**Hibernate JPA Cascade Types**

JPA Cascade Types control how state changes are cascaded from parent objects to child objects

* PERSIST – save operations will cascade to related entities
* MERGE - related entities are merged when the owning entity is merged
* REFRESH – related entities are refreshed when the owning entity is refreshed
* REMOVE – removes all related entities when the owning entity is deleted
* DETACH – detaches all related entities if a manual detach occurs
* ALL – applies all the above
* default – no cascade

**Embeddable Types**

* these are used to define a common set of properties
* for example, an order might have a billing address and a shipping address
* an embeddable type could be used for the address properties
* JPA/Hibernate support embeddable types

**Timestamps**

* often a best practice to use create and update timestamps on your entities for audit purposes
* JPA supports @PrePersist and @PreUpdate which can be used to support audit timeshamps via JPA lifecycle callbacks
* Hibernate provides @CreationTimestamp and @UpdateTimestamp – they require a little less logic by dev

-To-Many : Lazy Load

-To-One : Eager

***@Lob*** *–* Large Object

* CLOB – Character Large Object stores large text data (over 256 chars)
* BLOB – Binary Large Object is for storing binary data like image, audio, or video

***@Enumerated –*** for mapping an enum value to and from its database representation

* String
* ordinal – drawback is that the order must be kept intact, if you add an option it must be to the end of the enum list or prior data will be changed

**Hiberate DDL Auto**

* DDL – Data Definition Language : SQL language for the actual data structure like defining tables and relationships
* DML – Data Manipulation Language : SQL language for running queries like inserts, updates, deletes
* Hibernate property is set by the Spring property spring.jpa.hibernate.ddl-auto
* options are- none, validate, update, create, create-drop
* Spring Boot will use create-drop for embedded dbs (hsql, h2, herby) or none

Initializing

* Hibernate
  + data can be loaded from import.sql
    - hibernate feature
    - must be on root of class path
    - only executed if Hibernate’s ddl-auto property is set to create or create-drop
* Springs JDBC
  + spring’s DataSource initializer via Spring Boot will by default load schema.sql and data.sql from the classpath root
  + Spring Boot will also load form schema-${platform}.sql data-${platform}.sql
    - must set spring.datasource.platform
  + may conflict with Hibernate’s DDL Auto property
    - should use setting of ‘none’ or ‘validate’

**Mapping Strategies**

relational dbs don’t have a straightforward way to map class hierarchies onto db tables

hibernate wraps JPA

* MappedSuperclass- the parent classes can’t be a @Entity
  + inheritance is only evident in the class, but not the entity model (or db)
  + class will not have @Entity tag
    - @MappedSuperclass
  + ancestors cannot contain associations with other entities
  + the simplest approach but you can NOT use polymorphic queries that select all of the tree, nor could you use the relationships of the super class
    - so you can use relationships between subclasses but you CANT select all of the item (because that includes the super class) or use the super’s relationships in any other way
* Single Table – the entities from different classes with a common ancestor are placed in a single table
  + creates one table for each class hierarchy
  + default strategy
  + best performance and efficiency for polymorphic queries
  + define the strategy we want by adding the @Inheritance annotation to the super class
    - @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
  + because they all share one table, they have null values for members they do not have, thus you cannot use ***not null*** constraints in any of the involved classes which aren’t universal, this can create data integrity issues
  + BEST when you require the best performance and need to use polymorphic queries and relationships
* Joined Table – each class has its table and querying a subclass entity requires joining the tables
  + each class in the hierarchy is mapped to its table including abstract base class
  + the only column present in ALL tables is the identifier which will be used for joins
  + tables of the subclasses are much smaller than the table-per-class strategy because they hold only the columns specific for the mapped entity class and a primary key
    - the superclass contains columns for all shared entity attributes
  + also uses @Inheritance tag in the super class
    - @Inheritance(strategy = InheritanceType.JOINED)
  + sub classes will have foreign key of super class’s @ID, can customize join column in sub class
    - @PrimaryKeyJoinColumn(name=”petID”)
  + each query of a subclass requires a join which increases the complexity but it also allows you to use *not null* constraints on subclass attributes
  + disadvantage is that retrieving entities requires joins which lowers performance
  + BEST when data consistency is more important than performance and you need polymorphic queries and relationships
* Table-per-Class – all the properties of a class are in its own table
  + this mapping allows polymorphic queries and to define relationships to the super class but the table structure adds a lot of complexity to polymorphic queries
  + similar to MappedSuperclass except the parent classes is also an entity, allowing associations and polymorphic queries as a result
    - @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
  + similar to not using inheritance except the base class will return all the sub-class records too
  + BEST when you need very little polymorphic queries or relationships

Single Table

Discriminator Values – since the records for all entities will be in the same table, Hibernate needs a way to differentiate between them. By default, this is done through a discriminator column called DTYPE

* @Entity(name="products")
* @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
* @DiscriminatorColumn(name="product\_type", discriminatorType = DiscriminatorType.INTEGER)
  + in super class, this tells hibernate that all of its children will have a unique Integer value to differentiate between the columns (like id)
* public class MyProduct {
* // ...
* } SUBCLASS
  + @Entity
  + @DiscriminatorValue("1")
  + public class Book extends MyProduct {
  + // ...

}

@DiscriminatorValue(“null”) means that any row without a discriminator value will be mapped to the entity class with this annotation

@DiscriminatorValue(“not null”) – any row with a discriminator value not matching any of the ones associated with entity definitions will be mapped to the class with this annotation

entity.java

* @Id
* @GeneratedValue(strategy = GenerationType.AUTO)
* @Column(name=”id”, updatable=false, nullable=false)
* protected Long id;
* @Version
* @Column(name = “version”)
* @NonNull private int version;
* @ManyToMany
* @JoinTable(name=”PublicationAuthor”, joinColumns={ @JoinColumn(name= “publicationId”, referencedColumnName=”id”)}, inverseJoinColumns= {@JoinColumn(name=”authorId”, referencedColumnName=”id”)})
* private Set authors = new HashSet();
* @Column
* @Temporal(TemporalType.DATE)
* private Date publishingDate;

DB Filler

* @PostConstruct – this method must be invoked after the context is fully loaded
* @Transactional – keeps the entire method execution inside a db transaction
  + a transaction is a unit of work that you want treated as a whole, it has to happen in full or not at all
  + a transaction is a way of representing a state change and they have 4 properties
    - atomic –the entire transaction is committed at the same time (if committed)
    - consistent – the change can only happen if it is valid, any attempt to commit an invalid change will fail, leaving the system in its previous state
    - isolated – no one else sees any part of the transaction until its committed
    - durable – once the change has happened, no further action is needed
* application.properties file
  + spring.jpa.show-sql=true
  + spring.jpa.generate-ddl=true

Enterprise Beans-

* Session Beans
  + encapsulates business logic that can be invoked programmatically by a client
  + can be invoked locally by another class in the JVM or remotely over the network
  + the bean performs the task for the client, abstracting its complexity
  + lifecycle of a session bean is managed by the EJB container
    - stateless
      * shared by multiple clients
      * fast and easily managed by the container
      * may be singletons
      * devs are responsible for ensuring they are thread safe
    - stateful
      * represent a client’s state
      * unique to each client
      * destroyed when the client terminates
    - singleton
      * designed for when state must be shared across all clients
      * instantiated once per application and exists for the lifecycle of the app
      * devs must ensure thread safe
* writing an EJB class is just a matter of adding the annotation (above the class)
  + @Stateless
  + @Stateful
  + @Singleton
* Message Driven Beans
  + allows you to process messages asynchronously
  + normally acts as a JMS message listener (like an event listener but receives JMS messages instead of events)
  + they are event-driven, not invoked by a client
  + @MessageDriven(mappedName=”jms/TestQueue”)
  + public class TestMessageDrivenBean implements MessageListener{
  + to invoke the methods of an EJB locally, the bean can be injected in any managed class running in the container
    - @EJB
    - TestStatelessEjb testStatelessEjb;

**Hibernate Integration**

the Spring bean configuration file is the key for integration of spring and hibernate

@Bean

public LocalSessionFactoryBean sessionFactory(){

LocalSessionFactorBean sF = new LocalSessionFactoryBean();

sf.setDataSource(restDataSource());

sf.setPackagesToScan(new String[] { "org.baeldung.spring.persistence.model" });

sf.setHibernateProperties(hibernateProperties());

return sf; }

@Bean

public DataSource dataSource(){

BasicDataSource dataSource = new BasicDataSource();

dataSource.set/DriverClassName/Url/Username/Password // ie setUsername(“sa”);

return dataSource;

}

dependencies

• spring-orm dependency is required for hibernate integration

• spring-context and spring-tx for core Spring functionalities

• hibernate-entitymanager and hibernate-core dependencies for Hibernate framework

public class PersonDAOImp implements PersonDAO {

private SessionFactory sF;

public void setSessionFactory()

two ways we can provide db connection details to Hibernate

• by passing in everything in hibernateProperties

• creating a DataSource and then passing it to hibernate

Spring ORM used to provide two classes-

org.springframework.orm.hibernate3.LocalSessionFactoryBean //XML

org.springframework.orm.hibernate3.annotation.AnnotatoinSessionFactoryBean

now its just-

**org.springframework.orm.hibernate4.LocalSessionFactoryBean**

**Spring Boot EE Security**

password: secret

**Java EE Security**

Application-Layer Security

* pros-
  + security is unique suited to the needs of the application
  + security is fine grained with application-specific settings
* cons-
  + the app is dependent on security attributes that are not transferable
  + support for multiple protocols make it vulnerable

Transport Level Security

* the client and server agree on an appropriate algorithm
* a key is exchanged using public-key encryption and certificate-based authentication
* a symmetric cipher is used during the information exchange
* pros-
  + relatively simple, well understood, standard technology
  + applies to both a message body and its attachments
* cons-
  + it is tightly coupled with the transport-layer protocol
  + represent an all-or-nothing approach to security. you cannot selectively apply security to portions of the message
  + protection is transient. the message is protected only while in transit as protection is removed automatically by the endpoint
  + not an end-to-end solution, simply point-to-point

Message Level Security

* pros:
* Security stays with the message over all hops and after the message arrives at its destination.
* Security can be selectively applied to different portions of a message and, if using XML Web Services Security, to attachments.
* Message security can be used with intermediaries over multiple hops.
* Message security is independent of the application environment or transport protocol.
  + cons-
  + complex and adds more overhead to processing

@ServletSecurity

@HttpConstraint

@HttpMethodConstraint

Simple Application Walkthrough

1. Initial Request - client requests the main application URL, web server detects unauthenticated request and invokes the appropriate authentication mechanism
2. Initial Authentication – web server returns a form that the web client uses to collect authentication data from the user. the web client forwards the authentication data to the web server where it is validated. if successful, the web server sets a credential for the user
3. URL Authorization – credential is used for future determinations of whether the user is authorized to access restricted resources. the web server consults the security policy for any requested resource to determine if user is allowed access. (the security policy is derived from annotations or from the deployment descriptor). this is done via the users Role (credential)
4. Fulfilling the Original Request – if the user is authorized, the web server returns the result of the original URL request
5. Invoking Enterprise Bean Business Methods – [user posts form data] the web page performs the remote method call to the enterprise bean using the user’s credential to establish a secure associate between the web page and the enterprise bean. the association is implemented as two related security contexts: one in the web server and one in the EJB container. The EJB container is responsible for enforcing access control on the EB method. The container consults the security policy to determine whether the user’s role is permitted access to the method. results in “is authorized” or “not authorized”

Java Persistence API

* JPA defines a set of concepts that can be implemented by any tool or framework
* is concerned with persistence
  + persistence- any mechanism by which Java objects outlive the application process that created them
* JPA specification lets you define which objects should be persisted and how those objects should be persisted in your Java applications
* Hibernate ORM is often combined with JPA and is a JPA implementation
* Object-Relational Mapping is a task that has been codified into a ORM layer by Hibernate
* **ORM layer is responsible for managing the conversion of classes and objects so that they can be stored in a relational database**
  + *by default, the name of the object being persisted becomes the table name, and its fields become columns*
* ORM layer is an adapter layer- it adapts the language of object graphs to the language of relational tables
* **configure a *datastore connector* to connect to your chosen database**
* **define the mapping between objects and your db, then invoke JPA to persist them**
* JDBC (Java Database Connectivity) – layer of abstraction that lets an app issue SQL commands without regard for the underlying db implementation
  + Connection conn = DriverManager.getConnection(myURL, “root”, “”)
  + String query = “ insert into users (id, name) value (?, ?);
  + PreparedStatement pS = conn.prepareStatement(query);
  + all this is easier with an ORM layer instead
* @Entity informs JPA that the class and its objects should be persisted
* @Id to specify the required primary key
* 4 kinds of entity relationships
  + one-to-many – lazy load
    - @OneToMany
    - @JoinColumn(name=”musicianId”) // tells JPA what column on the Performance table will map to the Musician identity
    - private List<Performance> performances = new ArrayList();
  + many-to-one - eager
  + many-to-many - lazy
  + one-to-one - eager
* common way to install JPA is to include a JPA provider in your project, and its driver

***JAVA***

**identifiers**

* are case sensitive
* must start with a letter, $ or connecting character (\_ or -)
* cannot start with number!
* after first char, can contain letters, numbers, connecting characters, or currency characters
* can’t be a keyword

**Date and Time**

LocalDateTime class

java.time.\*

Import java.time.LocalDateTime

Now() -- get current date and time

LocalDateTime date = LocalDateTime.now();

Returns year-mn-day-T-time

date.getDayofMonth()

date.getHour()

date.minusDays(2) –gives date of 2 days ago

//convert from gregorian calendar

LocalDateTime myDate = LocalDateTime.now()

JapaneseDate jDate = JapaneseDate.from(myDate);

Or- static method

JapaeseDate.now()

import Java.time.format.\*

printing myDate calls toString() and gives unformatted date

String formattedDate = myDate.format(DateTimeFormatter.ISO\_LOCAL\_DATE); returns 2020-9-3

String formattedTime = myDate.format(DateTimeFormatter.ISO\_LOCAL\_TIME); returns 8:15:30

ISO\_DATE\_TIME = default format

String myDate mediumString = myDate.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);

returns September 4, 2020, 8:15:30 p.m.

from() = obtains an instance from a temporal object

of() = obtains an instance from integers. Month can be an int or enum.

parse() = obtains an instance from a String

**LocalDate**

Import java.time.LocalDate

LocalDate myDate = LocalDate.now();

2020-9-20

LocalDate someDate = LocalDate.of(2018, 1, 20); returns 2018-01-20

String dec = “2018-12-31”

LocalDate lastDate = LocalDate.parse(dec); returns 2018-12-31

use parse() to convert String to LocalDate or LocalDateTime

**Java Memory**

2 types of java memory

stack(short term)- holds local variables, primitives or reference types

heaps(long term)- holds actual objects

String Constant Pool – special part of the heap just for Strings

local variables are stored in short term memory – stack

Stack is only used for the methods currently running

instance variables are in longer term memory- heap

Heap Space contains all created objects

value passed into a method is an argument

a variable that is defined in the method declaration is a parameter(declaration)

method signature- method name, type and order of parameters

* public int adds(int a, int b){
  + return a + b; }
* public main()
  + int x = 2;
  + int y = 3;
  + int sum = adds(x, y)

x and y are arguments

a and b are parameters

**Error Handling**

exceptions that are not handled are propagated all the way up the call stack, at which point the JVM outputs the exception and a stack trace

should try to throw specific exceptions where possible //as opposed to general Exception e

checked exception – an exception that the caller is forced to catch or rethrow. if not caught then the method signature needs to include a throws statement

Throwable class of type error – an exceptional condition that is external to the applicatoin

exceptions that a method can throw are shown in the throws section of its documentation in the Java API

* variables have a type
* references have a class

**Object Class**

parent of all Java classes

Methods

* toString() + hashCode()
* equals()
* getClass()
* clone()
* finalize()

**Class class**

everyone object in Java belongs to a certain class. the Class objects contains metadata about the class

* name
* package
* methods
* fields
* constructors
* annotations

**Class Loaders**

* Bootstrap or Primordial Class Loader – parent of all other class loaders.
  + it is written in native code (unlike others) and thus won’t show up as a Java class
  + it is mainly responsible for loading JDK internal classes, rt.jar and other core libraries located in $JAVA\_HOME/jre/lib directory
* Extension Class Loader - loads classes that are an extensions of the standard core Java classes
  + child of Bootstrap Class Loader
* Application or System Class Loader = loads our own files in the class path
  + child of Extension
* **Multithreaded Custom Class Loaders**
  + public class CustomClassLoader extends ClassLoader {
  + parallel capable class loader to avoid deadlocks
  + public Class<?> loadClass(String name, Boolean resolve) throws ClassNotFoundException
    - loadClass() method is responsible for loading the class given a name parameter
  + must override the findClass() method and possibly loadClass()
  + avoid deadlocks by-
    - ensure your custom class loader is multithread safe
      * decide upon an internal locking scheme
      * ensure that critical sections are safe for multiple threads
    - invoke ClassLoader’s static method – registerAsParallelCapable()
    - check that all class loader classes that this custom class extends also invokes the above method

**Annotations**

* Marker – mark a declaration. contain no members and consist no data @Override
* Single Value – contain only one member and allow a shorthand form of specifying the value of the member @TestAnnotation(“testing”)
* Full Annotations – consist of multiple data members, value, pairs @TestAnnotation(owner=”Rah”, value=”classic”)
* Repeating Annotations – can be applied to a single item more than once, requires @Repeatable, its value field specifies the container type
* Type Annotations – can be applied to any place where a type is being used
  + for doing enhanced compile time checking or enforcing rules and issue errors/warnings
    - enhance Java strong typing to reduce runtime errors (primary)
    - @NotNull String str
    - @Authenticated User user
    - @Positive int i
    - @Readonly Date date;
    - public class MyClass extends @ThreadSafe otherClass{
  + can generate Java source code that can effect the annotated code in some way
  + use reflection to customize the runtime behavior

**Default Methods**

concrete method in an interface that is not required to be implemented (but can be executed)

provides backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class

also called- defender methods and virtual extension methods

interface TestInterface

default void show(){

System.out.println(“Default method executed”)

**instanceof()**

* if (arrayItem instanceof String)

**final keyword**

* variables – a final variable can be set once and only once
* fields – a final field can also be set only once, by the constructor of the class
* methods – a final method cannot be overridden or hidden
* classes – a final class cannot be extended
* used in parameter
  + system creates a copy of the reference variable and passes the copy, protecting it from reassignment outside of the scope

**Integral literals**

* decimals (standard)
* octal (base 8)
* hexa-decimal (base 16)
* binary
  + prefixed with 0b or 0B
  + int x = 0b1111;

**Numeric Literals**

any number of underscore characters can appear anywhere between digits in a numerical literal. this enables the separation of groups of digits to improve readability

* long creditCardNumber = 1234\_5678\_3245\_3456L;
* int socialSecurity = 999\_99\_9999;
* float pi = 3.14\_15F;

you can place underscores only between digits.

* not at beginning or end
* not adjacent to a decimal point
* not prior to a F or L suffix
* …where a string of digits is expected

**Java Streams**

**Optional Class**

Optional<String> checkNull = Optional.ofNullable(words[5]);

to avoid NullPointerException

* isPresent() returns boolean
* orElse() returns a default value if no value is present
* ifPresent() performs an action if a value is present
* get()

must import java.util package

is a value-based class, use of identity operations (==) will have unpredictable results

**Spliterator<T> Interface**

used for traversing and partitioning sequences

base utility for Streams, especially parallel ones

the source of elements covered by a Spliterator could be an array, Collection, IO channel, or a generator function

* tryAdvance() – traverse elements individually
  + if operation (on next element) is successfully executed, then returns true (also performs the operation)
* forEachRemaining() – traverse elements in bulk
* trySplit() – partitions off some elements to be used in possibly-parallel operations
  + for parallel streaming, this method splits off data into smaller steps based on a chosen criteria (file size, number of lines, etc)
* characteristics
  + SIZED – capable of returning the number of its elements
  + SORTED
  + SUBSIZED – still SIZED after trySplit()
  + CONCURRENT – thread safe
  + DISTINCT – all keys are distinct
  + IMMUTABLE – if elements cant be structurally modified
  + NONNULL
  + ORDERED

**class Collectors**

for streaming

implementations of Collector

* List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
* Set<String> set = …collect(Collectors.toCollectiono(TreeSet::new)
* String joined = …collect(Collectors.joining(“, “));
* int total = …collect(Collectors.summingInt(Employee::getSalary)
* Map<Department, List<Employee>> byDept = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));

**volatile keyword**

* makes it thread safe
* every read and write of a volatile variable will be done from main memory (instead of cache)
* ensures that changes in one thread, also affects parallel threads

**ANNOTATIONS**

annotations are used to provide supplemental information. They start with @ and help to associate metadata to the program elements. Annotations do not change action of a compiled program but they can change the way a program is treated by a compiler.

1. Marker Annotations – contains no members or data. ex = @Override
2. Single Value Annotations – contain only one member and allow a shorthand form of specifying the value of the member (if the member is value). @Test(“testing”)
3. Full Annotations – consists of multiple data members/name, value, pairs. @Test(owner=”Ra”, value=”Man”).
4. Type Annotations – applied to any place where a type is being used. These are declared annotated with @Target annotation.
5. Repeating Annotations – annotations that can be applied to a single item more than once. These must be annotated with the @Repeatable annotation.