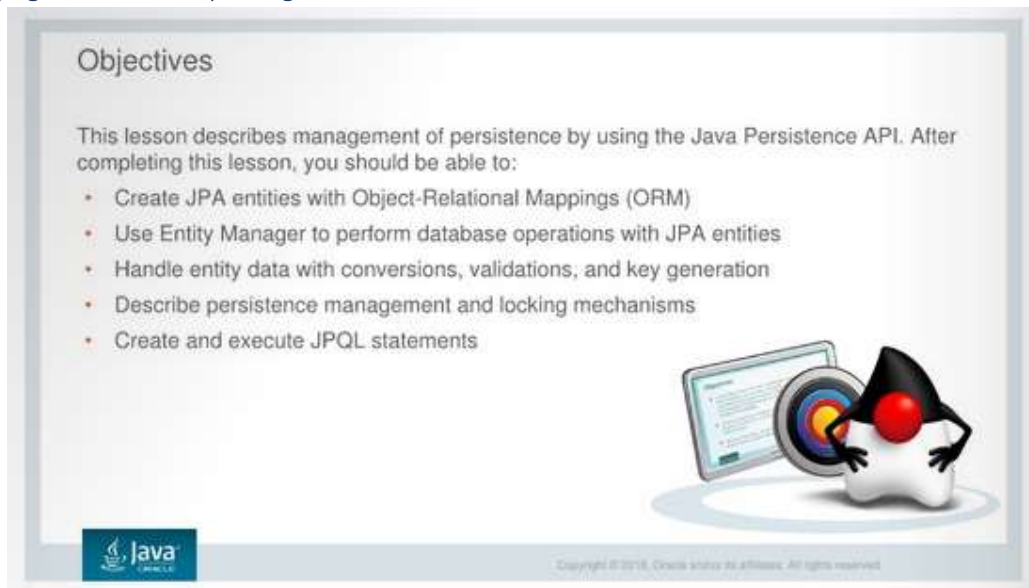


3. MANAGING PERSISTENCE BY USING JPA ENTITIES

3.1. Managing Persistence by Using JPA Entities: Java Persistence API



Echemos un vistazo a las formas de administrar la persistencia con la API de persistencia de Java. En primer lugar, en este capítulo, cubriremos diferentes áreas de la gestión de la persistencia. Primero, debemos averiguar cómo presentamos el contenido de la base de datos relacional, las tablas y los registros en el mundo de Java. Y eso es lo que hacemos con los mapeos relacionales de objetos: mapear clases de Java, que son entidades JPA, entidades API de persistencia de Java, a tablas de base de datos, vistas del contenido de la base de datos.

Luego, una vez que hemos resuelto las asignaciones, comenzamos a usarlas con transacciones de base de datos, con interacciones, consultas, datos de la base de datos, actualización, eliminación, creación de nuevos registros. Entonces, para eso, usamos la clase llamada Entity Manager. Esa es otra parte importante de este capítulo.

Por supuesto, mientras manejamos las interacciones con la base de datos, tendremos que pensar en la forma de convertir los valores de cualquier formato que tengan en Java a los formatos que la base de datos pueda entender. Validar datos y también algunas otras cosas, como, por ejemplo, generar claves primarias a partir de secuencias. Generación de claves.

Parte del contenido del capítulo también cubriría las transacciones y los mecanismos de bloqueo que podríamos usar en la API de persistencia de Java. Sin embargo, y este es un punto importante, el tema de la transacción continúa más allá de este capítulo. En el Capítulo 4, que cubre los beans Java empresariales, retomamos el tema de las transacciones y lo continuamos allí. Así que tendremos una parte en este capítulo, solo la parte que se relaciona con la parte de la base de datos, pero luego tendremos que ir más allá de las interacciones de la base de datos y descubrir cómo funcionan las transacciones en Java EE en general, no solo para la base de datos.

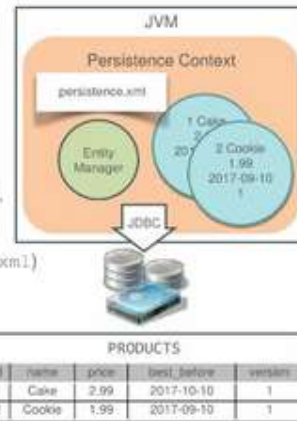
Y finalmente, una cosa más en este capítulo que vamos a cubrir son las formas de usar el lenguaje de consulta de persistencia de Java, JPQL. Es una mezcla interesante entre el mundo sintáctico de SQL y la sintaxis del mundo de objetos de Java. Así que vamos a echar un vistazo a ese lenguaje JPQL y cómo podemos utilizarlo en nuestras aplicaciones.

Java Persistence API

JPA is a lightweight framework that is designed to represent relational database data as POJOs.

JPA is built on top of JDBC and addresses the complexity of managing both SQL and Java code.

- JPA is designed to facilitate object-relational mapping.
- JPA works in both Java SE and Java EE environments.
- ORM Providers (EclipseLink, Hibernate) implement JPA operations.
- The key JPA concepts include:
 - Entities (POJOs mapped to database objects)
 - Persistence units (configuration described in `persistence.xml`)
 - Persistence contexts (In-Memory set of Entity instances)
- EntityManager is used to perform operations on entities:
 - `find`
 - `persist`
 - `merge`
 - `remove`



The Java Persistence API (1.0) began as a part of the Enterprise JavaBeans 3.0 specification (JSR-220) to standardize a model from object-relational mapping.

JPA 2.0 (JSR-317) set out to improve on the original JPA specification and was defined in its own JSR. JPA is currently at version 2.1 (JSR-338).

JDBC was the first mechanism that Java developers used for persistent storage of data. However, working with JDBC requires that you understand how to map a Java object to a database table and maintain the set of SQL queries that is used to transform relational data into Java objects. JPA provides a framework for this object-relational mapping while preserving the ability to manipulate databases directly.

The key JPA concepts in this lesson are the starting point for writing JPA applications:

- An entity can be used to represent a relational table by a Java object. (This is a one-to-one mapping.)
- A persistence unit defines the set of all classes that are related or grouped by the application and that must be collocated in their mapping to a single database.
- A persistence context is a set of entity instances in which there is a unique entity instance for any persistent entity identity.

Comencemos con los conceptos básicos de la persistencia de Java de la API de JPA. En primer lugar, no tiene que ser exclusivamente algo que haga en un contenedor Java EE. También puede usar la API de persistencia de Java en un Java SE, no solo en Java EE, por lo que es válido en cualquier entorno.

Lo que establecemos es un objeto de contexto de persistencia que será administrado por la clase denominada Entity Manager. Por lo tanto, el contexto de persistencia se administra a través del Entity Manager. Entity Manager será responsable de llevar a cabo acciones contra la base de datos: crear, eliminar, actualizar, consultar registros, este tipo de cosas. Entonces acciones que hacemos con nuestras tablas.

Ahora, en un mundo Java, los datos de estas tablas se representarán como objetos Java. Por lo tanto, una tabla se puede asignar a la clase. Un registro en una tabla se representará como una instancia de esa clase. Bueno, aquí hay un ejemplo bastante trivial de tabla de productos. Algunos productos. Pasteles y galletas. Solo porque son dulces. Asignado a la clase Java y luego representado allí como objetos Java.

Entonces podríamos operar en los objetos Java relevantes y, a cambio, obtener acciones contra la base de datos. Y la API de JPA traducirá nuestras acciones contra el objeto a las acciones contra la tabla subyacente y persistirá en que se consulten los datos. Ahora, la API de JPA tiene una historia interesante. Se ha desarrollado un poco al revés. No surgió de la especificación que luego fue implementada por diferentes proveedores.

Todo lo contrario, en realidad fue una implementación primero. Había empresas como, ya sabes, API, Eclipse Link, Hibernate, Top Link, lo que sea. Bastantes API diferentes que estaban haciendo el trabajo, y luego se combinaron bajo el estándar común bajo el paraguas de la API de persistencia de Java y ahora se conocen como proveedores de mapeo relacional de objetos. Entonces, hay implementaciones físicas detrás de la especificación JPA. Pero la forma

en que surgieron históricamente, la forma en que se desarrollaron históricamente, fue sin esa especificación general de JPA. Eso surgió después.

Así que vamos a usar el proveedor ORM detrás de un JPA, pero en este curso no vamos a escribir nada que sea específico del proveedor ORM. Vamos a ceñirnos a la norma. No vamos a hacer algo que solo funcione en Hibernate o solo en Eclipse Link. Intentaremos mantener las cosas neutrales con respecto al proveedor. Cíñete a la especificación JPA estándar.

Por supuesto, los proveedores pueden tener algunas funciones únicas que solo están disponibles en un proveedor determinado, pero con suerte, al final de esta sesión y del ejercicio, probablemente verá que no hay mucha lógica detrás del uso de tales funciones. Entonces, en estos días, la API de JPA cubre prácticamente todas las bases. Derecha.

Ahora, archivo de configuración. Hay un archivo de configuración aquí y se llama XML de persistencia y, a diferencia de otras configuraciones que hemos encontrado hasta ahora en este curso, recuerde que teníamos todos los descriptores de implementación. Era Web XML, ejb-jar.xml, Beans XML. ¿Qué fue lo que se dijo de ellos? Todos eran opcionales, ¿verdad? Podrías haber hecho solo anotaciones.

XML de persistencia no lo es. Debe crear este archivo para configurar el contexto de persistencia utilizado por Entity Manager. Así que eso es bastante obligatorio. Tienes que tenerlo.

Sin embargo, las asignaciones relacionales de objetos, es decir, los bits que asignamos con estos proveedores de ORM, podrían regirse completamente por anotaciones y, opcionalmente, puede crear un archivo ORM XML para realizar las asignaciones, pero eso es opcional. Así que podría mapear sus entidades para subrayar tablas a través de anotaciones y eso es todo.

JPA Entities: Basics

```
package demos.db;
import javax.persistence.*;

@Entity
@Table(name="PRODUCTS")
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    @NotNull
    private String name;
    @NotNull
    private BigDecimal price;
    @NotNull
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    private LocalDate bestBefore;
    @Version
    @NotNull
    private Integer version;
    /* constructors, get, set,
    equals, hashCode etc, operations */
}
```

JPA Entities:

- **Classes mapped to Database Tables or Views**
- **Attributes mapped to Columns**
- **Unique Identity** must be specified.
- **Validations and Constraints** may be applied.
- **Temporal** attributes represent Date, Time, or Timestamp database types.
- **Names may be overridden if required.**
- **Version** attribute may be used to handle locking.
- **Entity instances** represent records.



PRODUCTS				
id	name	price	best_before	version
1	Cake	2.99	2017-10-10	1
2	Cookie	1.99	2017-09-10	1



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

A JPA entity is a Java object that represents something to be persisted—a Customer, an Employee, a Book, and so on. Basically, anything that is typically represented in a relational database can be represented by a POJO that defines the elements of the entity and includes methods to set and get the values of each element.

The JPA characteristics of entities include:

- **Persistence:** Entities that are created can be stored in a database (persisted) yet treated as objects.
- **Identity:** Each entity has a unique object identity, which is generally associated with a key in the persistent store (database). Typically, the key is the database primary key.
- **Transactions:** JPA requires a transactional context for operations that commit changes to the database.
- **Granularity:** JPA entities can be as fine-grained or coarse-grained as a developer wants, typically representing a single row in a table.

Entities are managed by the EntityManager API.

- An entity can be created from a POJO through annotations.
- The entity class must be annotated with the `@Entity` annotation.
- The entity class must have a `no-arg` constructor. The `no-arg` constructor must be public or protected. Other constructors can still be used.
- The entity class must be a top-level class but can be a concrete or an abstract class.
- The persistent state of an entity is represented by instance fields.
- Entities support inheritance and must not be final.

- If an entity instance is to be passed by value as a detached object (for example, through a remote interface), the entity class must implement the `java.io.Serializable` interface.
- Note that instance fields must be accessed only from within the methods of the entity by the entity instance itself; they must be declared `private`.
- An entity cannot be an enum or an interface.
- No methods or persistent instance variables of the entity class may be `final`.
- Given the definition of an entity class and what you know about CDI beans, it might be tempting to simply inject an entity as a managed bean into another managed bean. This is discouraged by the designers of CDI because JPA cannot persist injected CDI proxies.

In the example in the slide, you can see the `@Entity` annotation to declare the `Product` class as an entity class to manage. Because the name of the entity class differs from the name of the table, a `@Table` annotation is used to map these object names.

The `@Id` annotation declares the primary key.

The field names match the table column names (attributes), except the `bestBefore` field, which is mapped to a database column with the help of the `@Column` annotation to match the column name.

The `@Temporal` annotation is required for any persisted field that is of the `java.util.Date` or `java.util.Calendar` type. Because databases store dates in different ways, some as `TimeStamp` types and some as `Date` types, the `@Temporal` annotation indicates that you want Java to manage the conversion to and from the Java `Date` or `Calendar` type to the appropriate type in the database.

Alternatively to Annotations, Entities can also be mapped to database object with the `orm.xml` file.

The `orm.xml` file is typically specified in the `META-INF` directory in the root of the persistence unit, or in the `META-INF` directory of any JAR file that is referenced by `persistence.xml`. Additionally, one or more mapping files can be referenced by the mapping-file elements of the persistence-unit element. These mapping files can be present anywhere on the class path.

Note: If you give the object-relational mapping XML file a name other than `orm.xml`, you must include the mapping-file element in `persistence.xml`.

```
<persistence>
  <persistence-unit name="ProductPU">
    <mapping-file>customMappings.xml</mapping-file>
  </persistence-unit>
</persistence>
```

This is an example content of the object relational mappings file:

```
<entity-mapping ...>
  <entity class="demos.db.Product">
    <attributes>
      <id name="id">
        <column name="ID" />
      </id>
      <basic name="name" />
      <basic name="price" />
    </attributes>
  </entity>
</entity-mapping>
```

Entonces, ¿cómo funcionan las asignaciones JPA? Echemos un vistazo a los conceptos básicos. Tenemos esta clase, una clase de producto, y la asignamos a la tabla de la base de datos. Todo lo que tenemos que hacer para hacer eso es simplemente decir entidad. Si el nombre de la tabla es el mismo que el nombre de la clase, en realidad no necesita hacer nada más, lo suficientemente divertido. Bueno, casi nada más. Pero si el nombre de la tabla es diferente, no importa. Siempre puede sobrescribir el nombre de la tabla, digamos productos. Bueno, como ves, la clase se llama producto. Bueno, allá vas.

Hay una cosa más que en realidad es obligatoria. Debe especificar dónde está su clave principal. Por lo tanto, algún atributo debe designarse como su identidad del registro. De lo contrario, ¿cómo demonios averiguaría la API de Java qué instancia de producto corresponde a qué registro en la tabla de productos? Entonces eso se hace usando esa columna.

Si el nombre de la columna es el mismo que el nombre del atributo de Java, no necesita hacer nada más. Si el nombre de la columna es diferente y el atributo de Java tiene un nombre diferente, simplemente use la anotación de

la columna. No es un gran trato. Simplemente anule ese nombre para que coincida. Porque tal vez en Java le gustaría usar una convención de nomenclatura diferente. Eso está perfectamente bien. Así también, sus atributos de Java pueden estar asociados con una serie de validaciones, pero hablaremos más sobre validaciones y restricciones un poco más adelante. En esta etapa, solo tenemos una restricción NOT NULL aquí que solo dice que el elemento es obligatorio, pero habrá una página posterior en este capítulo en la que hablaremos sobre otros casos de validación.

Con fechas, horas y marcas de tiempo, es posible que deba designar un tipo particular de objeto de fecha que esté utilizando. Puede ser una fecha o puede ser el tiempo o pueden ser ambos. Marca de tiempo o imagínense. Y verá, en Java, estos valores pueden presentarse de la misma manera que, por ejemplo, la fecha de Java, pero en una base de datos, puede haber diferentes tipos de columnas. Por lo tanto, es posible que deba designar de cuál de ellos está hablando al mapear algo que es una fecha.

Ahora, otro punto interesante que observa aquí es esta anotación llamada Versión. Una vez más, una página posterior de este capítulo revelará más información al respecto, pero es básicamente la forma de realizar un seguimiento de los cambios que se producen en un registro. Cada vez que lo actualice, puede decir, por ejemplo, incrementar el valor entero de esa versión. Alternativamente, puede usar la marca de tiempo, pero eso no es particularmente recomendable, en realidad. Entonces es algo de lo que podemos beneficiarnos cuando manejamos el bloqueo de estos registros. Una vez más, habrá más información al respecto en una página posterior.

Entonces, en resumen. ¿Quieres mapear tu clase de Java a una tabla? Digamos que es una entidad. Designa el DNI. Todo lo demás según sea necesario. ¿Sí? ESTÁ BIEN.

Persistent Field Versus Persistent Property

Using Property instead of the Field annotation allows you to perform extra processing of value.

- Properties are annotated against "get" operations.
- Fields are annotated against attributes.
- If a field has the same name as a column, no annotations are required (default).
- Transient fields are not persisted.

Persistent Field "price"

```
@Entity
public class Product implements Serializable {
    ...
    private BigDecimal price;

    public BigDecimal getPrice() {
        return price;
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

Persistent Property "price" Transient Field "discount"

```
@Entity
public class Product implements Serializable {
    ...
    private BigDecimal price;
    @Transient
    private BigDecimal discount;

    @ColumnName("price")
    public BigDecimal getPrice() {
        return price.subtract(discount);
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
        this.discount =
            price.multiply(BigDecimal.valueOf(0.1));
    }
}
```

Entity fields:

- They cannot be public.
- They should not be read by clients directly.
- Unless they are annotated with the `@Transient` annotation or the transient keyword, all fields are persisted. (The use of the `@Column` annotation defines only the column name to use.)

When fields are annotated, the persistence provider uses reflection to get and set the fields of the entity.

Note: Accessor methods can be present in an entity class that uses field-based access and may be useful for other types of operations, but they are ignored by the persistence provider.

When using persistent properties, the persistence provider will retrieve an object's state by calling the accessor methods of the entity class.

- Methods must be declared as public or protected.
- Methods must follow the JavaBeans naming convention.
- `@Column` may also be used to define the column name in the database.
- Persistence annotations can be placed only on getter methods.

The choice between property and field is really about whether you want to do any additional processing in a getter/setter method.

Cuando asigna una clase de Java a una tabla, hay un pequeño truco con respecto a qué asigna exactamente en una clase. Y aquí hay dos opciones, conocidas como un campo o una propiedad. Un campo significa que está considerando una asignación de un atributo. Una propiedad, está mapeando un par de métodos getter setter. Una asignación predeterminada es la asignación de una variable de instancia de Java. Ese es el valor predeterminado.

Si desea mapear no solo el precio variable sino algo que haría con ese precio, digamos que tiene algo de lógica aquí. Estás calculando cosas. Estás restando descuento, lo estás sumando o lo que sea, ¿sí? Entonces, algo de lógica en estos métodos getter setter. Luego, en lugar de asignar el atributo de precio, coloca la anotación, se llama column, en un método Get que funcionará tanto para Get como para Set. No repite la anotación en el método Set. Eso no es necesario. Solo con el método Get es suficiente.

E inmediatamente lo que sucede es que, en una base de datos, el valor del atributo precio se almacenará todo bien. Ese sigue siendo el caso. Pero en la aplicación, cada vez que manejemos el precio, buscaremos esta lógica adicional. Bueno, no sé, tal vez necesitamos calcular el descuento cuando fijamos el precio o algo así, ¿sí?

Ah, y si desea desactivar por completo la asignación de un elemento en particular a una tabla, este campo Descuento, por ejemplo, imagine que desea que el campo Descuento esté presente en la memoria JPA en Java, pero no desea un column y una tabla así, por la razón que sea, ¿no? Puede marcarlo como transitorio.

¿Notó que transitorio no es una palabra clave de Java sino una anotación? Así que no es un campo transitorio, per se, desde el punto de vista del lenguaje. Es un campo transitorio desde la perspectiva de la API JPA. No está tratando de persistir ese elemento en particular en la mesa, tal vez por la razón de que no desea almacenarlo allí.

De forma predeterminada, los campos se asignan a las columnas de la tabla de la base de datos. Sin embargo, esto es algo que puede sobrescribir y así es como se hace.

Using Access Annotation

Access Annotation overrides Property and Field Access Types.

In this example:

- At class level, access type is set to FIELD.
- All fields are persisted as usual.
- Field price changes its access type handling to PROPERTY in order to enable additional processing in its get/set operations.
- Field price is marked as Transient (not persisted as other fields) to avoid duplication with the getPrice operation marked with PROPERTY access type.
- Field discount is Transient and will not be persisted.

```
@Entity
@Access(AccessType.FIELD)
public class Product implements Serializable {
    @Id
    private Integer id;
    @Transient
    private BigDecimal discount;
    @Transient
    private BigDecimal price;

    @Access(AccessType.PROPERTY)
    public BigDecimal getPrice() {
        return price.subtract(discount);
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
        this.discount =
            price.multiply(BigDecimal.valueOf(0.1));
    }
}
```

To use both Field and Property access types for the same entity, you should add a class-level `@Access` annotation to define the default access type for the entity class and additional `@Access` annotations to the individual persistent fields or properties that should be different from the default access type.

- If the class is designated with field access, annotate the property (getter methods) with `@Access(AccessType.PROPERTY)`.
- If the class is designated with property access, annotate the fields with `@Access(AccessType.FIELD)`.

Note: When different access types are combined within the same class, the `@Transient` annotation should be used to avoid duplicate mappings.

Es la anotación llamada Acceso. Entonces, por defecto, es un campo.

En otras palabras, lo que está escrito aquí es realmente excesivo. Cuando está diciendo, Campo de tipo de acceso, es el valor predeterminado. Pero, de nuevo, puede decir propiedad de tipo de acceso. Bueno, eso no lo es, así que

puedes sobrescribirlo. Y lo que sucede aquí es que, cuando cambia el tipo de acceso a la propiedad, permite que se asignen los métodos getter setter en lugar del atributo.

Tenga cuidado, si está jugando con la anotación de tipo de acceso, podría mapear accidentalmente un atributo, como el precio, aquí. Y el método getter también. El valor predeterminado en una clase dice campo, por lo que se asigna el atributo de precio, pero luego la propiedad de tipo de acceso se coloca en el método getPrice, por lo que se asigna el método.

Entonces, ¿de qué manera está mapeado? Ambas cosas. Estamos creando aquí una duplicación de un mapeo. Eso es algo que preferirías evitar, ¿verdad? Entonces, para no crear tal duplicación, puede marcar esa columna como transitoria. Confíe en mí, seguiremos siendo persistentes porque getPrice se asigna como la propiedad de tipo de acceso y simplemente desactivó la asignación para el campo de ese elemento en particular.

Es un poco raro. No recomiendo que esté haciendo eso, pero este PowerPoint lo ayuda a comprender cómo puede cambiar toda la clase, por defecto, de campo a propiedad si lo necesita y cómo puede hacerlo para un atributo individual. Solo tenga cuidado de no abusar de esta función porque puede dificultar qué es lo que realmente está mapeando. Está bien.

Converters

A converter defines an alternative type of transformation.

- Converters can be applied **globally**, to all fields of a particular type or
- A specific entity field can directly **reference a converter**
- `@Convert(disableConversion=true)` disables global automatic conversion for a specific field.

```
@Converters(autoApply=true)
public class DateConverter implements AttributeConverter<LocalDate, Date> {
    @Override
    public Date convertToDatabaseColumn(LocalDate value) {
        return (value==null) ? null :
            Date.from(value.atStartOfDay(ZoneId.systemDefault()).toInstant());
    }
    @Override
    public LocalDate convertToEntityAttribute(Date value) {
        return (value==null) ? null :
            Instant.ofEpochMilli(value.getTime()).atZone(ZoneId.systemDefault()).toLocalDate();
    }
}
```

```
package demos.db;
@Entity
public class Product implements Serializable {
    ...
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    @Convert(converter="DateConverter.class")
    private LocalDate bestBefore;
}
```

When the optional `autoApply` property is set to "true", the converter will be applied to all attributes of a specific type—in the example above, it is `LocalDate`.

To disable automatic conversion for a particular attribute, the `@Convert(disableConversion=true)` annotation should be applied to the attribute.

Persistence fields or properties can be of the following data types:

Java primitive types and Java wrappers, including:

- Integer, Boolean, Float, and Double

Serializable types, including:

- String, `BigDecimal`, and `BigInteger`

Arrays of bytes and characters, including:

- `byte[]` and `Byte[]`, `char[]` and `Character[]`

Temporal types, including (but not limited to):

- `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, and `java.sql.Timestamp`

Enums and collections

When implementing relationship mappings, another entity, or a collection of entities becomes an attribute type.

In this example a `LocalDate` class is used to represent the `bestBefore` Date item. Depending on the version of the JPA provider, this may require additional conversion. The JPA 2.1 (latest) specification is not part of Java EE, but the Java SE product set, and has been created before Java SE 8. Therefore, unless a JPA provider specifically supports Java SE 8 features, it would not be able to directly handle Java SE 8 specific types such as `LocalDate`, `LocalTime`, or `LocalDateTime`.

Convertidores. Un convertidor lo ayuda a convertir el valor que tiene en Java al valor que desea almacenar en una base de datos y viceversa. Hay diferentes casos de uso para los convertidores. Tal vez, en una tabla desee almacenar

varias columnas y en Java desee presentarlas como un tipo integrable. Quizás. Así que convierta varios valores en un valor o viceversa.

Tal vez quiera almacenar algo en una base de datos y presentarlo en Java como su clase personalizada. Verá, la cuestión es que los convertidores para tipos de datos básicos se proporcionan de forma inmediata. No necesita escribir un convertidor para convertir un número entero o un flotante o un decimal grande o una cadena o incluso una fecha. Estos convertidores están disponibles para usted.

Pero si está creando su propio tipo de datos personalizado y está utilizando una clase que la API JPA no conoce, entonces debe proporcionar un convertidor para convertir valores de ese tipo a lo que sea que esté almacenando en una base de datos -- varchar, número, lo que sea. Cualquiera que sea el tipo que esté almacenando en la base de datos. Así que usted será responsable de dicha conversión.

Ahora aquí hay uno complicado. Verá, este ejemplo se almacena en una fecha de base de datos. Entonces, la conversión es al tipo de fecha. Conversión desde, conversión a, hasta tipo de fecha. Pero, ¿qué es lo que estamos convirtiendo hasta la fecha? Fecha local. La fecha local es una característica nueva en Java SE8 y se recomienda usar. Se recomienda usar la fecha local de la API, la hora local, la hora de la fecha local en lugar de la fecha anterior, que aún puede usar, obviamente.

Desafortunadamente, el estándar JPA se finalizó antes de que Java SE8 finalizara la API de fecha local como estándar. Y como resultado, la mayoría de los proveedores de JPA, proveedores de ORM, tratan la fecha local como una clase desconocida. Ellos no saben lo que es. Ellos no consideran que en realidad sea una cita. Así que no proporcionan un convertidor automático.

Así que este es un ejemplo muy práctico. Convierte fecha local a fecha y viceversa. Es algo muy simple de hacer. Sólo tienes que proporcionar dos métodos. Uno se llama columna Convertir a base de datos. Otro se llama atributo Convertir en entidad. Entonces, uno acepta la fecha local devuelve la fecha y otro acepta la fecha devuelve la fecha local. Y lo haces para cualquier tipo de conversión que necesites. No se trata solo de fechas locales, así es como escribes convertidores de todos modos.

Así que usted proporciona la lógica. Cómo se deben convertir los valores. Todo lo que queda es aplicar el convertidor a su código real. Hay dos formas de hacerlo. Puede aplicar globalmente, `alt apply true`, en una anotación de convertidor, o puede aplicarlo para un atributo individual para decir qué convertidor le gustaría usar. Lo que sea.

Ah, y también puedes deshabilitar los convertidores. Hay una anotación de conversión deshabilitada allí si no desea realizar la conversión por algún motivo. Ahí tienes Así que supongo que es muy-- en realidad, esto es algo que harás en el ejercicio. Durante la práctica, se le pedirá que use la fecha local en lugar de la fecha, y sin un convertidor, su proveedor de ORM no funcionaría, por lo que deberá proporcionarlo como parte de la práctica.

Generated Keys

Primary Key column values can be generated by using database sequences or tables.

SequenceGenerator

```
import javax.persistence.*;

@Entity
public class Product implements Serializable {
    @Id
    @GeneratedValue(strategy=SEQUENCE,
        generator="seq_idGen")
    @SequenceGenerator(name="seq_idGen",
        sequenceName="PID_SEQ")
    private Integer id;
    ...
}
```

TableGenerator

```
import javax.persistence.*;

@Entity
public class Product implements Serializable {
    @Id
    @GeneratedValue(strategy=TABLE,
        generator="tab_idGen")
    @TableGenerator(name="tab_idGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="PID_SEQ")
    private Integer id;
    ...
}
```



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

The GeneratedValue annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the Id annotation. The use of the GeneratedValue annotation is required to be supported only for simple primary keys. Use of the GeneratedValue annotation is not supported for derived primary keys.

The Generated Value annotation has following properties:

- **generator (Optional):** The name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation
- **strategy (Optional):** The primary key generation strategy that the persistence provider must use to generate the annotated entity primary key. This could be SequenceGenerator or TableGenerator.

A Sequence Generator defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A sequence generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

The SequenceGenerator annotation has following properties:

- **allocationSize (Optional):** The amount to increment by when allocating sequence numbers from the sequence
- **catalog (Optional):** The catalog of the sequence generator
- **initialValue (Optional):** The value from which the sequence object is to start generating
- **schema (Optional):** The schema of the sequence generator
- **sequenceName (Optional):** The name of the database sequence object from which to obtain primary key values

Defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A table generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

The TableGenerator annotation has the following properties:

- **allocationSize (Optional):** The amount to increment by when allocating ID numbers from the generator
- **catalog (Optional):** The catalog of the table
- **indexes (Optional):** Indexes for the table
- **initialValue (Optional):** The initial value to be used to initialize the column that stores the last value generated
- **pkColumnName (Optional):** Name of the primary key column in the table
- **pkColumnValue (Optional):** The primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table
- **schema (Optional):** The schema of the table
- **table (Optional):** Name of the table that stores the generated ID values
- **uniqueConstraints (Optional):** Unique constraints that are to be placed on the table
- **valueColumnName (Optional):** Name of the column that stores the last value generated

Claves generadas. Su clave principal puede generarse a partir de una secuencia. De hecho, es algo que la gente suele hacer. Existen diferentes estrategias de generación de claves. Aquí hay un par de ejemplos, llamados secuencia, tabla. Hay mas. Hay una opción en la que no especifica cómo se genera la clave. Le dices al proveedor ORM, imagínate. Automáticamente generarme el siguiente valor. Hay una opción de columna de ID. Pero estos son un par de ejemplos que usan la secuencia de la base de datos. Y luego el otro está usando un table.

Por cierto, ¿sabía que la anotación de valor generado hace referencia al generador simplemente usando el nombre? Así que esa es la parte que tiene que coincidir. En otras palabras, esta instrucción se puede colocar en otro lugar. Podría simplemente describir el generador de secuencias en alguna otra parte de su código, y luego simplemente, cualquier entidad en la que desee usarlo, solo diga cuál y simplemente use el nombre. Lo mismo sucede aquí. Así que solo hacemos coincidir el nombre.

A menudo, las personas lo describen en la misma entidad porque tiene sentido. Sí, haces ambas cosas y dices que así es como me gustaría proporcionar valores de columna de clave principal. Pero no tienes que hacerlo. Por lo que sé, puede usar el mismo generador para diferentes entidades. Eso no está prohibido. ESTÁ BIEN.

JPA Lifecycle Callback Methods


JPA Lifecycle Callback Methods are invoked by the EntityManager when it handles the Entity instance.

- Pre and Post Persist events are triggered around the insert of the new record.
- Pre and Post Update events are triggered around the update of the existing record.
- Pre and Post Remove events are triggered around the delete of the existing record.
- The PostLoad method is invoked when an entity is loaded into the current persistence context from the database or after the refresh operation is applied to it.

In these operations, you can:

- Validate Data
- Handle Data Denormalizations
- Set defaults and calculate values and so on

```
@Entity
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    @NotNull
    private String name;
    @PrePersist
    protected void beforeInsert() { ... }
    @PostPersist
    protected void afterInsert() { ... }
    @PreUpdate
    protected void beforeUpdate() { ... }
    @PostUpdate
    protected void afterUpdate() { ... }
    @PreRemove
    protected void beforeDelete() { ... }
    @PostRemove
    protected void afterDelete() { ... }
    @PostLoad
    protected void afterSelectOrRefresh() { ... }
    ...
}
```



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

An entity has a life cycle depending on the operations performed on that entity. Life-cycle callbacks are annotations that can be applied to entity methods that are invoked when an event occurs.

For entities to which a merge operation has been applied and causes the creation of newly managed instances, the `PrePersist` callback method is invoked for the managed instance after the entity state has been copied to it. These `PrePersist` and `PreRemove` callbacks are also invoked on all entities to which the operations are cascaded. The `PrePersist` and `PreRemove` methods are always invoked as part of the synchronous persist, merge, and remove operations.

The following rules apply to life-cycle callbacks:

- The callback methods can have public, private, protected, or package level access, but must not be static or final.
- A single method may be annotated with multiple life-cycle events, but only one callback method per life-cycle event should be used in a given entity. For example, an entity cannot have two different callback methods annotated with `@PostLoad`.
- Life-cycle callback methods may throw unchecked or runtime exceptions. A runtime exception thrown by a callback method that executes in a transaction causes that transaction to be marked for rollback.
- Lifecycle callbacks can invoke JNDI, JDBC, JMS, and enterprise beans. The life-cycle method of a portable application should not invoke `EntityManager` or `Query` operations, access other entity instances, or modify relationships in the same persistence context.
- A life-cycle callback method can modify the nonrelationship state of the entity on which it is invoked.

Cuando manipula con una entidad, puede conservarla, crear un nuevo registro en la tabla, puede actualizarla, puede eliminarla, eliminar el registro y puede consultar. Cargue el registro de la base de datos en la memoria de Java como objeto de entidad. Aparentemente, puede crear métodos interceptores para todos estos eventos. Es tan simple como eso.


Sabes que es como un disparador previo y posterior. Antes de la actualización, después de la actualización. Antes de borrar, después de borrar. Antes de insertar, después de insertar. Y después de la carga. Bueno, cualquier lógica adicional que le gustaría poner en estos métodos: mantenimiento, normalización, columnas, ponen validaciones adicionales. Tal vez calcular algunos valores. Calcule valores predeterminados para diferentes campos. Algo como eso. Así que podría ponerlos en los métodos del ciclo de vida.

Validating Entities

Bean Validation 1.1 (JSR-349) enables developers to use annotations to apply constraints to types, methods, parameters, fields, or other annotations in JPA, JSF, CDI, and JAX-RS APIs.

- JPA controls application of Bean Validation via the `persistence.xml` file.
- Validation Mode can be used to switch validations on and off.
 - NONE: Turn off validation.
 - AUTO: Turn on validation when there is a validation provider on the classpath (the default).
 - CALLBACK: Turn on validation. The persistence provider throws a `PersistenceException` if the validation provider is not available.
- JPA delegates validation to the Bean Validation implementation during the `persist`, `preupdate` entity life-cycle events.
- `Preremove` events do not trigger validation, unless enabled via validation group property.

```
<persistence-unit>
  <validation-mode>CALLBACK</validation-mode>
</persistence-unit>
<properties>
  <property name="javax.persistence.validation.group.pre-persist"
    value="javax.validation.groups.Default"/>
  <property name="javax.persistence.validation.group.pre-update"
    value="javax.validation.groups.Default"/>
  <property name="javax.persistence.validation.group.pre-remove"
    value="javax.validation.groups.Default"/>
</properties>
```



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

One of the goals of Bean Validation is to move validation from the presentation layer to the domain model, to avoid duplication of validation code in each layer of an enterprise application. This is sometimes referred to as the DRY principle: "Don't repeat yourself." Rather than clutter the domain model with validation logic, Bean Validation defines a default set of annotations that can be applied as predefined constraints, and also defines an API and metadata model to allow developers to override and extend the annotations.

Bean Validation is currently at version 1.1. The specification provides validation for any EE technology or specification. However, currently only the JPA, JSF, CDI, and JAX-RS specifications have implemented bean validation into their life cycles.

Before version 1.1, Bean Validation could be applied only to fields or properties (getter methods). Bean Validation 1.1 provides the capability of adding constraints to the parameters that are passed in a method and the values that a method returns. This capability applies also for constructors.

Bean Validation for JSF is enabled by default and can be turned off through a context parameter in the `web.xml` file:

```
<context-param>
  <param-name>
    javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR
  </param-name>
  <param-value>true</param-value>
</context-param>
```

Bean Validation for JSF beans is invoked automatically for every user-specified validation constraint whenever the components are normally validated.

Bean Validation can be invoked using an instance of a `Validator` on any Java class, class method, or constructor.

In addition to primitive and wrapper types, Bean Validation can be applied to collections, arrays, and Iterable fields and properties.

Hablando de validaciones, hay una API llamada Bean Validation 1.1 y esta API en realidad no solo usa la API de persistencia de Java. Se usa en todos lados. Bean Validation se puede usar con Java Server Faces, con servicios web JAX-RS, con beans CDI. Es una forma de validar universalmente el código Java. En general, validar cosas en código Java en cualquier lugar, no necesariamente solo con entidades. Pero ciertamente tiene sentido que le gustaría

validar los datos que coloca en una base de datos, por lo que con JPA, es el uso más obvio para la API de validación de Bean.

Bean Validation, de forma predeterminada: está configurando XML de persistencia. Puede activarlo para la API de JPA utilizando XML de persistencia. Y de forma predeterminada, tiene la propiedad llamada Auto: activar la validación cuando hay un proveedor de validación disponible en un classpath. Bean Validation se implementa por separado de JPA, por lo que el proveedor de JPA API y Bean Validation API pueden ser dos proveedores diferentes.

Si ejecuta JPA en el servidor Java EE, el servidor Java EE ya está configurado para admitir Bean Validation. Por lo tanto, cuando tiene una propiedad Auto, simplemente funciona porque la tiene en su classpath. Ya está allí. Pero, ¿no le dije al comienzo de este capítulo que un JPA puede usarse en un entorno Java SE? ah Bueno, no hay ningún servidor que lo configure por usted, por lo que es posible que Bean Validation no esté en su classpath, en cuyo caso el valor predeterminado de Auto conduce a ninguna validación.

Pones todas las restricciones de validación, has escrito todo el código, lo ejecutas, no pasa nada. Sin errores. No pasa nada. Ingresa valores claramente no válidos y no se marcan como no válidos. Si desea evitar que eso suceda, cambie el modo de validación a Devolución de llamada. Porque verá, con una devolución de llamada, obligará al proveedor de JPA a lanzarle una excepción en lugar de continuar con las validaciones desactivadas. Así que estás diciendo, no corras. Lanza el error si no he configurado correctamente la implementación detrás de una API de validación de Bean. Tiene que estar en el classpath.

El primer ejercicio que estaba haciendo con la API de persistencia de Java en este curso, bueno, este capítulo, configura el JPA para ejecutarse en el entorno Java SE. Entonces, su trabajo, parte del ejercicio, será configurar la biblioteca classpath correcta para Bean Validation. Más adelante, haremos los mismos JPA. Los ejecutaremos en el contenedor Java EE. No necesitarás hacerlo. En el contenedor Java EE, en el servidor Java EE, eso ya está configurado para que simplemente funcione.

Así que ese es el comentario sobre un motivo de devolución de llamada de validación. Sí, ¿por qué cambiarías de Auto a Devolución de llamada? Porque desea forzar la excepción de persistencia si no está configurada correctamente.

Hay una cosa más de la que me gusta hablar en esta página y estos son los puntos en el tiempo en los que se activa la validación. ¿Cuándo se activa? De forma predeterminada, se activa solo antes de la persistencia y antes de la actualización. Ese es el valor predeterminado. Bueno, es un poco oscuro por qué desea activar la validación en la eliminación previa. Bien tu puedes. Quiero decir, si estás eliminando el registro, ¿cuál es el punto de validarlo? Tal vez lo sean, no lo sé. Pero está bien, si cree que tiene sentido hacerlo, puede modificar ese archivo de configuración e incluir la eliminación previa, pero eso depende de usted. De forma predeterminada, como dije, la eliminación previa no se activa en el ciclo de validación a menos que lo reconfigure. Estos son conocidos como los grupos de validación.

Using Bean Validation Constraints

Bean Validation Constraints are applied to Entity Attributes

- Custom Error Messages can be applied:
 - Directly, using message attribute
 - Via the `ValidationMessages.properties` file
- Default or Custom messages can be used.
- Expressions can reference values.
- Valid annotation triggers validation

```
javax.validation.constraints.Size.message=
Size must be between {min} and {max}
product.bestBefore=
Product Best Before Date must be in the future

@NotNull(message="Discount must be applied")
private BigDecimal discount;
@Size(min=1, max=100)
private String description;
@Future(message="{product.bestBefore}")
private Date bestBefore;

public void updateProduct(@Valid Product p) {...}
```

Examples of Bean Validation Constraints:

```
@AssertFalse
private boolean isSomething;
@AssertTrue
private boolean isSomethingElse;
@DecimalMax(value="999.99")
@DecimalMin(value="0.99")
private BigDecimal price;
@Digits(integer=4, fraction=2)
private BigDecimal discount;
@Size(min=2, max=100)
private String description;
@Max(10)
@Min(1)
private Integer quantity;
@NotNull
@Null
@Future
private Date bestBefore;
@Past
private Date dateManufactured;
@Pattern(regexp="\\(\\d\\)\\(\\d\\)\\(\\d\\)")
private String phoneNumber;
```



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

Some Bean Validation annotations do not have required attribute elements (for example, `@NotNull`, `@Null`, `@Past` and `@Future`).

Note: Each of the annotations also has an `@<Annotation>.List` type that can be used to define a set of constraints, as in the following example:

```
@Pattern.List( {
    @Pattern(regexp="[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}"),
    @Pattern(regexp=".*?emmanuel.*?")
} )
```

Each element in the array is processed by the Bean Validation implementation as regular constraint annotations. This means that each constraint in the array is applied to the target. The targets of the constraint list should be of the same type as the initial constraint.

`ConstraintValidationException` will contain a message from the violated constraints. The default messages are declared in the implementation.

The advantage of the `ValidationMessages.properties` file is the ability to localize the messages.

See the JSR 349 specification for a complete list of standard error messages.

Constraint violation messages can use Expression Language (enclosed between `${}`), as shown in the third code snippet in the slide.

The properties listed below are resolved by the default message interpolator:

```
javax.validation.constraints.AssertFalse.message=must be false
javax.validation.constraints.AssertTrue.message=must be true
javax.validation.constraints.DecimalMax.message=must be less than
${inclusive == true ? 'or equal to ' : ''}{value}
javax.validation.constraints.DecimalMin.message=must be greater than
${inclusive == true ? 'or equal to ' : ''}{value}
javax.validation.constraints.Digits.message=numeric value out of bounds
(<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message=must be a future date
javax.validation.constraints.Max.message=must be less than or equal to
{value}
javax.validation.constraints.Min.message=must be greater than or equal to
{value}
javax.validation.constraints.NotNull.message=must not be null
javax.validation.constraints.Null.message=must be null
javax.validation.constraints.Past.message=must be a past date
javax.validation.constraints.Pattern.message=must match the following
regular expression: {regex}
javax.validation.constraints.Size.message=size must be between {min} and
{max}
```

@javax.validation.Valid annotation can be used to trigger entity object validation.

You can also programmatically validate entities using following classes:

```
javax.validation.ConstraintViolation
javax.validation.Validation
javax.validation.Validator
javax.validation.ValidatorFactory
```

Code snippet demonstrating programmatic validation:

```
ValidatorFactory validatorFactory =
Validation.buildDefaultValidatorFactory();
Validator validator = validatorFactory.getValidator();
Set<ConstraintViolation<Product>> violations =
validator.validate(product);
for (ConstraintViolation violation: violations) {
    String message = violation.getMessage();
    ...
}
```

Ahora hablando de las restricciones de validación. ¿Está bien? En la página anterior, vimos esta restricción de validación NOT NULL y les dije que hay más. Wow, bastantes cosas más. Una búsqueda verdadera, búsqueda falsa para valores booleanos. Media decimal y máx.

dígitos. Número total de dígitos. Los dígitos de fracción o dos dígitos después del punto en este caso particular para algún número de coma flotante. Tamaño entre 2 y 100. Esa es la cantidad de caracteres en una cadena, por ejemplo. mín. y máx. Rango de valores para un entero.

Ni nulo ni nulo. De hecho, hay algo que usted designa que tiene que ser nulo. Futuro. Y eso es realmente solo futuro y pasado. Le permite saber si la fecha es anterior o posterior a la hora actual. Así que también puedes validar estas cosas. Y la expresión regular para que también pueda hacer el acceso de registro. Bastante ordenado, ¿eh? Estas son las restricciones de Validación de Bean.

De hecho, puede crear restricciones de validación personalizadas. Un ejemplo de una restricción de validación personalizada se coloca en el apéndice, por lo que lo cubriremos como parte del apéndice. Ahora, ¿cómo especifica un mensaje que desea producir cuando falla una validación?

Cada restricción tiene un mensaje predeterminado. Está justo aquí. Por ejemplo, tamaño. Restricción de tamaño... este, ¿sí? Restricción de tamaño. Déjame resaltar eso para ti. Tiene un mensaje predeterminado de tamaño de restricción de validación de Java X. El tamaño tiene atributos de min y max. mín. y máx. Se puede hacer referencia desde el mensaje real como tipo de variables. Y podría poner este mensaje estándar en un archivo llamado Propiedades del mensaje de validación, que se colocan en su classpath.

Puede diseñar mensajes que no sean estándar sino personalizados. Éste. Producto mejor antes. Así que ese es un mensaje personalizado. ¿Cómo se referencian los mensajes? Si es un mensaje estándar, no necesita hacer nada. Por lo tanto, este campo que usa la anotación de tamaño mostrará automáticamente el mensaje estándar proveniente del tamaño de restricción de validación de Java X. Viniendo automáticamente de eso. Y sí, ese será el mensaje adjunto.

Pero si desea uno personalizado, todo lo que necesita hacer es hacer referencia al mensaje como una propiedad de la anotación. Entonces, en este caso particular, anotación futura, el mensaje es igual a... ¿Notas los corchetes? Esa es una expresión. Eso es lo que hace que `product.bestbefore` se interprete como la clave dentro de un paquete de recursos en lugar de solo como un texto simple. Entonces se interpreta como una referencia al paquete.

Por supuesto, puede simplemente codificar el texto, como en el primer ejemplo. Bueno, eso no es particularmente ingenioso, pero puede codificar el texto. Obviamente, se prefiere usar el paquete de recursos porque entonces puede modificar ese paquete de recursos sin tener que tocar su código. Es mucho más flexible si cambia de opinión sobre qué mensaje de error le gustaría mostrar.

Puede validar cualquier cosa, en cualquier momento que desee, siguiendo esta anotación, Válido. Solo en un método. Solo un método. No tiene que estar vinculado a un ciclo de vida de JPA o simplemente al bean arbitrario que recibe el parámetro le gustaría verificar si ese parámetro está a la altura. Solo piense anotación válida y termine con eso. Muy fácil de hacer.

3.2. Managing Persistence by Using JPA Entities: container Managed Persistence

Container Managed Persistence

Entity Manager:

- Handles entity instances by executing select, insert, update, and delete operations
- Is initialized with Persistence Unit configuration
 - Persistence Unit is a set of entities representing data from the same data store
 - Persistence Units are configured via the persistence.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="...>
  <persistence-unit name="ProductPU" transaction-type="JTA">
    <jta-data-source>jdbc/productDB</jta-data-source>
    <properties/>
  </persistence-unit>
</persistence>
```

Container Managed (Java EE) deployment:

- Entity Manager is directly injected.
- Container can control transactions.

```
package demo.model;
import javax.persistence.*;
public class ProductManager {
    @PersistenceContext (unitName="ProductPU")
    private EntityManager em;
    ...
}
```



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

The EntityManager interface performs the work of persisting entities.

- Entities that an entity manager instance holds references to are managed by the entity manager.
- A set of entities within an entity manager at any given time is called its persistence context.
- Entity instances in the persistence context are unique (for example, only one Employee instance with an ID of 110).
- A persistence provider creates the implementation details to read from and write to in a given database.
- An instance of an EntityManager is injected through the @PersistenceContext annotation using an optional persistence unit name in the injection.

Persistence Unit defines a set of entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence Unit Elements:

- **Provider:** This is an ORM implementation used behind JPA. Typical choices are EclipseLink or Hibernate. Provider must be an implementation of the javax.persistence.spi.PersistenceProvider class.

- **class, jar-file or mapping-file:** These are elements that describe a set of managed persistence classes, which are managed by a persistence unit. These could be defined by using one or more of the following:
 - Annotated managed persistence classes that are contained in the root of the persistence unit unless the `exclude-unlisted-classes` element is specified
 - One or more object-relational mapping XML files (`orm.xml`)
 - One or more JAR files that are searched for the classes
 - An explicit list of classes
- **exclude-unlisted-classes:** This element excludes all entities that are not explicitly listed in the `class`, `jar-file`, or `mapping-file` property.
- **shared-cache-mode:** This element controls whether second-level caching is in effect for the persistence unit.
- **validation-mode:** This element controls whether automatic life-cycle event time validation is in effect.

Persistence Unit can be configured to use two transaction-type attribute values:

- **JTA:** EntityManager will support Java Transactional API (JTA).
- **RESOURCE_LOCAL:** EntityManager will require you to manage transactions through the code.

Note: In a Java EE environment, if this element is not specified, the default is JTA. In a Java SE environment, if this element is not specified, the default is RESOURCE_LOCAL.

Persistence Unit configuration may define additional properties and hints:

- **javax.persistence.lock.timeout:** Value in milliseconds for pessimistic lock timeout. This is only a hint.
- **javax.persistence.query.timeout:** Value in milliseconds for query timeout. This is only a hint.
- **javax.persistence.validation.group.pre-persist:** Groups that are targeted for validation upon the `prepersist` event (overrides the default behavior)
- **javax.persistence.validation.group.pre-update:** Groups that are targeted for validation upon the `preupdate` event (overrides the default behavior)
- **javax.persistence.validation.group.pre-remove:** Groups that are targeted for validation upon the `preremove` event (overrides the default behavior)

Example of the `persistent.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
```

```
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="ProductPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>demos.db.Product</class>
    <jta-data-source>jdbc/productDB</jta-data-source>
    <properties/>
  </persistence-unit>
</persistence>
```

Container Managed Transactions with JTA are covered later in the course.

Ahora hablemos no solo de las asignaciones de entidades, sino de lo que hacemos con estas entidades cuando ejecutamos el código de la aplicación. Los usamos para administrar la persistencia, ¿verdad? Y eso hay que configurarlo. Y hay un par de opciones sobre cómo puede configurar el contexto persistente en el que se utilizarán las entidades.

La clase que manejará la persistencia para nosotros es un Entity Manager. Pero Entity Manager necesita una configuración. Esa configuración se proporciona como un archivo `persistence.xml`.

Debe crear una configuración que tenga dos propiedades importantes, propiedad de tipo de transacción y nombre. La propiedad de nombre es bastante sencilla. Cuando está inyectando Entity Manager en su código cuando usa la inyección de contexto persistente, ¿qué contexto de persistencia desea inyectar? ¿Qué configuración desea activar? Utiliza el nombre de la unidad de persistencia para hacerlo.

Entonces, la unidad persistente es una configuración dentro del archivo persistence.xml. El contexto de persistencia es un entorno en memoria creado por Entity Manager, que utiliza la unidad persistente como su configuración.

Y hay un par de opciones aquí. Entonces, como digo, el tipo de transacción es otro factor clave aquí. El tipo de transacción podría ser JTA, en cuyo caso será administrado por contenedor. Así que es algo que probablemente haga en el entorno del servidor Java EE. Solo está haciendo referencia aquí a la fuente de datos que le gustaría usar. Y espera que el contenedor Java EE se ocupe del resto del código de plomería.

Locally Managed Persistence

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="ProductPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="...">
      <property name="javax.persistence.jdbc.user" value="...">
      <property name="javax.persistence.jdbc.driver" value="...">
      <property name="javax.persistence.jdbc.password" value="...">
    </properties>
  </persistence-unit>
</persistence>
```

Locally Managed (Java SE) deployment:

- Entity Manager is acquired from EntityManagerFactory.
- Transactions are controlled programmatically.

❖ Transaction management is covered in the lesson titled "Implementing Business Logic by using EJBs".

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ProductPU");
EntityManager em = emf.createEntityManager();
try {
    em.getTransaction().begin();
    em.persist(...);
    em.merge(...);
    em.remove(...);
    em.getTransaction().commit();
} catch (Exception e) {
    em.getTransaction().rollback();
}
```



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

In a Container Managed Persistence scenario, Persistence provider could be assumed as system default. However, in a Locally Managed scenario Persistence provider needs to be specified.

Also container may provide access to preconfigured datasource; otherwise, database connection properties must be specified.

Example of the persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="ProductJPAPU" transaction-
type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url"
```

```

value="jdbc:derby://localhost:1527/ProductDB"/>
    <property name="javax.persistence.jdbc.user" value="oracle"/>
    <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="javax.persistence.jdbc.password" value="welcome1"/>
    <property name="javax.persistence.schema-
generation.database.action" value="create"/>
  </properties>
</persistence-unit>
</persistence>

```

The UserTransaction interface defines the methods that allow an application to explicitly manage transaction boundaries.

It defines operations to:

`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.

`setTransactionTimeout(int seconds)` : Modify the timeout value that is associated with transactions started by the current thread with the `begin` method.

La otra opción es de tipo transacción RECURSO LOCAL. Y eso es lo que debe hacer si desea ejecutar el código fuera del contenedor Java EE. Entonces no podrá beneficiarse de cosas como las fuentes de datos proporcionadas por el servidor Java EE, por lo que debe configurar todo el proveedor y la conectividad con la base de datos, como URL, nombre de usuario, contraseña. Todo lo que espera que el servidor Java EE configure para usted, debe configurarlo en el archivo `persistence.xml`, porque está ejecutando este código fuera del servidor Java EE. Y luego puede comenzar a administrar sus transacciones.

Ahora, la opción JTA, en la página anterior, implicaba que las transacciones eran administradas por el contenedor Java EE. La opción RECURSO LOCAL implica que debe administrar las transacciones mediante programación, comenzarlas, realizar cualquier acción, confirmar, retroceder al final, depende de lo que desee hacer.

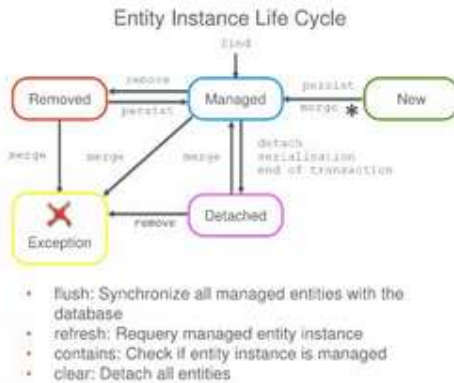
Además, hay una forma ligeramente diferente de inicializar Entity Manager. En una página anterior, permítanme volver muy rápidamente, el Entity Manager se inicializó simplemente con esta anotación de `PersistenceContext`. Eso fue todo. El resto se hizo por contenedor.

En el entorno Java SE, Entity Manager se inicializa con un objeto `EntityManagerFactory`. Tienes que crear una fábrica de persistencia con ese nombre particular de unidad de persistencia. Y luego esa fábrica tiene un método llamado `createEntityManager` que en realidad lo inicializa.

Entonces, en un entorno Java SE, es un poco más extenso. Pero ahí tienes. En este caso, no tiene el soporte del contenedor Java EE.

Se cubrirá más sobre la gestión de transacciones con respecto al entorno gestionado por contenedor en el capítulo que trata sobre Enterprise Java Beans, implementando la lógica empresarial con Enterprise Java Beans. Hablaremos sobre lo que sucedió exactamente con las transacciones JTA en un servidor Java EE allí.

Entity Manager Operations



```

package demos.model;
import javax.persistence.*;
public class ProductManager {
    @PersistenceContext (unitName="ProductPU")
    private EntityManager em;
    public void doThings() {
        Product p1 = new Product(7, "Cookie", 1.99);
        em.persist(p1);
        Product p2 = em.find(Product.class, 2);
        p2.setPrice(2.99);
        em.remove(p2);
        em.detach(p1);
        p1.setPrice(2.99);
        Product p3 = em.merge(p1);

        em.flush();
        em.refresh(p1);
        boolean isManaged = em.contains(p1);
        em.clear();
    }
}
    
```

The life cycle of an entity is depicted in the slide. An entity can have one of several states:

- **New:** An entity object instance that has no persistent identity and is not yet associated with a persistence context (for example, when an Employee object is created by using the new keyword)
- **Managed:** An instance with a persistent identity that is currently associated with a persistence context
- **Detached:** An instance with a persistent identity that is not (or no longer) associated with a persistence context
 - Detached entities are still object instances.
 - They can be read and modified, but they are not persisted to the database.
 - A detached entity is an "offline" entity.
- **Removed:** An instance with a persistent identity that is associated with a persistence context and that will be removed from the database on transaction commit

Entities can become detached in a number of ways:

- When the transaction associated with a transaction-scoped persistence context commits
- If a stateful session bean with an extended context is removed, all of its managed entities become detached.
- If the `clear()` method is used on an entity manager, all entities associated with that entity manager become detached.
- If the `detach(entity)` method is used, the entity specified becomes detached.
- When a transaction rollback exception occurs, all entities in all of the persistent contexts associated with the transaction become detached.
- When an entity is serialized, the serialized form of the entity is detached.

Managed entities do not need merge to be updated.

Detached entities may accumulate changes, before they are merged.

* When the merge method is used on an entity that does not already exist, a new entity is created as a copy of the entity passed to the method. The entity returned from the method is managed:

```

Product p1 = new Product(7, "Cookie", 1.99);
Product p2 = em.merge(p1);
    
```

To locate an entity in the database, the EntityManager will locate a row in the database based on the primary key—the equivalent of a SQL SELECT statement.

```

Product p2 = em.find(Product.class, 2);
    
```

The find method uses the entity class that is passed in the first argument to determine what type to use for the primary key and for the return type. (Therefore, an extra cast is not required.)

When the call returns, the Product instance is a managed entity.

If no Product with the ID 2 is found, a null is returned.

Ahora, una vez que haya inyectado el Entity Manager, ¿qué hace con eso? Así que tienes el Entity Manager inicializado. Esta disponible.

Entity Manager gestiona el ciclo de vida de la entidad. Y estas son las fases del ciclo de vida de la entidad. Intentemos averiguar qué está pasando en este ejemplo de código.

Comencemos con este, Producto p1, nuevo Producto. Esto se conoce como un nuevo estado. Acaba de crear un nuevo objeto.

Este objeto aún no está vinculado a ningún registro de la base de datos. No lo hemos guardado en ningún lado. Es solo un objeto Java en memoria. Y ese será el caso hasta que lo persistamos. Cuando llama a la operación persistente, toma ese objeto y le dice al Entity Manager que desea guardarlo en la base de datos.

Además de guardarlo en la base de datos, Entity Manager hace algo más. En esta etapa, solo su código hace referencia a ese Producto 1. Su variable p1 hace referencia a ese producto.

Sin embargo, después de llamar al método persistente, Entity Manager ahora también hace referencia a ese producto. Tienes un puntero para ello. Entity Manager tiene un puntero hacia él.

Este producto pasa al estado administrado. Un área de memoria, conocida como contexto de persistencia, administrada por Entity Manager tiene una referencia a ese objeto en particular. Eso es lo que significa administrado.

¿De qué otra manera el objeto podría llegar al estado administrado? Llamas a un método find. Si consulta con Entity Manager un producto de la tabla, digamos que desea encontrar un producto con ID 2, algo así, como su clave principal, para encontrar el producto número 2. El objeto Java devuelto también está en estado administrado. No solo tiene una referencia a la variable p2, a ese producto, tiene la referencia, sino que Entity Manager también hace referencia al mismo objeto.

¿Qué significa ser administrado? Significa que si configuro la propiedad en el objeto p1 o p2, no importa ninguno de ellos, p1.setPrice, p2.setPrice, a quién le importa, inmediatamente provoca la actualización del registro de la base de datos. El Entity Manager simplemente sigue adelante y cambia ese registro por usted. Eso es lo que significa el estado administrado. Simplemente llama al método set y el registro cambia, no solo el objeto Java, sino el registro de la tabla real.

Además, cuando desee eliminar ese registro, puede eliminarlo, em.remove, Entity Manager remove. Y el método de eliminación lleva el objeto del estado administrado al estado eliminado, ya que es otro estado. Por supuesto, si intenta hacer algo con el objeto eliminado, no se actualizará en la base de datos.

Supongo que puedes persistirlo de nuevo, al igual que con el nuevo. Cuando creaste un nuevo objeto, podrías llamar a persistir. Y eso lo lleva al estado administrado. Si lo eliminó y llamó a persistir, en realidad también lo devolverá al estado administrado, para que pueda volver a insertar el objeto. Supongo que puedes hacer eso.

A continuación, puede mover el objeto del estado administrado al estado separado. Separar significa que le está pidiendo al Entity Manager que excluya ese objeto del contexto de persistencia. Entonces, todavía tiene un puntero que apunta al objeto p1 en esa etapa, pero Entity Manager no.

Entonces, ¿qué sucede cuando estableces su precio? Diga p1.setPrice, el registro no se actualiza en una tabla porque el Entity Manager no lo está rastreando. Lo separaste.

ESTÁ BIEN. Así que acabas de cambiar el objeto Java. El registro no ha cambiado. ¿Quieres actualizar el registro? Supongo que sí, eventualmente. Y puede llamar al método de combinación para hacerlo.

El método de combinación tomará los cambios que haya realizado en el objeto y los aplicará a la tabla subyacente, actualizará los registros reales. Por cierto, el método de combinación le devuelve otra referencia de un objeto administrado esta vez, por lo que lo vuelve a colocar en el estado administrado si es necesario.

¿Por qué separarías el objeto? Bueno, supongo que la respuesta es bastante obvia. Desea realizar una serie de cambios antes de intentar actualizar la tabla. Desea establecer diferentes atributos y luego actualizarlos todos de una vez. Tiene sentido.

Sin embargo, puede haber otras circunstancias en las que el objeto se desprenda. Por lo tanto, puede llamar al método de desconexión. Pero también, otras circunstancias incluyen el final de la transacción o la serialización del objeto. Si tomó ese objeto y lo serializó, o si comprometió o revirtió la transacción, el objeto pasará del estado administrado al estado separado. Es sólo mientras dure la transacción. Entonces se separa.

Si intenta eliminar un objeto separado, obtendrá un error. Si intenta fusionar un objeto eliminado, obtendrá un error. Si intenta fusionar un objeto administrado, obtendrá un error, porque recuerde que el objeto administrado se actualiza automáticamente de todos modos, entonces, ¿por qué fusionarlo?

Un par de otras cosas. Si llama a merge en un nuevo objeto, lo persistirá de todos modos. Pero no es recomendable. Se recomienda llamar a persistir. Es más como decir explícitamente que está insertando aquí en lugar de intentar una actualización.

Hay algunas operaciones que también puede realizar en grupos de entidades. Flush, por ejemplo, sincronizará todas las entidades administradas con la base de datos. Actualizar, lo opuesto a vaciar si lo desea, volverá a consultar las instancias de entidades administradas de la base de datos en la memoria. Entonces, tal vez haya algún otro cambio de transacción en ellos. Si desea recuperarlos de la base de datos en la memoria, vuelva a consultarlos.

contiene método, uno muy simple. Booleano, verdadero, falso, ¿se gestiona o no? Eso es lo que contiene Managed te dice. Recuerde, administrado significa que la referencia al objeto está presente dentro del contexto de persistencia. Entonces contiene el método dice, pregunta el contexto de persistencia, ¿tiene una referencia para este objeto? Si devuelve un verdadero, se gestiona. Si devuelve falso, se separa.

Y finalmente, método claro, es una operación de grupo que le permite separar todas las entidades del contexto de persistencia juntas, en lugar de llamar a la separación una por una. Es sólo una operación de grupo, supongo. ESTÁ BIEN. Así que esto es lo que espera que un Entity Manager haga por usted.

Locking and @Version

Locking Modes and Version attributes are designed to ensure data integrity and support different levels of concurrency.

```
package demo.db;  
import javax.persistence.*;  
@Entity  
@Table(name="PRODUCTS")  
public class Product implements Serializable {  
    @Id  
    @NotNull  
    private Integer id;  
    ...  
    @Version  
    private Integer version;  
    ...  
}
```

incremented every time
record is updated

Locking Modes

	Pessimistic		Optimistic
	Read	Write	
find entity	LOCK		
update or remove entity		LOCK	
post changes to database			LOCK



The example above assumes that two transactions attempt to update products 2 Cake price to 1.5 and 1.2. Only the transaction that posted changes first succeeds. Version column value is incremented and another transaction is prevented from saving changes.

LockModeType Enum defines locking mechanism supported by EntityManager

Possible values are:

- NONE: No lock will be performed.
- OPTIMISTIC
 - Optimistic lock places the lock on a record only when record is posted to the database.
 - Version values in the entity instance and in the database record are compared when record is updated.
 - If version value is not the same, it is assumed that record has been changed by another transaction. Current Transaction will rollback and OptimisticLockException will be thrown.
 - If version value matched, then version is incremented when the record is updated can be committed.
- OPTIMISTIC_FORCE_INCREMENT
 - Similar to the Optimistic mode
 - Version update that occurs even if the entity has not been updated
 - Useful when working with entities that form relationships; for example, changing version of the Order if you modified any of its Items

- **PESSIMISTIC_FORCE_INCREMENT:** Pessimistic write lock, with version update. This option is used to achieve consistent behavior when same tables are used in both pessimistic and optimistic locking strategies.
- **PESSIMISTIC_READ**
 - Depending on the database type, JPA provider may switch to **PESSIMISTIC_WRITE** mode instead.
 - Locks the record with read lock state when you read the record. This prevents other transactions from changing the record, but still allows them to read it
- **PESSIMISTIC_WRITE**
 - Locks the record, preventing other transaction from both reading and writing it
- **READ:** Synonymous with **OPTIMISTIC**
- **WRITE:** Synonymous with **OPTIMISTIC_FORCE_INCREMENT**

Version annotation specifies the version field or property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control.

Only a single Version property or field should be used per class; applications that use more than one Version property or field will not be portable.

The Version property should be mapped to the primary table for the entity class; applications that map the Version property to a table other than the primary table will not be portable.

The following types are supported for version properties: int, Integer, short, Short, long, Long, java.sql.Timestamp.

Note: Timestamp stratagem is not reliable.

Mencionamos las opciones de bloqueo y versión un poco antes en este capítulo. Y te prometí que habría más información al respecto. Así que aquí viene esta parte de más información.

Hay diferentes estrategias de bloqueo que podría usar, pesimista, optimista, lectura, escritura. Así que déjame darte algunas explicaciones. En primer lugar, el bloqueo pesimista significa que le gustaría bloquear el registro en la base de datos real.

Entonces, por ejemplo, un usuario intenta actualizar el registro. No desea confirmarlo todavía, pero desea ir y bloquear el registro. Así que tal vez hay una serie de cambios que le gustaría hacer y luego le gustaría aplicarlos todos o revertirlos todos, por lo que la confirmación y la reversión se realizarán en una fase posterior. Pero ya ha aplicado bloqueos a los registros subyacentes.

Aplicar un bloqueo al registro evitará que cualquier otra sesión actualice ese mismo registro junto con usted. Esto podría tener graves consecuencias cuando se trata de una aplicación cuyo cliente no está conectado con estado, con un protocolo de conectividad transitorio. Déjame explicar.

Imagine que está tratando con un cliente de navegador web. Así que abres el navegador, vas a algún sitio web, dices, me gustaría actualizar ese producto. Supongamos que la aplicación web detrás de ella usa un bloqueo pesimista. imaginemos

Estás diciendo que me gustaría actualizar el producto. Hiciste una actualización, pero no te comprometiste. El registro está bloqueado.

Pones la referencia a eso en la sesión HTTP para que puedas devolver la próxima llamada con otra solicitud y hacer otra cosa. Desafortunadamente, su usuario cerró el navegador, solo accidentalmente. Ay. O el cliente colapsó, lo que sea. Las cosas pasan.

Entonces, el cliente abre el navegador nuevamente, regresa al mismo sitio web y dice, bueno, ahí está ese registro en el que estaba trabajando. ¿Puedo continuar con eso? No. Está bloqueado por otras sesiones. Esa otra sesión es la sesión anterior del mismo cliente, que acaba de bloquearse a sí mismo. Quiero decir, ya sabes, la sesión terminará y el bloqueo finalmente se liberará y se revertirá.

¿Cuánto dura el tiempo de espera de su sesión? Pregúntele al administrador de su servidor. ¿20 minutos? ¿Así que te bloqueas durante 20 minutos para no hacer nada en ese sitio web con ese registro en el que trabajaste? Eso no es divertido, ¿sí?

Entonces, el otro enfoque... quiero decir, si tuvieras un cliente conectado permanentemente, eso no sería un problema. Si el cliente usó algo como el protocolo RMI para la conectividad, la conexión se interrumpe, eso es todo. Sabes exactamente lo que pasó. El cliente se ha ido. Puede revertir lo que quiera revertir.

Pero con el protocolo HTTP, no se puede saber. No sabes si el cliente se ha ido o no. Todos pueden retransmitir en un tiempo de espera. Entonces, el bloqueo pesimista con un cliente conectado transitoriamente podría ser un desastre.

Ahora bien, ¿cuál es la otra opción? Cerradura optimista. Bueno, se llama optimista porque no estamos buscando el registro en la base de datos. Esperamos que nadie más lo actualice.

Entonces está actualizando el registro, pero solo en la memoria de Java. En realidad, no estás haciendo nada con la tabla de la base de datos. Eso significa que otro usuario simultáneamente con usted podría ir e intentar una actualización en ese registro, ¿verdad? Oh, eso será un desastre. Podemos comprometer fácilmente la integridad de la base de datos.

Para evitar ese compromiso de integridad, el bloqueo optimista a menudo se usa junto con un atributo de versión. Todo lo que necesita hacer es crear un atributo en su entidad, obviamente mapeado con la columna apropiada. Así que hay una columna de versión en la tabla. Hay un atributo de versión en la entidad. Y está anotado como versión.

ESTÁ BIEN. Así que esa es esa versión. Ahí tienes

Supongamos que dos usuarios de twp recuperaron simultáneamente el mismo registro. Hasta aquí todo bien. Ellos pueden hacerlo. No hay problema.

Entonces, está este usuario que recuperó el pastel y ese usuario que recuperó el pastel. Y un usuario dice, bueno, cambiemos el precio de ese pastel a 1.5. Y el otro usuario dice, cambiemos el precio de la torta a 1.2. ESTÁ BIEN.

Si es un bloqueo pesimista, solo uno de ellos podrá realizar el cambio, porque bloqueará la base de datos. Pero con un bloqueo optimista, ambos pueden realizar el cambio en la memoria de Java. Nada les impide hacer ese cambio, porque no están bloqueando el registro en la tabla en esa etapa.

Entonces cambian ese precio. Y ahora, quien vaya a confirmar la transacción primero publica los cambios en la base de datos y confirma la transacción. Cualquiera de estos muchachos que lo haga primero ganará esa competencia. Así que supongamos que ese es el tipo que cambió a 1.2. Supongamos que es ese tipo el que hace la publicación primero.

Así que publica el registro en la base de datos. Y lo que sucede en esa etapa es que se incrementará el número de versión, que era 1. Se convertirá en 2. Y ya ves lo que está pasando. Lo que hace Entity Manager es justo antes de publicar ese registro y antes de incrementar ese número, en realidad verifica si el número en la memoria de Java es el mismo que el número en la tabla.

Así que esto es 1 y en la tabla es 1. Lo va a incrementar a 2, pero primero verifica. Está bien. Entonces la transacción tiene éxito. Lo comete. El registro se convierte en 2.

Que pasa con el otro que lo actualizo a la 1.5. Intenta comparar su versión 1 y ya no coincide, porque ahora es 2. En la tabla que ya no es 1. Ahí tienes

Entonces, la segunda transacción obtendrá una excepción de bloqueo optimista que dice que otro usuario ha cambiado ese registro. Está bien. No anulamos la actualización de otra persona. Podemos volver a consultarlo. Podemos decidir qué queremos hacer al respecto.

Tal vez aún... después de volver a consultar, es posible que aún deseemos continuar con esa actualización o no. No sé. Depende Pero al menos no pisas los dedos de los pies de otra persona. No anulas ciegamente ese registro.

Y me temo que un bloqueo optimista es lo que tiene que hacer en las aplicaciones basadas en la web la mayor parte del tiempo, porque, como digo, un bloqueo pesimista podría causar un problema grave de concurrencia cuando su cliente usa una conexión transitoria, cuando está utilizando clientes conectados de forma transitoria, porque no sabe cuándo se desconectan. Y luego el bloqueo pesimista quedó colgando por un tiempo.

Así que otro punto importante que me gustaría hacer en general. Para hacer que la aplicación sea más escalable, intente escribir código que funcione con transacciones más cortas. No haga muchos bloqueos y luego lave muchas cosas de una sola vez. Mover transacciones más cortas significa que no tiene que mantener en la memoria de Java cosas que no están en la base de datos durante mucho tiempo. Y corre menos riesgos de desincronización o... a medida que avanza tratando de actualizar las cosas según corresponda, un bloqueo optimista sin duda podría ayudar.

Una cosa más. Atributo de versión, a qué se podría asignar en la tabla. Por lo general, un número entero, podría ser corto, podría ser un número entero largo. Podría ser la marca de tiempo. Todos los proveedores de JPA admiten la marca de tiempo para el número de versión.

Mi consejo no utilice la marca de tiempo. Mala idea. Adivina lo que sucede en el clúster.

Dos usuarios simultáneos en diferentes nodos de clúster pueden coincidir fácilmente con exactamente la misma marca de tiempo en milisegundos. Y sí, eso es generalmente lo que puede suceder. Entonces tendrás una situación muy extraña. ¿Quién va a ganar ese concurso? Y potencialmente, puede actualizar accidentalmente el registro de alguien sin saber que lo actualizó, solo porque sucede en el mismo milisegundo. Entonces, un número entero como versión y un número numérico como versión, un número simple, probablemente funcionará mejor. Es bastante sencillo de implementar.

ESTÁ BIEN. Así que buenas noticias, no necesitas escribir nada de ese código que acabo de explicar. Simplemente pega la anotación @Version. Y el bloqueo optimista es predeterminado de todos modos, así que ta-da.

Si desea anularlo y convertirlo en un bloqueo pesimista, considere el tipo de cliente. Si eso es apropiado, ¿por qué no? Tal vez tenga un cliente de escritorio con conexión permanente al servidor. Está bien. Usa una mirada pesimista entonces.

Changing Locking Mode

Optimistic Lock is a default option. Locking mode can be changed:

- For a specific Entity Instance in managed state using operations:
 - find
 - refresh
 - lock
- When defining a JPQL or SQL query programmatically or via annotation

```
@PersistenceContext
EntityManager em;
...
Product p1 = ...;

p1 = em.find(Products.class, 42, LockModeType.PESSIMISTIC_WRITE);

em.refresh(p1, LockModeType.OPTIMISTIC_FORCE_INCREMENT);

em.lock(p1, LockModeType.OPTIMISTIC);

Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);

@NamedQuery(name="lockProductQuery",
    query="SELECT p FROM Product p
    WHERE p.name LIKE :name",
    lockMode=PESSIMISTIC_READ)
```

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Así es como cambias el modo de bloqueo. Tan optimista es predeterminado. Puede cambiar el modo de bloqueo para una consulta en particular. Ver aquí, método de búsqueda. Y donde estamos ubicando el registro, decimos bloquear el registro inmediatamente.

Cuando actualizamos el registro, hay un método separado llamado bloqueo para que pueda llamarlo explícitamente para bloquear. Puede cambiar el modo de bloqueo en una consulta. Todavía no hemos cubierto las consultas JPQL. Lo haremos en un momento. Eso es lo próximo de lo que hablaremos. Así que estaremos cubriendo JPQL. Pero cuando ejecuta la consulta JPQL o la consulta SQL, también puede decir qué modo de bloqueo en particular le gustaría usar.

Java Persistence Query Language (JPQL)

Define JPQL statements

- Refer to Java Object names.
- Use bind parameters.
 - Named :name
 - Positional ?1
- Improve performance by using bulk actions instead of selecting, updating, or deleting individual entities one by one.
- They can be annotated or created programmatically.

```
@Entity
@Table(name="PRODUCTS")
@NamedQuery({
    @NamedQuery(name="Product.findByNameAndPrice",
        query="SELECT p.name, p.price FROM Product p
            WHERE p.bestBefore > :someDate
            AND p.price BETWEEN :minPrice AND :maxPrice"),
    @NamedQuery(name="Product.updateOld",
        query="UPDATE Product AS c SET p.price=p.price*0.5
            WHERE p.bestBefore < ?1"),
    @NamedQuery(name="Product.removeOld",
        query="DELETE FROM Product c
            WHERE p.bestBefore > :someDate")})
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    private String name;
    private BigDecimal price;
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    private LocalDate bestBefore;
    ...
}
```



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

The Java Persistence Query Language is a query specification language for dynamic queries and for static queries expressed through metadata. JPQL can be compiled to a target language (such as SQL) of a database or other persistent store. This feature enables the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, queries can be optimized as well as portable.

JPA supports two methods for constructing queries:

- Query languages, such as the Java Persistence Query Language (JPQL)
- Native SQL

JPQL is a database-independent query language that operates over the logical entity model rather than the physical data model.

JPQL and Native SQL Statements can be constructed via annotations or programmatically with the help of Criteria API.

All JPQL statements reference Java object and attribute names, rather than database names.

SELECT can also be used to return scalar data using a scalar expression or function, as in the example, `SELECT p.price*0.9 FROM Product p`.

UPDATE and DELETE statements can greatly improve performance, when bulk actions are required.

Note that the type of operand depends on the expression operation:

- =, >, >=, <, <=, <>: Arithmetic
- [NOT] BETWEEN: Arithmetic, string, date, time
- [NOT] LIKE: String
- [NOT] IN: All types
- IS [NOT] EMPTY: Collection
- [NOT] MEMBER OF: Collection
- [NOT] LIKE: String

Note: The LIKE operator operates on a string expression. The string expression must have a string value. The pattern value is a string literal or a string-valued input parameter in which an underscore (_) stands for any single character, a percent (%) character stands for any sequence of characters including an empty sequence, and all other characters stand for themselves.

Echemos un vistazo al lenguaje de consulta de persistencia de Java, JPQL. Se parece mucho a SQL, pero no lo es exactamente. Verá, no se refiere a tablas y columnas. Se refiere a objetos de Java para que el producto, el nombre o el precio no sean nombres de tablas y columnas. Son nombres de entidades y atributos de Java.

Y a qué tablas y columnas se traducirá la consulta JPQL decidirá a través de las asignaciones de las entidades JPA para las asignaciones relacionales de objetos a la tabla subyacente. Eso se traducirá eventualmente a algún código SQL basado en las anotaciones JPA. Ahora, podría usar una variedad de declaraciones diferentes aquí, como SELECT, cláusulas WHERE, GROUP BY, ORDER BY, cualquier cosa realmente, casi todos los operadores SQL habituales.

Usas parámetros. Hay dos estilos de parámetros, parámetro con nombre y parámetro posicional. Los parámetros posicionales tendrían un signo de interrogación, 1, 2, 3, 4, enumeraciones de posición. Y los parámetros con nombre tendrían nombres aquí.

Además, me gustaría señalar que además de escribir sentencias JPQL SELECT como en el primer ejemplo, también puede usar JPQL para crear, actualizar y eliminar sentencias. Piensa en ello de esta manera. Si tiene varias entidades que desea actualizar, ¿quiere ir y fusionarlas una por una? Eso hará que un Entity Manager realice declaraciones de actualización separadas en cada uno de ellos. ¿Cuántos viajes de ida y vuelta en la red van a ser?

O eliminar. ¿Qué ocurre si desea realizar una actualización masiva o eliminar varios registros de una sola vez? Seguramente la declaración JPQL será más eficiente que una actualización minuciosa registro por registro con fusionar o eliminar. Así que las acciones de grupo, bastante útiles.

Sin embargo, una cosa. ¿Dime que piensas? Atributo de versión, ¿se actualizará a más 1 cuando ejecute la actualización de JPQL?

Lo siento. Tiene que hacer la versión más 1 usted mismo como parte de la declaración de actualización. Cuando estás llamando a métodos de combinación o cuando estableces atributos, sí, esa versión de bloqueo optimista funciona de maravilla. Cuando estás haciendo acciones masivas, no lo es. Entonces, debe actualizar explícitamente el atributo de versión en ese caso.

Está bien. Pero aparte de eso, es bastante sencillo. Entonces, acciones masivas, consultas, actualizaciones, eliminaciones disponibles con estas declaraciones JPQL. Hay más, y te mostraré algunos ejemplos más.

Using JPQL with non-Entity classes

JPQL statements can query data into non-Entity objects

- Use NEW operator to create instances of object that is not an Entity
- Populate each of these objects with data returned by the query
- Often can be used to represent information from different Entities that needs to be joined

```
@NamedQuery(name="FindProductOffers",
    query="SELECT NEW demon.ProductOffer(p.id, p.name, p.price, o.discount)
    FROM Product p JOIN p.offer o")

package demon;
public class ProductOffer {
    private int id;
    private String name;
    private float price;
    private float discount;
    public ProductOffer(int id, String name, float price, float discount) {...}
}
```

Note that NEW operator requires a relevant constructor name to be fully qualified.

If you use NEW operator to select an Entity then the resulting Entity object would be in a "new" state. This coding technique may be a bit controversial, since a simple object instantiation could have been used to achieve same result.

Este es inusual. Echa un vistazo a este operador, NUEVO. Lo que le permite hacer, le permite consultar cosas de diferentes tablas tal vez, desde diferentes objetos de almacenamiento de bases de datos, en la no entidad.

Mira la Oferta de Producto. No tiene una anotación de entidad. Una consulta JPQL normal implica que lo que está recuperando es una entidad. En este caso, tiene Producto de entidad y tiene oferta de entidad, sean cuales sean, no lo sé, así que cualquier cosa que tenga.

Pero no tiene la entidad de oferta de producto. Lo tienes como un objeto Java de acuerdo. Pero no está asignado a ninguna tabla específica. Y esa es la razón. Es posible que desee consultar cosas de la base de datos en varias combinaciones, no necesariamente exactamente de una manera en la que mapeó sus entidades. Por lo tanto, se requeriría cierta flexibilidad.

Si llama al operador SELECCIONAR NUEVO y opera en la entidad aquí, bueno, simplemente crearán una cantidad de entidades en un nuevo estado, lo que me pregunto por qué quiere hacer eso, porque ¿qué estaba mal con solo consultar? ¿a ellos? ¿Por qué desea crear-- consultarlos como nuevos registros? No sé. Tal vez desee crear un nuevo registro como una copia del existente o algo así, tal vez. Pero de lo contrario, probablemente lo usaría para seleccionar objetos que no son entidades.

Executing JPQL Statements

Executing named queries


- If query returns multiple records, use:
`.getResultList();`
- If query returns a single record, use:
`.getSingleResult();`
- To update or delete records, use:
`.executeUpdate();`

```
@PersistenceContext
public EntityManager em;
...
List<Product> products =
    em.createNamedQuery("Product.findByNameAndPrice")
        .setParameter("productName", "LocalDate.now()")
        .setParameter("minPrice", 1)
        .setParameter("maxPrice", 10)
        .getResultList();
```

JPQL statements can be defined dynamically.

- Use the `createQuery` method to define TypedQuery Object.

```
TypedQuery<Product> query =
    em.createQuery("SELECT p.name FROM Product p
        WHERE p.price > :value",
        Product.class);
query.setParameter("value", 5);
List<Product> products = query.getResultList();
```



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

Use `StoredProcedureQuery` object to execute database stored procedures.

You may use Native SQL statements instead of JPQL with

`@NativeQuery` annotation or `em.createNativeQuery` method.

However, because native queries use direct database-specific SQL, they are less portable than JPQL statements.

Query can be conducted dynamically using JPQL statements, or Query Criteria API. Here is an example of a query that retrieves product with price greater than 5 and name starting with "C":

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = builder.createQuery(Product.class);
Root<Product> product = cq.from(Product.class);
cq.where(builder.gt(product.get(Product_.price), 5).and(builder.like(product.get(Product_.name), "C%")));
Query q = em.createQuery(cq);
q.getResultList();
```

For more information please reference <https://docs.oracle.com/javaee/7/tutorial/persistence-criteria003.htm>

Query object can be used to implement pagination - i.e. getting results in sets of records. Here is an example code that retrieves first 10 records:

```
Query q = em.createNamedQuery("Product.findProducts");
q.setFirstResult(1);
q.setMaxResult(10);
q.getResultList();
```

Está bien. Entonces puede ejecutar las declaraciones JPQL. ¿Puedo regresar por un segundo? Cuando está creando las declaraciones JPQL, les está dando nombres, ¿verdad?

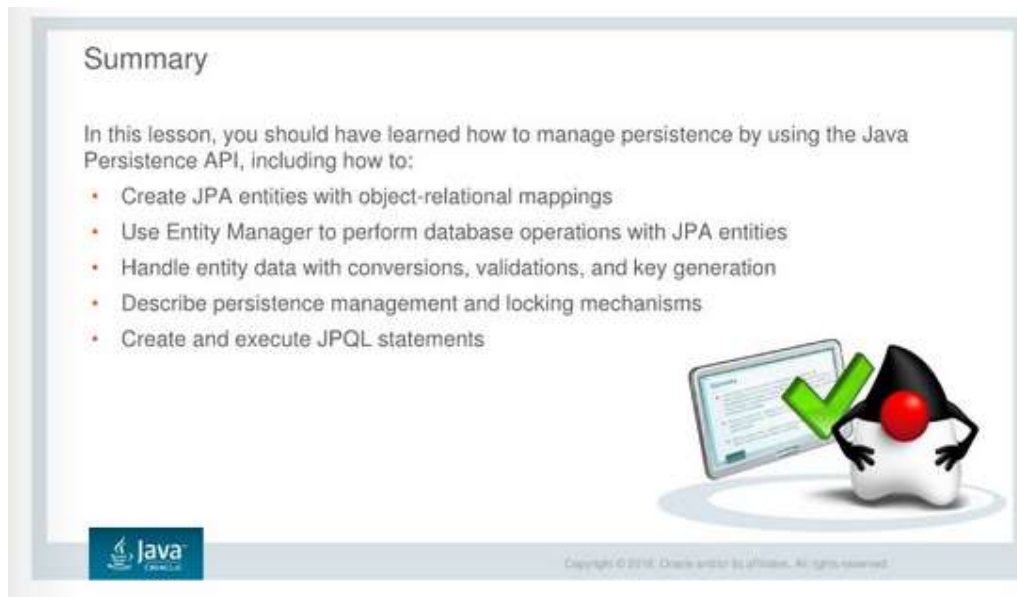
Este, ese es un nombre, ¿verdad? Y si regresas, sí, todos tienen nombres, consultas con nombre. Estos son los nombres. Podrías llamarlos como quieras. Es solo un string.

Entonces, cuando desea ejecutar una consulta con ese nombre, ¿qué hace? Entity Manager, `createNamedQuery`. Básicamente, este método recupera la definición de consulta nombrada, establece parámetros en ella, lo que necesite y ejecuta. La ejecución se puede realizar obteniendo una lista de resultados si cree que le devolverá más de

un registro. Si cree que va a devolver un solo registro, llame a `getSingleResult`. Y si no es SELECCIONAR, sino ACTUALIZAR o ELIMINAR, entonces puede ejecutar la actualización. Ahí tienes

Además, puede crear consultas JPQL sobre la marcha, dinámicamente. Entonces, este segundo ejemplo aquí no está usando una consulta nombrada precodificada a través de la anotación, sino simplemente creándola sobre la marcha. Hay toda una API con la que puedes hacerlo. Se llama Generador de consultas. Por lo tanto, podría usar el tipo de notación punto punto punto para crear una cláusula SELECT, una cláusula WHERE, una cláusula FROM, latido a latido construya esta declaración, o simplemente construya como una declaración JPQL como una cadena, de la forma que desee hacerlo.

Y luego lo mismo. Le gustaría establecer parámetros y ejecutar la consulta. ESTÁ BIEN. Estos son los conceptos básicos de JPQL.



Y creo que eso es todo para esta sesión en particular. Así que hemos visto cómo tomamos una entidad y la asignamos a los objetos de la base de datos subyacente. Examinamos el ciclo de vida de la entidad con respecto a cómo es administrado por el administrador de la entidad dentro del contexto de persistencia, todos estos estados diferentes, nuevo, administrado, separado. ¿Cómo manejamos las validaciones, la generación de claves, cuáles son las opciones de bloqueo y cómo podemos ejecutar declaraciones JPQL?

Hay mucho más sobre este tema en el Apéndice, donde hablamos sobre validaciones personalizadas más allá de las restricciones de validación estándar y otros tipos de asignaciones. En el Apéndice, encontraremos información sobre asignaciones de relación de entidades, claves compuestas, asignación a varias tablas, así que más tipos de casos de asignación avanzada.

Pero independientemente de la complejidad del caso de mapeo, lo que hizo el capítulo fue mostrar todos los conceptos importantes de cómo funciona esta API. El mapeo adicional se puede tomar del Apéndice más adelante. Primero debe comprender conceptualmente qué espera de la API de JPA, qué tipo de funcionalidad espera que cubra.

Por último, la práctica para este capítulo en particular, se le pedirá que cree una aplicación Java que se ejecute en un cliente Java SE, no en Java EE, que administre la entidad del producto. Asigne eso a la tabla de la base de datos subyacente, generando claves a partir de la secuencia, aplicando convertidores de datos, aplicando validadores, manejando errores y transacciones. En un capítulo posterior, ese mismo código JPA se moverá al entorno Java EE, donde será manejado por algunos Enterprise Java Beans. Pero primero debemos cubrir Enterprise Java Beans. ESTÁ BIEN.