format
- `ant undeploy` - remove the example from the server
- `ant clean` - clean the example

To deploy the numberguess example:

```
$ cd examples/numberguess
```

```
ant deploy
```

Start JBoss AS:

```
$ /Application/jboss-5.0.0.GA/bin/run.sh
```

Tip

If you use Windows, use the `run.bat`script.

Wait for the application to deploy, and enjoy hours of fun at *http://localhost:8080/webbeans-numberguess*!

Web Beans includes a second simple example that will translate your text into Latin. The numberguess example is a war example, and uses only simple beans; the translator example is an ear example, and includes enterprise beans, packaged in an EJB module. To try it out:

```
$ cd examples/translator
ant deploy
```

Wait for the application to deploy, and visit *http://localhost:8080/webbeans-translator*!

6.2. Using Apache Tomcat 6.0

You'll need to download Tomcat 6.0.18 or later from *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/apache-tomcat-6.0.18.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Next, we need to tell Web Beans where Tomcat is located. Edit `jboss-as/build.properties` and set the `tomcat.home` property. For example:

```
tomcat.home=/Applications/apache-tomcat-6.0.18
```

Tip

The build scripts for the examples offer a number of targets for Tomcat, these are:

- `ant tomcat.restart` - deploy the example in exploded format
- `ant tomcat.explode` - update an exploded example, without restarting the deployment
- `ant tomcat.deploy` - deploy the example in compressed jar format
- `ant tomcat.undeploy` - remove the example from the server
- `ant tomcat.clean` - clean the example

To deploy the numberguess example for tomcat:

```
$ cd examples/tomcat
ant tomcat.deploy
```

Start Tomcat:

```
$ /Applications/apache-tomcat-6.0.18/bin/startup.sh
```

Tip

If you use Windows, use the `startup.bat`script.

Wait for the application to deploy, and enjoy hours of fun at *http://localhost:8080/webbeans-numberguess*!

6.3. Using GlassFish

TODO

6.4. The numberguess example

In the numberguess application you get given 10 attempts to guess a number between 1 and 100. After each attempt, you will be told whether you are too high, or too low.

The numberguess example is comprised of a number of Web Beans, configuration files, and Facelet JSF pages, packaged as a war. Let's start with the configuration files.

All the configuration files for this example are located in `WEB-INF/`, which is stored in `WebContent` in the source tree. First, we have `faces-config.xml`, in which we tell JSF to use Facelets:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="1.2"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-facesconfig_1_2.xsd">

    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
    </application>

</faces-config>
```

There is an empty `web-beans.xml` file, which marks this application as a Web Beans application.

Finally there is `web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">

   <display-name>Web Beans Numbergues example</display-name>

   <!-- JSF -->


   <servlet>                                         1
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>


  <servlet-mapping>                                    (2)
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>


  <context-param>                                      (3)
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>


  <session-config>                                     (4)
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

1. Enable and load the JSF servlet
2. Configure requests to `.jsf` pages to be handled by JSF
3. Tell JSF that we will be giving our source files (facelets) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Note

Whilst this demo is a JSF demo, you can use Web Beans with any Servlet based web framework.

Let's take a look at the Facelet view:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:s="http://jboss.com/products/seam/taglib">
```

```
 <ui:composition template="template.xhtml">
  <ui:define name="content">
     <h1>Guess a number...</h1>
     <h:form id="NumberGuessMain">

        <div style="color: red">
           <h:messages id="messages" globalOnly="false"/>
              <h:outputText id="Higher" value="Higher!" rendered="#{game.number gt game.guess
and game.guess ne 0}"/>
           <h:outputText id="Lower" value="Lower!" rendered="#{game.number lt game.guess and
game.guess ne 0}"/>
        </div>

        <div>
           I'm thinking of a number between #{game.smallest} and #{game.biggest}.
           You have #{game.remainingGuesses} guesses.
        </div>

        <div>
           Your guess:
           <h:inputText id="inputGuess"
                        value="#{game.guess}"
                        required="true"
                        size="3"
                        disabled="#{game.number eq game.guess}">
              <f:validateLongRange maximum="#{game.biggest}"
                                   minimum="#{game.smallest}"/>
           </h:inputText>
          <h:commandButton id="GuessButton"
                           value="Guess"
                           action="#{game.check}"
                           disabled="#{game.number eq game.guess}"/>
        </div>
        <div>
                     <h:commandButton  id="RestartButton"  value="Reset"  action="#{game.reset}"
immediate="true" />
        </div>
     </h:form>
  </ui:define>
 </ui:composition>
</html>
```

1. Facelets is a templating language for JSF, here we are wrapping our page in a template which defines the header.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentance changes to make sure they know what range to guess in.
4. This input field is bound to a Web Bean, using the value expression.
5. A range validator is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of range number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the Web Bean.

The example exists of 4 classes, the first two of which are binding types. First, there is the `@Random` binding type, used for injecting a random number:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface Random {}
```

There is also the `@MaxNumber` binding type, used for injecting the maximum number that can be injected:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface MaxNumber {}
```

The `Generator` class is responsible for creating the random number, via a producer method. It also exposes the maximum possible number via a producer method:

```
@ApplicationScoped
public class Generator {

   private java.util.Random random = new java.util.Random( System.currentTimeMillis() );

   private int maxNumber = 100;

   java.util.Random getRandom()
   {
      return random;
   }

   @Produces @Random int next() {
      return getRandom().nextInt(maxNumber);
   }

   @Produces @MaxNumber int getMaxNumber()
   {
      return maxNumber;
   }

}
```

You'll notice that the `Generator` is application scoped; therefore we don't get a different random each time.

The final Web Bean in the application is the session scoped `Game`.

You'll note that we've used the `@Named` annotation, so that we can use the bean through EL in the JSF page. Finally, we've used constructor injection to initialize the game with a random number. And of course, we need to tell the player when they've won, so we give feedback with a `FacesMessage`.

```
package org.jboss.webbeans.examples.numberguess;


import javax.annotation.PostConstruct;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.webbeans.AnnotationLiteral;
import javax.webbeans.Current;
import javax.webbeans.Initializer;
import javax.webbeans.Named;
import javax.webbeans.SessionScoped;
import javax.webbeans.manager.Manager;

@Named
@SessionScoped
public class Game
{
   private int number;
   private int guess;
   private int smallest;
   private int biggest;
   private int remainingGuesses;

   @Current Manager manager;

   public Game()
   {
   }

   @Initializer
   Game(@MaxNumber int maxNumber)
   {
      this.biggest = maxNumber;
   }

   public int getNumber()
   {
      return number;
   }

   public int getGuess()
   {
      return guess;
   }

   public void setGuess(int guess)
   {
      this.guess = guess;
   }

   public int getSmallest()
   {
      return smallest;
   }

   public int getBiggest()
   {
      return biggest;
   }

   public int getRemainingGuesses()
```

```
   {
      return remainingGuesses;
   }

   public String check()
   {
      if (guess>number)
      {
         biggest = guess - 1;
      }
      if (guess<number)
      {
         smallest = guess + 1;
      }
      if (guess == number)
      {
         FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Correct!"));
      }
      remainingGuesses--;
      return null;
   }

   @PostConstruct
   public void reset()
   {
      this.smallest = 0;
      this.guess = 0;
      this.remainingGuesses = 10;
                    this.number    =    manager.getInstanceByType(Integer.class,    new
 AnnotationLiteral<Random>(){});
   }

}
```

6.4.1. The numberguess example in Tomcat

The numberguess for Tomcat differs in a couple of ways. Firstly, Web Beans should be deployed as a Web Application library in `WEB-INF/lib`. For your convenience we provide a single jar suitable for running Web Beans in any servlet container `webbeans-servlet.jar`.

Tip

Of course, you must also include JSF and EL, as well common annotations (`jsr250-api.jar`) which a JEE server includes by default.

Secondly, we need to explicitly specify the servlet listener (used to boot Web Beans, and control it's interaction with requests) in `web.xml`:

```
<listener>
   <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

6.4.2. The numberguess example for Apache Wicket

Whilst JSR-299 specifies integration with Java ServerFaces, Web Beans allows you to inject into Wicket components, and also allows you to use a conversation context with Wicket. In this section, we'll walk you through the Wicket version of the numberguess example.

Note

You may want to review the Wicket documentation at *http://wicket.apache.org/*.

Like the previous example, the Wicket WebBeans examples make use of the `webbeans-servlet` module. The use of the *Jetty servlet container* [http://jetty.mortbay.org/] is common in the Wicket community, and is chosen here as the runtime container in order to facilitate comparison between the standard Wicket examples and these examples, and also to show how the webbeans-servlet integration is not dependent upon Tomcat as the servlet container.

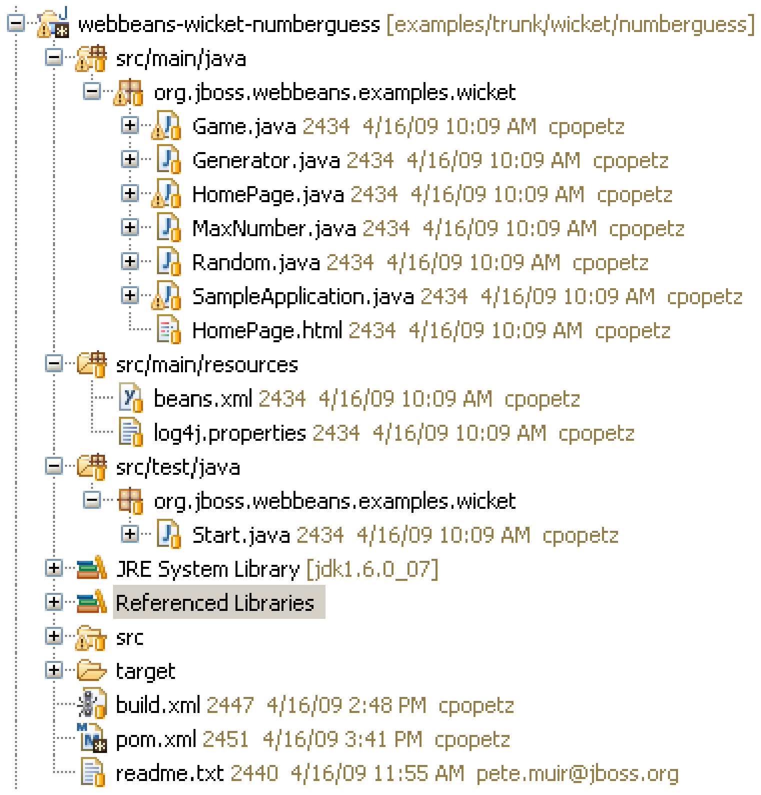
These examples make use of the Eclipse IDE; instructions are also given to deploy the application from the command line.

6.4.2.1. Creating the Eclipse project

To generate an Eclipse project from the example:

```
cd examples/wicket/numberguess
mvn -Pjetty eclipse:eclipse
```

Then, from eclipse, choose *File -> Import -> General -> Existing Projects into Workspace*, select the root directory of the numberguess example, and click finish. Note that if you do not intend to run the example with jetty from within eclipse, omit the "-Pjetty." This will create a project in your workspace called `webbeans-wicket-numberguess`



```
webbeans-wicket-numberguess [examples/trunk/wicket/numberguess]
  src/main/java
    org.jboss.webbeans.examples.wicket
      Game.java 2434  4/16/09 10:09 AM  cpopetz
      Generator.java 2434  4/16/09 10:09 AM  cpopetz
      HomePage.java 2434  4/16/09 10:09 AM  cpopetz
      MaxNumber.java 2434  4/16/09 10:09 AM  cpopetz
      Random.java 2434  4/16/09 10:09 AM  cpopetz
      SampleApplication.java 2434  4/16/09 10:09 AM  cpopetz
      HomePage.html 2434  4/16/09 10:09 AM  cpopetz
  src/main/resources
    beans.xml 2434  4/16/09 10:09 AM  cpopetz
    log4j.properties 2434  4/16/09 10:09 AM  cpopetz
  src/test/java
    org.jboss.webbeans.examples.wicket
      Start.java 2434  4/16/09 10:09 AM  cpopetz
  JRE System Library [jdk1.6.0_07]
  Referenced Libraries
  src
  target
  build.xml 2447  4/16/09 2:48 PM  cpopetz
  pom.xml 2451  4/16/09 3:41 PM  cpopetz
  readme.txt 2440  4/16/09 11:55 AM  pete.muir@jboss.org
```

6.4.2.2. Running the example from Eclipse

This project follows the `wicket-quickstart` approach of creating an instance of Jetty in the `Start` class. So running the example is as simple as right-clicking on that Start class in `src/test/java` in the *Package Explorer* and choosing *Run as Java Application*. You should see console output related to Jetty starting up; then visit able `http://localhost:8080` to view the app. To debug choose *Debug as Java Application*.

6.4.2.3. Running the example from the command line in JBoss AS or Tomcat

This example can also be deployed from the command line in a (similar to the other examples). Assuming you have set up the `build.properties` file in the `examples` directory to specify the location of JBoss AS or Tomcat, as previously described, you can run `ant deploy` from the `examples/wicket/numberguess` directory, and access the application at `http://localhost:8080/webbeans-numberguess-wicket`.

6.4.2.4. Understanding the code

JSF uses Unified EL expressions to bind view layer components in JSP or Facelet views to beans, Wicket defines it's components in Java. The markup is plain html with a one-to-one mapping between html elements and the view components. All view logic, including binding of components to models and controlling the response of view actions, is handled in Java. The integration of

Web Beans with Wicket takes advantage of the same binding annotations used in your business layer to provide injection into your WebPage subclass (or into other custom wicket component subclasses).

The code in the wicket numberguess example is very similar to the JSF-based numberguess example. The business layer is identical!

Differences are:

- Each wicket application must have a `WebApplication` subclass, In our case, our application class is `SampleApplication`:

```
public class SampleApplication extends WebBeansApplication {
   @Override
   public Class getHomePage() {
      return HomePage.class;
   }
}
```

  This class specifies which page wicket should treat as our home page, in our case, `HomePage.class`

- In `HomePage` we see typical wicket code to set up page elements. The bit that is interesting is the injection of the `Game` bean:

```
 @Current Game game;
```

  The `Game` bean is can then be used, for example, by the code for submitting a guess:

```
final Component guessButton = new AjaxButton("GuessButton") {
   protected void onSubmit(AjaxRequestTarget target, Form form) {
      if (game.check()) {
```

Note

All injections may be serialized; actual storage of the bean is managed by JSR-299. Note that Wicket components, like the HomePage and it subcomponents, are *not* JSR-299 beans.

Wicket components allow injection, but they *cannot* use interceptors, decorators and lifecycle callbacks such as `@PostConstruct` or `@Initializer` methods.

- The example uses AJAX for processing of button events, and dynamically hides buttons that are no longer relevant, for example when the user has won the game.
- In order to activate wicket for this webapp, the Wicket filter is added to web.xml, and our application class is specified:

```
<filter>
  <filter-name>wicket.numberguess-example</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>org.jboss.webbeans.examples.wicket.SampleApplication</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>wicket.numberguess-example</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

  Note that the servlet listener is also added, as in the Tomcat example, in order to boostrap Web Beans when Jetty starts, and to hook Web Beans into the Jetty servlet request and session lifecycles.

6.4.3. The numberguess example for Java SE with Swing

This example can be found in the `examples/se/numberguess` folder of the Web Beans distribution.

To run this example:

- Open a command line/terminal window in the `examples/se/numberguess` directory
- Ensure that Maven 2 is installed and in your PATH
- Ensure that the `JAVA_HOME` environment variable is pointing to your JDK installation
- execute the following command

```
mvn -Drun
```

There is an empty `beans.xml` file in the root package (`src/main/resources/beans.xml`), which marks this application as a Web Beans application.

The game's main logic is located in `Game.java`. Here is the code for that class, highlighting the changes made from the web application version:

```
@ApplicationScoped                                          (1)(2)
public class Game implements Serializable
{

   private int number;
   private int guess;
   private int smallest;

   @MaxNumber
   private int maxNumber;

   private int biggest;
   private int remainingGuesses;
   private boolean validNumberRange = true;

   @Current Generator rndGenerator;


   ...

   public boolean isValidNumberRange()
   {
      return validNumberRange;
   }

   public boolean isGameWon()
                                                     (3)
   {
      return guess == number;
   }

   public boolean isGameLost()
   {
      return guess != number && remainingGuesses <= 0;
   }

   public boolean check()
   {
      boolean result = false;
```

```
        if ( checkNewNumberRangeIsValid() )                    (4)
        {
            if ( guess > number )
            {
                biggest = guess - 1;
            }

            if ( guess < number )
            {
                smallest = guess + 1;
            }

            if ( guess == number )
            {
                result = true;
            }

            remainingGuesses--;
        }

        return result;
    }

    private boolean checkNewNumberRangeIsValid()
    {
        return validNumberRange = ( ( guess >= smallest ) && ( guess <= biggest ) );
    }

    @PostConstruct
    public void reset()                                  (5)
    {
        this.smallest = 0;
        ...
        this.number = rndGenerator.next();
    }
}
```

(1) The bean is application scoped instead of session scoped, since an instance of the application represents a single 'session'.

(2) The bean is not named, since it doesn't need to be accessed via EL

(3) There is no JSF `FacesContext` to add messages to. Instead the `Game` class provides additional information about the state of the current game including:

- If the game has been won or lost
- If the most recent guess was invalid

This allows the Swing UI to query the state of the game, which it does indirectly via a class called `MessageGenerator`, in order to determine the appropriate messages to display to the user during the game.

(4) Validation of user input is performed during the `check()` method, since there is no dedicated validation phase

(5) The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. It cannot use `manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random>(){})` as the JSF example does because there will not be any active contexts like there is during a JSF request.

The `MessageGenerator` class depends on the current instance of `Game`, and queries its state in order to determine the appropriate messages to provide as the prompt for the user's next guess and the response to the previous guess. The code for `MessageGenerator` is as follows:

```
public class MessageGenerator
{
   @Current Game game;                                        (1)

   public String getChallengeMessage()                         (2)
   {
      StringBuilder challengeMsg = new StringBuilder( "I'm thinking of a number between " );
      challengeMsg.append( game.getSmallest() );
      challengeMsg.append( " and " );
      challengeMsg.append( game.getBiggest() );
      challengeMsg.append( ". Can you guess what it is?" );

      return challengeMsg.toString();
   }

   public String getResultMessage()                            (3)
   {
      if ( game.isGameWon() )
      {
         return "You guess it! The number was " + game.getNumber();
```

```
        } else if ( game.isGameLost() )
        {
            return "You are fail! The number was " + game.getNumber();
        } else if ( ! game.isValidNumberRange() )
        {
            return "Invalid number range!";
        } else if ( game.getRemainingGuesses() == Game.MAX_NUM_GUESSES )
        {
            return "What is your first guess?";
        } else
        {
            String direction = null;

            if ( game.getGuess() < game.getNumber() )
            {
                direction = "Higher";
            } else
            {
                direction = "Lower";
            }

            return direction + "! You have " + game.getRemainingGuesses() + " guesses left.";
        }
    }
}
```

1. The instance of `Game` for the application is injected here.
2. The `Game`'s state is interrogated to determine the appropriate challenge message.
3. And again to determine whether to congratulate, console or encourage the user to continue.

Finally we come to the `NumberGuessFrame` class which provides the Swing front end to our guessing game.

```
public class NumberGuessFrame  extends javax.swing.JFrame
{
    private @Current Game game;                                   (1)

    private @Current MessageGenerator msgGenerator;                   (2)


    public void start( @Observes @Deployed Manager manager )             (3)
    {
        java.awt.EventQueue.invokeLater( new Runnable()
            {
                public void run()
                {
                    initComponents();
                    setVisible( true );
                }
            } );
    }


    private void initComponents() {                              4

        buttonPanel = new javax.swing.JPanel();
        mainMsgPanel = new javax.swing.JPanel();
        mainLabel = new javax.swing.JLabel();
        messageLabel = new javax.swing.JLabel();
        guessText = new javax.swing.JTextField();
        ...
        mainLabel.setText(msgGenerator.getChallengeMessage());
        mainMsgPanel.add(mainLabel);

        messageLabel.setText(msgGenerator.getResultMessage());
        mainMsgPanel.add(messageLabel);
        ...
    }


    private void guessButtonActionPerformed( java.awt.event.ActionEvent evt )   5
    {
        int guess =  Integer.parseInt(guessText.getText());

        game.setGuess( guess );
        game.check();
        refreshUI();

    }


    private void replayBtnActionPerformed( java.awt.event.ActionEvent evt )   6
    {
       game.reset();
       refreshUI();
    }


    private void refreshUI()                                   7
```

```
    {
        mainLabel.setText( msgGenerator.getChallengeMessage() );
        messageLabel.setText( msgGenerator.getResultMessage() );
        guessText.setText( "" );
        guessesLeftBar.setValue( game.getRemainingGuesses() );
        guessText.requestFocus();
    }

    // swing components
    private javax.swing.JPanel borderPanel;
    ...
    private javax.swing.JButton replayBtn;

}
```

1. The injected instance of the game (logic and state).
2. The injected message generator for UI messages.
3. This application is started in the usual Web Beans SE way, by observing the `@Deployed Manager` event.
4. This method initialises all of the Swing components. Note the use of the `msgGenerator`.
5. `guessButtonActionPerformed` is called when the 'Guess' button is clicked, and it does the following:

    - Gets the guess entered by the user and sets it as the current guess in the `Game`
    - Calls `game.check()` to validate and perform one 'turn' of the game
    - Calls `refreshUI`. If there were validation errors with the input, this will have been captured during `game.check()` and as such will be reflected in the messeges returned by `MessageGenerator` and subsequently presented to the user. If there are no validation errors then the user will be told to guess again (higher or lower) or that the game has ended either in a win (correct guess) or a loss (ran out of guesses).
6. `replayBtnActionPerformed` simply calls `game.reset()` to start a new game and refreshes the messages in the UI.
7. `refreshUI` uses the `MessageGenerator` to update the messages to the user based on the current state of the Game.

6.5. The translator example

The translator example will take any sentences you enter, and translate them to Latin.

The translator example is built as an ear, and contains EJBs. As a result, it's structure is more complex than the numberguess example.

Note

EJB 3.1 and Jave EE 6 allow you to package EJBs in a war, which will make this structure much simpler!

First, let's take a look at the ear aggregator, which is located in `webbeans-translator-ear` module. Maven automatically generates the `application.xml` for us:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-ear-plugin</artifactId>
   <configuration>
      <modules>
         <webModule>
            <groupId>org.jboss.webbeans.examples.translator</groupId>
            <artifactId>webbeans-translator-war</artifactId>
            <contextRoot>/webbeans-translator</contextRoot>
         </webModule>
      </modules>
   </configuration>
</plugin>
```

Here we set the context path, which gives us a nice url (*http://localhost:8080/webbeans-translator*).

Tip

If you aren't using Maven to generate these files, you would need `META-INF/application.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://
java.sun.com/xml/ns/javaee/application_5.xsd"
             version="5">
  <display-name>webbeans-translator-ear</display-name>
   <description>Ear Example for the reference implementation of JSR 299: Web
 Beans</description>

  <module>
```

```
    <web>
      <web-uri>webbeans-translator.war</web-uri>
      <context-root>/webbeans-translator</context-root>
    </web>
  </module>
  <module>
    <ejb>webbeans-translator.jar</ejb>
  </module>
</application>
```

Next, lets look at the war. Just as in the numberguess example, we have a `faces-config.xml` (to enable Facelets) and a `web.xml` (to enable JSF) in `WebContent/WEB-INF`.

More intersting is the facelet used to translate text. Just as in the numberguess example we have a template, which surrounds the form (ommitted here for brevity):

```
<h:form id="NumberGuessMain">

  <table>
    <tr align="center" style="font-weight: bold" >
      <td>
        Your text
      </td>
      <td>
        Translation
      </td>
    </tr>
    <tr>
      <td>
      <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80" />
      </td>
      <td>
        <h:outputText value="#{translator.translatedText}" />
      </td>
    </tr>
  </table>
  <div>
    <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
  </div>

</h:form>
```

The user can enter some text in the lefthand textarea, and hit the translate button to see the result to the right.

Finally, let's look at the ejb module, `webbeans-translator-ejb`. In `src/main/resources/META-INF` there is just an empty `web-beans.xml`, used to mark the archive as containing Web Beans.

We've saved the most interesting bit to last, the code! The project has two simple beans, `SentenceParser` and `TextTranslator` and two enterprise beans, `TranslatorControllerBean` and `SentenceTranslator`. You should be getting quite familiar with what a Web Bean looks like by now, so we'll just highlight the most interesting bits here.

Both `SentenceParser` and `TextTranslator` are dependent beans, and `TextTranslator` uses constructor initialization:

```
public class TextTranslator {
   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Initializer
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator)
   {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
```

`TextTranslator` is a stateless bean (with a local business interface), where the magic happens - of course, we couldn't develop a full translator, but we gave it a good go!

Finally, there is UI orientated controller, that collects the text from the user, and dispatches it to the translator. This is a request scoped, named, stateful session bean, which injects the translator.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController
{

   @Current TextTranslator translator;
```

The bean also has getters and setters for all the fields on the page.

As this is a stateful session bean, we have to have a remove method:

```
   @Remove
   public void remove()
   {

   }
```

The Web Beans manager will call the remove method for you when the bean is destroyed; in this case at the end of the request.

That concludes our short tour of the Web Beans examples. For more on Web Beans , or to help out, please visit *http://www.seamframework.org/WebBeans/Development*.

We need help in all areas - bug fixing, writing new features, writing examples and translating this reference guide.

Part II. Meteor Basics

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

7.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

7.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, "Deployment types"*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

7.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

7.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

7.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

7.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

7.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

7.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

7.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

7.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

7.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

7.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

8.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
   public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

8.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

8.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

8.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

8.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

8.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

8.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

8.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

8.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

8.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

8.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

8.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

9.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
   public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

9.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

9.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

9.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

9.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

9.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

9.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

9.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

9.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

9.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

9.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

9.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

10.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

10.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

10.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

10.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

10.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

10.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

10.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

10.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

10.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

10.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

10.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

10.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

11.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
   public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

11.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

11.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

11.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

11.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

11.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

11.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

11.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

11.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

11.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

11.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

11.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

12.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

12.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

12.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

12.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

12.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

12.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

12.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

12.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

12.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

12.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

12.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

12.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

13.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

13.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

13.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

13.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

13.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

13.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
   ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

13.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

13.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

13.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

13.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/ `@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

13.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

13.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

14.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

14.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

14.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

14.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

14.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

14.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

14.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

14.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

14.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

14.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

14.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

14.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Decorators

Interceptors are a powerful way to capture and separate concerns which are *orthogonal* to the type system. Any interceptor is able to intercept invocations of any Java type. This makes them perfect for solving technical concerns such as transaction management and security. However, by nature, interceptors are unaware of the actual semantics of the events they intercept. Thus, interceptors aren't an appropriate tool for separating business-related concerns.

The reverse is true of *decorators*. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. This makes decorators a perfect tool for modeling some kinds of business concerns. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types.

Suppose we have an interface that represents accounts:

```
public interface Account {
    public BigDecimal getBalance();
    public User getOwner();
    public void withdraw(BigDecimal amount);
    public void deposit(BigDecimal amount);
}
```

Several different Web Beans in our system implement the `Account` interface. However, we have a common legal requirement that, for any kind of account, large transactions must be recorded by the system in a special log. This is a perfect job for a decorator.

A decorator is a simple Web Bean that implements the type it decorates and is annotated `@Decorator`.

```
@Decorator
public abstract class LargeTransactionDecorator
        implements Account {

    @Decorates Account account;

    @PersistenceContext EntityManager em;

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

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

}
```

Unlike other simple Web Beans, a decorator may be an abstract class. If there's nothing special the decorator needs to do for a particular method of the decorated interface, you don't need to implement that method.

15.1. Delegate attributes

All decorators have a *delegate attribute*. The type and binding types of the delegate attribute determine which Web Beans the decorator is bound to. The delegate attribute type must implement or extend all interfaces implemented by the decorator.

This delegate attribute specifies that the decorator is bound to all Web Beans that implement `Account`:

```
@Decorates Account account;
```

A delegate attribute may specify a binding annotation. Then the decorator will only be bound to Web Beans with the same binding.

```
@Decorates @Foreign Account account;
```

A decorator is bound to any Web Bean which:

- has the type of the delegate attribute as an API type, and
- has all binding types that are declared by the delegate attribute.

The decorator may invoke the delegate attribute, which has much the same effect as calling `InvocationContext.proceed()` from an interceptor.

15.2. Enabling decorators

We need to *enable* our decorator in `web-beans.xml`.

```
<Decorators>
   <myapp:LargeTransactionDecorator/>
</Decorators>
```

This declaration serves the same purpose for decorators that the `<Interceptors>` declaration serves for interceptors:

- it enables us to specify a total ordering for all decorators in our system, ensuring deterministic behavior, and
- it lets us enable or disable decorator classes at deployment time.

Interceptors for a method are called before decorators that apply to that method.

Interceptors

Web Beans re-uses the basic interceptor architecture of EJB 3.0, extending the functionality in two directions:

- Any Web Bean may have interceptors, not just session beans.
- Web Beans features a more sophisticated annotation-based approach to binding interceptors to Web Beans.

The EJB specification defines two kinds of interception points:

- business method interception, and
- lifecycle callback interception.

A *business method interceptor* applies to invocations of methods of the Web Bean by clients of the Web Bean:

```
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

A *lifecycle callback interceptor* applies to invocations of lifecycle callbacks by the container:

```
public class DependencyInjectionInterceptor {
   @PostConstruct public void injectDependencies(InvocationContext ctx) { ... }
}
```

An interceptor class may intercept both lifecycle callbacks and business methods.

16.1. Interceptor bindings

Suppose we want to declare that some of our Web Beans are transactional. The first thing we need is an *interceptor binding annotation* to specify exactly which Web Beans we're interested in:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Now we can easily specify that our `ShoppingCart` is a transactional object:

```
@Transactional
public class ShoppingCart { ... }
```

Or, if we prefer, we can specify that just one method is transactional:

```
public class ShoppingCart {
    @Transactional public void checkout() { ... }
}
```

16.2. Implementing interceptors

That's great, but somewhere along the line we're going to have to actually implement the interceptor that provides this transaction management aspect. All we need to do is create a standard EJB interceptor, and annotate it `@Interceptor` and `@Transactional`.

```
@Transactional @Interceptor
public class TransactionInterceptor {
    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

All Web Beans interceptors are simple Web Beans, and can take advantage of dependency injection and contextual lifecycle management.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

    @Resource Transaction transaction;

    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

Multiple interceptors may use the same interceptor binding type.

16.3. Enabling interceptors

Finally, we need to *enable* our interceptor in `web-beans.xml`.

```
<Interceptors>
    <tx:TransactionInterceptor/>
</Interceptors>
```

Whoah! Why the angle bracket stew?

Well, the XML declaration solves two problems:

- it enables us to specify a total ordering for all the interceptors in our system, ensuring deterministic behavior, and
- it lets us enable or disable interceptor classes at deployment time.

For example, we could specify that our security interceptor runs before our `TransactionInterceptor`.

```
<Interceptors>
    <sx:SecurityInterceptor/>
    <tx:TransactionInterceptor/>
</Interceptors>
```

Or we could turn them both off in our test environment!

16.4. Interceptor bindings with members

Suppose we want to add some extra information to our `@Transactional` annotation:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
    boolean requiresNew() default false;
}
```

Web Beans will use the value of `requiresNew` to choose between two different interceptors, `TransactionInterceptor` and `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {
    @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Now we can use `RequiresNewTransactionInterceptor` like this:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

But what if we only have one interceptor and we want the manager to ignore the value of `requiresNew` when binding interceptors? We can use the `@NonBinding` annotation:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
    @NonBinding String[] rolesAllowed() default {};
}
```

16.5. Multiple interceptor binding annotations

Usually we use combinations of interceptor bindings types to bind multiple interceptors to a Web Bean. For example, the following declaration would be used to bind `TransactionInterceptor` and `SecurityInterceptor` to the same Web Bean:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

However, in very complex cases, an interceptor itself may specify some combination of interceptor binding types:

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Then this interceptor could be bound to the `checkout()` method using any one of the following combinations:

```
public class ShoppingCart {
    @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
    @Transactional public void checkout() { ... }
}
```

```
@Transactionl
public class ShoppingCart {
    @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
    public void checkout() { ... }
}
```

16.6. Interceptor binding type inheritance

One limitation of the Java language support for annotations is the lack of annotation inheritance. Really, annotations should have reuse built in, to allow this kind of thing to work:

```
public @interface Action extends Transactional, Secure { ... }
```

Well, fortunately, Web Beans works around this missing feature of Java. We may annotate one interceptor binding type with other interceptor binding types. The interceptor bindings are transitive # any Web Bean with the first interceptor binding inherits the interceptor bindings declared as meta-annotations.

```
@Transactional @Secure
@InterceptorBindingType
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Any Web Bean annotated `@Action` will be bound to both `TransactionInterceptor` and `SecurityInterceptor`. (And even `TransactionalSecureInterceptor`, if it exists.)

16.7. Use of `@Interceptors`

The `@Interceptors` annotation defined by the EJB specification is supported for both enterprise and simple Web Beans, for example:

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
    public void checkout() { ... }
}
```

However, this approach suffers the following drawbacks:

- the interceptor implementation is hardcoded in business code,
- interceptors may not be easily disabled at deployment time, and
- the interceptor ordering is non-global # it is determined by the order in which interceptors are listed at the class level.

Therefore, we recommend the use of Web Beans-style interceptor bindings.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

17.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

17.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

17.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

17.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

17.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

17.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

17.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

17.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

17.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

17.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

17.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

17.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Part III. Meteor Modules

Dependency injection

Web Beans supports three primary mechanisms for dependency injection:

Constructor parameter injection:

```
public class Checkout {

    private final ShoppingCart cart;

    @Initializer
    public Checkout(ShoppingCart cart) {
        this.cart = cart;
    }

}
```

Initializer method parameter injection:

```
public class Checkout {

    private ShoppingCart cart;

    @Initializer
    void setShoppingCart(ShoppingCart cart) {
        this.cart = cart;
    }

}
```

And direct field injection:

```
public class Checkout {

    private @Current ShoppingCart cart;

}
```

Dependency injection always occurs when the Web Bean instance is first instantiated.

- First, the Web Bean manager calls the Web Bean constructor, to obtain an instance of the Web Bean.
- Next, the Web Bean manager initializes the values of all injected fields of the Web Bean.
- Next, the Web Bean manager calls all initializer methods of Web Bean.
- Finally, the `@PostConstruct` method of the Web Bean, if any, is called.

Constructor parameter injection is not supported for EJB beans, since the EJB is instantiated by the EJB container, not the Web Bean manager.

Parameters of constructors and initializer methods need not be explicitly annotated when the default binding type `@Current` applies. Injected fields, however, *must* specify a binding type, even when the default binding type applies. If the field does not specify a binding type, it will not be injected.

Producer methods also support parameter injection:

```
@Produces Checkout createCheckout(ShoppingCart cart) {
    return new Checkout(cart);
}
```

Finally, observer methods (which we'll meet in ???), disposal methods and destructor methods all support parameter injection.

The Web Beans specification defines a procedure, called the *typesafe resolution algorithm*, that the Web Bean manager follows when identifying the Web Bean to inject to an injection point. This algorithm looks complex at first, but once you understand it, it's really quite intuitive. Typesafe resolution is performed at system initialization time, which means that the manager will inform the user immediately if a Web Bean's dependencies cannot be satisfied, by throwing a `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

The purpose of this algorithm is to allow multiple Web Beans to implement the same API type and either:

- allow the client to select which implementation it requires using *binding annotations*,
- allow the application deployer to select which implementation is appropriate for a particular deployment, without changes to the client, by enabling or disabling *deployment types*, or
- allow one implementation of an API to override another implementation of the same API at deployment time, without changes to the client, using *deployment type precedence*.

Let's explore how the Web Beans manager determines a Web Bean to be injected.

18.1. Binding annotations

If we have more than one Web Bean that implements a particular API type, the injection point can specify exactly which Web Bean should be injected using a binding annotation. For example, there might be two implementations of `PaymentProcessor`:

```
@PayByCheque
public class ChequePaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

```
@PayByCreditCard
public class CreditCardPaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

Where `@PayByCheque` and `@PayByCreditCard` are binding annotations:

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

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

A client Web Bean developer uses the binding annotation to specify exactly which Web Bean should be injected.

Using field injection:

```
@PayByCheque PaymentProcessor chequePaymentProcessor;
@PayByCreditCard PaymentProcessor creditCardPaymentProcessor;
```

Using initializer method injection:

```
@Initializer
public void setPaymentProcessors(@PayByCheque PaymentProcessor chequePaymentProcessor,
                                 @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
   this.chequePaymentProcessor = chequePaymentProcessor;
   this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

Or using constructor injection:

```
@Initializer
public Checkout(@PayByCheque PaymentProcessor chequePaymentProcessor,
                @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
   this.chequePaymentProcessor = chequePaymentProcessor;
   this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

18.1.1. Binding annotations with members

Binding annotations may have members:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayBy {
    PaymentType value();
}
```

In which case, the member value is significant:

```
@PayBy(CHEQUE) PaymentProcessor chequePaymentProcessor;
@PayBy(CREDIT_CARD) PaymentProcessor creditCardPaymentProcessor;
```

You can tell the Web Bean manager to ignore a member of a binding annotation type by annotating the member `@NonBinding`.

18.1.2. Combinations of binding annnotations

An injection point may even specify multiple binding annotations:

```
@Asynchronous @PayByCheque PaymentProcessor paymentProcessor
```

In this case, only a Web Bean which has *both* binding annotations would be eligible for injection.

18.1.3. Binding annotations and producer methods

Even producer methods may specify binding annotations:

```
@Produces
@Asynchronous @PayByCheque
PaymentProcessor createAsyncPaymentProcessor(@PayByCheque PaymentProcessor
 processor) {
    return new AsynchronousPaymentProcessor(processor);
}
```

18.1.4. The default binding type

Web Beans defines a binding type `@Current` that is the default binding type for any injection point or Web Bean that does not explicitly specify a binding type.

There are two common circumstances in which it is necessary to explicitly specify `@Current`:

- on a field, in order to declare it as an injected field with the default binding type, and
- on a Web Bean which has another binding type in addition to the default binding type.

18.2. Deployment types

All Web Beans have a *deployment type*. Each deployment type identifies a set of Web Beans that should be conditionally installed in some deployments of the system.

For example, we could define a deployment type named `@Mock`, which would identify Web Beans that should only be installed when the system executes inside an integration testing environment:

```
@Retention(RUNTIME)
  @Target({TYPE, METHOD})
  @DeploymentType
  public @interface Mock {}
```

Suppose we had some Web Bean that interacted with an external system to process payments:

```
public class ExternalPaymentProcessor {

    public void process(Payment p) {
        ...
    }

}
```

Since this Web Bean does not explicitly specify a deployment type, it has the default deployment type `@Production`.

For integration or unit testing, the external system is slow or unavailable. So we would create a mock object:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

    @Override
    public void process(Payment p) {
        p.setSuccessful(true);
    }

}
```

But how does the Web Bean manager determine which implementation to use in a particular deployment?

18.2.1. Enabling deployment types

Web Beans defines two built-in deployment types: `@Production` and `@Standard`. By default, only Web Beans with the built-in deployment types are enabled when the system is deployed. We can identify additional deployment types to be enabled in a particular deployment by listing them in `web-beans.xml`.

Going back to our example, when we deploy our integration tests, we want all our `@Mock` objects to be installed:

```
<WebBeans>
    <Deploy>
        <Standard/>
        <Production/>
        <test:Mock/>
    </Deploy>
</WebBeans>
```

Now the Web Bean manager will identify and install all Web Beans annotated `@Production`, `@Standard` or `@Mock` at deployment time.

The deployment type `@Standard` is used only for certain special Web Beans defined by the Web Beans specification. We can't use it for our own Web Beans, and we can't disable it.

The deployment type `@Production` is the default deployment type for Web Beans which don't explicitly declare a deployment type, and may be disabled.

18.2.2. Deployment type precedence

If you've been paying attention, you're probably wondering how the Web Bean manager decides which implementation # `ExternalPaymentProcessor` or `MockPaymentProcessor` # to choose. Consider what happens when the manager encounters this injection point:

```
@Current PaymentProcessor paymentProcessor
```

There are now two Web Beans which satisfy the `PaymentProcessor` contract. Of course, we can't use a binding annotation to disambiguate, since binding annotations are hard-coded into the source at the injection point, and we want the manager to be able to decide at deployment time!

The solution to this problem is that each deployment type has a different *precedence*. The precedence of the deployment types is determined by the order in which they appear in `web-beans.xml`. In our example, `@Mock` appears later than `@Production` so it has a higher precedence.

Whenever the manager discovers that more than one Web Bean could satisfy the contract (API type plus binding annotations) specified by an injection point, it considers the relative precedence of the Web Beans. If one has a higher precedence than the others, it chooses the higher precedence Web Bean to inject. So, in our example, the Web Bean manager will inject `MockPaymentProcessor` when executing in our integration testing environment (which is exactly what we want).

It's interesting to compare this facility to today's popular manager architectures. Various "lightweight" containers also allow conditional deployment of classes that exist in the classpath, but the classes that are to be deployed must be explicity, individually, listed in configuration code or in some XML configuration file. Web Beans does support Web Bean definition and configuration via XML, but in the common case where no complex configuration is required, deployment types allow a whole set of Web Beans to be enabled with a single line of XML. Meanwhile, a developer browsing the code can easily identify what deployment scenarios the Web Bean will be used in.

18.2.3. Example deployment types

Deployment types are useful for all kinds of things, here's some examples:

- `@Mock` and `@Staging` deployment types for testing
- `@AustralianTaxLaw` for site-specific Web Beans
- `@SeamFramework`, `@Guice` for third-party frameworks which build on Web Beans
- `@Standard` for standard Web Beans defined by the Web Beans specification

I'm sure you can think of more applications...

18.3. Fixing unsatisfied dependencies

The typesafe resolution algorithm fails when, after considering the binding annotations and and deployment types of all Web Beans that implement the API type of an injection point, the Web Bean manager is unable to identify exactly one Web Bean to inject.

It's usually easy to fix an `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

To fix an `UnsatisfiedDependencyException`, simply provide a Web Bean which implements the API type and has the binding types of the injection point # or enable the deployment type of a Web Bean that already implements the API type and has the binding types.

To fix an `AmbiguousDependencyException`, introduce a binding type to distinguish between the two implementations of the API type, or change the deployment type of one of the implementations so that the Web Bean manager can use deployment type precedence to choose between them. An `AmbiguousDependencyException` can only occur if two Web Beans share a binding type and have exactly the same deployment type.

There's one more issue you need to be aware of when using dependency injection in Web Beans.

18.4. Client proxies

Clients of an injected Web Bean do not usually hold a direct reference to a Web Bean instance.

Imagine that a Web Bean bound to the application scope held a direct reference to a Web Bean bound to the request scope. The application scoped Web Bean is shared between many different requests. However, each request should see a different instance of the request scoped Web bean!

Now imagine that a Web Bean bound to the session scope held a direct reference to a Web Bean bound to the application scope. From time to time, the session context is serialized to disk in order to use memory more efficiently. However, the application scoped Web Bean instance should not be serialized along with the session scoped Web Bean!

Therefore, unless a Web Bean has the default scope `@Dependent`, the Web Bean manager must indirect all injected references to the Web Bean through a proxy object. This *client proxy* is responsible for ensuring that the Web Bean instance that receives a method invocation is the instance that is associated with the current context. The client proxy also allows Web Beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected Web Beans.

Unfortunately, due to limitations of the Java language, some Java types cannot be proxied by the Web Bean manager. Therefore, the Web Bean manager throws an `UnproxyableDependencyException` if the type of an injection point cannot be proxied.

The following Java types cannot be proxied by the Web Bean manager:

- classes which are declared `final` or have a `final` method,
- classes which have no non-private constructor with no parameters, and
- arrays and primitive types.

It's usually very easy to fix an `UnproxyableDependencyException`. Simply add a constructor with no parameters to the injected class, introduce an interface, or change the scope of the injected Web Bean to `@Dependent`.

18.5. Obtaining a Web Bean by programatic lookup

The application may obtain an instance of the interface `Manager` by injection:

```
@Current Manager manager;
```

The `Manager` object provides a set of methods for obtaining a Web Bean instance programatically.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class);
```

Binding annotations may be specified by subclassing the helper class `AnnotationLiteral`, since it is otherwise difficult to instantiate an annotation type in Java.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new AnnotationLiteral<CreditCard>(){});
```

If the binding type has an annotation member, we can't use an anonymous subclass of `AnnotationLiteral` # instead we'll need to create a named subclass:

```
abstract class CreditCardBinding
   extends AnnotationLiteral<CreditCard>
   implements CreditCard {}
```

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new CreditCardBinding() {
```

```
                    public void value() { return paymentType; }
                  } );
```

18.6. Lifecycle callbacks, `@Resource`, `@EJB` and `@PersistenceContext`

Enterprise Web Beans support all the lifecycle callbacks defined by the EJB specification: `@PostConstruct`, `@PreDestroy`, `@PrePassivate` and `@PostActivate`.

Simple Web Beans support only the `@PostConstruct` and `@PreDestroy` callbacks.

Both enterprise and simple Web Beans support the use of `@Resource`, `@EJB` and `@PersistenceContext` for injection of Java EE resources, EJBs and JPA persistence contexts, respectively. Simple Web Beans do not support the use of `@PersistenceContext(type=EXTENDED)`.

The `@PostConstruct` callback always occurs after all dependencies have been injected.

18.7. The `InjectionPoint` object

There are certain kinds of dependent objects # Web Beans with scope `@Dependent` # that need to know something about the object or injection point into which they are injected in order to be able to do what they do. For example:

- The log category for a `Logger` depends upon the class of the object that owns it.
- Injection of a HTTP parameter or header value depends upon what parameter or header name was specified at the injection point.
- Injection of the result of an EL expression evaluation depends upon the expression that was specified at the injection point.

A Web Bean with scope `@Dependent` may inject an instance of `InjectionPoint` and access metadata relating to the injection point to which it belongs.

Let's look at an example. The following code is verbose, and vulnerable to refactoring problems:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

This clever little producer method lets you inject a JDK `Logger` without explicitly specifying the log category:

```
class LogFactory {

  @Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
  }

}
```

We can now write:

```
@Current Logger log;
```

Not convinced? Then here's a second example. To inject HTTP parameters, we need to define a binding type:

```
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
  @NonBinding public String value();
}
```

We would use this binding type at injection points as follows:

```
@HttpParam("username") String username;
@HttpParam("password") String password;
```

The following producer method does the work:

```
class HttpParams

  @Produces @HttpParam("")
  String getParamValue(ServletRequest request, InjectionPoint ip) {
    return request.getParameter(ip.getAnnotation(HttpParam.class).value());
  }

}
```

(Note that the `value()` member of the `HttpParam` annotation is ignored by the Web Bean manager since it is annotated `@NonBinding.`)

The Web Bean manager provides a built-in Web Bean that implements the `InjectionPoint` interface:

```
public interface InjectionPoint {
   public Object getInstance();
   public Bean<?> getBean();
   public Member getMember():
   public <T extends Annotation> T getAnnotation(Class<T> annotation);
   public Set<T extends Annotation> getAnnotations();
}
```

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

19.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

19.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

19.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

19.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

19.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

19.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

19.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

19.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

19.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

19.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

19.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

19.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

20.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

20.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

20.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

20.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

20.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

20.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

20.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

20.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

20.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

20.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

20.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

20.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

21.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

21.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

21.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

21.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

21.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

21.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

21.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

21.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

21.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

21.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

21.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in ???.

21.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in ???.

Part IV. Meteor Reference

Getting started with Web Beans

So you're already keen to get started writing your first Web Bean? Or perhaps you're skeptical, wondering what kinds of hoops the Web Beans specification will make you jump through! The good news is that you've probably already written and used hundreds, perhaps thousands of Web Beans. You might not even remember the first Web Bean you wrote.

22.1. Your first Web Bean

With certain, very special exceptions, every Java class with a constructor that accepts no parameters is a Web Bean. That includes every JavaBean. Furthermore, every EJB 3-style session bean is a Web Bean. Sure, the JavaBeans and EJBs you've been writing every day have not been able to take advantage of the new services defined by the Web Beans specification, but you'll be able to use every one of them as Web Beans # injecting them into other Web Beans, configuring them via the Web Beans XML configuration facility, even adding interceptors and decorators to them # without touching your existing code.

Suppose that we have two existing Java classes, that we've been using for years in various applications. The first class parses a string into a list of sentences:

```
public class SentenceParser {
    public List<String> parse(String text) { ... }
}
```

The second existing class is a stateless session bean front-end for an external system that is able to translate sentences from one language to another:

```
@Stateless
public class SentenceTranslator implements Translator {
    public String translate(String sentence) { ... }
}
```

Where `Translator` is the local interface:

```
@Local
public interface Translator {
    public String translate(String sentence);
}
```

Unfortunately, we don't have a preexisting class that translates whole text documents. So let's write a Web Bean that does this job:

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Initializer
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    public String translate(String text) {
        StringBuilder sb = new StringBuilder();
        for (String sentence: sentenceParser.parse(text)) {
            sb.append(sentenceTranslator.translate(sentence));
        }
        return sb.toString();
    }

}
```

We may obtain an instance of `TextTranslator` by injecting it into a Web Bean, Servlet or EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
    this.textTranslator = textTranslator;
}
```

Alternatively, we may obtain an instance by directly calling a method of the Web Bean manager:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

But wait: `TextTranslator` does not have a constructor with no parameters! Is it still a Web Bean? Well, a class that does not have a constructor with no parameters can still be a Web Bean if it has a constructor annotated `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean

manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

22.2. What is a Web Bean?

So what, *exactly*, is a Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

However, like a stateless or singleton model, but *unlike* stateful session beans, the client does not control the lifecycle of the instance by explicitly creating and destroying it. Instead, the *scope* of the Web Bean determines:

- the lifecycle of each instance of the Web Bean and
- which clients share a reference to a particular instance of the Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Clients (for example, other Web Beans) executing in the same context will see the same instance of the Web Bean. But clients in a different context will see a different instance.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages, and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- they interact via well-defined public APIs

- their lifecycles are completely decoupled

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Section 18.2, “Deployment types”*.

Note that not all clients of a Web Bean are Web Beans. Other objects such as Servlets or Message-Driven Beans # which are by nature not injectable, contextual objects # may also obtain references to Web Beans by injection.

Enough hand-waving. More formally, according to the spec:

> A Web Bean comprises:
>
> - A (nonempty) set of API types
> - A (nonempty) set of binding annotation types
> - A scope
> - A deployment type
> - Optionally, a Web Bean name
> - A set of interceptor binding types
> - A Web Bean implementation

Let's see what some of these terms mean, to the Web Bean developer.

22.2.1. API types, binding types and dependency injection

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- an API type, together with
- a set of binding types.

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Chapter 18, Dependency injection*.

22.2.2. Deployment types

Deployment types let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

We'll talk more about this unique and powerful feature in *Section 18.2, “Deployment types”*.

22.2.3. Scope

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

For example, any web application may have *session scoped* Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

By default, Web Beans belong to a special scope called the *dependent pseudo-scope*. Web Beans with this scope are pure dependent objects of the object into which they are injected, and their lifecycle is bound to the lifecycle of that object.

We'll talk more about scopes in *???*.

22.2.4. Web Bean names and Unified EL

A Web Bean may have a *name*, allowing it to be used in Unified EL expressions. It's easy to specify the name of a Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Now we can easily use the Web Bean in any JSF or JSP page:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable>
```

It's even easier to just let the name be defaulted by the Web Bean manager:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In this case, the name defaults to `shoppingCart` # the unqualified class name, with the first character changed to lowercase.

22.2.5. Interceptor binding types

Web Beans supports the interceptor functionality defined by EJB 3, not only for EJB beans, but also for plain Java classes. In addition, Web Beans provides a new approach to binding interceptors to EJB beans and other Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

However, it is more elegant, and better practice, to indirect the interceptor binding through an *interceptor binding type*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

We'll discuss Web Beans interceptors and decorators in *Chapter 16, Interceptors* and *Chapter 15, Decorators*.

22.3. What kinds of objects can be Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

22.3.1. Simple Web Beans

The Web Beans specification says that a concrete Java class is a *simple* Web Bean if:

- it is not an EE container-managed component, like an EJB, a Servlet or a JPA entity,
- it is not a non-static static inner class,
- it is not a parameterized type, and
- it has a constructor with no parameters, or a constructor annotated `@Initializer`.

Thus, almost every JavaBean is a simple Web Bean.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

22.3.2. Enterprise Web Beans

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

    ...

    @Remove
    public void destroy() {}

}
```

So when should we use an enterprise Web Bean instead of a simple Web Bean? Well, whenever we need the advanced enterprise services offered by EJB, such as:

- method-level transaction management and security,
- concurrency management,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remote and web service invocation, and
- timers and asynchronous methods,

we should use an enterprise Web Bean. When we don't need any of these things, a simple Web Bean will serve just fine.

Many Web Beans (including any session or application scoped Web Bean) are available for concurrent access. Therefore, the concurrency management provided by EJB 3.1 is especially useful. Most session and application scoped Web Beans should be EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Finally, it's usually obvious when method-level transaction management, method-level security, timers, remote methods or asynchronous methods are needed.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

22.3.3. Producer methods

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

    private Random random = new Random( System.currentTimeMillis() );

    @Produces @Random int next() {
        return random.nextInt(100);
    }

}
```

The result of a producer method is injected just like any other Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Some producer methods return objects that require explicit destruction:

```
@Produces @RequestScoped Connection connect(User user) {
    return createConnection( user.getId(), user.getPassword() );
}
```

These producer methods may define matching *disposal methods*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

This disposal method is called automatically by the Web Bean manager at the end of the request.

We'll talk much more about producer methods in *???*.

22.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *???*.