


4. IMPLEMENTING BUSINESS LOGIC BY USING EJBS


4.1. Implementing Business Logic by Using EJBS:EJBs and EJB Container

Objectives

This lesson teaches how to implement business logic with Enterprise JavaBeans. After completing this lesson, you should be able to:

- Create Session EJB components
- Create EJB business methods
- Manage EJB life cycle with container callbacks
- Use asynchronous EJB operations
- Control transactions
- Create EJB timers
- Create and apply interceptors



 Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Implementación de lógica de negocios con Enterprise JavaBeans. Bueno, en este capítulo, veremos cómo creamos componentes de Enterprise JavaBeans. Y estamos viendo beans de sesión aquí. Los beans controlados por mensajes se cubren en detalle en la siguiente lección, API JMS. Pero estamos empezando con la sesión M

¿Cómo crear beans de sesión? ¿Cómo agregarles métodos comerciales? ¿Cómo gestionamos sus ciclos de vida con devoluciones de llamadas de contenedores? ¿Cómo hacemos invocaciones EJB asíncronas? ¿Cómo controlamos las transacciones?

Comenzamos el tema de transacciones en el capítulo JEPA, pero necesitamos completar ese tema de transacciones poniéndolo en un contexto de transacciones administradas por contenedores. Y este capítulo nos dará la oportunidad de hacerlo. Además, en este capítulo, cubrimos los temporizadores e interceptores de Enterprise JavaBeans.

EJBs and EJB Container

Enterprise JavaBean

- Server-side component that encapsulates the business logic of an application

EJB container

- An environment in which EJBs are executed
- Provides system-level services, such as transactions and security, to the enterprise beans

Types of EJB containers:

- Full implementation
- Embeddable

```
import javax.ejb.embeddable.*;
...
EJBContainer container = EJBContainer.createEJBContainer();
Context ctx = container.getContext();
SomeBean foo = (SomeBean)ctx.lookup("java:global/SomeBean");
...
container.close();
```



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Features of EJBs:

1. **Reusable:** The application assembler can build new applications from existing beans
2. **Portable:** Use the standard APIs, these applications can run on any compliant Java EE server.
3. **Easy to Develop and use:** As the EJB container is responsible for system-level services, such as transaction management and security authorization, the bean developer can concentrate on solving business problems.

You should consider using enterprise beans if your application requires scalability, data integrity, and variety of clients.

The EJB container provides the following functionality:

- Encapsulates access to external resources such as databases and legacy systems
- Manages the life cycles of instances of the EJB component's implementation class
- Isolates the class that provides the implementation from its clients
- Provides timer services, and can invoke methods at certain times, which allows the EJB component to perform scheduled tasks
- Monitors, for message-driven beans, a message queue on behalf of the EJB component

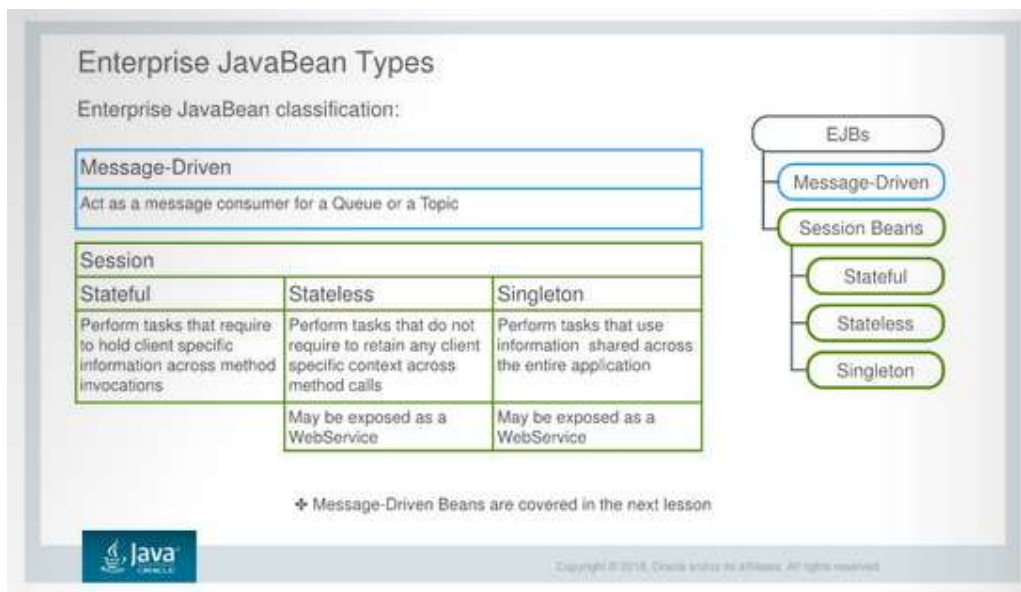
Embedded EJB Container

Most of the services present in the enterprise bean container in a Java EE server are available in the embedded enterprise bean container, including injection, container-managed transactions, and security. The enterprise bean components execute similarly in both embedded and Java EE environments and, therefore, the same enterprise bean can be easily reused in both stand-alone and networked applications.

Comencemos con el rol de un Enterprise JavaBean y un contenedor de Enterprise JavaBeans. Un Enterprise JavaBean es un componente del lado del servidor que lleva a cabo su lógica empresarial. Un contenedor EJB es el entorno que ejecuta estos componentes. Curiosamente, en realidad puede crear un contenedor EJB mediante programación. Aquí está.

Sí, hay un fragmento de código. Y dudo que lo hagas de verdad, porque la aplicación Java ya creó el contenedor EJB para ti. Pero, no sé, tal vez si desea probar algún Enterprise JavaBean sin implementarlo en un servidor real. Sí. Entonces, en realidad puede crear estos contenedores similares a EJB para pruebas locales similares, tal vez.

Y, por supuesto, recuerde que hablamos de ello en la sesión anterior de este curso, dos tipos de contenedores Enterprise JavaBeans, completos y livianos integrables. El contenedor ligero está disponible al instante para usted, incluso con el perfil web para Java. El contenedor completo requiere soporte de plataforma completa. Y hablamos de las diferencias, ¿no? Recuerde, los beans irrevocables remotos son compatibles con el contenedor completo, pero no con el ligero. Sí. ESTÁ BIEN.



De todos modos, los tipos Enterprise JavaBean. Hay beans de sesión y beans controlados por mensajes. Beans controlados por mensajes que se tratarán en la siguiente lección. Por lo tanto, son suscriptores de colas de mensajes y temas. Los beans de sesión están cubiertos en este momento. Y hay tres tipos de beans de sesión, con estado, sin estado y singleton.

Stateful realiza las tareas específicas de un cliente en particular. Porque, recuerde, los beans con estado tienen una conexión permanente con ese cliente. Los beans sin estado realizan tareas que no requieren una conectividad tan permanente. Se le dan al cliente por la duración de la llamada.

Y luego son devueltos. Y podría ser un conjunto de instancias de beans sin estado utilizados por varios clientes. Y, por último, singleton que crea una sola instancia de ese Enterprise JavaBean para compartir entre todos los clientes de su servidor.

Puede exponer un Enterprise JavaBean como un servicio web, pero debe ser sin estado o singleton. Entonces, solo estos dos tipos pueden exponerse como servicios web. Los beans con estado no pueden porque tienen que estar vinculados a un cliente específico a través de un protocolo no transitorio. Entonces no pueden ser servicios web. Su único propósito esencial es estar vinculado a clientes particulares.

Session EJBs

There are three types of EJB Session Beans:

- Stateless
 - Annotated with `@Stateless`
 - Not associated with any particular client
- Stateful
 - Annotated with `@Stateful`
 - One bean instance per client
- Singleton
 - Annotated with `@Singleton`
 - One bean instance per JVM

Optionally, inject `EJBContext` object that provides:

- Access to security properties
- Context data, such as `Interceptor` or `WebService` context
- Transaction control

```
package demo01;
import javax.ejb.*;
@Stateless
public class ABean {
}

package demo01;
import javax.ejb.*;
@Stateful
public class BBean implements Serializable {
}

package demo01;
import javax.ejb.*;
@Singleton
public class CBean {
}

@Inject
private EJBContext context;
```



Copyright © 2018. Oracle and/or its affiliates. All rights reserved.

By separating the presentation from the business logic, EJBs can serve the needs of a presentation object and their life cycle can be managed by another process.

- Scalability through pooling
- Transaction management
- Injection, rather than instantiation
- Services can be swapped out without rewriting presentation code

The Session Bean class must:

- Be top-level class
- Be public class
- Not be final
- Not be abstract
- Have a public constructor that takes no parameters
- Not define the `finalize` method
- Implement the methods of the bean's business interfaces, if any

Stateless Session Beans

- The bean does not retain client-specific information.
- A client might not use the same session bean instance during subsequent method calls.
- A large number of clients can be handled at the same time.

Stateful Session Beans

- Beans belong to a particular client for an entire conversation.
- The client connection exists until the client removes the bean or the session times out.
- The container maintains a separate EJB object and EJB instance for each client.

There is always a cost to maintain client state; maintaining more state than what is needed or using stateful session beans when a stateless bean would be adequate, negatively impacts performance. If an application must store client state, stateful session beans are an effective way to store the client state.

Singleton Session Beans

The `EJBContext` interface provides an instance with access to the container-provided runtime context of an enterprise bean instance.

This interface is extended by the `SessionContext`, `EntityContext`, and `MessageDrivenContext` interfaces to provide additional methods specific to the enterprise interface bean type.

¿Cómo los creas? De forma minimalista, todo lo que necesita hacer es poner una anotación, como sin estado, por ejemplo. Apátrida, con estado, singleton, eso es todo. Sólo eso, ¿sí? Derecha.

Opcionalmente, si desea inyectar un objeto de contexto Enterprise JavaBeans, también puede hacerlo en cualquiera de estos beans. Y eso es si necesita obtener acceso, acceso programático, a cosas como seguridad, propiedades, datos de contexto, como, no sé, interceptores, contexto de servicio web o realizar control de transacciones. Así que esa podría ser la razón por la que te lo inyectas.

Accessing Session Beans

An Session bean can have different views (ways of accessing the bean):

- **Remote:** Used by remote invokers (all parameters and returned values are serialized)
- **Local:** Used by local invokers
- **WebService:** Used by external web service invokers
- **No Interface:** Used by local invokers (All public methods are automatically exposed.)

```
@Stateless
@Remote(ProductFacadeRemote.class)
@Local(ProductFacadeLocal.class)
@WebService
@LocalBean
public class ProductFacade
    implements ProductFacadeLocal,
               ProductFacadeRemote {
    public Product findProduct(int id){...}
    public void createProduct(Product product){...}
    @WebMethod
    public void removeProduct(int id){...}
    public void updateProduct(Product product){...}
}
```

```
@Remote
public interface ProductFacadeRemote {
    public Product findProduct(int id);
    ...
}
```

```
@Local
public interface ProductFacadeLocal {
    public void createProduct(Product product);
    ...
}
```



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Remote Interface

A session bean that is implemented with a remote interface is accessible by components:

- Within the application
- Outside of the application
- Co-resident in the JVM
- On a different JVM (most likely)

Session beans with remote capabilities simplify the development of large-scale distributed solutions. However, there are potential challenges with remote beans, such as responsiveness and bandwidth. Remote beans are typically accessed by clients outside of the container and they act as an entry-point to a complex middleware solution.

Local Interface

A session bean that is implemented with a local interface is accessible only by the components and resources that are packaged within the application that is running on the same JVM.

Implementing session beans by using a local interface has several advantages:

- Local interfaces create encapsulation, in the same manner as access modifiers.
- Hiding certain components from other resources may be beneficial when you are dealing with sensitive algorithms.

Local interfaces adopt a pass-by-reference semantic, which is more efficient than remote communication.

No-Interface Implementation

Starting with EJB 3.1, EJB components can be developed without the creation of a separate business interface. This implementation strategy is known as a no-interface implementation.

- This involves a less cumbersome development process; however, it does have certain side effects.
- To create a no-interface implementation of a session bean, apply the `@LocalBean` annotation directly to the bean class, or assume the default local client access mode.
- When using the no-interface representation of a session bean, all the public methods defined within the bean are considered part of the local interface.

Web Service Implementation

The Stateless session and Singleton session beans may have web service clients. A web service client accesses a session bean through the web service client view. The web service client view is described by the WSDL document for the web service that the bean implements.

The web service client view of an enterprise bean is location-independent and remotable. Web service clients may be Java clients or clients not written in the Java programming language. A web service client that is a Java client accesses the web service by means of the JAX-WS or JAX-RPC client APIs. Such EJB classes must be annotated with either the `javax.jws.WebService` or the `javax.jws.WebServiceProvider` annotation and business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.

Note: Remote and Local interfaces can contain same operations. However, you need to remember that unlike local interface, remote interface parameters and return values are always serialized. This means that Local and Remote interface operation semantics may be quite different.

Note: It is advisable to package Remote interface and related classes (those that are used as parameters and return types) into a separate archive, so it could be distributed on to both client and server tiers.

Con respecto a los tipos de interfaces que podría usar para exponer el Enterprise JavaBean a las personas que llaman, hay varias opciones. Por supuesto, puede hacer la interfaz remota, exponer algunas de las operaciones del bean a las personas que llaman remotamente. Y ese es ese ejemplo, la interfaz remota define los métodos que desea que usen las personas que llaman remotamente.

Puede usar la interfaz local, definir los métodos que desea que usen las personas que llaman locales. Tal vez un conjunto diferente de métodos, no lo sé. Ciertamente, puede exponer el bean como un servicio web. Ahora permita que los métodos de beans se llamen como métodos de servicio. Pero hay otra opción interesante. Y se llama frijol local, o sin opción de interfaz.

No confunda frijol local con local. son diferentes La opción local implica que tiene una interfaz. Y en esa interfaz, usted describe qué métodos le gustaría que la persona que llama pueda invocar. Un bean local dice, no te molestes con una interfaz. Cualquier método público en el bean, solo permita que llame, y eso es todo.

Entonces, en un caso simple, en realidad, podría decir, oh, bueno, ese es un bean local, no es necesario crear ninguna interfaz en particular, solo use el bean. ¿Derecha? Si está satisfecho con que solo los métodos públicos estén expuestos a las personas que llaman. Si necesita más control, sobre qué método en el bean se expondrá a qué llamador, entonces use interfaces.

Stateless Session Bean Life Cycle

EJB Container instantiates stateless session EJBs as pooled instances, or when the caller injects or looks up the bean.

- The life cycle callback methods are annotated with : @PostConstruct and @PreDestroy annotations
- A @PostConstruct annotated method is invoked when a bean instance is created
- A @PreDestroy annotated method is invoked when the bean instance is destroyed and the bean's instance is then ready for garbage collection.

```
package demo;\nimport javax.ejb.*;\n\n@Stateless\npublic class SomeBean {\n    @PostConstruct\n    public void init() {...}\n    @PreDestroy\n    public void cleanup() {...}\n    public void doThings() {...}\n}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The stateless session bean life-cycle diagram shown in the slide provides the context for the stateless session bean life-cycle events. Stateless session beans have a life cycle that mirrors the common life cycle that was previously described.

A stateless session bean generates the following lifecycle events:

- PostConstruct
- PreDestroy

Instantiation of a Stateless Session Bean:

Stateless session beans experience instantiation on the first client request for the session bean in the naming system. That is, when a client uses dependency injection or the more traditional JNDI mechanism to look up a session bean, the container determines whether a stateless session bean exists, and if one does not, it creates one.

Most EJB containers use a pooling strategy for session beans. Pooling of session beans allows the container to reuse objects that are "idle" instead of creating new objects, handling the request, and then destroying them.

Also remember that stateless session beans are not tied to a specific client. As a result, subsequent "lookups" of a stateless session bean in the EJB container do not automatically translate into subsequent (and additional) new instance creation.

Ciclos de vida. Para cada tipo de bean necesitamos cubrir cuáles son las opciones del ciclo de vida. Entonces, el bean de sesión sin estado podría estar en estado inexistente y en estado listo. Cuando pasa de la inexistencia al estado listo, en esa etapa el contenedor puede invocar un método @PostConstruct sobre ese bean. Si se desecha, si pasa del estado listo al estado inexistente, el contenedor puede invocar el método @PreDestroy.

Simplemente puede escribir un par de métodos para @PostConstruct, @PreDestroy, simplemente anule los métodos, de verdad. En este caso particular, se llaman limpieza y limpieza, pero puedes llamarlo como quieras. Los nombres de los métodos no son significativos.

Echemos un vistazo al ciclo de vida de un bean de sesión sin estado. Podría estar en dos estados, inexistente y listo. Viaja de un estado a otro. Y a medida que pasa de la inexistencia al estado listo, un contenedor puede invocar el método que ha anotado con una anotación @PostConstruct.

La firma del método... bueno, cualquier método funcionará realmente. El nombre no es significativo. No tiene que llamarse init. Puedes llamarlo como quieras. Y luego, si está desechando el bean, un contenedor puede invocar el método @PreDestroy. Ahí está. Esa es la anotación.

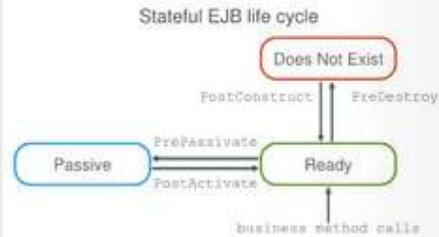
Nuevamente, no tiene que llamarse nada en particular, no tiene que llamarse limpieza, sino solo el método que le gustaría invocar en esa etapa, y cualquier otro método comercial que tenga el bean. Entonces, @PostConstruct y @PreDestroy, básicamente, se usan para asignar, inicializar cualquier recurso que desee que use ese bean, o desechar y limpiar estos recursos cuando el bean está a punto de ser recolectado como basura, cuando está marcado como listo para ser desechado.

Stateful Session Bean Life Cycle

Container allocates a Stateful session bean instance per caller.

- `PostConstruct` and `PreDestroy` work in the same way as in the Stateless session bean.
- `PrePassivate` is used when bean instance is swapped out of memory.
- `PostActivate` is used when bean instance is restored to memory.
- Passivation/Activation can be turned off with the `passivationCapable=false` attribute.
- `Remove` is used to define a business method that client call call to request this bean instance to be removed.

```
package demo2;  
import javax.ejb.*;  
@Stateful(passivationCapable=true)  
public class SomeBean implements Serializable {  
    @PostConstruct  
    public void init() {...}  
    @PreDestroy  
    public void cleanup() {...}  
    @PostActivate  
    public void restore() {...}  
    @PrePassivate  
    public void save() {...}  
    public void doThings() {...}  
    @Remove  
    public void remove() {...}  
}
```



Stateful Session Bean Operational Characteristics

With stateful session beans:

- The bean belongs to a particular client for an entire conversation
- The client connection exists until the client removes the bean or the session times out
- The container maintains a separate EJB object and EJB instance for each client

Comparison of Stateless and Stateful Behavior

Session beans can be stateless or stateful. The statefulness of a bean depends on the type of business function it performs.

In a stateless client-service interaction, no client-specific information is maintained beyond the duration of a single method invocation.

Alternatively, stateful services require that information that is obtained during one method invocation be available during subsequent method calls:

- Shopping carts
- Multipage data entry
- Online banking

Note: Stateful session beans are by default passivation capable and, therefore, there is no need to use the `passivationCapable` attribute, unless you want to switch passivation off. Alternatively, you can use `ejb-jar.xml` deployment descriptor to control this property.

Note: If Enterprise JavaBeans are invoked from the web tier, it is likely that state will be maintained by the `HttpSession` object in a web container. Therefore, EJB components can actually be stateless.

Ahora echemos un vistazo al ciclo de vida del bean de sesión con estado. `@PostConstruct` y `@PreDestroy`, idénticos. No repetiría toda la historia porque funciona de la misma manera. Pero el bean de sesión con estado tiene un estado adicional, que en realidad puede desactivar con el atributo de fallas con capacidad de pasivación. Pero el valor predeterminado es verdadero. La capacidad de pasivación por defecto es verdadera. Realmente no necesitas decir eso. Y lo que significa es que permite que se intercambie la instancia de un bean de sesión con estado, como tomarlo de la RAM y volcarlo en algún lugar y luego restaurarlo del almacenamiento a la memoria.

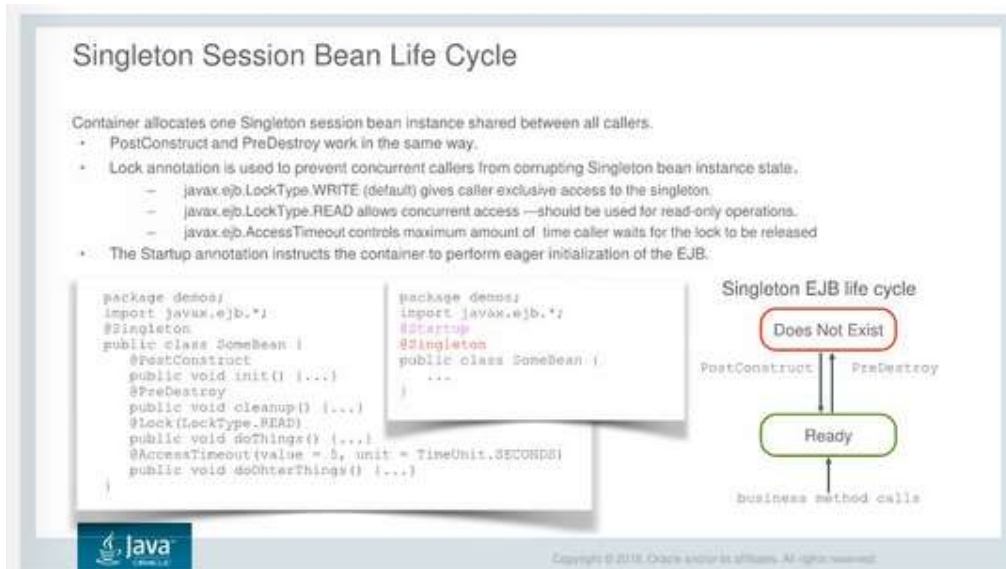
¿Por qué querías hacer eso? Porque los beans de sesión con estado están vinculados a clientes específicos. Si ese cliente está inactivo, un servidor todavía tiene que mantener el bean de sesión con estado en la memoria. Eso consume recursos. Eso es un lastre para sus recursos. Entonces puede cambiarlo de la memoria si el cliente no está usando ese bean actualmente.

Con los apátridas, no había ningún problema. El bean de sesión sin estado se puede reciclar para otro cliente de forma instantánea. Todos los clientes utilizan las mismas instancias de beans de sesión sin estado al mismo tiempo. Está bien. Pero con stateful, el problema sería que mantendrá el bean y la memoria sin usar mientras el cliente todavía tenga una conexión, aunque el cliente no esté haciendo nada con ese bean actualmente. Y por lo tanto, corre el riesgo de quedarse sin RAM.

Por lo tanto, si tiene demasiados clientes concurrentes, demasiadas instancias de beans de sesión con estado, es posible que desee cambiarlos de memoria. Si desea hacer algo en el momento en que intercambia el bean desde la RAM, llamado método `@PrePassivate`, ahí lo tiene. Y cuando restaure, necesita `@PostActivate`. Así que `@PrePassivate` y `@PostActivate` par de métodos.

Y una cosa más: un cliente puede solicitar explícitamente al servidor que finalice el bean de sesión con estado y lo elimine. El cliente ya no quiere ese frijol. Por supuesto, con stateless no tendría ningún sentido porque el cliente no tiene un enlace a una instancia específica de un bean de sesión sin estado. Entonces, el cliente simplemente deja de usar el bean de sesión sin estado. Luego es utilizado por otro cliente. Está bien.

Pero con stateful, el bean está robando memoria hasta que el cliente cierra la conexión. ¿Qué pasa si el cliente quiere mantener la conexión pero ya no quiere ese bean con estado? OK, bueno, eliminar el método, supongo. Esto es lo que podría hacer para decirle al servidor que elimine esa instancia.



Singleton session beans have a life cycle that mirrors the common session bean lifecycle transitions. In fact, the life cycle of a singleton bean is very similar to that of a stateless bean.

A singleton session bean generates the following life-cycle transitions:

- **Instantiation:** A singleton session bean transitions from the "Does not exist" state to the "Ready" state as part of the instantiation process.
- **Ready:** Unlike other session beans, clients can concurrently access the business methods of a singleton bean.
- **PreDestroy Callback:** There is no specific way for a client to initiate removal of a singleton session bean. Removal is determined by the container as part of the application shutdown process.

A singleton session bean has the following characteristics:

- The container maintains a single EJB instance per JVM. Typically, applications utilize a single JVM; however, in a clustered environment; there can be multiple singleton instances.
- The bean can belong to multiple clients simultaneously.
- Concurrent access is controlled by using Container Managed Concurrency by default.

Unlike stateless session beans, a singleton session bean maintains state across method calls. Additionally, state is shared by all clients because all clients access the same bean instance.

The methods of a singleton session bean can be concurrently accessed by multiple clients. By default, a singleton session bean's concurrent access is controlled by the container by using Container Managed Concurrency. Container Managed Concurrency supports a multiple reader, single writer locking strategy. Every method of a singleton session bean attempts to obtain a Write lock by default. Methods can be annotated with either of the following:

- `@javax.ejb.Lock(javax.ejb.LockType.WRITE)`
 - `@javax.ejb.Lock(javax.ejb.LockType.READ)`
- `@javax.ejb.AccessTimeout` annotation can set timeout to:
- A specific period of time
 - Or -1 in this case caller would be allowed to wait as long as it takes for the bean instance to become available after lock is released
 - Or 0 in this case caller is instructed not to wait for the lock to be released, so the caller would receive an error if the bean instance is currently locked by another caller
 - Lock and AccessTimeout annotations can also be used with Stateful Session beans to control access to the bean from the client that can make simultaneous calls to the bean.

Alternatively to the default concurrency management method, which is Container Managed, developers may also use Bean Managed Concurrency. Singletons that use bean-managed concurrency allow full concurrent access to all the business and timeout methods in the singleton. The developer of the singleton is responsible for ensuring that the state of the singleton is synchronized across all clients. Developers who create singletons with bean-managed concurrency are allowed to use the Java programming language synchronization primitives, such as synchronization and volatile, to prevent errors during concurrent access.

A `@DependsOn` annotation can be used to control the order in which Singleton beans are loaded.

Singleton, un tercer tipo de bean de sesión. Simplemente asignando una instancia del bean compartida entre todos los clientes. Los métodos `@PostConstruct` y `@PreDestroy` funcionan igual. Una vez más, no hay diferencia allí. La diferencia, supongo, es que el frijol solo se inicia una vez.

Ahora, ¿qué pasa si dos clientes llaman simultáneamente a ese singleton? De forma predeterminada, solo uno de estos clientes logrará realizar la llamada. Y una segunda persona que llama tendrá que esperar a que el primer cliente complete la llamada. Porque la política de bloqueo predeterminada en el singleton es un bloqueo de escritura. Eso es un valor predeterminado. Entonces singleton asume que, potencialmente, el cliente simultáneo que accede a la misma instancia puede hacerlo de una manera insegura. Y pueden corromper cierta información si al mismo tiempo intentan actualizar el bean.

Si dos clientes simultáneos leen información del bean pero no intentan escribirle nada, entonces es bastante seguro. Pueden leer simultáneamente todo lo que quieran. Entonces, si tiene una operación en un bean que sabe que no modifica ningún dato, solo lo está leyendo, puede marcarlo con un bloqueo de lectura. Y eso permitirá que los clientes simultáneos llamen simultáneamente a esa operación en una misma instancia de una manera segura para subprocesos.

Pero nadie más que tú sabe qué operación es esa. Entonces, el singleton, por defecto, implica que todos los métodos deben usar la anotación correcta. Si no coloca nada, ese es el tipo de bloqueo de escritura, que se asume de manera predeterminada solo para asegurarse de que sea seguro para subprocesos. Y luego, si cree que hay ciertas operaciones que solo están leyendo información, márkuelas como leídas, y eso permitirá que las personas que llaman concurrentes progresen con llamadas simultáneas a estas operaciones. A menos, por supuesto, que haya una llamada simultánea que llame a algún otro método que tenga un bloqueo de escritura. Eso evitará que todos accedan a esa instancia en ese momento.

Ahora las personas que llaman esperarán a que se libere el bloqueo. ¿Cuánto tiempo? Eso se controla con una anotación llamada tiempo de espera de acceso. De forma predeterminada, el tiempo de espera de acceso es indefinido. Es menos uno. Puede medirlo en un intervalo de tiempo específico, como en este caso, cinco segundos. Es decir, si hay varias personas que llaman accediendo simultáneamente al singleton, entonces ninguna persona que llama esperará más de cinco segundos. Y si ese es el caso, si alguien está en cola para obtener acceso a esa instancia por más tiempo, solo recibirá una excepción. Si logran obtener acceso a ese bean para que otras personas liberen

sus bloqueos de escritura, entonces tendrán éxito. Depende de si las otras personas que llaman logran liberar los bloqueos de escritura dentro de ese intervalo de cinco segundos o no.

Adivina qué sucede si lo pones a cero. Presumiblemente, deshabilitará las llamadas simultáneas para que nunca esperen, ¿verdad? Entonces, si dos personas llaman simultáneamente a ese singleton con el bloqueo de escritura, uno de ellos ingresará directamente. Pero el recuerdo predeterminado es una espera indefinida, menos uno.

Una cosa más sobre los solteros. Le permiten comenzar antes de que alguien los llame. Y para hacer eso, puede inicializar ansiosamente esta instancia de bean singleton colocando una anotación de inicio aquí. Eso le permite decirle al contenedor que, a medida que se inicia la aplicación que contiene ese código, debe cargar inmediatamente el singleton en la memoria. De lo contrario, el bean se usa cuando se inyecta por primera vez en algún lugar, por lo que cuando algún otro componente usa el bean. Ahí es cuando se cargará.

Asynchronous EJB operations

Entire EJB or specific operations can be marked with the asynchronous annotation.

- While asynchronous operation is executing its logic, the caller can perform other actions.
- The Future object presents results returned by the asynchronous method.

Asynchronous Invoker

```
@EJB
private OrderManagement om;
...
om.reserveStock(order);
...
Future<Invoice> future = om.getInvoice(order);
...
try {
    Invoice invoice = future.get();
} catch (InterruptedException |
        ExecutionException ex) {...}
```

Asynchronous EJB

```
@Stateless
@Asynchronous
public class OrderManagement {
    public void reserveStock(Order order) {
        ...
    }
    public Future<Invoice> getInvoice(Order order) {
        Invoice result = ...
        return new AsyncResult<Invoice>(result);
    }
}
```

In the case of an asynchronous method with a void return type, nothing is returned to the client. However, when an asynchronous method designates a Future<V> return type, the method must create an instance of the Future interface, and return the instance to the client. Future Object is a placeholder for the result to be produced by the asynchronous method. As part of EJB 3.1, a utility class known as AsyncResult can be used to represent a Future. The javax.ejb.AsyncResult<V> class is a concrete implementation of the Future<V> interface that is provided as a helper class for returning asynchronous results. Future object defines following operations:

- `cancel(boolean mayInterruptIfRunning)`: Attempts to cancel execution of this task
- `get()`: Waits if necessary for the computation to complete, and then retrieves its result
- `get(long timeout, TimeUnit unit)`: Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available
- `isCancelled()`: Returns true if this task was cancelled before it completed normally
- `isDone()`: Returns true if this task is completed

Enterprise JavaBeans se puede invocar de forma asíncrona. En este ejemplo en particular, estamos viendo una operación asíncrona en Enterprise JavaBean, que está habilitado con anotación asíncrona. Así que estamos diciendo que este es el Enterprise JavaBean capaz de ejecutar código asíncrono. Podría ser un método específico, podría ser el frijol completo, en este caso particular, el frijol completo.

Hay dos casos de llamadas a métodos asíncronos. El primer caso es cuando el método es nulo, no necesita devolver nada. Bueno, en cuyo caso el invocador simplemente llama al método y cuando el método ejecuta lo que sea que esté haciendo, ese código se ejecuta. El invocador avanza simultáneamente al siguiente fragmento de código. Así que estos fragmentos de código se ejecutan en paralelo. El invocador no está esperando que regrese el método de stock de reserva. Es disparar y olvidar. El invocador lo llama e inmediatamente pasa a la siguiente línea de código.

Si el método devuelve el valor, debe devolver un objeto que se conoce como futuro. Entonces, hay una clase de implementación para el futuro, y ese es un resultado asíncrono. es un envoltorio. La forma en que funciona es cuando llamas a ese método, cuando llamas a ese método, lo recuperas de esa llamada, ese envoltorio, ese objeto futuro. Luego, el método hace lo que hace y su código hace lo que hace como invocador. Estos latidos se ejecutan en paralelo.

El contenido dentro de un objeto futuro está inicialmente vacío. No hay nada dentro. Pero en algún momento, el método llegará al punto en que realmente forme el resultado. Y cuando forma el resultado, crea el contenido real, el resultado, dentro del objeto futuro para que pueda recuperarlo en un momento posterior. Si llama al punto get futuro en el momento en que el bean aún no ha producido el resultado, su método get simplemente estará esperando. Esperará sincrónicamente a que el método de obtención de factura genere su factura.

Pero si llama al método get, después de un tiempo, no inmediatamente, pero está haciendo otras cosas y luego llama al método get sobre ese objeto futuro, le da la oportunidad al método get de factura para crear esa factura y eventualmente devolverla para ti. Ahí vas. Cuando llama al método get y un futuro que contiene el resultado, simplemente obtiene el resultado. Usted llama a un método get antes de que el objeto futuro se haya llenado con el resultado, oh, bueno, solo tendrá que esperar a que se produzca este resultado. Pero ciertamente da una oportunidad para que el invocador y el bean hagan cosas en paralelo.

Java Transaction API

Transactions are ACID:

- **Atomic:** Transaction must be done, or undone completely.
- **Consistent:** Transaction brings system from one consistent state to another.
- **Isolated:** Transaction state is not visible to components outside of this transaction
- **Durable:** Once transaction completes, all changes become permanent.

JTA Transaction Implementations:

- **Programmatic - Bean Managed Transactions (BMT)**
 - Available for all Java Components Types
 - Not recommended for EJBs and CDI Beans
- **Declarative - Container Managed Transactions (CMT)**
 - Available and recommended for EJBs and CDI Beans

Various objects can participate in transactions.

All actions within the transaction have to succeed or must be undone.

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The Java EE application developer's concern is with scoping transactions. Scoping can be programmatic (bean-managed transactions) or declarative (container-managed transactions). Scoping refers to the selection of operations that the developer wants to group into a transaction. An alternative term is transaction boundary demarcation. The developer is never concerned with the technicalities of transaction coordination. That is, matters such as resource enlistment and log management are the responsibility of the application server.

For some component types, the Java EE application developer can decide whether to use programmatic transaction scoping or declarative transaction scoping. Some component types allow only one method or the other. Where both methods are allowed, the decision about which method to use operates at the component level. For example, a particular EJB component cannot use both declarative and programmatic transaction scoping.

Note: Java supports only flat transaction model; chained or nested transaction are not supported.

Flat Transaction Model characteristics:

- In a particular thread, only one transaction is in effect at a time.
- This is the only transaction model supported by the Java EE specification and supported by all database vendors.
- Flat transactions also allow for a simple declarative transaction model. More complex transaction models can be simulated in code if necessary.

Otra cosa que debemos retomar de la presentación anterior es la gestión de transacciones. Comenzamos ese tema en un capítulo de la API de persistencia de Java, pero debemos completarlo con una discusión sobre cómo se administran las transacciones en general, no solo con respecto a JPA, sino quizás con otras cosas en Java. Hay dos

enfoques para la gestión de transacciones: las transacciones de gestión de beans, donde escribe el código que comienza y finaliza las transacciones, o las transacciones de gestión de contenedores, o el enfoque declarativo, donde anota su objeto sobre cómo desea que se realicen las transacciones. Pero en realidad no está iniciando y finalizando transacciones mediante programación. Confía en el contenedor para que lo haga por usted.

La transacción tiene algunas propiedades muy importantes. Las conocemos como propiedades ACID. La transacción tiene que ser atómica, lo que significa que todas las cosas que se hacen con una transacción se hacen por completo o se deshacen por completo. La transacción tiene que llevar el sistema de un estado a otro, de un estado consistente a otro estado consistente. Esa es la propiedad consistente. Las cosas que hace dentro de las transacciones no son visibles para el mundo exterior hasta que finaliza la transacción. Entonces eso se conoce como aislamiento. Y, por último, duradero: una vez que está comprometido o una vez que se revierte, una vez que finaliza la transacción, independientemente de cómo termine, los cambios se hacen permanentes. Entonces, realmente no puede deshacer después de retroceder, a menos que haga otra transacción, supongo.

Programmatic Transactions (BMT)

Enabling Bean Managed Transactions:

- Initialize UserTransaction Object within Java EE Container using @Resource injection.
- For the EJB, set the BEAN TransactionManagement property.
- Initialize UserTransaction Object outside of the Java EE Container using JNDI lookup.

```
@RequestScoped
public class SomeBean {
    @Resource
    private UserTransaction tx;
    ...
}
```

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class SomeEJB {
    @Resource
    private UserTransaction tx;
    ...
}
```

```
Context ctx = new InitialContext();
UserTransaction t = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

Control transactions using operations:

- `begin()`: Start transaction
- `commit()` and `rollback()`: End transaction

```
try {
    tx.begin();
    ...
    tx.commit();
} catch (Exception e) {
    tx.rollback();
}
```

UserTransaction Object is used to control transactions in Java. It has the following operations:

- `begin()`: Create a new transaction and associate it with the current thread.
- `commit()`: Complete the transaction associated with the current thread.
- `getStatus()`: Obtain the status of the transaction associated with the current thread.
- `rollback()`: Roll back the transaction associated with the current thread.
- `setRollbackOnly()`: Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction, even if `commit` is invoked.
- `setTransactionTimeout(int seconds)`: Modify the timeout value that is associated with the transactions started by the current thread with the `begin()` method.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Enfoque programático... bueno, eso depende, ya ves. Lo que estamos haciendo en este PowerPoint, tenemos dos ejemplos. Tenemos un bean CDI en el lado izquierdo y un Enterprise JavaBean en el lado derecho. El CDI, de forma predeterminada, utiliza transacciones gestionadas por beans. Y un Enterprise JavaBean, de forma predeterminada, utiliza transacciones administradas por contenedor. Por lo tanto, si desea que sea bean, debe cambiarlo, bean de gestión de transacciones. Si decide que desea realizar transacciones administradas por beans, controle las transacciones mediante programación, inyecte el objeto de transacción del usuario. Oh, si lo está haciendo desde fuera del contenedor Enterprise JavaBeans, como en Java SE, en realidad, puede buscar el objeto de transacción para usar JNDI. Ahí vas.

Ahora, ¿qué haces con un objeto de transacción de usuario? Comience a revertir la confirmación. Es tan simple como eso. Comience la transacción y luego finalícela con una confirmación o una reversión. Eso es todo. Es bastante fácil.

Declarative Transactions (CMT)

Enabling Container Managed Transactions:

- EJB TransactionManagement property is CONTAINER by default.
- CDI Beans use Transactional Annotation.
- Demarcate Transactions boundaries using Transactional Attribute Annotations (default is REQUIRED).

To tell container to rollback:

- EJBs call the setRollbackOnly operation
- CDI Beans throw exceptions (see notes)

```
@RequestScoped
@Transactional
public class SomeBean {
    @Transactional(value = Transactional.TxType.REQUIRED,
        rollbackOn = {list of exceptions},
        dontRollbackOn = {list of exceptions})
    public void someMethod() {
        ...
        // just throw exception to rollback
    }
}
```

```
@Stateless
@TransactionManagement(CONTAINER)
public class SomeEJB {
    @Resource
    private SessionContext ctx;
    @TransactionalAttribute
    @TransactionalAttributeType(REQUIRED)
    public void someMethod() {
        try {
            ...
        } catch (Exception e) {
            ctx.setRollbackOnly();
        }
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By default, EJBs use CONTAINER Transactional Management.

EJB Declarative Transactions use the TransactionalAttribute annotation.

The default value for the Stateless EJB is REQUIRED and for Stateful it is REQUIRES_NEW.

CDI Beans Declarative Transactions use javax.transaction.Transactional annotation. It provides application the ability to declaratively control transaction boundaries on CDI-managed beans, as well as classes defined as managed beans by the Java EE specification, at both the class and method level where method level annotations override those at the class level.

TxType.REQUIRED is the default.

By default, checked exceptions do not result in the transactional interceptor marking the transaction for rollback and instances of RuntimeException and its subclasses do. This default behavior can be modified by specifying exceptions that result in the interceptor marking the transaction for rollback and/or exceptions that do not result in rollback.

The rollbackOn element can be set to indicate exceptions that must cause the interceptor to mark the transaction for rollback.

Conversely, the dontRollbackOn element can be set to indicate exceptions that must not cause the interceptor to mark the transaction for rollback.

When a class is specified for either of these elements, the designated behavior applies to subclasses of that class as well. If both elements are specified, dontRollbackOn takes precedence.

When constructing a custom exception class you may use @ApplicationException annotation to specify that this exception should be reported to the client directly (unwrapped) and if it should cause rollback or not.

Java EE 7 also introduced a new CDI scope, @TransactionScoped. This scope allows this object to participate in a transaction propagated across stateless session calls.

Ahora transacciones gestionadas por contenedores. Viceversa. Enterprise JavaBeans, el valor predeterminado es administrado por contenedor, por lo que realmente no necesita decir eso. Ese es el valor predeterminado de todos modos. Pero los beans CDI, en realidad puedes hacerlos transaccionales. Solo tengo que anotarlo. Por lo tanto, el valor predeterminado para ellos es administrado por beans, pero se pueden hacer administrados por contenedores.

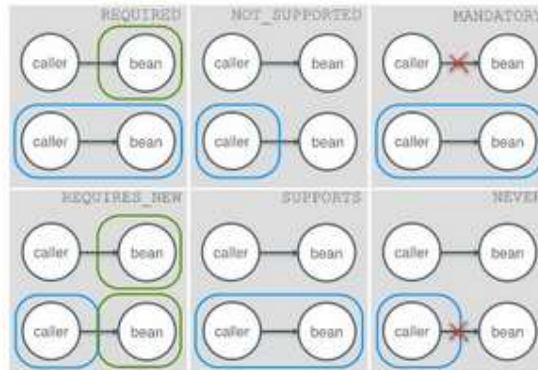
Con las transacciones administradas por contenedor, no llama a los métodos de inicio, compromiso o reversión. Uh-uh. El contenedor lo hace por ti. Si todo va bien, y su código se ejecuta y todo está bien, el contenedor se confirma. Si desea que un contenedor retroceda, puede llamar al conjunto de métodos rollback solamente. Así que es un-- no puedes llamar a la reversión. Pero puede pedirle al contenedor que retroceda. Por lo tanto, establecer la reversión solo le dice al contenedor que prefiere retroceder, y el contenedor lo hará por usted.

Alternativamente, en los beans CDI, en realidad puede designar una lista de excepciones en las que desea realizar la reversión. Puede designar que ciertas excepciones no deben causar reversión y ciertas otras excepciones sí. Ahí vas. Esa es la idea. Entonces simplemente lanza la excepción para hacer una reversión en ese escenario. Supongo que podría hacer lo mismo con un Enterprise JavaBean, pero luego tienen este conveniente método de solo reversión, así que lo que encuentre más conveniente, supongo.

Demarcate Transactional Attributes

Depending on the Transactional Attribute value, bean methods can:

- Join transaction of invoker
- Start new transaction
- Run with no transactional context
- Throw exception



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- **REQUIRED:** The method becomes part of the caller's transaction. If the caller does not have a transaction, the method runs in its own transaction.
- **REQUIRES_NEW:** The method always runs in its own transaction. Any existing transaction is suspended.
- **NOT_SUPPORTED:** The method never runs in a transaction. Any existing transaction is suspended.
- **SUPPORTS:** The method becomes part of the caller's transaction if there is one. If the caller does not have a transaction, the method does not run in a transaction.
- **MANDATORY:** It is an error to call this method outside of a transaction.
- **NEVER:** It is an error to call this method in a transaction.

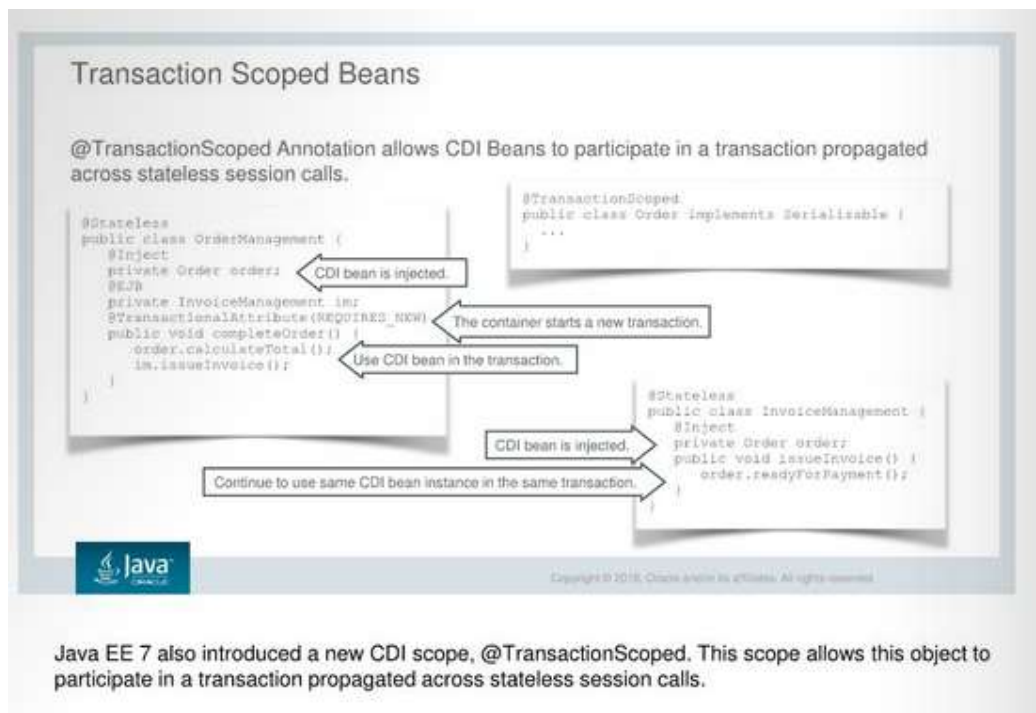
Lo notó en la página anterior: no comenté allí, pero se colocaron algunos atributos transaccionales en la página. Entonces estamos marcando los beans y sus materiales con estos atributos transaccionales. ¿Qué quieren decir? Bueno, en realidad es un requisito predeterminado para Enterprise JavaBean. Ese es el valor predeterminado para Enterprise JavaBean. Si no marca con nada, eso es un sellado, es la anotación.

Si la persona que llama llama a su bean, invoca el bean, el método y el bean que tiene un requisito, que es la propiedad transaccional predeterminada, y la persona que llama no tiene una transacción, la persona que llama aún no ha iniciado ninguna transacción, eso es multa. El bean iniciará una transacción. Si la persona que llama tiene la transacción antes de realizar la llamada al bean, el bean se unirá a la transacción o a la persona que llama. Entonces eso es requerido.

Echemos un vistazo a la siguiente, requiere nuevo. Similar a requerido, pero un bean iniciará una nueva transacción de cualquier manera, incluso si la persona que llama tiene una transacción en curso. Echemos un vistazo a la siguiente, no compatible. El bean se negará a unirse a la transacción de la persona que llama si existe. Simplemente ignóralo. El siguiente, soportes. El bean se unirá a la transacción de la persona que llama si existe, o se ejecutará sin contexto transaccional si ese es el caso, si la persona que llama lo llamó sin transacción.

Obligatorio. El bean se unirá a la transacción de la persona que llama, si la hay, o generará una excepción si la persona que llama llama al bean sin establecer un contexto transaccional antes. Y finalmente, nunca. Contrario de obligatorio. El bean lanzará una excepción si la persona que llama tiene el contexto transaccional, y solo aceptará ejecutar sin ninguna transacción actual en curso de la persona que llama. Esa es la opción de nunca.

Por lo tanto, puede convertir todo el bean o un método particular en bean con estos atributos transaccionales. Como ha visto, se puede hacer tanto para CDI como para Enterprise JavaBeans. Y simplemente le dice declarativamente al contenedor lo que desea hacer, cómo desea iniciar transacciones o unirse a las existentes y sus circunstancias.



Otra característica interesante: ¿recuerdas? Hablamos sobre diferentes alcances de beans CDI. Y les dije que en esa etapa hay más alcances de los que puedo cubrir humanamente porque pueden crearlos personalizados. De todos modos, bueno, hay un alcance que podría interesarle. Se llama bean de código sin transacciones. Mm.

Si crea un bean que tiene un alcance transaccional, esto es lo que sucederá. Inyectas este frijol. Digamos que lo inyecta aquí, este pedido, Inyéctelo en la gestión de pedidos. La gestión de pedidos se anota con una anotación sin estado, lo que significa que es un bean de sesión sin estado, ¿no? ESTÁ BIEN. ¿Cuál es el atributo transaccional predeterminado del bean de sesión sin estado? Requiere. Bueno, hacemos que este método en particular requiera nuevo, por lo que definitivamente comienza una nueva transacción. Bien. Entonces, quien llame al pedido completo en realidad comenzará una nueva transacción. Hermoso.

Ahora, ¿qué más sucede? Tenemos otro bean, también un Enterprise JavaBean, otro bean sin estado, gestión de facturas. Y de hecho inyectamos el bean CDI llamado ámbito de transacción, ese bean de orden, también lo inyectamos allí. Vea lo que sucede a continuación. Quien llama al método de pedido completo inicia una nueva transacción. El orden completo opera, hace algo con el orden. OK, estaba haciendo algo allí. Y luego llama a este método llamado emitir factura.

¿Cuál es el atributo transaccional del método de emisión de factura? No anotado, lo que significa que es obligatorio. Se une a la transacción de la persona que llama. En otras palabras, está en la misma transacción que el método de pedido completo. ESTÁ BIEN. Cuando llame a este método listo para el pago, se garantiza que se invocará en la misma instancia del pedido que la instancia que estaba aquí, y calculará el total. Porque el pedido tenía un alcance transaccional.

Si esa gestión de facturas será en otra transacción, será otra instancia de pedido. Pero está en la misma transacción, por lo que es la misma instancia de pedido. ¿Pasé una orden como argumento? No, solo uso una anotación. Enfoque interesante. ¿no es así? Le está pidiendo al contenedor que averigüe qué instancia del bean de orden inyectar en estos dos Enterprise JavaBeans. Y el contenedor lo resuelve por usted, basándose únicamente en el hecho de que se encuentra en la misma transacción. Ta-da. De hecho, creo que es un muy buen ejemplo del uso del bean CDI fuera del contenedor web. Es puramente un ejemplo de Enterprise JavaBeans en este caso.

4.2.Implementing Business Logic by Using EJBs: Timers

Timers

Purpose of timers is to execute code at some point in time or periodically.

- Timers can be applied to Stateless, Singleton, Message-Driven, and 2.1 Entity Beans.
- By default timers are stateful, so they are redelivered after server restart.
- Automatic timers can be configured with annotations or deployment descriptors.

Timer types:

- Programmatic
 - Created using TimerService object
 - When such timer expires (goes off), the container calls single method annotated @Timeout in the bean's implementation class.
- Automatic
 - Created with multiple @Schedule or @Schedules

Timeout methods:


- Must not return values (be declared with void return type)
- Optionally take Timer object as the only parameter.


```
@Resource
private TimerService timerService;
public void someMethod() {
    timerService.createTimer(...);
}

@Timeout
public void doThings(Timer timer) {...}

@Schedule(dayOfWeek="Sun", hour="0")
public void doThings() {...}

@Schedule(minute="*/15", hour="9-17")
public void doMoreThings() {...}
```





Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

The EJB timer service is a container-managed service that provides a way to allow methods to be invoked at specific times or time intervals.

The EJB timer may be set to invoke certain EJB methods at a specific time, after a specific duration, or at specific recurring intervals.

Timers can be created programmatically by interacting with TimerService or automatically through annotations.

These Timer tasks apply to the following types of enterprise beans:

- Stateless and singleton session beans
- Message-driven beans
- 2.1 entity beans, except calendar-based timer and nonpersistent timer functionality that are not supported for 2.1 Entity beans.

Timers are intended for long-lived business processes and are by default persistent.

Types of timers:

- Programmatic timers are defined:
 - By calling one of the timer creation methods of the TimerService interface
 - When Timer object expires. EJB container triggers an EJB method that is marked with Timeout annotation.
- Automatic timers are created on successful deployment of an enterprise bean that contains methods that are annotated with the java.ejb.Schedule or java.ejb.Schedules annotations, or configured with ejb-jar.xml.

In the event of the application server shutdown, timers behave differently, depending on their persistency type:

- Persistent Timer (default)
 - A timer is guaranteed to survive application server crashes and shutdowns. Timers that expire during application server shutdowns are redelivered on server restart. In most cases, the application server, on power up, notifies the enterprise bean of all expired timers.
- Nonpersistent Timer
 - Nonpersistent timers have a life span that is tied to the life span of the JVM in which it is created. However, they are canceled upon server shutdown, crash, and so on. They can be created again programmatically or automatically.

Ahora hablemos de otra característica del contenedor Enterprise Java Beans, y son los temporizadores. Puede adjuntar temporizadores a diferentes tipos de frijoles. Podrías hacerlo con frijoles sin estado. Puedes hacerlo con singletons. De hecho, puede hacerlo con beans controlados por mensajes. Bueno, eso es por razones de compatibilidad con versiones anteriores. Los antiguos beans de entidad 2.1, pero ya no los usamos.

Entonces, los temporizadores le permiten ejecutar su código en función de un tiempo, un cronograma, un punto particular en el tiempo. Podría ser un evento recurrente. Podría ser un evento único. Entonces, un cierto punto o

puntos en el tiempo en los que desea ejecutar su código. Los temporizadores por defecto tienen estado, pero puede hacerlos sin estado. Temporizador con estado significa que si su servidor se cierra y luego se reinicia, se utiliza el temporizador. Digamos que en el último temporizador, su servidor se apaga. El temporizador se pierde. Así que no los estás reanudando. El valor predeterminado es con estado, pero puede jugar con los temporizadores y hacerlos de la forma que desee.

Ahora, hay dos formas de configurar los temporizadores. Puede configurar un temporizador con un método llamado tiempo de espera. Marque un método en su bean con tiempo de espera de anotación, solo uno. Luego inyecte `TimerService`, cree el temporizador. Bueno, aquí especificaremos cuándo va a sonar. Hay más. Hay parámetros allí. Los discutiremos en un momento.

Entonces podríamos configurar propiedades específicas cuando el temporizador debería sonar. Y luego tiene este método de tiempo de espera que se ejecutará cuando expire ese temporizador. Esto se llama un temporizador programático. Hay otro tipo de temporizador que se llama temporizador automático. Si voy a ser una persona que formularía el estándar de temporizadores, nunca lo llamaría automático. Lo llamaré, como en cualquier otra API, declarativo.

Siempre tienes este programático versus declarativo. Pero por algunas razones misteriosas, el estándar que describe los temporizadores, en lugar de la palabra "declarativo" usa la palabra "automático". no me preguntes Yo tampoco puedo comprenderlo. De todos modos, básicamente estás creando un temporizador usando anotaciones de programación.

Solo está definiendo contra un método particular o una cantidad de métodos en el bean. ¿Cuándo desea que se ejecuten estos métodos? Así que construye el horario con diferentes expresiones. Supongo que este temporizador automático o declarativo es probablemente nuestro enfoque más flexible.

En primer lugar, porque no está restringido a un solo bean final de método. Puede tener múltiples métodos allí. Y luego puedes construir todo tipo de horarios. Y es muy fácil de hacer a través de la anotación. Pero si desea crear temporizadores programáticos, seguro que puede hacerlo. Así que supongo que es cuando quieres configurar dinámicamente ese objeto de temporizador. Ese es el beneficio de eso. Desafortunadamente, estará restringido a un solo método de tiempo de espera en ese caso.

Los métodos de temporizador deben ser nulos. No debe devolver valores. Pueden tomar el objeto `Timer` como argumento, pero en realidad es opcional. No tienes que hacerlo. Tu decides. Si desea un mayor control del temporizador, acepte el objeto `Timer` como argumento. Pero entonces de nuevo.

Calendar-Based Timer Expressions

Timer expressions can be used within:

- @Schedule annotation
- ScheduleExpression Object
- The `ejb-jar.xml` file

Attribute	Allowed Values	Default	Examples
second	0 to 59	0	second="30"
minute	0 to 59	0	minute="15"
hour	0 to 23	0	hour="13"
dayOfWeek	0 to 7 (both 0 and 7 refer to Sunday) Sun, Mon, Tue, Wed, Thu, Fri, Sat	*	dayOfWeek="3" dayOfWeek="Mon"
dayOfMonth	1 to 31 -7 to -1 (a negative number means the nth day or days before the end of the month) Last [1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat]	*	dayOfMonth="15" dayOfMonth="-3" dayOfMonth="Last" dayOfMonth="2nd Fri"
month	1 to 12 Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	*	month="7" month="July"
year	A four-digit calendar year	*	year="2010"



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Setting an attribute to an asterisk symbol (*) - wildcard, represents all allowable values for the attribute.

- The following expression represents every minute: `minute=""`
- The following expression represents every day of the week: `dayOfWeek=""`

To specify two or more values for an attribute, use a comma (,) to separate the values. A range of values is allowed as part of a list. Wildcards and intervals, however, are not allowed.

Duplicates within a list are ignored.

- The following expression sets the day of the week to Tuesday and Thursday:
`dayOfWeek="Tue, Thu"`
- The following expression represents 4:00 a.m., every hour from 9:00 a.m. to 5:00 p.m. using a range, and 10:00 p.m.: `hour="4,9-17,22"`

Use a dash character (-) to specify an inclusive range of values for an attribute. Members of a range cannot be wildcards, lists, or intervals. A range of the form `x-x`, is equivalent to the single-valued expression `x`. A range of the form `x-y` where `x` is greater than `y` is equivalent to the expression `x-maximum value, minimum value-y`. That is, the expression begins at `x`, rolls over to the beginning of the allowable values, and continues up to `y`.

- The following expression represents 9:00 a.m. to 5:00 p.m.: `hour="9-17"`
- The following expression represents Friday through Monday: `dayOfWeek="5-1"`
- The following expression represents the 25th day of the month to the end of the month, and the beginning of the month to the fifth day of the month: `dayOfMonth="25-5"`
- It is equivalent to the following expression: `dayOfMonth="25-Last,1-5"`

The forward slash (/) constrains an attribute to a starting point and an interval and is used to specify every N seconds, minutes, or hours within the minute, hour, or day. For an expression of the form `x/y`, `x` represents the starting point and `y` represents the interval. The wildcard character may be used in the `x` position of an interval and is equivalent to setting `x` to 0.

- Intervals may be set only for second, minute, and hour attributes.
- The following expression represents every 10 minutes within the hour: `minute=""/10"`
- It is equivalent to: `minute="0,10,20,30,40,50"`
- The following expression represents every 2 hours starting at noon: `hour="12/2"`

Expresiones. Se pueden usar en la creación del temporizador programático. Se pueden usar en la anotación del cronograma. Podrían usarse, en realidad, en EJB JAR XML como parte de su descriptor de implementación. Y hay muchos ejemplos aquí. Podría haber puntos específicos en el tiempo. Podrían ser días de la semana. Podría haber días del mes.

No sé. Digamos menos siete. Numero negativo. ¿Qué significa eso? Bueno, significa siete días desde el final del mes. Así que no estás contando desde el principio sino desde el final. De todos modos, hay bastantes anotaciones diferentes.

Este es el valor predeterminado, por cierto. Si no especifica un segundo, minuto u hora en particular, cuál sería por defecto. Y también hay comodines. Si usa un comodín, eso básicamente significa que, por ejemplo, si dice 15 minutos, eso significa ejecutarlo 15 minutos después de la 1:00, por ejemplo. Si dice estrella 15, significa ejecutar cada 15 minutos. Así que eso es un evento repetitivo en ese caso. Entonces se ejecutará, digamos, a las 15 y 1:00, y luego a la 1 y media, y a las 45 y 1:00, etcétera, etcétera. Y posiblemente en la próxima hora, dependiendo de lo que diga la hora.

Define Programmatic Timers

Programmatic Timers allow developers to create Timer Objects dynamically.

- **TimerService** object controls creation of timers, with various methods, such as:

- createCalendarTimer
- createIntervalTimer
- createSingleActionTimer
- createTimer

- Time may be triggered as a one-off or periodic event, or both.
- Timers may use Schedule Expressions.
- **Timeout** annotation designated the method that timer should trigger.
- Timer persistency can be switch off:
 - TimerConfig.setPersistent(false)

```
@Stateless
public class SomeBean {
    @Resource
    private TimerService timerService;

    @PostConstruct
    public void init() {
        ScheduleExpression e = new ScheduleExpression();
        e.second("30").minute("*/10").hour("*/");
        timerService.createCalendarTimer(e);
    }

    public void createSingleTimer(Date date) {
        timerService.createSingleActionTimer(date,
            new TimerConfig());
    }

    @Timeout
    public void someMethod(Timer timer) { ... }
}
```



Copyright © 2018. Oracle and/or its affiliates. All rights reserved.

TimerService can be obtained from EJB SessionContext object:

```
@Stateless
public class HelloWorldBean implements HelloWorldRemote {
    @Resource
    private SessionContext sessionContext;
    public void doThings() {
        TimerService timerService = sessionContext.getTimerService();
        timerService.createTimer(...);
        ...
    }
}
```

or via direct resource injection of the TimerService object:

```
@Stateless
public class HelloWorldBean implements HelloWorldRemote {
    @Resource
    private TimerService timerService;
    public void doThings() {
        timerService.createTimer(...);
        ...
    }
}
```

EJB Timer Service allows stateless session beans, singleton session beans, message-driven beans to be registered for timer callback events at a specified time, after a specified elapsed time, after a specified interval, or according to a calendar-based schedule with the following operations:

`createCalendarTimer(ScheduleExpression schedule)`: Create a calendar-based timer based on the input schedule expression.

`createCalendarTimer(ScheduleExpression schedule, TimerConfig timerConfig)`: Create a calendar-based timer based on the input schedule expression.

`createIntervalTimer(Date initialExpiration, long intervalDuration, TimerConfig timerConfig)`: Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

`createIntervalTimer(long initialDuration, long intervalDuration, TimerConfig timerConfig)`: Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.

`createSingleActionTimer(Date expiration, TimerConfig timerConfig)`: Create a single-action timer that expires at a given point in time.

`createSingleActionTimer(long duration, TimerConfig timerConfig)`: Create a single-action timer that expires after a specified duration.

`createTimer(Date initialExpiration, long intervalDuration, Serializable info)`: Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

`createTimer(Date expiration, Serializable info)`: Create a single-action timer that expires at a given point in time.

`createTimer(long initialDuration, long intervalDuration, Serializable info)`: Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.

`createTimer(long duration, Serializable info)`: Create a single-action timer that expires after a specified duration.

`getAllTimers()`: It returns all active timers associated with the beans in the same module in which the caller bean is packaged.

`getTimers()`: It returns all active timers associated with this bean.

Timer Identification

If you associate multiple timers with an enterprise bean, you must also apply a strategy to identify the timer that issues the notification. You can use the `info` parameter of the `createTimer` method to develop your identification and response strategy.

Cómo construir un temporizador programático. Entonces, hay todo tipo de diferentes tipos de temporizadores que puede crear. Calendarios, intervalos, acciones individuales. Hay un método genérico `createTimer`. Así que lo construyes con una expresión de programación. Ahí está. Y solo construye esto. En este caso particular, dice cada 30 segundos cada 10 minutos de cada hora. Por lo tanto, el fragmento de código se ejecutará periódicamente el 30 segundo de cada 10 minutos.

Y luego creas un temporizador. Y debido a que es un temporizador programático, se invocará el método que está anotado con la anotación de tiempo de espera. Si desea controlar el tiempo de persistencia, ese método `setPersistent true/false` podría usarse para desactivar la persistencia si es necesario.

Define Automatic Timers

Automatic Timers allow developers to create a number of Schedules to trigger different operations upon timer expiration.

- EJB Container creates automatic timers upon successful deployment of an enterprise bean that contains one or more method that is annotated with the `java.ejb.Schedule` or `java.ejb.Schedules` annotations, or describes automatic timers in the `ejb-jar.xml` file.
- An `info` element can contain any serializable information that you want to associate with this timer.
- Timer persistency can be switched off using the `persistent=false` attribute.

```
@Singleton
public class SomeBean {
    @Schedules({
        @Schedule(hour="*", minute="*", second="*/10", persistent=false),
        @Schedule(hour="*", minute="*", second="*/45")
    })
    public void doThings() {...}

    @Schedule(minute="*/15", hour="9-17", info="Something")
    public void doMoreThings() {...}
}
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Alternatively, similar functionality to that one described on the page above, can be achieved by defining timer using the `ejb-jar.xml` file:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SomeBean</ejb-name>
      <ejb-class>demos.SomeBean</ejb-class>
      <session-type>Stateless</session-type>
      <timer>
        <schedule>
          <second>\*30</second>
          <minute>\*.10</minute>
          <hour>\*</hour>
          <month>\*</month>
          <year>\*</year>
        </schedule>
        <timeout-method>
          <method-name>someMethod</method-name>
          <method-params>
            <method-param>javax.ejb.Timer</method-param>
          </method-params>
        </timeout-method>
      </timer>
    </session>
  </enterprise-beans>
</ejb-jar>
```

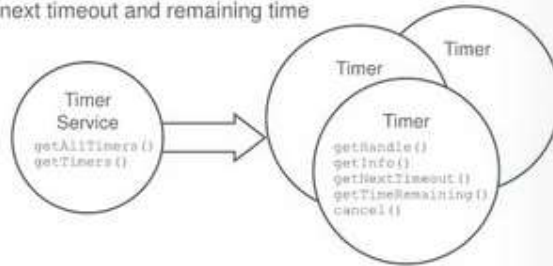
Temporizador automático o declarativo. Eso se hace con una anotación de programación. Puede crear varios horarios de una sola vez. Así que horarios. En realidad, también puedes hacerlos transitorios. Persistente es igual a falso. Entonces eso apaga la persistencia de un temporizador en particular. En este caso particular, ¿qué está diciendo? Cada hora, cada minuto, cada 10 segundos. Otro, cada 45 segundos.

¿Que tal este? Cada 15 minutos entre las 9 de la mañana y las 5 de la tarde. Ahí tienes Info es cualquier objeto serializable. es opcional Si desea asociar cualquier objeto serializado con un temporizador, puede poner lo que quiera allí.

Manage Timers

TimerService object can be used to retrieve active Timer objects. Each Timer object can be used to:

- Obtain its handler
- Obtain an info object associated with this timer
- Obtain information about its next timeout and remaining time
- Cancel Timer



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can retrieve active timers from the `TimerService` object:

- `getAllTimers`: Returns all active timers associated with any bean in the same ejb-module as the caller bean
- `getTimers`: Returns all active timers associated with this bean

Each `Timer` object allows:

- `getHandle`: Returns a serializable handle to the timer object. By persisting this handle, you can obtain a reference to the timer object at a future date.
- `getInfo`: Returns the info object associated with the timer during timer creation. The info object can contain any kind of information as long as it is serializable.
- `getNextTimeout`: Returns a date object that represents the absolute time until the next scheduled timer expiration
- `getTimeRemaining`: Returns a long value that represents the time in milliseconds until the next scheduled timer expiration
- `cancel`: To cancel a timer, you invoke the `cancel` method on the timer. You can perform this task in any method that contains a valid reference to a timer object. Only the bean that created the timer can invoke the `cancel` method. Invoking the `cancel` method removes the timer from the container. After a timer has been canceled, subsequent method invocations on the timer will generate `NoSuchObjectException`.

Por último, controlar los temporizadores. Servicio de temporizador, que recuerda, solo puedes inyectar. Tiene un método para obtener todos los temporizadores. ¿Cuál es la diferencia entre obtener todos los temporizadores y obtener temporizadores? `Get timers` devuelve los temporizadores que son para este bean. Todos los temporizadores devuelve todos los temporizadores, para todos los beans. De todos modos, puedes iterar con una colección de temporizadores.

Y luego puede obtener un controlador para cada temporizador, obtener información, ese objeto serializable para cada dímerno, saber dónde está el próximo tiempo de espera, obtener el tiempo restante hasta el próximo tiempo de espera o incluso cancelar el temporizador por completo. Por lo tanto, puede cancelarlos programáticamente si lo desea. Ahí tienes

Define Interceptors

Interceptors allows you to separate reusable/repeated code fragments such as logging, auditing and security, from the business logic implementation code contained in CDI or EJB components.

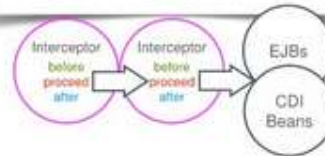
How interceptors are defined:

- A method in the target component class itself
- A separate class that can be applied to a number of CDI and EJB components

How interceptors are applied:

- Caller invokes components as usual.
- Each interceptor applied to this component executes its "before" part.
- Then it passes control to the next interceptor and eventually to the target component.
- Then it executes the "after" part.
- Priority annotation can be used to set interceptor execution order

```
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+1)
public class SomeInterceptor {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx){
        // Perform "Before" Actions
        Object obj = ctx.proceed();
        // Perform "After" Actions
        return obj;
    }
    @AroundInvoke
    public Object method2(InvocationContext ctx){...}
}
```



The `@AroundInvoke` annotation defines an interceptor method that interposes on business methods. May be applied to any nonfinal, nonstatic method with a single parameter of type `InvocationContext` and return type `Object` of the target class (or superclass) or of any interceptor class.

Such operations can be placed in dedicated interceptor classes:

```
@Interceptor
public class SomeInterceptor {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx){
        ...
        Object obj = ctx.proceed();
        ...
        return obj;
    }
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Or be added directly to target component class:

```
@Stateless
public class SomeBean {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx){
        ...
        Object obj = ctx.proceed();
        ...
        return obj;
    }
    ...
}
```

Priorities that define the order in which interceptors are invoked. These values should be used with the Priority annotation.

- Interceptors defined by platform specifications should have priority values in the range PLATFORM_BEFORE up until LIBRARY_BEFORE, or starting at PLATFORM_AFTER.
- Interceptors defined by extension libraries should have priority values in the range LIBRARY_BEFORE up until APPLICATION, or LIBRARY_AFTER up until PLATFORM_AFTER.
- Interceptors defined by applications should have priority values in the range APPLICATION up until LIBRARY_AFTER.

An interceptor that must be invoked before or after another defined interceptor can choose any appropriate value.

Interceptors with smaller priority values are called first. If more than one interceptor has the same priority, the relative order of these interceptors is undefined.

Y la siguiente parte de este capítulo habla de los interceptores. Así que terminamos con los temporizadores. Nos vamos a los interceptores. Interceptor es una pieza de código que le gustaría ejecutar antes, después o alrededor de una llamada a un CDI o un Enterprise Java bean. Interceptor es transparente para el invocador. Una persona que llama está llamando al bean, no al interceptor.

Una persona que llama está llamando al frijol. Un interceptor detrás de escena secuestra la llamada y le permite poner un poco de código antes de llamar a ese bean, un poco de código después de llamar a ese bean. La llamada al bean es controlada por el interceptor con un método simple llamado proceder. Llamar al método de proceder significa que le estás pidiendo al interceptor que vaya al próximo interceptor en la línea y eventualmente vaya al bean. Haz esa progresión.

Lo que ponga antes de llamar al método de proceder en el interceptor será el código que se invoca. Antes de ese punto, cualquier cosa que coloque después del método de proceder será el código que invocó después de que se haya ejecutado el bean. Los interceptores pueden contener código que no desea colocar en un bean específico. Cualquier cosa que desee que sea genérica o que se aplique a múltiples beans, cualquier rutina, fragmentos de código repetidos y reutilizables, no sé, registro, auditoría, aspectos de seguridad. Lo que desee, código que se adapta a múltiples componentes CDI o Enterprise Java beans diferentes.

En realidad, podría estar interceptando llamadas de tiempo de espera, por cierto. ¿Ves eso alrededor del tiempo de espera? Podría hacer eso. Absolutamente.

Por último, estos métodos, como alrededor del tiempo de espera y alrededor de la invocación, puede colocarlos directamente en el bean. Puedes. Pero si quiere que sean realmente reutilizables, probablemente quiera ponerlos en una clase separada. Entonces, en este caso, marca esta clase como un interceptor. ¿Cómo se establece la orden de un interceptor?

Alguien llama al bean y luego los interceptores secuestran la llamada. Pero, ¿cuál es el orden en que se invocan? Hay interceptores que no aplicas tú, como ves. Son interceptores que se aplican mediante el contenedor Java EE. Entonces tienen su propio orden, y este orden está controlado por el número de prioridad entero.

Por lo tanto, no interferiría accidentalmente con los interceptores provistos por el contenedor y accidentalmente entraría en su secuencia. Se le da una constante, que se llama prioridad de aplicación, que es básicamente un

número. Pero se garantiza que este número será posterior al número del contenedor. Es más tarde que cualquier interceptor de contenedores.

Así que permites que el contenedor haga lo suyo, y luego dices, OK, bueno, he escrito algunos interceptores adicionales, y luego simplemente los controlas agregando cualquier número a la prioridad de la aplicación. Más 1, más 2, más 3. Eso los ordena. Ahí tienes Entonces, cualquier tipo de código genérico que desee aplicar a una cantidad de beans diferentes, puede secuestrar las llamadas y poner ese código en interceptores.

Types of Interceptors

Interceptors can be applied to business methods, timeout methods, and life-cycle operations:

- Business Method Interceptor:
@AroundInvoke
- Timeout Interceptor:
@AroundTimeout
- Lifecycle Interceptors:
@AroundConstruct
@PostConstruct
@PreDestroy

```
@Interceptors({SomeInterceptor.class})
public class SomeBean {
    public SomeBean() {...}
    @PostConstruct
    public void init() {...}
    ...
}
```

```
@Interceptor
public class SomeInterceptor {
    @AroundConstruct
    public Object methodA(InvocationContext ctx) {
        // Perform "Before" Actions
        Object obj = ctx.proceed();
        // Perform "After" Actions
        return obj;
    }
    @PostConstruct
    public Object methodB(InvocationContext ctx) {...}
    @PreDestroy
    public Object methodC(InvocationContext ctx) {...}
}
```

The `@AroundConstruct` annotation designates an interceptor method that receives a callback when the target class constructor is invoked.

The method to which this annotation is applied must have one of the following signatures:

@AroundConstruct

```
public void <METHOD>(InvocationContext ctx) { ... }
```

@AroundConstruct

```
public Object <METHOD>(InvocationContext ctx) throws Exception { ... }
```

@PostConstruct and **@PreDestroy** annotations define interceptor operations that are applied to the corresponding bean life-cycle events.

The **@AroundTimeout** annotation defines an interceptor method that interposes on timeout methods. It may be applied to any nonfinal, nonstatic method with a single parameter of type `InvocationContext` and return type `Object` of the target class (or superclass) or of any interceptor class. `InvocationContext.getTimer()` allows any `@AroundTimeout` method to retrieve the timer object associated with the timeout.

También puede interceptar métodos de ciclo de vida. ¿Recuerdas el `PostConstruct`, `PreDestroy`? Así que también puedes interceptar estas llamadas. Invocaciones de constructor, método `AroundConstruct`. Interceptar la llamada al constructor de la clase. Así que estos son interceptores de ciclo de vida. Ah, y opcionalmente, si te apetece, un método interceptor tiene esta capacidad.

Realmente debería resaltar esa habilidad. ¿Qué sucederá si no invoca el método de proceder? No solo el ciclo de vida, cualquier método de interceptor. ¿Qué va a pasar? No estás avanzando al próximo interceptor en una cadena. No estás progresando al frijol. Entonces, si no está llamando a proceder en `AroundConstruct`, ¿tendría que construir el bean? No.

Eso es lo que sucede si no llamas a continuar. Puede ser una decisión consciente. Por ejemplo, el interceptor está haciendo algo que evita que se cree el bean o evita que se ejecute el método. No sé. Tal vez hay algún aspecto de seguridad. Revisas algo de seguridad y dices que no. Si ese control de seguridad falla, no continúe. ¿Por qué no? Ciertamente podría usar el interceptor para prohibir la llamada al bean.

Apply Interceptors

- An interceptor can be applied to multiple components.
- Multiple interceptors can be applied to the same target component.
 - A, B, C, and D interceptors are applied to operationX.
 - A and B interceptors are applied to operationY.
 - No interceptor is applied to operationZ.

```
@Stateless
@Interceptors({InterceptorA.class, InterceptorB.class})
public class SomeBean {
    @Interceptors({InterceptorC.class, InterceptorD.class})
    public void operationX() {...}
    public void operationY() {...}
    @ExcludeInterceptors
    public void operationZ() {...}
}
```



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Interceptor operations may be described directly in the target component class. In this case they are applied to all business operations of this component:

```
@Stateless
public class SomeBean {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx){
        ...
        Object obj = ctx.proceed();
        ...
        return obj;
    }
    ...
    public void operationL() {...}
    public void operationM() {...}
}
```

Los interceptores podrían estar allí aplicados a sus clases. Puedes aplicarlos a nivel de clase. Puede aplicarlos a nivel de método. Puede excluirlos. Entonces, el resultado es que para la operación x, se aplicarán los interceptores A, B, C, D. En el orden que sea, no lo sé. Tenemos que mirar la anotación de prioridad en ellos. Entonces, la propiedad de prioridad define el orden. Al menos un poco de orden.

Y en la operación Y, solo se aplican los interceptores A y B. Y en la operación Z, no se aplican interceptores porque acaba de excluirlos. Ahí tienes Así que es fácil de controlar.

Summary

In this lesson, you should have learned how to:

- Create Session EJB components
- Create EJB business methods
- Manage EJB life cycle with container callbacks
- Use asynchronous EJB operations
- Control transactions
- Create EJB timers
- Create and apply interceptors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Y eso es todo por esta sesión en particular. Así que hemos visto cuáles son los tipos de bean de sesión, cómo definir sus métodos, cómo exponerlos a los clientes usando una variedad de interfaces: remoto, ley, código, incluso servicio web. Que las operaciones pueden ser síncronas y asíncronas. Que se puede participar en las transacciones. Que puede ejecutar métodos basados en temporizadores. Que se pueden aplicar interceptores. Y, por supuesto, gestionar el ciclo de vida del bean.

En la práctica para esta sesión en particular, creará un bean Enterprise Java que será un bean de sesión sin estado que contendrá métodos de lógica comercial y administrará su producto entre bastidores. Ese producto es el mismo código JPA que creó en un ejercicio anterior. Es solo que esta vez se ejecutará en el entorno Java EE en lugar del entorno Java SE.

Vamos a exponer este código a través de una interfaz remota al cliente Java. Entonces llamaremos a ese bean de forma remota desde el cliente. Y esa es otra tarea. Tendremos que escribir a ese cliente.

Luego se le pedirá que cree otro Enterprise Java Bean que será un singleton que contendrá un temporizador. Estaremos revisando productos. Están a punto de caducar. Así que los que tienen fecha de caducidad, mejor antes de la fecha de mañana. Y luego si están a punto de caducar pues vamos a montar un evento al respecto. Lo que haremos cuando caduquen, de hecho continuaremos con ese tema en particular en ejercicios posteriores. Entonces habrá más lógica adjunta a ese código en particular más adelante.

Pero de todos modos, primero debemos activar ese evento y ver qué sucede si se activa el temporizador. Ah, y una cosa más que harás en la práctica. Tendrás una interacción entre un par de beans y un servidor. El singleton llamará a ProductFacade. Solo una llamada local entre estos beans.