Hibernate and JPA

About me


Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

Where to Find The Code and Materials?

https://github.com/iproduct/course-hibernate

Hibernate Architecture
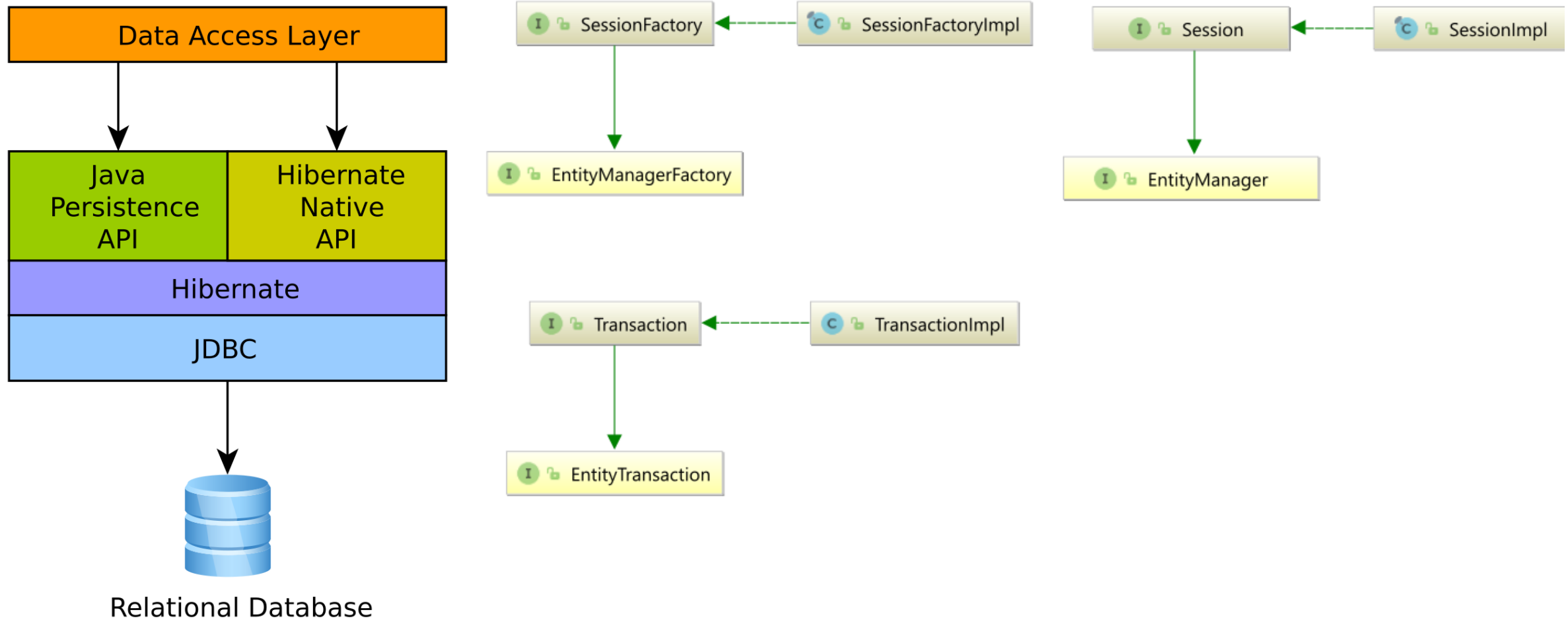
Hibernate Architecture - II

- **SessionFactory (org.hibernate.SessionFactory)** - a thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for org.hibernate.Session instances. The EntityManagerFactory is the JPA equivalent of a SessionFactory and basically, those two converge into the same SessionFactory implementation.

- Very expensive to create, so, for any given database, the application should have only one associated SessionFactory.

- The SessionFactory maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations, etc.

Hibernate Architecture - III

- **Session (org.hibernate.Session)** - a single-threaded, short-lived object conceptually modeling a "Unit of Work" (PoEAA). In JPA nomenclature, the Session is represented by an EntityManager.

- Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

- **Transaction (org.hibernate.Transaction)** - a single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

Domain Model

- Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose. With the coming of JPA, most of this information is now defined in a way that is portable across ORM/JPA providers using annotations (and/or standardized XML format).

- We usually prefer the JPA mappings where possible.

- For Hibernate mapping features not supported by JPA we will prefer Hibernate extension annotations.

Mapping Types

- Hibernate understands both the Java and JDBC representations of application data.

- Hibernate type – provides the ability to read/write this data from/to the database. It is an implementation of the org.hibernate.type.Type interface. Also describes various behavioral aspects of the Java type such as how to check for equality, how to clone values, etc.

- Hibernate type is **neither a Java type nor a SQL data type**. It provides information about mapping a Java type to an SQL type as well as how to persist and fetch a given Java type to and from a relational database.

- When you encounter the term type in discussions of Hibernate, it may refer to the Java type, the JDBC type, or the Hibernate type, depending on the context.

Hiberbate Native Bootstrapping

- There are two types of ServiceRegistry and they are hierarchical:

- **BootstrapServiceRegistry**, which has no parent and holds these three required services:
  - ClassLoaderService: allows Hibernate to interact with the ClassLoader of the various runtime environments
  - IntegratorService: controls the discovery and management of the Integrator service allowing third-party applications to integrate with Hibernate
  - StrategySelector: resolves implementations of various strategy contracts

- **StandardServiceRegistry**

Building BootstrapServiceRegistry

```
BootstrapServiceRegistry bootstrapServiceRegistry =

    new BootstrapServiceRegistryBuilder()

            .applyClassLoader()

            .applyIntegrator()

            .applyStrategySelector()

            .build();
```

Building **StandardServiceRegistry**

```java
BootstrapServiceRegistry bootstrapServiceRegistry =

    new BootstrapServiceRegistryBuilder().build();

StandardServiceRegistryBuilder standardServiceRegistryBuilder =

    new StandardServiceRegistryBuilder(bootstrapServiceRegistry);

StandardServiceRegistry standardServiceRegistry = standardServiceRegistryBuilder

    .configure()

    .build();
```

Building Metadata

```java
MetadataSources metadataSources =

    new MetadataSources(standardServiceRegistry);

metadataSources.addPackage( … );

metadataSources.addAnnotatedClass( … );

metadataSources.addResource( … )

Metadata metadata = metadataSources.buildMetadata();
```

Building and Using `SessionFactory` and `Session`

```java
// Get SessionFactory
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder().build();

// Get Session
Session session = sessionFactory.openSession();

// Persist entity
Contact contact = new Contact(1,
        new Name("Ivan", "Dimitrov", "Petrov"),
        "From work", new URL("http://ivan.petrov.me/"), true);
session.beginTransaction();
session.persist(contact);
session.getTransaction().commit();
session.close();
sessionFactory.close();
```

WEB-INF/applicationContext.xml -I

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx.xsd
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:property-placeholder location="classpath:jdbc.properties" />

    <context:component-scan base-package="org.iproduct.spring.webmvc.dao,
        org.iproduct.spring.webmvc.service"/>

    <context:annotation-config />
```

WEB-INF/applicationContext.xml -II

```xml
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>

<bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list><value>article.hbm.xml</value></list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=org.hibernate.dialect.HSQLDialect
            hibernate.hbm2ddl.auto=update
        </value>
    </property>
</bean>
```

WEB-INF/applicationContext.xml III

```xml
<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<tx:annotation-driven/>

</beans>
```

Hibernate Mapping: article.hbm.xml

```xml
<hibernate-mapping>
    <class name="org.iproduct.spring.webmvc.model.Article" table="ARTICLES">

        <meta attribute="class-description">
            This class contains the articles details.
        </meta>

        <id name="id" type="long" column="id">
            <generator class="identity"/>
        </id>

        <property name="title" column="title" type="string"/>
        <property name="content" column="content" type="string"/>
        <property name="createdDate" column="created_date" type="timestamp"/>
        <property name="pictureUrl" column="picture_url" type="string"/>

    </class>
</hibernate-mapping>
```

Java Persistence API (JPA)

- **JPA four main parts:**
  - ❑ Java Persistence API
  - ❑ JPA Query Language
  - ❑ Java Persistence Criteria API
  - ❑ Object to Relational Mapping (ORM) metadata

- **JPA Entity Classes**
  - ❑ persistent fields
  - ❑ persistent properties
- **@Entity** annotation

Advantages of Spring ORM

❖Easier testing

❖Common data access exceptions

❖General resource management

❖Integrated transaction management

Persistent Units

- Persistent Unit description in persistence.xml file:

–description

–provider

–jta-data-source

–non-jta-data-source

–mapping-file

–jar-file

–class

–exclude-unlisted-classes

–properties

Persistent Unit Example 1

```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="CustomerDBPU" transaction-type="JTA">
        <jta-data-source>jdbc/sample</jta-data-source>
        <class>customerdb.Customer</class>
        <class>customerdb.DiscountCode</class>
        <properties/>
    </persistence-unit>
</persistence>
```

Persistent Unit Example 2

```xml
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="invoicingPU" transaction-type="RESOURCE_LOCAL">
   <provider>oracle.toplink.essentials.PersistenceProvider</provider>
     <class>myinvoice.dbentities.Product</class>
     <class>myinvoice.dbentities.Invoice</class>
     <properties>
        <property name="javax.persistence.jdbc.user" value="root"/>
        <property name="javax.persistence.jdbc.password" value="root"/>
        <property name="javax.persistence.jdbc.url"
                    value="jdbc:mysql://localhost:3306/invoicing"/>
        <propery name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
     </properties>
  </persistence-unit>
</persistence>
```

JPA Setup in Spring

```xml
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

```xml
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

```xml
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="someDataSource"/>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
    </bean>
</beans>
```

Mapping Example

```java
@Entity(name = "Contact")
@Data
public class Contact {

    @Id
    private Integer id;

    @Embedded
    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    public Name getName() {
        return name;
    }
}
```

```java
@Embeddable
@Data
public class Name {

    private String firstName;

    private String middleName;

    private String lastName;
}
```

```sql
create table Contact
(
    id      integer not null,
    first   varchar(255),
    last    varchar(255),
    middle  varchar(255),
    notes   varchar(255),
    starred boolean not null,
    website varchar(255),
    primary key (id)
)
```

## Code First

DB Schema First

24

Value and Entity Types

- **Value types** - a value type is a piece of data that does not define its own lifecycle. It is, in effect, owned by an entity, which defines its lifecycle -> persistent attributes.
  - Basic types – e.g. in mapping the Contact table, all attributes except for name would be basic types.
  - Embeddable types - the name attribute is an example of an embeddable type, which is discussed in details in Embeddable types
  - Collection types - although not featured in the aforementioned example, collection types are also a distinct category among value types. Collection types are

- **Entity types** - by nature of their unique identifier, entities exist independently and define their own lifecycle, whereas values do not. Entities are domain model classes which correlate to rows in a database table, using a unique identifier.

Naming Strategies

Part of the mapping of an object model to the relational database is mapping names from the object model to the corresponding database names. Hibernate looks at this as 2-stage process:

1. The first stage is determining a proper logical name from the domain model mapping. A logical name can be either explicitly specified by the user (e.g., using @Column or @Table) or it can be implicitly determined by Hibernate through an ImplicitNamingStrategy contract.

2. Second is the resolving of this logical name to a physical name which is defined by the PhysicalNamingStrategy contract.

Naming Strategies
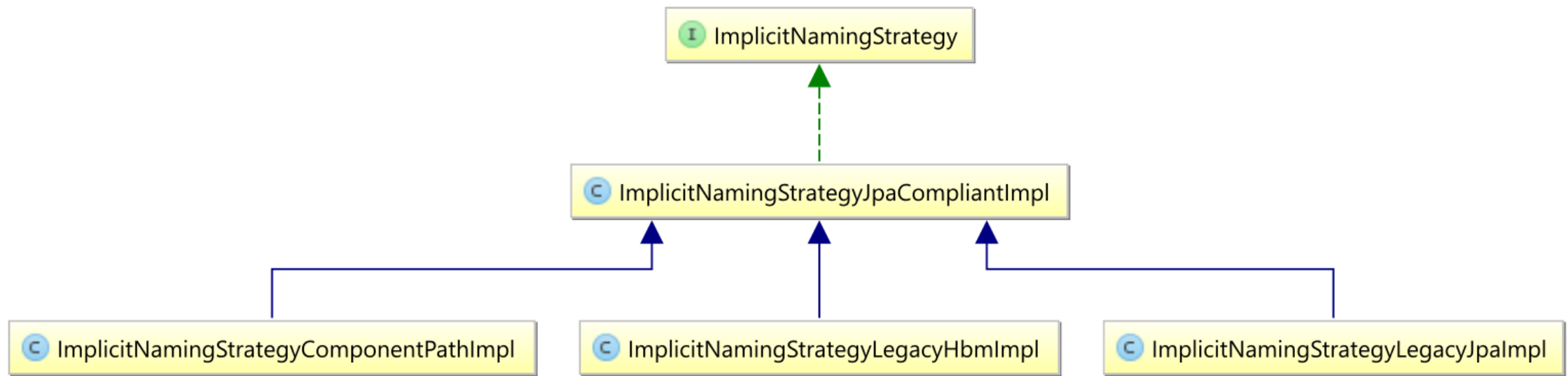
Implicite Naming Strategies

- Hibernate defines multiple ImplicitNamingStrategy implementations out-of-the-box. Applications are also free to plug in custom implementations.
- Applications can specify the implementation using the hibernate.implicit_naming_strategy configuration setting which accepts:
  - **default** – for org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl - an alias for jpa
  - **jpa** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl - the JPA 2.0 compliant naming strategy
  - **legacy-hbm** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyHbmImpl - compliant with the original Hibernate NamingStrategy
  - **legacy-jpa** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl - compliant with the legacy NamingStrategy developed for JPA 1.0, which was unfortunately unclear in many respects regarding implicit naming rules
  - **component-path** –org.hibernate.boot.model.naming.ImplicitNamingStrategyComponentPathImpl - mostly follows ImplicitNamingStrategyJpaCompliantImpl rules, except that it uses the full composite paths, as opposed to just the ending property part – e.g.
- By calling org.hibernate.boot.MetadataBuilder#applyImplicitNamingStrategy

Physical Naming Strategies

- There are multiple ways to specify the PhysicalNamingStrategy to use. First, applications can specify the implementation using the hibernate.physical_naming_strategy configuration setting which accepts:

  - reference to a Class that implements the org.hibernate.boot.model.naming.PhysicalNamingStrategy contract
  - FQN of a class that implements the org.hibernate.boot.model.naming.PhysicalNamingStrategy contract

- Secondly, applications and integrations can leverage org.hibernate.boot.MetadataBuilder#applyPhysicalNamingStrategy

Basic Types

- Basic value types usually map a single database column, to a single, non-aggregated Java type

- Internally Hibernate uses a registry of basic types when it needs to resolve a specific org.hibernate.type.Type:

https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#basic-provided

- The @Basic annotation - defines 2 attributes:
    - optional - boolean (defaults to true) - whether this attribute allows nulls.
    - fetch - FetchType (defaults to EAGER) - whether should be fetched eagerly or lazily. Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

Explicite Basic Types

```java
@Entity(name = "Product")
public class Product {
        @Id
        private Integer id; private String sku;

        @org.hibernate.annotations.Type( type = "nstring" )
        private String name;

        @org.hibernate.annotations.Type( type = "materialized_nclob" )
        private String description;
}
```

Basic Types

- Basic value types usually map a single database column, to a single, non-aggregated Java type

- Internally Hibernate uses a registry of basic types when it needs to resolve a specific org.hibernate.type.Type:

https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#basic-provided

- The @Basic annotation - defines 2 attributes:
  - optional - boolean (defaults to true) - whether this attribute allows nulls.
  - fetch - FetchType (defaults to EAGER) - whether should be fetched eagerly or lazily. Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

Embeddable Types

```java
@Embeddable
public class Publisher {

    private String name;


    @Embedded

    private Location location;

    public Publisher(String name, Location location) {
        this.name = name;
        this.location = location;
    }


    private Publisher() {}

    //Getters and setters are omitted for brevity

}
```

```java
@Embeddable
public class Location {

    private String country;

    private String city;

    public Location(String country, String city) {
        this.country = country;
        this.city = city;
    }

    private Location() {}

    //Getters and setters are omitted for brevity

}
```

Embeddable Types - II

```java
@Entity(name = "Book")
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    @Embedded
    private Publisher2 publisher;

    public Publisher2 getPublisher() {
        return publisher;
    }
}
```

```java
@Embeddable
@Data
class Publisher2 {

    @Column(name = "publisher_name")
    private String name;

    @Column(name = "publisher_country")
    private String country;
}
```

Embeddable Types – Attribute Overrides

```java
@Entity(name = "Book")
@AttributeOverrides({
    @AttributeOverride(
        name = "ebookPublisher.name",
        column = @Column(name = "ebook_publisher_name")
    ),
    @AttributeOverride(
        name = "paperBackPublisher.name",
        column = @Column(name = "paper_back_publisher_name")
    )
})
@AssociationOverrides({
    @AssociationOverride(
        name = "ebookPublisher.country",
        joinColumns = @JoinColumn(name = "ebook_publisher_country_id")
    ),
    @AssociationOverride(
        name = "paperBackPublisher.country",
        joinColumns = @JoinColumn(name = "paper_back_publisher_country_id")
    )
})
```

```java
public class Book2 {
    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    @Embedded
    private Publisher2 ebookPublisher;

    @Embedded
    private Publisher2 paperBackPublisher;

    public Publisher2 getPaperBackPublisher() {
        return paperBackPublisher;
    }

    public Publisher2 getEbookPublisher() {
        return ebookPublisher;
    }
}
```

@Target Mapping

```java
@Embeddable
class GPS implements Coordinates {
    private double latitude;
    private double longitude;

    public GPS() {
    }

    public GPS(double latitude, double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    @Override
    public double x() {
        return latitude;
    }
    @Override
    public double y() {
        return longitude;
    }
}
```

```java
interface Coordinates {
    double x();
    double y();
}


@Entity(name = "City")
public class City {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    @Target( GPS.class )
    private Coordinates coordinates;
}
```

@Parent Mapping

```java
@Embeddable
@Data
public class GPS {

    private double latitude;

    private double longitude;

    @Parent
    private City city;
}

entityManager -> {
    City cluj = new City(); cluj.setName( "Cluj" );

    cluj.setCoordinates( new GPS( 46.77120, 23.62360 ) );

    entityManager.persist( cluj );

    City cluj = entityManager.find( City.class, 1L );

    assertSame( cluj, cluj.getCoordinates().getCity() );
}
```

```java
@Entity(name = "City")
@Data
public class City {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    @Target( GPS.class )
    private GPS coordinates;
}
```

37

@Entity and @Table Mappings

```java
@Entity(name = "Book")
@Table( catalog = "public", schema = "store", name = "book" )
public static class Book {
    @Id
    private Long id;
    private String title;
    private String author;
    …
}
```

```sql
create table public.book (
    id bigint not null,
    author varchar(255),
    title varchar(255),
    primary key (id)
) engine=InnoDB
```

Identifiers

- UNIQUE - the values must uniquely identify each row.

- NOT NULL - the values cannot be null. For composite ids, no part can be null.

- IMMUTABLE - the values, once inserted, can never be changed. This is more a general guide, than a hard-fast rule as opinions vary. JPA defines the behavior of changing the value of the identifier attribute to be undefined; Hibernate simply does not support that. In cases where the values for the PK you have chosen will be updated, Hibernate recommends mapping the mutable value as a natural id, and use a surrogate id for the PK.

- EVER-INCREASING - fragmentation problem - because UUIDs are random, they have no natural ordering so cannot be used for clustering. This is why SQL Server has implemented a newsequentialid() function that is suitable for use in clustered indexes, and UUID PKs.

Entity Objects Identity – equals() and hashCode()

```java
Book book1 = new Book();
book1.setTitle("High-Performance Java Persistence");

Book book2 = new Book();
book2.setTitle("Java Persistence with Hibernate");

Library library = doInJPA(entityManager -> {
    Library _library = entityManager.find(Library.class, 1L);

    entityManager.persist(book1);
    entityManager.persist(book2);
    entityManager.flush();

    _library.getBooks().add(book1);
    _library.getBooks().add(book2);

    return _library;
});

assertTrue(library.getBooks().contains(book1));
assertTrue(library.getBooks().contains(book2));
```

Entity Objects Identity – equals() and hashCode() - II

```java
@Entity(name = "Library")
public class Library {
    @Id
    private Long id;
    private String name;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "book_id")
    private Set<Book> books = new HashSet<>();
}

@Entity(name = "Book")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String author;
    @NaturalId
    private String isbn;
```

```java
    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Book book = (Book) o;
        return Objects.equals( isbn, book.isbn );
    }

    @Override
    public int hashCode() {
        return Objects.hash( isbn );
    }
}
```

Entity Objects Identity – equals() and hashCode() - II

```java
@Entity(name = "Library")
public class Library {
    @Id
    private Long id;
    private String name;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "book_id")
    private Set<Book> books = new HashSet<>();
}

@Entity(name = "Book")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String author;
    @NaturalId
    private String isbn;
```

```java
@Override
public boolean equals(Object o) {
    if ( this == o ) {
        return true;
    }
    if ( o == null || getClass() != o.getClass() ) {
        return false;
    }
    Book book = (Book) o;
    return Objects.equals( isbn, book.isbn );
}

@Override
public int hashCode() {
    return Objects.hash( isbn );
}
}
```

Object-Relational Mapping (ORM)

- Package: javax.persistence
- Simple keys - @Id annotation
- Composite keys
  - Primary Key Class – requirements and structure
  - Annotations – @EmbeddedId, @IdClass
- Realtions between entity objects –
  - uni- and bi-directional,
  - 1:1, 1:many, many:1 many:many

Main JPA Annotations

- @PersistenceUnit,

- @PersistenceContext

- @Entity

- @Id

- @OneToOne

- @OneToMany

- @ManyToMany

- @DiscrimainatorColumn

- @Column

- @JoinTable

- @JoinColumn

- @Embeddable

- @Embedded

Entity Embeddables

- **@Embeddable** – annotates class that is a value type (not Entity), but can be embedded into one or more Entities

- **@Embedded** – embeds Embeddable class into Entity class

- Embedding can be hierarchical on multiple levels

- Annotations: **@AttributeOverride, @AttributeOverrides, @AssociationOverride, @AssociationOverrides**

Collection Type Persistent Fields

- Field or properties should be of Collection or Map type (usually generic):
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map
- @ElementCollection
- @CollectionTable – name of additional table
- @Embeddable, @Column
- @AttributeOverride, @AttributeOverrides

JPA Entities: @ManyToMany

```java
@Entity
public class Book {
    @Id @GeneratedValue
    private int id;

    @NotNull
    private String title;

    @ManyToOne
    @JoinColumn(name = "PUBLISHER_ID",
                referencedColumnName = "id")
    private Publisher publisher;

    @Column(name = "PUBLISHED_DATE") @PastOrPresent
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private LocalDate publishedDate;

    @Pattern(regexp = "\\d{10}|\\d{13}")
    private String isbn;

    @NotNull @Min(0)
    private double price;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name="BOOK_AUTHOR", joinColumns=
      @JoinColumn(name="BOOK_ID",referencedColumnName="ID"),
            inverseJoinColumns=
      @JoinColumn(name="AUTHOR_ID",referencedColumnName="ID")
    )
    private List<Author> authors = new ArrayList<>();
… }
```

```java
@Entity
public class Author {
    @Id @GeneratedValue
    private int id;

    @NotNull
    @Length(min=2, max=60)
    @Column(name = "first_name")
    private String firstName;

    @NotNull
    @Length(min=2, max=60)
    @Column(name = "last_name")
    private String lastName;

    @ManyToMany(mappedBy = "authors",
        fetch = FetchType.EAGER)
    List<Book> books = new ArrayList<>();
… }
```
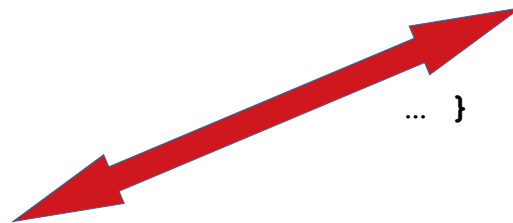
Optimizers

- None - no optimization is performed. We communicate with the database each and every time an identifier value is needed from the generator.

- pooled-lo - The pooled-lo optimizer works on the principle that the increment-value is encoded into the database table/sequence structure. In sequence-terms, this means that the sequence is defined with a greater-than-1 increment size.

- pooled - just like pooled-lo, except that here the value from the table/ sequence is interpreted as the high end of the value pool.

- hilo; legacy-hilo - custom algorithm for generating pools of values based on a single value from a table or sequence.

- Applications can also implement and use their own optimizer strategies, as defined by the org.hibernate.id.enhanced.Optimizer contract.

ORM Cascade Updates

- Entities that have a dependency relationship can be managed declaratively by JPA using CascadeType:

  - **ALL** – всички операции са каскадни

  - **DETACH** – каскадно отстраняване

  - **MERGE** – каскадно сливане

  - **PERSIST** – каскадно персистиране

  - **REFRESH** – каскадно обновяване

  - **REMOVE** – каскадно премахване

@OneToMany(cascade=REMOVE, mappedBy="customer")

public Set<Order> getOrders() { return orders; }

ArticlesDaoHibernate Class - I

```java
@Repository
@Transactional
public class ArticleDaoHibernate implements ArticleDao {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    public Collection<Article> findAll() {
        return this.sessionFactory.getCurrentSession()
            .createQuery("select article from Article article",
Article.class)
            .list();
    }
    @Override
    public Article find(long id) {
        return this.sessionFactory.getCurrentSession()
            .byId(Article.class).load(id);
    }
}
```

JPA Entity Annotations Example

- ```java
  @Entity
  public class Article {
      @Id
      @GeneratedValue
      private Long id;

      @Length(min=3, max=80)
      private String title;

      @Length(min=3, max=2048)
      private String content;

      @NotNull
      @ManyToOne
      @JoinColumn(name="AUTHOR_ID", nullable=false)
      private User author;

      @Length(min=3, max=256)
      private String pictureUrl;

      @Temporal(TemporalType.TIMESTAMP)
      private Date created = new Date();
  ```
- ```java
      @Temporal(TemporalType.TIMESTAMP)
      private Date updated = new Date();
  ```
- … }

```java
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue
    private long id;

    @NotNull
    @Length(min = 3, max = 30)
    private String username;
    ...

    @NonNull
    private String roles = "ROLE_USER";

    @OneToMany(mappedBy = "author",
               cascade = CascadeType.ALL,
               orphanRemoval=true)
    Collection<Article> articles =
        new ArrayList<>();

    @Temporal(TemporalType.TIMESTAMP)
    private Date created = new Date();

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated = new Date();
… }
```

Entity Inheritance

- Entity / Abstract entity

- Mapped superclass

- Non-entity superclass

- Entity -> DB tables mapping strategies

  - SingleTable per Class Hierarchy

  - TheTable per Concrete Class

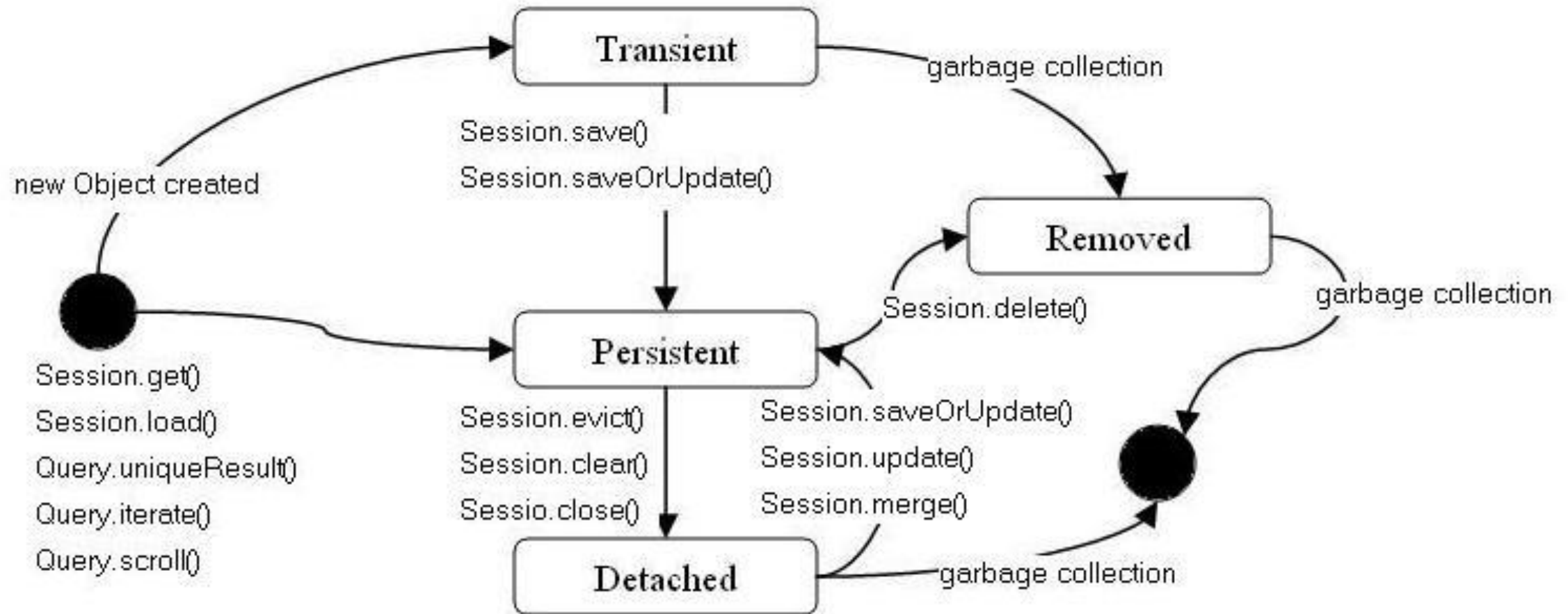 - The Joined Subclass Strategy

Entity Inheritance
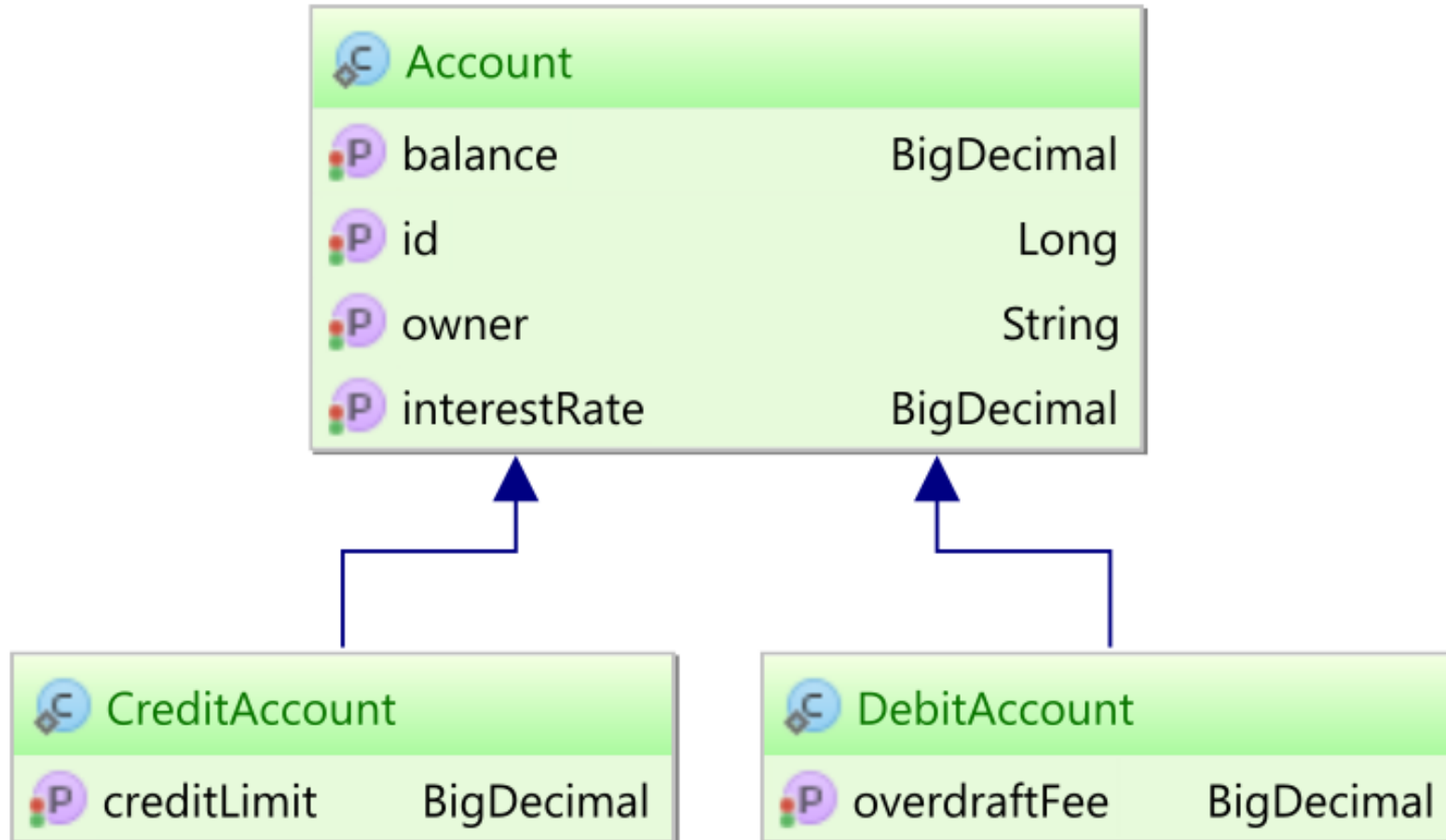
JPA Entity Lifecycle

persist()

Hibernate Entity Lifecycle

Hibernate Entity Lifecycle

ArticlesDaoHibernate Class - II

```java
@Override
public Article create(Article article) {
    this.sessionFactory.getCurrentSession()
            .persist(article);
    return article;
}
@Override
public Article update(Article article) {
    Article toBeDeleted = find(article.getId());
    if (toBeDeleted == null) {
        throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    return (Article) this.sessionFactory.getCurrentSession()
        .merge(article);
}
@Override
public Article remove(long articleId) {
    Article toBeDeleted = find(articleId);
    if (toBeDeleted == null) {
        throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    this.sessionFactory.getCurrentSession()
        .delete(toBeDeleted);
    return toBeDeleted;
}}
```

JPA Entities: ER Diagram
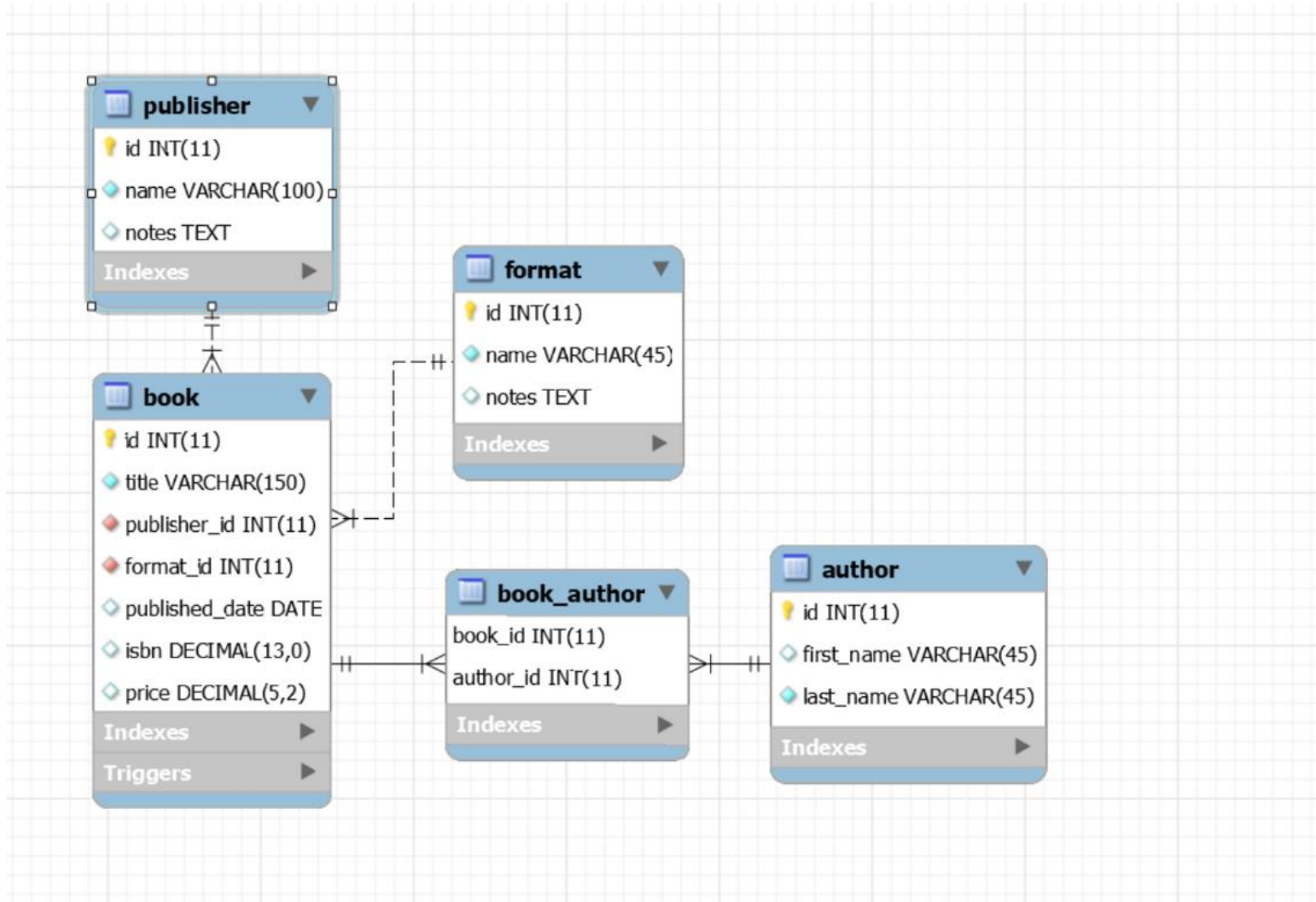
Second Level Caching

- If an instance is already present in the first-level cache, it is returned from there

- If an instance is not found in the first-level cache, and the corresponding instance state is cached in the second-level cache, then the data is fetched from there and an instance is assembled and returned

- Otherwise, the necessary data are loaded from the database and an instance is assembled and returned

Enabling Second Level Caching

hibernate.cache.use_second_level_cache=true

hibernate.cache.region.factory_class=jcache

hibernate.cache.use_second_level_cache: true

hibernate.cache.default_cache_concurrency_strategy: read-write

hibernate.cache.use_query_cache: true

Cache Concurrency Strategy

- READ_ONLY - used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don't change

- NONSTRICT_READ_WRITE: cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency

- READ_WRITE: this strategy guarantees strong consistency which it achieves by using 'soft' locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database

- TRANSACTIONAL: cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both database and cache in the same XA transaction

Cache Modes

- **GET** - the session may read items from the cache, but will not add items, except to invalidate items when updates occur.

- **IGNORE** - the session will never interact with the cache, except to invalidate cache items when updates occur.

- **NORMAL** - the session may read items from the cache, and add items to the cache.

- **PUT** - the session will never read items from the cache, but will add items to the cache as it reads them from the database.

- **REFRESH** - the session will never read items from the cache, but will add items to the cache as it reads them from the database.

Using Query Cache JPA & Hibernate

- List<Person> persons = entityManager.createQuery(
            "select p " +
                    "from Person p " +
                    "where p.name = :name", Person.class)
        .setParameter("name", "John Doe")
        .setHint("org.hibernate.cacheable", "true")
        .getResultList();

- List<Person> persons = session.createQuery(
            "select p " +
                    "from Person p " +
                    "where p.name = :name")
        .setParameter("name", "John Doe")
        .setCacheable(true)
        .list();

Caching query in custom region using JPA

```java
List<Person> persons = entityManager.createQuery(
        "select p " +
            "from Person p " +
            "where p.id > :id", Person.class)
    .setParameter( "id", 0L)
    .setHint( QueryHints.HINT_CACHEABLE, "true")
    .setHint( QueryHints.HINT_CACHE_REGION, "query.cache.person" )
    .setHint( "javax.persistence.cache.storeMode", CacheStoreMode.REFRESH )
    .getResultList();
```

Caching query in custom region using Hibernate

- List<Person> persons = session.createQuery(
  "select p " +
      "from Person p " +
      "where p.id > :id")
  .setParameter("id", 0L)
  .setCacheable(true)
  .setCacheRegion("query.cache.person")
  .setCacheMode(CacheMode.*REFRESH*)
  .list();

When using CacheStoreMode.REFRESH or CacheMode.REFRESH in conjunction with the region you have defined for the given query, Hibernate will selectively force the results cached in that particular region to be refreshed. This behavior is particularly useful in cases when the underlying data may have been updated via a separate process and is a far more efficient alternative to the bulk eviction of the region via SessionFactory eviction which looks as follows:

session.getSessionFactory().getCache().evictQueryRegion("query.cache.person");

Cache modes relationships

Hibernate	JPA	Description
CacheMode.NORMAL	CacheStoreMode.USE and CacheRetrieveMode.USE	Default. Reads/writes data from/into the cache
CacheMode.REFRESH	CacheStoreMode.REFRESH and CacheRetrieveMode.BYPASS	Doesn't read from cache, but writes to the cache upon loading from the database
CacheMode.PUT	CacheStoreMode.USE and CacheRetrieveMode.BYPASS	Doesn't read from cache, but writes to the cache as it reads from the database
CacheMode.GET	CacheStoreMode.BYPASS and CacheRetrieveMode.USE	Read from the cache, but doesn't write to cache
CacheMode.IGNORE	CacheStoreMode.BYPASS and CacheRetrieveMode.BYPASS	Doesn't read/write data from/into the cache

Using custom cache modes for queries

- List<Person> persons = entityManager.createQuery(
          "select p from Person p", Person.class)
      .setHint(QueryHints.*HINT_CACHEABLE*, "true")
      .setHint("javax.persistence.cache.retrieveMode" , CacheRetrieveMode.*USE*)
      .setHint("javax.persistence.cache.storeMode" , CacheStoreMode.*REFRESH*)
      .getResultList();

- List<Person> persons = session.createQuery(
          "select p from Person p")
      .setCacheable(true)
      .setCacheMode(CacheMode.*REFRESH*)
      .list();

Query Cache Best Practice

- As is case with collections, only ids of entities returned as a result of a cacheable query are cached, so it is strongly recommended that second-level cache is enabled for such entities.

- There is one cache entry per each combination of query parameter values (bind variables) for each query, so queries for which you expect lots of different combinations of parameter values are not good candidates for caching.

- Queries that involve entity classes for which there are frequent changes in the database are not good candidates for caching either, because they will be invalidated whenever there is a change related to any of the entity classed participating in the query, regardless whether the changed instances are cached as part of the query result or not.

Query Cache Best Practice

- By default, all query cache results are stored in org.hibernate.cache.internal.StandardQueryCache region. As with entity/collection caching, you can customize cache parameters for this region to define eviction and expiration policies according to your needs. For each query you can also specify a custom region name in order to provide different settings for different queries.

- For all tables that are queried as part of cacheable queries, Hibernate keeps last update timestamps in a separate region named org.hibernate.cache.spi.UpdateTimestampsCache. Being aware of this region is very important if you use query caching, because Hibernate uses it to verify that cached query results are not stale. The entries in this cache must not be evicted/expired as long as there are cached query results for the corresponding tables in query results regions. It is best to turn off automatic eviction and expiration for this cache region, as it does not consume lots of memory anyway.

Itering Entities and Associations

Hibernate offers two options if you want to filter entities or entity associations:

- static (e.g. @Where and @WhereJoinTable) - which are defined at mapping time and cannot change at runtime.

- dynamic (e.g. @Filter and @FilterJoinTable) - which are applied and configured at runtime.

JPA Query Language Syntax

- **Select Statements** - SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY.
- The SELECT clause defines the types of the objects or values returned by the query.
- The FROM clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the SELECT and WHERE clauses. An identification variable represents one of the following elements:
  - The abstract schema name of an entity
  - An element of a collection relationship
  - An element of a single-valued relationship
  - A member of a collection that is the multiple side of a one-to-many relationship
- The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query. Although the clause is optional, most queries have a WHERE clause.
- The GROUP BY clause groups query results according to a set of properties.
- The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.
- The ORDER BY clause sorts the objects or values returned by the query into a specified order.

JPA Query Language Syntax

Update and delete statements provide bulk operations over sets of entities. These statements have the following syntax:

- update_statement :: = update_clause [where_clause]

- delete_statement :: = delete_clause [where_clause]

- The update and delete clauses determine the type of the entities to be updated or deleted. The WHERE clause may be used to restrict the scope of the update or delete operation.

Java Persistence Query Language

- Object-oriented database queries

- Navigation

- Abstract schema

- Path expression

- State field

- Relationship field

Java Persistence Query Language

- SELECT

- FROM

- WHERE

- GROUP BY

- HAVING

- ORDER BY

- UPDATE

- DELETE

- AS, IN

- LIKE

- EXISTS, ANY, ALL

- NEW

Basic JPA Query usage

```java
Query query = entityManager.createQuery(
                "select p " +
                        "from Person p " +
                        "where p.name like :name")
        // timeout - in milliseconds
        .setHint("javax.persistence.query.timeout", 2000)
        // flush only at commit time
        .setFlushMode(FlushModeType.COMMIT);
```

Hibernate Flush Modes

ALWAYS

Flushes the Session before every query.

AUTO

This is the default mode, and it flushes the Session only if necessary.

COMMIT

The Session tries to delay the flush until the current Transaction is committed, although it might flush prematurely too.

MANUAL

The Session flushing is delegated to the application, which must call Session.flush() explicitly in order to apply the persistence context changes.

JPA defines some standard hints - I

javax.persistence.query.timeout - Defines the query timeout, in milliseconds.

javax.persistence.fetchgraph - Defines a fetchgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). For details, see the EntityGraph discussions in Fetching.

javax.persistence.loadgraph - Defines a loadgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). Attributes that are not specified are treated as FetchType.LAZY or FetchType.EAGER depending on the attribute's definition in metadata. For details, see the EntityGraph discussions in Fetching.

org.hibernate.cacheMode - Defines the CacheMode to use. See org.hibernate.query.Query#setCacheMode.

org.hibernate.cacheable - Defines whether the query is cacheable. true/false. See org.hibernate.query.Query#setCacheable.

JPA defines some standard hints - II

org.hibernate.cacheRegion - For queries that are cacheable, defines a specific cache region to use. See org.hibernate.query.Query#setCacheRegion.

org.hibernate.comment - Defines the comment to apply to the generated SQL. See org.hibernate.query.Query#setComment.

org.hibernate.fetchSize - Defines the JDBC fetch-size to use. See org.hibernate.query.Query#setFetchSize.

org.hibernate.flushMode - Defines the Hibernate-specific FlushMode to use. See org.hibernate.query.Query#setFlushMode. If possible, prefer using javax.persistence.Query#setFlushMode instead.

org.hibernate.readOnly - Defines that entities and collections loaded by this query should be marked as read-only. See org.hibernate.query.Query#setReadOnly.

JPA retrieving result set

In terms of execution, JPA Query offers 3 different methods for retrieving a result set:

Query#getResultList() - executes the select query and returns back the list of results.

Query#getResultStream() - executes the select query and returns back a Stream over the results.

Query#getSingleResult() - executes the select query and returns a single result. If there were more than one result an exception is thrown.

Basic Hibernate Query usage

- org.hibernate.query.Query query = session.createQuery(
            "select p " +
                    "from Person p " +
                    "where p.name like :name")
        // timeout - in seconds
        .setTimeout(2)
        // write to L2 caches, but do not read from them
        .setCacheMode(CacheMode.*REFRESH*)
         // assuming query cache was enabled for the SessionFactory
        .setCacheable(true)
        // add a comment to the generated SQL if enabled via the
        // hibernate.use_sql_comments configuration property
        .setComment("+ INDEX(p idx_person_name)");

Hibernate query scrolling

```java
try ( ScrollableResults scrollableResults = session.createQuery(
            "select p " +
                    "from Person p " +
                    "where p.name like :name" )
        .setParameter( "name", "J%" )
        .scroll()
) {
    while(scrollableResults.next()) {
        Person person = (Person) scrollableResults.get()[0];
        process(person);
    }
}
```

Hibernate query streaming

```
try ( Stream<Object[]> persons = session.createQuery(
            "select p.name, p.nickName " +
                "from Person p " +
                "where p.name like :name" )
        .setParameter( "name", "J%" )
        .stream() ) {

    persons
        .map( row -> new PersonNames(
                (String) row[0],
                (String) row[1] ) )
        .forEach( this::process );
}
```

JPA & Hibernate Locking
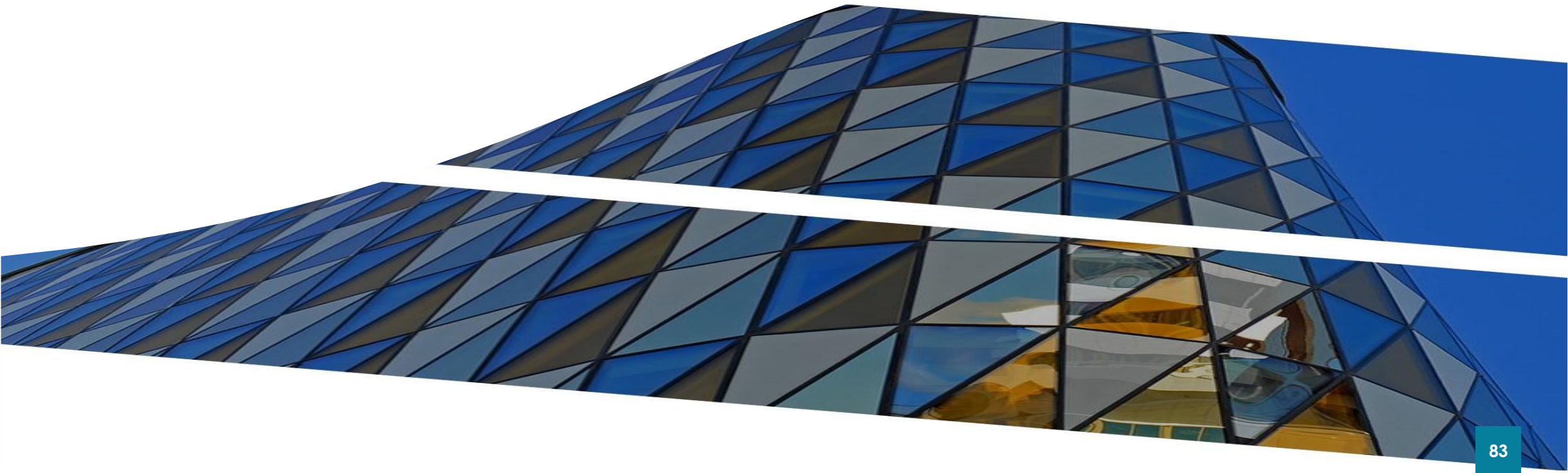
DB Locking

- In a relational database, locking refers to actions taken to prevent data from changing between the time it is read and the time is used.

- Your locking strategy can be either optimistic or pessimistic. Hibernate provides mechanisms for implementing both types in your applications.

- Optimistic locking – assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

- Pessimistic locking – assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Optimistic Locking

- When your application uses long transactions or conversations that span several database transactions, you can store versioning data so that if the same entity is updated by two conversations, the last to commit changes is informed of the conflict, and does not override the other conversation's work. This approach guarantees some isolation, but scales well and works particularly well in **read-often-write-sometimes** situations.

- Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.

- A version or timestamp property can never be null for a detached instance. Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys

@Version Based Optimistic Locking - I

```java
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Version
    private long version;

    //Getters and setters are omitted for brevity

}
```

@Version Based Optimistic Locking - II

```java
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Version
    private Timestamp version;

    //Getters and setters are omitted for brevity

}
```

@Version Based Optimistic Locking - III

```java
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Version
    private Instant version;

    //Getters and setters are omitted for brevity

}
```

@Version Optimistic Locking

- @Version property is mapped to the version column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would otherwise be overwritten by a last-commit-wins strategy.

- The version column can be any kind of type, as long as you define and implement the appropriate UserVersionType.

- Your application is forbidden from altering the version number set by Hibernate. You can artificially increase the version number, using the LockModeType.OPTIMISTIC_FORCE_INCREMENT or LockModeType.PESSIMISTIC_FORCE_INCREMENT.

- If the version number is generated by the database, such as a trigger, use the annotation @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) on the version attribute.

Timestamp Optimistic Locking

- Timestamp - a less reliable way of optimistic locking than version numbers but can be used by applications for other purposes as well. Timestamping is automatically used if you the @Version annotation on a Date or Calendar property type. E.g.:

@Version private Date version;

- Hibernate can retrieve the timestamp value from the database or the JVM, by reading the value you specify for the @org.hibernate.annotations.Source annotation. The value can be org.hibernate.annotations.SourceType.DB or VM. Default is database.

- The timestamp can also be generated by the database instead of Hibernate if you use the @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) or the @Source annotation.

Timestamp Optimistic Locking

```java
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    @Version
    @Source(value = SourceType.DB)
    private Date version;
}
```

Timestamp Optimistic Locking

Person person = new Person();

person.setId(1L);

person.setFirstName("John");

person.setLastName("Doe");

assertNull(person.getVersion());

entityManager.persist(person);

assertNotNull(person.getVersion());

CALL current_timestamp()

INSERT INTO
   Person
   (firstName, lastName, version, id)
VALUES
   (?, ?, ?, ?)

-- binding parameter [1] as [VARCHAR]   - [John]

-- binding parameter [2] as [VARCHAR]   - [Doe]

-- binding parameter [3] as [TIMESTAMP] - [2017-05-18 12:03:03.808]

-- binding parameter [4] as [BIGINT]   - [1]

Excluding Attributes from Version Increment

```java
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OptimisticLock( excluded = true )
    private long callCount;

    @Version
    private Long version;

    //Getters and setters are omitted for brevity

    public void incrementCallCount() {
        this.callCount++;
    }
}
```

Excluding Attributes from Version Increment

```java
doInJPA( this::entityManagerFactory, entityManager -> {
    Phone phone = entityManager.find( Phone.class, 1L );
    phone.setNumber( "+123-456-7890" );

    doInJPA( this::entityManagerFactory, _entityManager -> {
        Phone _phone = _entityManager.find( Phone.class, 1L );
        _phone.incrementCallCount();

        log.info( "Bob changes the Phone call count" );
    } );

    log.info( "Alice changes the Phone number" );
} );
```

Excluding Attributes from Version Increment

-- Bob changes the Phone call count

```
update
    Phone
set
    callCount = 1,
    "number" = '123-456-7890',
    version = 0
where
    id = 1
    and version = 0
```

-- Alice changes the Phone number

```
update
    Phone
set
    callCount = 0,
    "number" = '+123-456-7890',
    version = 1
where
    id = 1
    and version = 0
```

Versionless optimistic locking

- The idea is that you can get Hibernate to perform "version checks" using either all of the entity's attributes or just the attributes that have changed. This is achieved through the use of the @OptimisticLocking annotation which defines a single attribute of type OptimisticLockType:

- NONE- optimistic locking is disabled even if there is a @Version annotation present

- VERSION (the default)- performs optimistic locking based on a @Version as described above

- ALL - performs optimistic locking based on all fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

- DIRTY- performs optimistic locking based on dirty fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

Versionless optimistic locking

```java
@Entity(name = "Person")
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
@SelectBeforeUpdate
public static class Person {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    private String country;

    private String city;

    @Column(name = "created_on")
    private Timestamp createdOn;
}
```

- The main advantage of OptimisticLockType.DIRTY over OptimisticLockType.ALL and the default OptimisticLockType.VERSION used implicitly along with the @Version mapping, is that it allows you to minimize the risk of OptimisticLockException across non-overlapping entity property changes.

- When using OptimisticLockType.DIRTY, you should also use @DynamicUpdate because the UPDATE statement must take into consideration all the dirty entity property values, and also the @SelectBeforeUpdate annotation so that detached entities are properly handled by the Session#update(entity) operation.

LOCK MODES

Lock Mode	Description
OPTIMISTIC	Obtain an optimistic read lock for all entities with version attributes.
OPTIMISTIC_FORCE_INCREMENT	Obtain an optimistic read lock for all entities with version attributes, and increment the version attribute value.
PESSIMISTIC_READ	Immediately obtain a long-term read lock on the data to prevent the data from being modified or deleted. Other transactions may read the data while the lock is maintained, but may not modify or delete the data. The persistence provider is permitted to obtain a database write lock when a read lock was requested, but not vice versa.
PESSIMISTIC_WRITE	Immediately obtain a long-term write lock on the data to prevent the data from being read, modified, or deleted.
PESSIMISTIC_FORCE_INCREMENT	Immediately obtain a long-term lock on the data to prevent the data from being modified or deleted, and increment the version attribute of versioned entities.
READ	A synonym for OPTIMISTIC. Use of LockModeType.OPTIMISTIC is to be preferred for new applications.
WRITE	A synonym for OPTIMISTIC_FORCE_INCREMENT. Use of LockModeType.OPTIMISTIC_FORCE_INCREMENT is to be preferred for new applications.
NONE	No additional locking will occur on the data in the database.

LockModeType -> Hibernate Native LockMode

LockModeType	LockMode	Description
NONE	NONE	The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode.
READ and OPTIMISTIC	READ	The entity version is checked towards the end of the currently running transaction.
WRITE and OPTIMISTIC_FORCE_INCREMENT	WRITE	The entity version is incremented automatically even if the entity has not changed.
PESSIMISTIC_FORCE_INCREMENT	PESSIMISTIC_FORCE_INCREMENT	The entity is locked pessimistically and its version is incremented automatically even if the entity has not changed.
PESSIMISTIC_READ	PESSIMISTIC_READ	The entity is locked pessimistically using a shared lock if the database supports such a feature. Otherwise, an explicit lock is used.
PESSIMISTIC_WRITE	PESSIMISTIC_WRITE, UPGRADE	The entity is locked using an explicit lock.
PESSIMISTIC_WRITE with a javax.persistence.lock.timeout setting of 0	UPGRADE_NOWAIT	The lock acquisition request fails fast if the row s already locked.
PESSIMISTIC_WRITE with a javax.persistence.lock.timeout setting of -2	UPGRADE_SKIPLOCKED	The lock acquisition request skips the already locked rows. It uses a SELECT … FOR UPDATE SKIP LOCKED in Oracle and PostgreSQL 9.5, or SELECT … with (rowlock, updlock, readpast) in SQL Server.

Hibernate Specific Lock Options

- NO_WAIT - indicates that the database should not wait at all to acquire the pessimistic lock.

- NONE - represents LockMode.NONE (timeout + scope do not apply).

- READ - represents LockMode.READ (timeout + scope do not apply).

- SKIP_LOCKED - indicates that rows that are already locked should be skipped.

- UPGRADE - represents LockMode.UPGRADE (will wait forever for lock, and scope of false meaning only entity is locked).

- WAIT_FOREVER - indicates that there is no timeout for the acquisition.

Setting the Lock Mode

- Call the EntityManager.lock method, passing in one of the lock modes:

EntityManager em = ...;

Person person = ...;

em.lock(person, LockModeType.OPTIMISTIC);

- Call one of the EntityManager.find methods that take the lock mode as a parameter:

EntityManager em = ...;

String personPK = ...;

Person person = em.find(Person.class, personPK,

    LockModeType.PESSIMISTIC_WRITE);

Setting the Lock Mode

- Call one of the EntityManager.refresh methods that take the lock mode as a parameter:

EntityManager em = ...;

String personPK = ...;

Person person = em.find(Person.class, personPK);

...

em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);

- Call the Query.setLockMode or TypedQuery.setLockMode method, passing the lock mode as the parameter:

Query q = em.createQuery(...);

q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);

Setting the Lock Mode

- Add a lockMode element to the @NamedQuery annotation:

@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)

JPA Locking Query Hints

- javax.persistence.lock.timeout - it gives the number of milliseconds a lock acquisition request will wait before throwing an exception. Not all JDBC database drivers support setting a timeout value for a locking request. If not supported, the Hibernate dialect ignores this query hint.

- javax.persistence.lock.scope - defines the scope of the lock acquisition request. The scope can either be NORMAL (default value) or EXTENDED. The EXTENDED scope will cause a lock acquisition request to be passed to other owned table structured (e.g. @Inheritance(strategy=InheritanceType.JOINED), @ElementCollection). Accordint tot Hibernate 5.5.9 user guide, javax.persistence.lock.scope is **not yet supported** as specified by the JPA standard.

Example Specifing javax.persistence.lock.timeout

```
entityManager.find(
        Person.class, id, LockModeType.PESSIMISTIC_WRITE,
        Collections.singletonMap( "javax.persistence.lock.timeout", 200 )
);


SELECT explicitlo0_.id     AS id1_0_0_,
       explicitlo0_."name" AS name2_0_0_
FROM   person explicitlo0_
WHERE  explicitlo0_.id = 1
FOR UPDATE wait 2
```

The Hibernate Native buildLockRequest API

```java
Person person = entityManager.find( Person.class, id );
Session session = entityManager.unwrap( Session.class );
session
        .buildLockRequest( LockOptions.NONE )
        .setLockMode( LockMode.PESSIMISTIC_READ )
        .setTimeOut( LockOptions.NO_WAIT )
        .lock( person );
```

```sql
SELECT p.id AS id1_0_0_ ,
        p.name AS name2_0_0_
FROM   Person p
WHERE  p.id = 1

SELECT id
FROM   Person
WHERE  id = 1
FOR    SHARE NOWAIT
```

Follow-on-locking (DISTINCT, GROUP BY, UNION, views – Oracle)

```
List<Person> persons = entityManager.createQuery(
        "select DISTINCT p from Person p", Person.class)
    .setLockMode( LockModeType.PESSIMISTIC_WRITE )
    .getResultList();
```

```
SELECT DISTINCT p.id as id1_0_, p."name" as name2_0_
FROM Person p


SELECT id
FROM Person
WHERE id = 1 FOR UPDATE


SELECT id
FROM Person
WHERE id = 1 FOR UPDATE
```

Secondary Query Entity Locking

```
List<Person> persons = entityManager.createQuery(
        "select DISTINCT p from Person p", Person.class)
    .getResultList();

entityManager.createQuery(
        "select p.id from Person p where p in :persons")
    .setLockMode( LockModeType.PESSIMISTIC_WRITE )
    .setParameter( "persons", persons )
    .getResultList();
```

```
SELECT DISTINCT p.id as id1_0_, p."name" as name2_0_
FROM Person p

SELECT p.id as col_0_0_
FROM Person p
WHERE p.id IN ( 1 , 2 )
FOR UPDATE
```

Disabling Follow-on-locking

```
List<Person> persons = entityManager.createQuery(
        "select p from Person p", Person.class)
    .setMaxResults( 10 )
    .unwrap( Query.class )
    .setLockOptions(
        new LockOptions( LockMode.PESSIMISTIC_WRITE )
            .setFollowOnLocking( false ) )
    .getResultList();
```

```
SELECT *
    FROM (
        SELECT p.id as id1_0_, p."name" as name2_0_
        FROM Person p
    )
WHERE rownum <= 10
FOR UPDATE
```

Locking Exceptions

- Pessimistic Locking Exceptions
  - **LockTimeoutException** – – indicates that obtaining a lock or converting a shared to exclusive lock times out, and the result is a **statement-level rollback**
  - **PessimisticLockException** – indicates that obtaining a lock or converting a shared to exclusive lock fails, and the result is a **transaction-level rollback**
  - **PersistanceException** – indicates that a persistence problem occurred. PersistanceException and its subtypes (except LockTimeoutException, QueryTimeoutException, NoResultException, and NonUniqueResultException), results in a **transaction-level rollback**.

- Optimistic Locking Exceptions
  - **OptimisticLockException** – when persistence provider discovers optimistic locking conflicts between entity versions, results in a **transaction-level rollback**.
  - Usually (but not always) it contains a reference to the conflicting entity.
  - The recommended way of handling the exception is to try reloading (refreshing) the entity and applying required changes again in a new transaction (e.g. traying to call the domain service method again with same data until success.

Hibernate Batching

JDBC Batching I

- JDBC offers support for batching together SQL statements that can be represented as a single PreparedStatement. Implementation wise this generally means that drivers will send the batched operation to the server in one call, which can save on network calls to the database. Hibernate can leverage JDBC batching. The following settings control this behavior:

- hibernate.jdbc.batch_size - maximum number of statements Hibernate will batch together before asking the driver to execute the batch. Zero or a negative number disables this feature.

- hibernate.jdbc.batch_versioned_data - some JDBC drivers return incorrect row counts when a batch is executed. If your JDBC driver falls into this category this setting should be set to false. Otherwise, it is safe to enable this which will allow Hibernate to still batch the DML for versioned entities and still use the returned row counts for optimistic lock checks. Since 5.0, it defaults to true. Previously (versions 3.x and 4.x), it used to be false.

JDBC Batching II

- hibernate.jdbc.batch.builder - names the implementation class used to manage batching capabilities. It is almost never a good idea to switch from Hibernate's default implementation.

- hibernate.order_updates - forces Hibernate to order SQL updates by the entity type and the primary key value of the items being updated. This allows for more batching to be used. It will also result in fewer transaction deadlocks in highly concurrent systems. Comes with a performance hit, so benchmark before and after to see if this actually helps or hurts.

- hibernate.order_inserts - forces Hibernate to order inserts to allow for more batching to be used. Comes with a performance hit, so benchmark before and after to see if this actually helps or hurts your application.

JDBC Batching Configuration Using Hibernate API

entityManager
      .unwrap(Session.class)
      .setJdbcBatchSize(10);

Session Batching – Is there a problem with following code:

```java
EntityManager entityManager = null;
EntityTransaction txn = null;
try {
    entityManager = entityManagerFactory().createEntityManager();

    txn = entityManager.getTransaction();
    txn.begin();

    for ( int i = 0; i < 100_000; i++ ) {
        Person Person = new Person( String.format( "Person %d", i ) );
        entityManager.persist( Person );
    }

    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
    throw e;
} finally {
    if (entityManager != null) {
        entityManager.close();
    }
}
```

Session Batching – Is there a problem with following code:

There are several problems associated with this example:

- Hibernate caches all the newly inserted Customer instances in the session-level cache, so, when the transaction ends, 100 000 entities are managed by the persistence context. If the maximum memory allocated to the JVM is rather low, this example could fail with an OutOfMemoryException. The Java 1.8 JVM allocated either 1/4 of available RAM or 1Gb, which can easily accommodate 100 000 objects on the heap.

- long-running transactions can deplete a connection pool so other transactions don't get a chance to proceed.

- JDBC batching is not enabled by default, so every insert statement requires a database roundtrip. To enable JDBC batching, set the hibernate.jdbc.batch_size property to an integer between 10 and 50.

Session Batching – Solution:

```java
EntityManager entityManager = null;
EntityTransaction txn = null;
try {
    entityManager = entityManagerFactory().createEntityManager();
    txn = entityManager.getTransaction();
    txn.begin();
    int batchSize = 25;
    for ( int i = 0; i < entityCount; i++ ) {
        if ( i > 0 && i % batchSize == 0 ) {
            //flush a batch of inserts and release memory
            entityManager.flush();
            entityManager.clear();
        }
        Person Person = new Person( String.format( "Person %d", i ) );
        entityManager.persist( Person );
    }
    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
    throw e;
} finally {
    if (entityManager != null) {  entityManager.close();  }
}
```

Session scroll

```java
ScrollableResults scrollableResults = null;
try {
    entityManager = entityManagerFactory().createEntityManager();
    txn = entityManager.getTransaction();
    txn.begin();
    int batchSize = 25;
    Session session = entityManager.unwrap( Session.class );
    scrollableResults = session
            .createQuery( "select p from Person p" )
            .setCacheMode( CacheMode.IGNORE )
            .scroll( ScrollMode.FORWARD_ONLY );
    int count = 0;
    while ( scrollableResults.next() ) {
        Person Person = (Person) scrollableResults.get( 0 );
        processPerson(Person);
        if ( ++count % batchSize == 0 ) {
            //flush a batch of updates and release memory:
            entityManager.flush();
            entityManager.clear();
        }
    }
    txn.commit();
```

```java
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
    throw e;
} finally {
    if (scrollableResults != null) {
        scrollableResults.close();
    }
    if (entityManager != null) {
        entityManager.close();
    }
}
}
```

StatelessSession API

- StatelessSession is a command-oriented API provided by Hibernate. Use it to stream data to and from the database in the form of detached objects. A StatelessSession has no persistence context associated with it and does not provide many of the higher-level lifecycle semantics.

- Some of the things not provided by a StatelessSession include:
  - a first-level cache
  - interaction with any second-level or query cache
  - transactional write-behind or automatic dirty checking

- Limitations of StatelessSession:
  - Operations performed using a stateless session never cascade to associated instances.
  - Collections are ignored by a stateless session.
  - Lazy loading of associations is not supported.
  - Operations performed via a stateless session bypass Hibernate's event model and interceptors.
  - Due to the lack of a first-level cache, Stateless sessions are vulnerable to data aliasing effects.
  - A stateless session is a lower-level abstraction that is much closer to the underlying JDBC.

StatelessSession API Example

```java
StatelessSession statelessSession = null;
Transaction txn = null;
ScrollableResults scrollableResults = null;
try {
    SessionFactory sessionFactory = entityManagerFactory()
        .unwrap( SessionFactory.class );
    statelessSession = sessionFactory.openStatelessSession();

    txn = statelessSession.getTransaction();
    txn.begin();

    scrollableResults = statelessSession
        .createQuery( "select p from Person p" )
        .scroll(ScrollMode.FORWARD_ONLY);

    while ( scrollableResults.next() ) {
        Person Person = (Person) scrollableResults.get( 0 );
        processPerson(Person);
        statelessSession.update( Person );
    }

    txn.commit();
}
```

```java
} catch (RuntimeException e) {
    if ( txn != null && txn.getStatus() == TransactionStatus.ACTIVE)
txn.rollback();
    throw e;
} finally {
    if (scrollableResults != null) {
        scrollableResults.close();
    }
    if (statelessSession != null) {
        statelessSession.close();
    }
}
```

JPQL/HQL for Bulk UPDATE and DELETE

int updatedEntities = entityManager

    .createQuery("update Person p set p.name = :newName where p.name = :oldName")

    .setParameter("oldName", oldName)

    .setParameter("newName", newName) .executeUpdate();

int updatedEntities = session

    .createQuery("update versioned Person set name = :newName where name = :oldName")

    .setParameter("oldName", oldName)

    .setParameter("newName", newName) .executeUpdate();

HQL Bulk INSERT

```
int insertedEntities = session

    .createQuery( "insert into Partner (id, name) select p.id, p.name from Person p ")

    .executeUpdate();
```

Hibernate Fetching

Fetching

The concept of fetching breaks down into two different questions:

- When should the data be fetched? Now? Later?

- How should the data be fetched?

Fetching Options with Hibernate – Static:

Static definition of fetching strategies is done in the mappings. The statically-defined fetch strategies are used in the absence of any dynamically defined strategies.

- SELECT - performs a separate SQL select to load the data. This can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed). This trategy is generally termed N+1.

- JOIN - inherently an EAGER style of fetching. The data to be fetched is obtained through the use of an SQL outer join.

- BATCH - performs a separate SQL select to load a number of related data items using an IN-restriction as part of the SQL WHERE-clause based on a batch size. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).

- SUBSELECT - performs a separate SQL select to load associated data based on the SQL restriction used to load the owner. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).

Fetching Options with Hibernate – Dynamic:

Dynamic (sometimes referred to as runtime) definition is really use-case centric. There are multiple ways to define dynamic fetching:

- fetch profiles - defined in mappings, but can be enabled/disabled on the Session.

- HQL / JPQL - both Hibernate and JPA Criteria queries have the ability to specify fetching, specific to said query.

- entity graphs - starting in Hibernate 4.2 (JPA 2.1), this is also an option there are two types of entity graphs:

    - load graph – In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, but attributes not specified use their static mapping specification.
    - fetch graph - In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, and all attributes not specified will **ALWAYS** be treated as FetchType.LAZY.

Fetching Options with Hibernate

- The Hibernate recommendation is to statically mark all associations lazy and to use dynamic fetching strategies for eagerness.

- This is unfortunately at odds with the JPA specification which defines that all one-to-one and many-to-one associations should be eagerly fetched by default. Hibernate, as a JPA provider, honors that default.

No Fetching

```
Integer accessLevel = entityManager.createQuery(
        "select e.accessLevel " +
              "from Employee e " +
              "where " +
              "       e.username = :username and " +
              "       e.password = :password",
        Integer.class)
    .setParameter( "username", username)
    .setParameter( "password", password)
    .getSingleResult();
```

Dynamic fetching via queries

```
Employee employee = entityManager.createQuery(
        "select e " +
            "from Employee e " +
            "left join fetch e.projects " +
            "where " +
            "    e.username = :username and " +
            "    e.password = :password",
        Employee.class)
    .setParameter("username", username)
    .setParameter("password", password)
    .getSingleResult();
```

Dynamic fetching via queries – using Criteria API

```java
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = builder.createQuery( Employee.class );
Root<Employee> root = query.from( Employee.class );
root.fetch( "projects", JoinType.LEFT);
query.select(root).where(
        builder.and(
                builder.equal(root.get("username"), username),
                builder.equal(root.get("password"), password)
        )
);
Employee employee = entityManager.createQuery( query ).getSingleResult();
```

Dynamic fetching via JPA entity graph

```
@Entity(name = "Employee")
@NamedEntityGraph(name = "employee.projects",
    attributeNodes = @NamedAttributeNode("projects")
)


Employee employee = entityManager.find(
    Employee.class,
    userId,
    Collections.singletonMap(
        "javax.persistence.fetchgraph",
        entityManager.getEntityGraph( "employee.projects" )
    )
);}
```

Dynamic fetching via JPA entity graph

- When executing a JPQL query, if an EAGER association is omitted, Hibernate will issue a secondary select for every association needed to be fetched eagerly, which can lead to N+1 query issues.

- For this reason, it's better to use LAZY associations, and only fetch them eagerly on a per-query basis.

Dynamic fetching via JPA entity graph – using @NamedSubgraph

```java
@Entity(name = "Project")
@NamedEntityGraph(name = "project.employees",
    attributeNodes = @NamedAttributeNode(
        value = "employees",
        subgraph = "project.employees.department"
    ),
    subgraphs = @NamedSubgraph(
        name = "project.employees.department",
        attributeNodes = @NamedAttributeNode( "department" )
    )
)
public static class Project {

    @Id
    private Long id;

    @ManyToMany
    private List<Employee> employees = new ArrayList<>();
}
```

Dynamic fetching via JPA entity graph – using @NamedSubgraph

```java
Project project = doInJPA( this::entityManagerFactory,

entityManager -> {
    return entityManager.find(
        Project.class,
        1L,
        Collections.singletonMap(
            "javax.persistence.fetchgraph",
            entityManager.getEntityGraph( "project.employees" )
        )
    );
} );
```

```sql
select
p.id as id1_2_0_, e.id as id1_1_1_, d.id as id1_0_2_,
    e.accessLevel as accessLe2_1_1_,
e.department_id as departme5_1_1_,
    decrypt( 'AES', '00', e.pswd  ) as pswd3_1_1_,
e.username as username4_1_1_,
    p_e.projects_id as projects1_3_0__,
p_e.employees_id as employee2_3_0__
    from
Project p
inner join
Project_Employee p_e
on p.id=p_e.projects_id
inner join
Employee e
on p_e.employees_id=e.id
inner join
Department d
on e.department_id=d.id
where
p.id = ?

-- binding parameter [1] as [BIGINT] - [1]
```

Creating and applying JPA graphs from text representations

```java
final EntityGraph<Project> graph = GraphParser.parse(
        Project.class,
        "employees( department )",
        entityManager
);


final EntityGraph<Movie> graph = GraphParser.parse(
        Movie.class,
        "cast.key( name )",
        entityManager
);


final EntityGraph<Ticket> graph = GraphParser.parse(
        Ticket.class,
        "showing.key( movie( cast ) )",
        entityManager
);
```

Subtypes and Subgraphs. Combining Subgraphs

```java
Class<Invoice> invoiceClass = ...;
javax.persistence.EntityGraph<Invoice> invoiceGraph = entityManager.createEntityGraph( invoiceClass );
invoiceGraph.addAttributeNode( "responsibleParty" );
invoiceGraph.addSubgraph( "responsibleParty" ).addAttributeNode( "taxIdNumber" );
invoiceGraph.addSubgraph( "responsibleParty", Corporation.class ).addAttributeNode( "ceo" );
invoiceGraph.addSubgraph( "responsibleParty", NonProfit.class ).addAttributeNode( "sector" );

final EntityGraph<Project> a = GraphParser.parse(
        Project.class, "employees( username )", entityManager
);
final EntityGraph<Project> b = GraphParser.parse(
        Project.class, "employees( password, accessLevel )", entityManager
);
final EntityGraph<Project> c = GraphParser.parse(
        Project.class, "employees( department( employees( username ) ) )", entityManager
);

final EntityGraph<Project> all = EntityGraphs.merge( entityManager, Project.class, a, b, c );
```

Dynamic fetching via Hibernate profiles

```java
@Entity(name = "Employee")
    @FetchProfile(
        name = "employee.projects",
        fetchOverrides = {
            @FetchProfile.FetchOverride(
                entity = Employee.class,
                association = "projects",
                mode = FetchMode.JOIN
            )
        }
    )


session.enableFetchProfile( "employee.projects" );
Employee employee = session.bySimpleNaturalId( Employee.class ).load( username );
```

Batch fetching - I

```java
@Entity(name = "Department")
public static class Department {
    @Id
    private Long id;
    @OneToMany(mappedBy = "department")
    //@BatchSize(size = 5)
    private List<Employee> employees = new ArrayList<>();
}

@Entity(name = "Employee")
public static class Employee {
    @Id
    private Long id;
    @NaturalId
    private String name;
    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;
}
```

- Without @BatchSize, you'd run into a N+1 query issue, so, instead of 2 SQL statements, there would be 10 queries needed for fetching the Employee child entities.

- However, although @BatchSize is better than running into an N+1 query issue, most of the time, a DTO projection or a JOIN FETCH is a much better alternative since it allows you to fetch all the required data with a single query.

Batch fetching - II

```java
List<Department> departments = entityManager.createQuery(
            "select d " +
                "from Department d " +
                "inner join d.employees e " +
                "where e.name like 'John%'", Department.class)
        .getResultList();
    for ( Department department : departments ) {
        log.infof(
            "Department %d has {} employees",
            department.getId(),
            department.getEmployees().size()
        );
    }
    SELECT
    d.id as id1_0_
        FROM
    Department d
    INNER JOIN
    Employee employees1_
    ON d.id=employees1_.department_id
```

```sql
 SELECT
e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
e.name as name2_1_0_
    FROM
Employee e
WHERE e.department_id IN ( 0, 2, 3, 4, 5)
SELECT
e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
e.name as name2_1_0_
    FROM
Employee e
WHERE  e.department_id IN ( 6, 7, 8, 9, 1)
```

The @Fetch annotation mapping – FetchMode s:

- SELECT - the association is going to be fetched lazily using a secondary select for each individual entity, collection, or join load. It's equivalent to JPA FetchType.LAZY fetching strategy.

- JOIN - use an outer join to load the related entities, collections or joins when using direct fetching. It's equivalent to JPA FetchType.EAGER fetching strategy.

- SUBSELECT - available for collections only. When accessing a non-initialized collection, this fetch mode will trigger loading all elements of all collections of the same role for all owners associated with the persistence context using a single secondary select.

Dynamic fetching via Hibernate profiles

```java
@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
  @Fetch(FetchMode.SUBSELECT)
  private List<Employee> employees = new ArrayList<>();

  List<Department> departments = entityManager.createQuery(
          "select d " +
              "from Department d " +
              "where d.name like :token", Department.class )
      .setParameter( "token", "Department%" )
      .getResultList();

log.infof( "Fetched %d Departments", departments.size());

for (Department department : departments ) {
    assertEquals( 3, department.getEmployees().size() );
}
```

```sql
SELECT
    d.id as id1_0_
        FROM
    Department d
    where
    d.name like 'Department%'
        -- Fetched 2 Departments

        SELECT
    e.department_id as departme3_1_1_,
        e.id as id1_1_1_,
    e.id as id1_1_0_,
        e.department_id as departme3_1_0_,
    e.username as username2_1_0_
        FROM
    Employee e
    WHERE
    e.department_id in (
        SELECT
    fetchmodes0_.id
        FROM
    Department fetchmodes0_
    WHERE
    d.name like 'Department%'
)
```

@LazyCollection

```java
@Entity(name = "Department")
public static class Department {
    @Id  private Long id;

    @OneToMany(mappedBy = "department",
        cascade = CascadeType.ALL)
    @OrderColumn(name = "order_id")
    @LazyCollection( LazyCollectionOption.EXTRA )
    private List<Employee> employees = new ArrayList<>();
}

@Entity(name = "Employee")
public static class Employee {
    @Id    private Long id;

    @NaturalId
    private String username;
    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;
}
```

- TRUE - Load it when the state is requested.

- FALSE - Eagerly load it.

- EXTRA - Prefer extra queries over full collection loading.

- LazyCollectionOption.EXTRA only works for ordered collections, either List(s) that are annotated with @OrderColumn or Map(s).

- For bags (e.g. regular List(s) of entities that do not preserve any certain ordering), the @LazyCollection(LazyCollectionOption.EXTRA) behaves like any other FetchType.LAZY collection (the collection is fetched entirely upon being accessed for the first time).

142

@LazyCollection

@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints({@QueryHints("javax.persistence.lock.timeout", value ="5000")})

Criteria API and Metamodel API

- Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.

- JPA 2 defines a typesafe Criteria API which allows Criteria queries to be constructed in a strongly-typed manner, utilizing so called static metamodel classes. For developers it is important that the task of the metamodel generation can be automated.

- Hibernate Static Metamodel Generator is an annotation processor based on JSR_269 with the task of creating JPA 2 static metamodel classes. The following example shows two JPA 2 entities Order and Item, together with the metamodel class Order_ and a typesafe query.

@LazyCollection

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Person> persons = entityManager.createQuery( criteria ).getResultList();
```

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<String> criteria = builder.createQuery( String.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root.get( Person_.nickName ) );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<String> nickNames = entityManager.createQuery( criteria ).getResultList();
```

References

- [PoEAA] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Professional. 2002.

- [JPwH] Christian Bauer & Gavin King. Java Persistence with Hibernate, Second Edition. Manning. 2015.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD