Spring Documentation Part 1

Duy NTC

Version	**0.1**
Status:	**Draft**
Date:	**03.08.2021**
Prepared by:	**OpenWay**
Author:	**Duyntc**
Approved By:	**Duyntc**

1. Table of Contents

2. Spring configuration

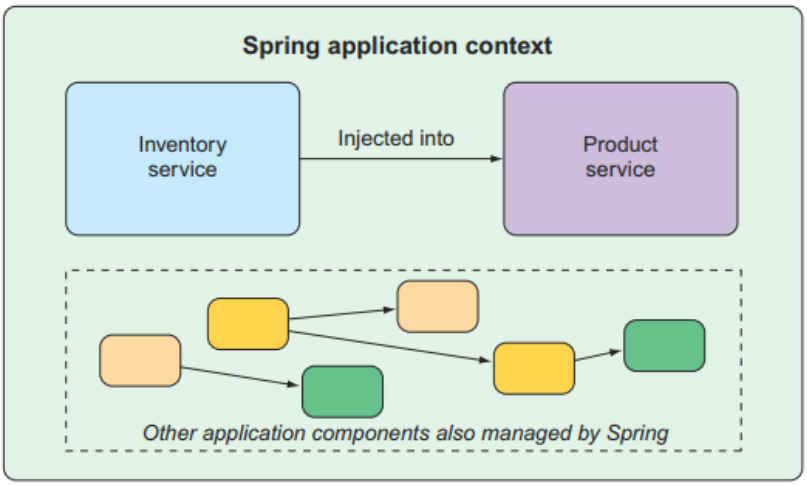
2.1. Spring application context

2.1.1. Getting start with spring

At its core, Spring offers a container, often referred to as the Spring application context, that creates and manages application components.

These components, or beans, are wired together inside the Spring application context to make a complete application, much like bricks, mortar, timber, nails, plumbing, and wiring are bound together to make a house.

The act of wiring beans together is based on a pattern known as dependency injection (DI). Rather than have components create and maintain the lifecycle of other beans that they depend on, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those into the beans that need them. This is done typically through constructor arguments or property accessor methods.



Automatic configuration has its roots in the Spring techniques known as autowiring and component scanning. With component scanning, Spring can automatically discover components from an application's classpath and create them as beans in the Spring application context. With autowiring, Spring automatically injects the components with the other beans that they depend on.

The choice of JAR packaging is a cloud-minded choice. Whereas WAR files are perfectly suitable for deploying to a traditional Java application server, they're not a natural fit for most cloud platforms.

Although some cloud platforms (such as Cloud Foundry) are capable of deploying and running WAR files, all Java cloud platforms are capable of running an executable JAR file. Therefore, the Spring Initializr defaults to JAR packaging unless you tell it to do otherwise.

Boostrapping the application:

Listing 1.2 The Taco Cloud bootstrap class

```
package tacos;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication                          <-- Spring Boot application
public class TacoCloudApplication {

  public static void main(String[] args) {
    SpringApplication.run(TacoCloudApplication.class, args);   <-- Runs the application
  }

}
```

@SpringBootApplication:

- **@SpringBootConfiguration**
- **@EnableAutoConfiguration**
- **@ComponentScan**

Testing application execute with two annotation:

@RunWith (SpringRunner.class) // Spring runner is alias of @SpringJUnit4ClassRunner

@SpringBootTest

Controller:

Listing 1.4 The homepage controller

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller                                     <-- The controller
public class HomeController {

  @GetMapping("/")                              <-- Handles requests for the root path /
  public String home() {
    return "home";                              <-- Returns the view name
  }

}
```

Thymeleaf:

Listing 1.5 The Taco Cloud homepage template

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Welcome to...</h1>
    <img th:src="@{/images/TacoCloud.png}"/>
  </body>
</html>
```

Testing controller:

```
@RunWith(SpringRunner.class)
@WebMvcTest(HomeController.class)
public class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @org.junit.Before
    public void setUp() throws Exception {
    }

    @Test
    public void testController() throws Exception{
        mockMvc.perform(get("/home"))
                .andExpect(status().isOk())
                .andExpect(view().name("home"))
                .andExpect(content().string(
                        containsString("Welcome to...")));
    }

    @org.junit.After
    public void tearDown() throws Exception {
    }
}
```

@WebMvcTest (HomeController.*class*)

Intialize Spring MVC context and put controller into so that request can be send against controller.

2.1.2. Developing Spring application.

Simple MVC model

Domain:

Sample or order domain class:

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

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private Date createAt;

    @NotBlank(message="Name is required")
    private String name;
    @NotBlank(message="Street is required")
    private String street;
    @NotBlank(message="City is required")
    private String city;
    @NotBlank(message="State is required")
    private String state;
    @NotBlank(message="Zip code is required")
    private String zip;
    @CreditCardNumber(message="Not a valid credit card number")
    private String ccNumber;
    @Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$",
            message="Must be formatted MM/YY")
    private String ccExpiration;
    @Digits(integer=3, fraction=0, message="Invalid CVV")
    private String ccCVV;

    @ManyToMany(targetEntity = Design.class)
    @Size(min = 1, message = "You must at least choose one component")
    @JoinTable(
            name = "DESIGN_ORDER",
            joinColumns = @JoinColumn(name = "ORDER_ID"),
            inverseJoinColumns = @JoinColumn(name = "DESIGN_ID")
    )
    private List<Design> designs = new ArrayList<>();

    public void addDesign(Design design) {
        this.designs.add(design);
    };

    @PrePersist
    void createAt(){
        this.createAt = new Date();
    }
}
```

Data Validation require explicit:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Controller:

```
@Slf4j
@Controller
@RequestMapping("/orders")
public class OrderController {
    @GetMapping("/current")
    public String viewCurrentOrder(Model model){
        model.addAttribute("order", new Order());
        return "orders/orderForm";
    };

    @PostMapping
    public String processOrder(@Valid @ModelAttribute("order") Order order, Errors errors) {
        if(errors.hasErrors()){
            return "orders/orderForm";
        }
        log.info("Errors {}", errors);
        log.info("Order {}", order);
        log.info("Order Saved");
        log.info("Order submitted: " + order);
        return "redirect:/home";
    }
}
```

If view action is empty it POST to the same path.

Registry simple view handler:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry){
        registry.addViewController("/")
                .setViewName("home");
    };
}
```

2.2. Spring configuration

2.2.1. Fine-tuning autoconfiguration

Before we dive in too deeply with configuration properties, it's important to establish that there are two different (but related) kinds of configurations in Spring:

- **Bean wiring**—Configuration that declares application components to be created as beans in the Spring application context and how they should be injected into each other.

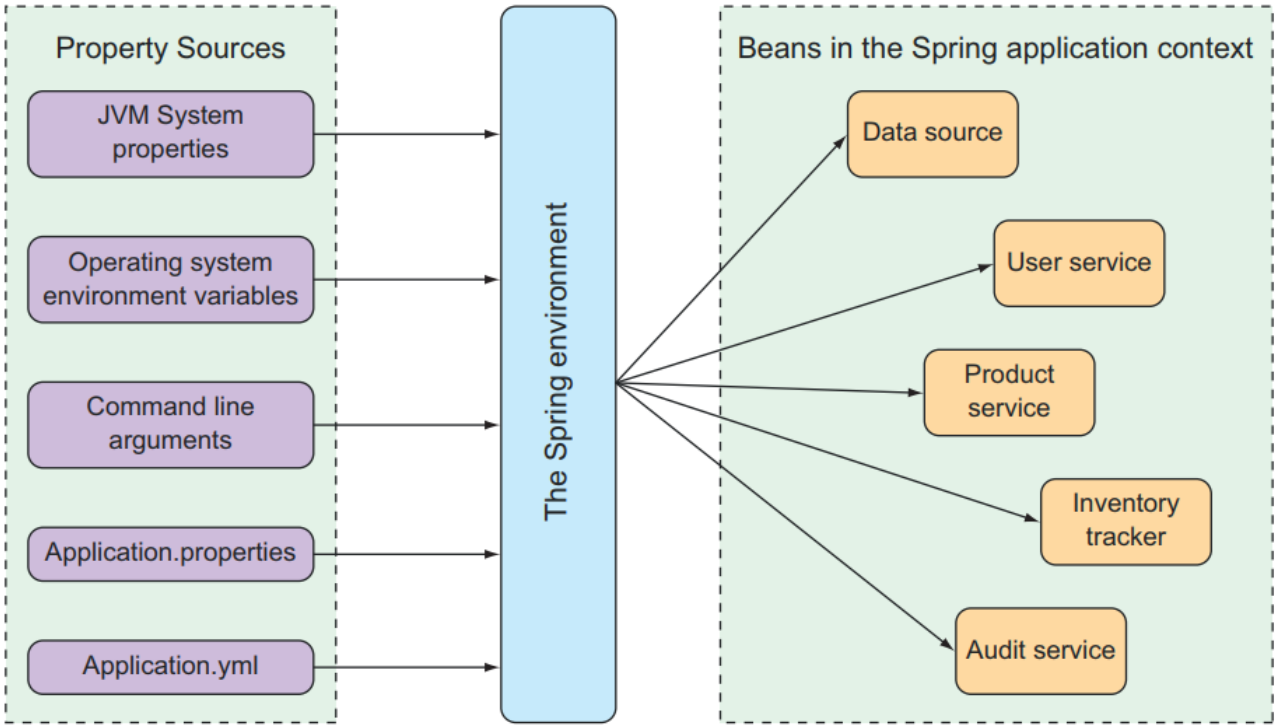
```
@Bean
public DataSource dataSource() {
  return new EmbeddedDataSourceBuilder()
       .setType(H2)
       .addScript("taco_schema.sql")
       .addScripts("user_data.sql", "ingredient_data.sql")
       .build();
}
```

- **Property injection**—Configuration that sets values on beans in the Spring application context.

► ***Spring env abstraction***

The Spring environment abstraction is a one-stop shop for any configurable property. It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself. The Spring environment pulls from several property sources, including:

- JVM system properties
- Operating system environment variables
- Command-line arguments
- Application property configuration files



- ***Sample data source configuration***

```
  spring:
    datasource:
      url: jdbc:mysql://localhost/tacocloud
      username: tacodb
      password: tacopassword
      driver-class-name: com.mysql.jdbc.Driver
```

```
spring:
  datasource:
    schema:
    - order-schema.sql
    - ingredient-schema.sql
    - taco-schema.sql
    - user-schema.sql
    data:
    - ingredients.sql
```

- ***Sample configuration embedded server***

But there's more to the underlying server than just a port. One of the most common things you'll need to do with the underlying container is to set it up to handle HTTPS requests. To do that, the first thing you must do is create a keystore using the JDK's `keytool` command-line utility:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

You'll be asked several questions about your name and organization, most of which are irrelevant. But when asked for a password, remember what you choose. For the sake of this example, I chose letmein as the password.

Next, you'll need to set a few properties to enable HTTPS in the embedded server. You could specify them all on the command line, but that would be terribly inconvenient. Instead, you'll probably set them in the file's application.properties or application.yml. In application.yml, the properties might look like this:

```
server:
  port: 8443
  ssl:
    key-store: file:///path/to/mykeys.jks
    key-store-password: letmein
    key-password: letmein
```

- ***Sample configuring logging***

Most applications provide some form of logging. And even if your application doesn't log anything directly, the libraries that your application uses will certainly log their activity.

By default, Spring Boot configures logging via Logback (http://logback.qos.ch) to write to the console at an INFO level. You've probably already seen plenty of INFO-level entries in the application logs as you've run the application and other examples.

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here's an example of a simple logback.xml file you might use:

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <logger name="root" level="INFO"/>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

2.2.2. Custom configuration properties

Listing 5.1 Enabling configuration properties in `OrderController`

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
@ConfigurationProperties(prefix="taco.orders")
public class OrderController {

  private int pageSize = 20;

  public void setPageSize(int pageSize) {
    this.pageSize = pageSize;
  }

  ...

  @GetMapping
  public String ordersForUser(
      @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, pageSize);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
  }

}
```

The most significant change made in listing 5.1 is the addition of the **@ConfigurationProperties** annotation. Its prefix attribute is set to **taco.orders**, which means that when setting the pageSize property, you need to use a configuration property named **taco.orders.pageSize**.

The new **pageSize** property defaults to 20. But you can easily change it to any value you want by setting a **taco.orders.pageSize** property. For example, you could set this property in application.yml like this: taco: orders: pageSize: **10.**

- ***Configuration holder***

Listing 5.2 Extracting `pageSize` to a holder class

```
package tacos.web;
import org.springframework.boot.context.properties.
                                        ConfigurationProperties;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {

  private int pageSize = 20;

}

@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

  Pageable pageable = PageRequest.of(0, props.getPageSize());
  model.addAttribute("orders",
      orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

  return "orderList";
}
```

- ***Sample profile configuration***

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

logging:
  level:
    tacos: WARN
```

If you're running the application as an executable JAR file, you might also set the active profile with a command-line argument like this:

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

3. Spring JPA

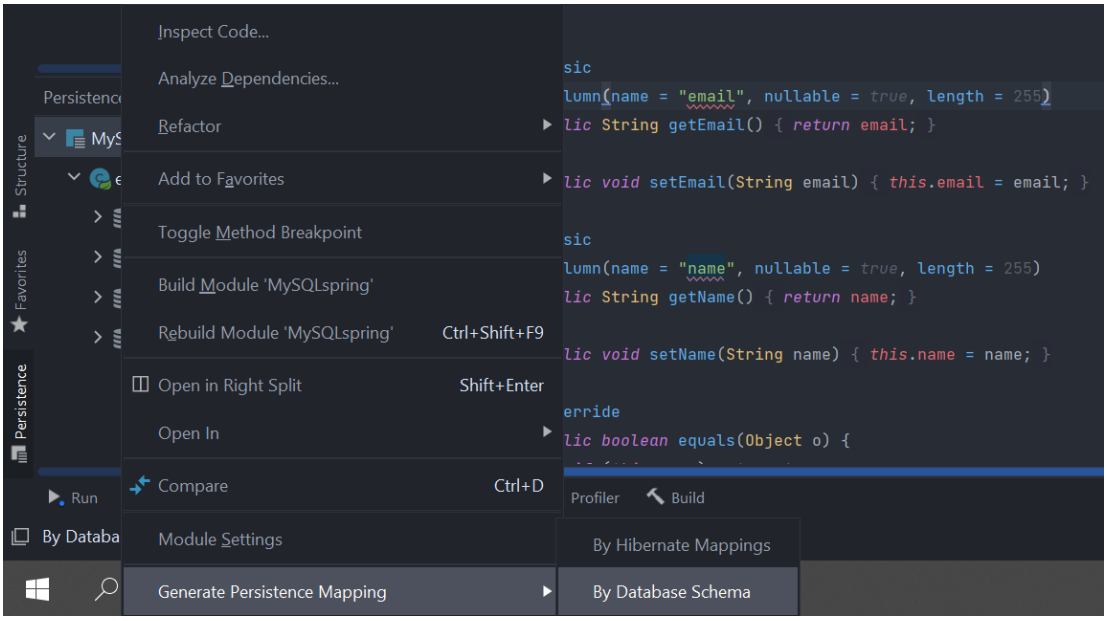
Quick note: Spring DB Mysql

How to configure MySQL:

1. Configurate Appliation-dev properties

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://10.91.24.2:3306/sfg_dev
spring.datasource.username=dev_user
spring.datasource.password=dev_user
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-
platform=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true
```

2. Configurate data source, import entity:



3.1. JPA Overview

- ***Spring JDBC***

Spring JDBC support is rooted in the JdbcTemplate class. JdbcTemplate provides a means by which developers can perform SQL operations against a relational database without all the ceremony and boilerplate typically required when working with JDBC.

3.1.1.1. *Requirement:*

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jdbc</artifactId>
    <version>2.2.5</version>
  </dependency>

</dependencies>
```

3.1.1.2. *Spring Data JDBC configuration*

```
@Configuration
@EnableJdbcRepositories                                                    (1)
class ApplicationConfig extends AbstractJdbcConfiguration {                (2)

    @Bean
    public DataSource dataSource() {                                       (3)

        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setType(EmbeddedDatabaseType.HSQL).build();
    }

    @Bean
    NamedParameterJdbcOperations namedParameterJdbcOperations(DataSource dataSource) {  (4)
        return new NamedParameterJdbcTemplate(dataSource);
    }

    @Bean
    TransactionManager transactionManager(DataSource dataSource) {         (5)
        return new DataSourceTransactionManager(dataSource);
    }
}
```

(1) `@EnableJdbcRepositories` creates implementations for interfaces derived from `Repository`

(2) `AbstractJdbcConfiguration` provides various default beans required by Spring Data JDBC

(3) Creates a `DataSource` connecting to a database.
This is required by the following two bean methods.

(4) Creates the `NamedParameterJdbcOperations` used by Spring Data JDBC to access the database.

(5) Spring Data JDBC utilizes the transaction management provided by Spring JDBC.

The configuration class in the preceding example sets up an embedded HSQL database by using the **EmbeddedDatabaseBuilder** API of spring-jdbc.

The **DataSource** is then used to set up **NamedParameterJdbcOperations** and a **TransactionManager**.

3.1.1.3. *Dialect:*

- ***A database dialect is a configuration setting for platform independent software (JPA, Hibernate, etc) which allows such software to translate its generic SQL statements into vendor specific DDL, DML.***

Spring Data JDBC uses implementations of the interface Dialect to encapsulate behavior that is specific to a database or its JDBC driver.

By default, the AbstractJdbcConfiguration tries to determine the database in use and register the correct Dialect.

This behavior can be changed by overwriting jdbcDialect(NamedParameterJdbcOperations).

3.1.1.4. *Persisting entities*

Saving an aggregate can be performed with the CrudRepository.save(…) method. If the aggregate is new, this results in an insert for the aggregate root, followed by insert statements for all directly or indirectly referenced entities.

If the aggregate root is not new, all referenced entities get deleted, the aggregate root gets updated, and all referenced entities get inserted again. Note that whether an instance is new is part of the instance's state.

3.1.1.5. *Object Mapping fundamentals*

This section covers the fundamentals of Spring Data object mapping, object creation, field and property access, mutability and immutability.

Note, that this section only applies to Spring Data modules that do not use the object mapping of the underlying data store (like JPA).

Also be sure to consult the store-specific sections for store-specific object mapping, like indexes, customizing column or field names or the like.

Core responsibility of the Spring Data object mapping is to create instances of domain objects and map the store-native data structures onto those. This means we need two fundamental steps:

- **Instance creation by using one of the constructors exposed.**
- **Instance population to materialize all exposed properties.**

► ***Object creation***

- If there is a single constructor, it is used.
- If there are multiple constructors and exactly one is annotated with `@PersistenceConstructor`, it is used.
- If there's a no-argument constructor, it is used. Other constructors will be ignored.

▶ *Object internal creation*

Object creation internals

To avoid the overhead of reflection, Spring Data object creation uses a factory class generated at runtime by default, which will call the domain classes constructor directly. I.e. for this example type:

JAVA

```
class Person {
  Person(String firstname, String lastname) { … }
}
```

we will create a factory class semantically equivalent to this one at runtime:

JAVA

```
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

This gives us a roundabout 10% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

▶ *Property Population*

```
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              2

  private Person person;                                    1

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             2
    } else if ("id".equals(name)) {
      this.person = person.withId((Long) value);            3
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);              4
    }
  }
}
```

Sample entity

```
class Person {

  private final @Id Long id;                                                1
  private final String firstname, lastname;                                 2
  private final LocalDate birthday;
  private final int age;                                                    3

  private String comment;                                                   4
  private @AccessType(Type.PROPERTY) String remarks;                        5

  static Person of(String firstname, String lastname, LocalDate birthday) { 6

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age) { 6

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                                  1
    return new Person(id, this.firstname, this.lastname, this.birthday, this.age);
  }

  void setRemarks(String remarks) {                                         5
    this.remarks = remarks;
  }
}
```

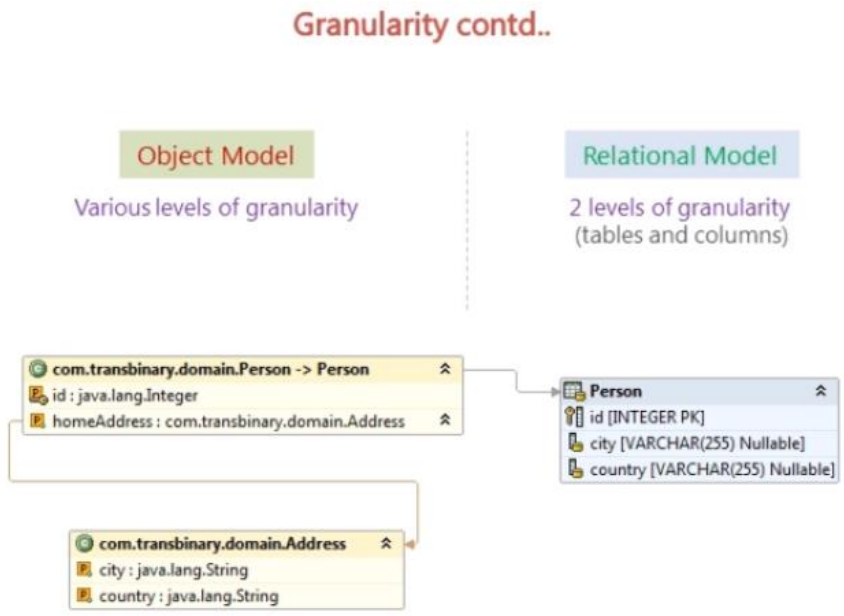
Entity convention:

- **Create immutable property**, use constructor only to materialize entity is faster.
- **Provide all-args constructor**, this will skip the part of property population.
- **Use factory methods** instead of overloaded constructors, example withID
- For **identifiers** to be generated, still use a final field in combination with an all-arguments persistence constructor (preferred) or a with… method
- Use Lombok

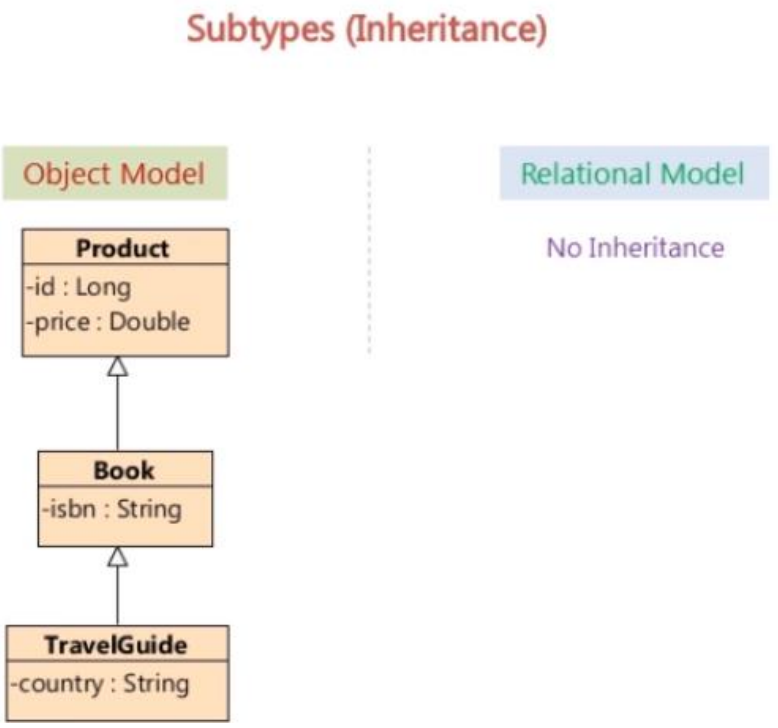
3.2. Hibernate

3.2.1.1. *Object relational impdedance mismatch.*

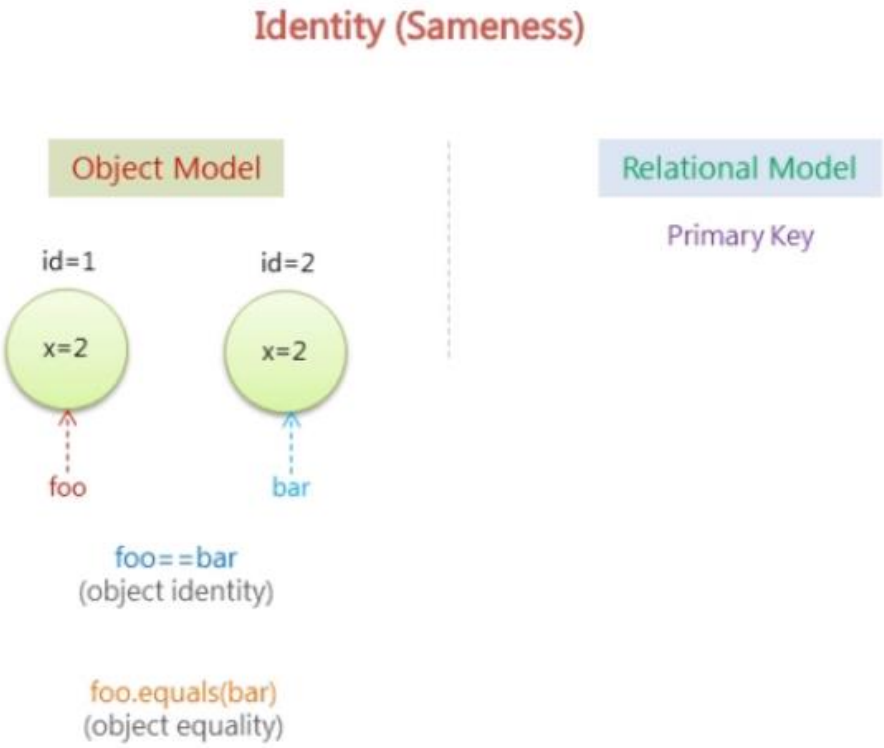
- ***Granularity:***



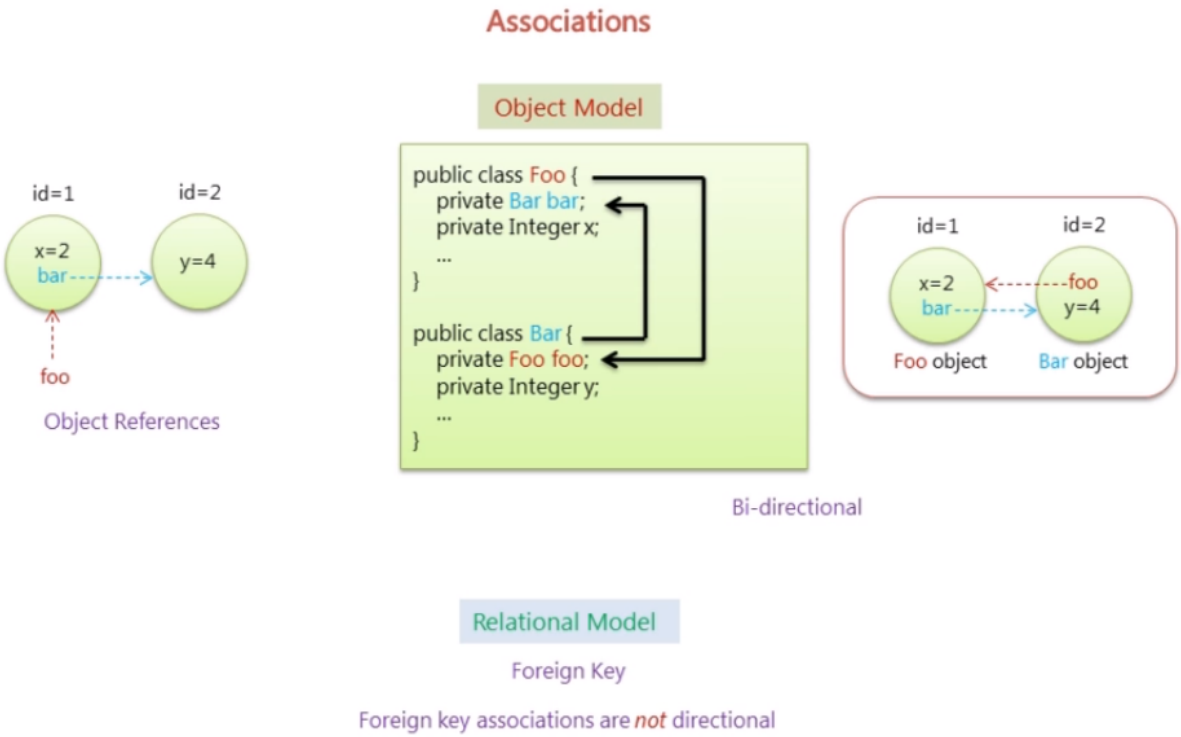
- ***Inheritance***



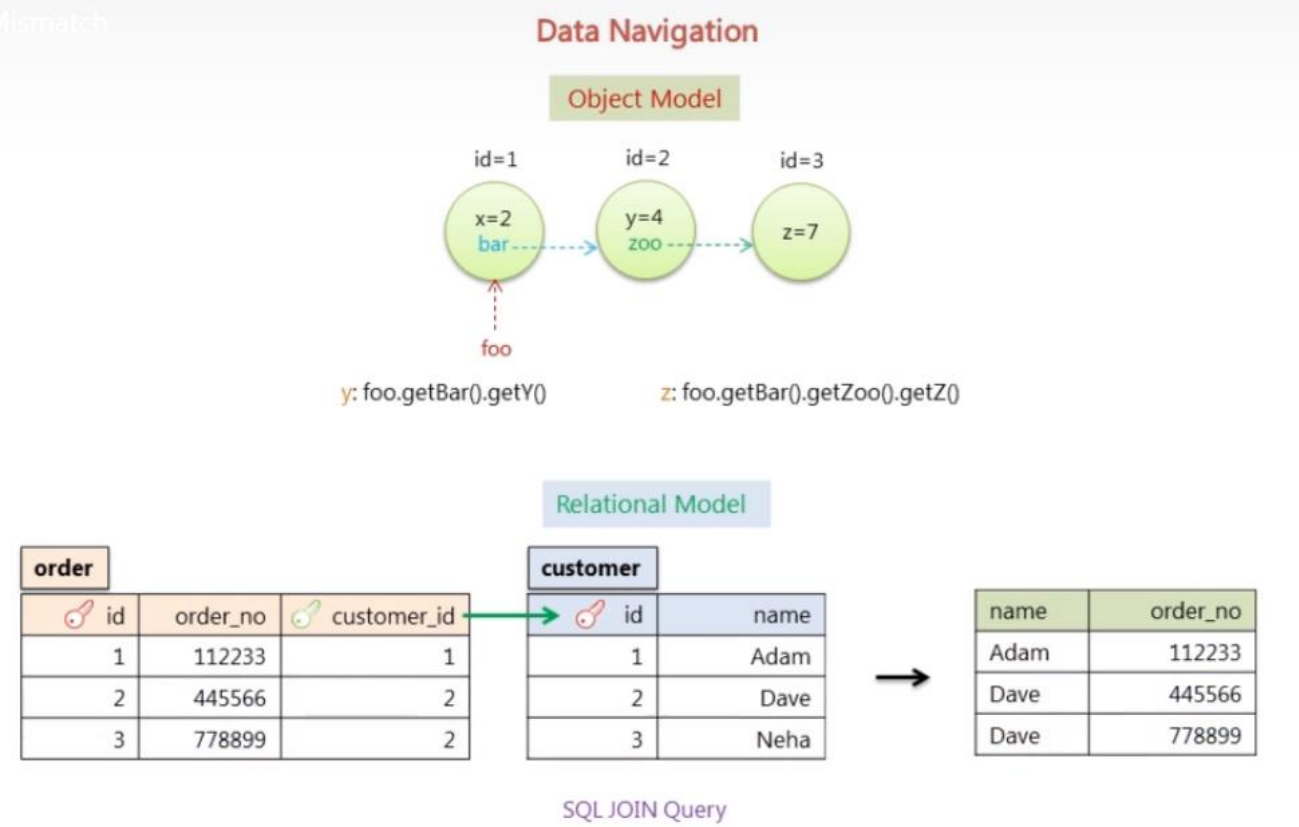
- ***Identity***



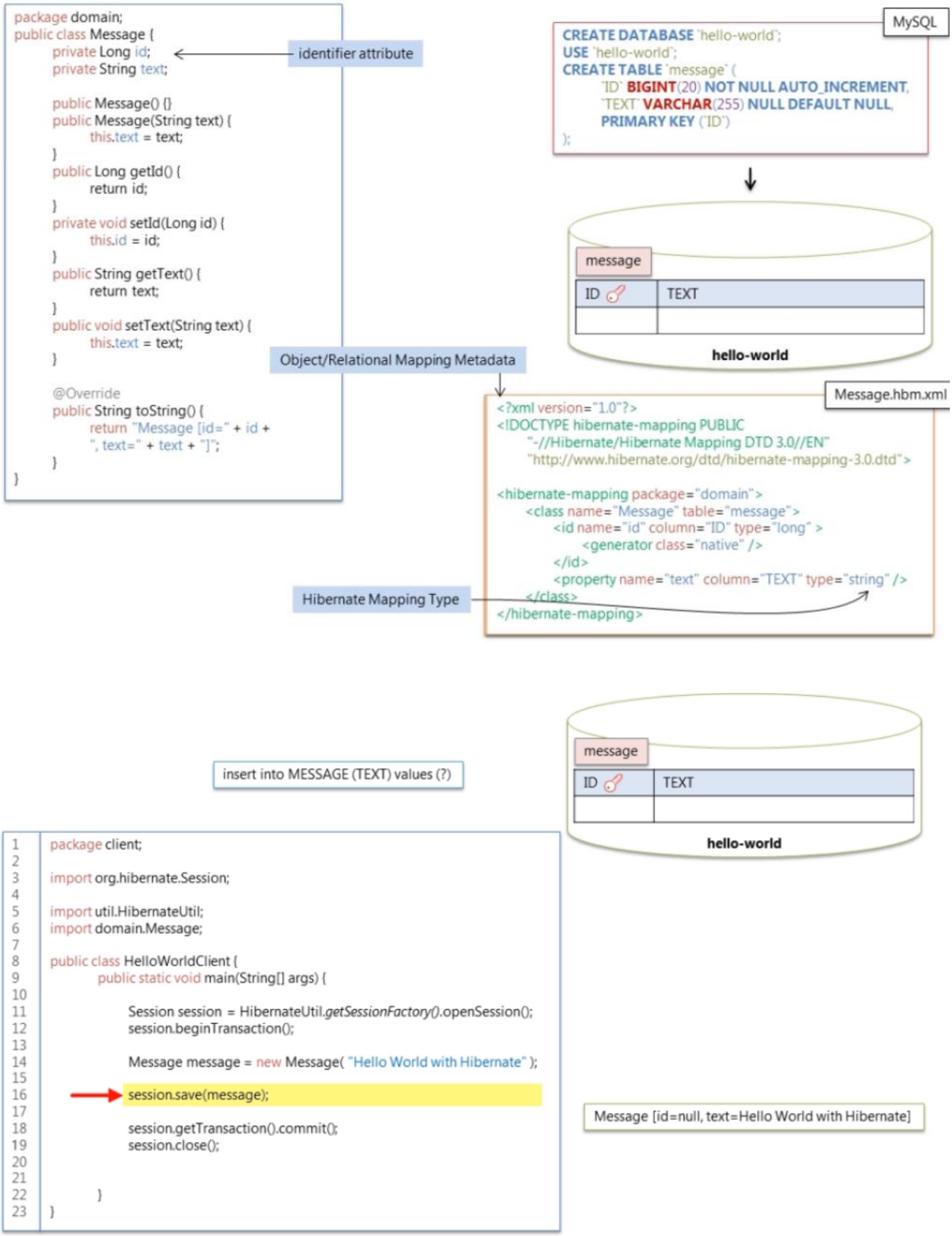
- ***Association***



► *Data navigation*



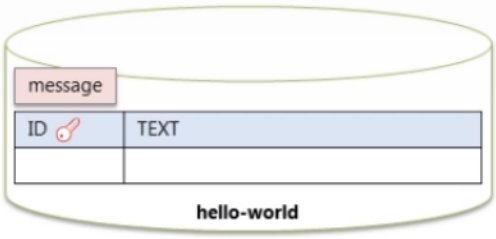
3.2.1.2. *Hibernate and JPA annotation.*



Annotation-based Metadata

Using JPA Annotations

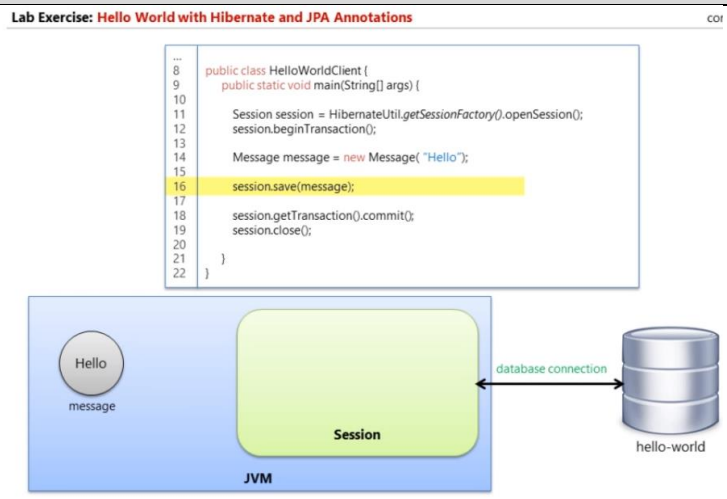
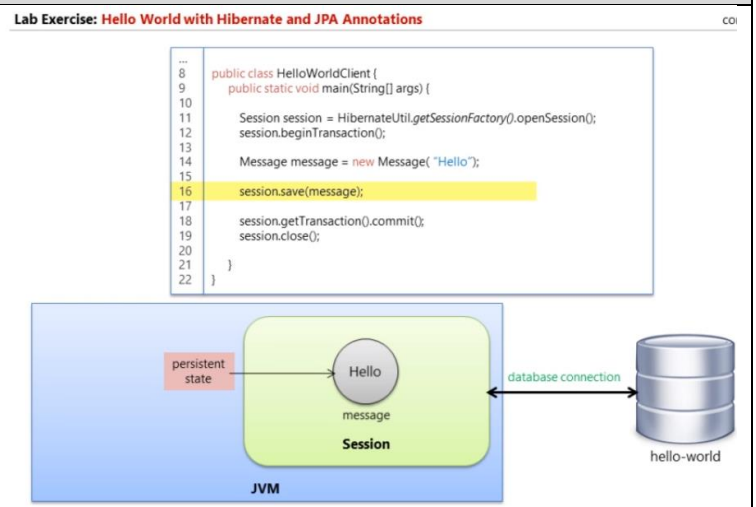
```java
package entity;

import javax.persistence.*;

@Entity
@Table(name="message")
public class Message {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private Long id;

    @Column(name="TEXT")
    private String text;

    public Message() {}
    public Message(String text) {
        this.text = text;
    }

    //other methods
}
```



Message.hbm.xml

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="domain">
    <class name="Message" table="message">
        <id name="id" column="ID">
            <generator class="native" />
        </id>
        <property name="text" column="TEXT" type="string" />
    </class>
</hibernate-mapping>
```

XML-based Metadata

Transistent state	Persistence state
	

► *Transaction handling*

Transactions

A transaction is a group of operations that are run as a single unit of work

```
1 start transaction;
2
3 delete from book where id=4;
4 insert into invoice(id, detail) values(4, 'isbn 9781617290459 ordered');
5
6 commit;
7
```

← tells the database to commit any changes made in this transaction

```
1 start transaction;
2
3 delete from book where id=4;
4 insert into invoice(id, detail) values(4, 'isbn 9781617290459 ordered');
5
6 rollback;
7
```

← tells the database to rollback any changes made in this transaction

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction txn = session.getTransaction();
try {
    txn.begin();

    //saving, finding, updating, deleting objects

    txn.commit();
}   catch(Exception e) {
        if(txn != null) { txn.rollback(); }
        e.printStackTrace();
}   finally {
        if(session != null) { session.close(); }
}
```

Finding Objects

```
1  package client;
2
3  import org.hibernate.Session;
4  import org.hibernate.Transaction;
5
6  import util.HibernateUtil;
7  import entity.Message;
8
9
10 public class HelloWorldClient {
11         public static void main(String[] args) {
12                 Session session = HibernateUtil.getSessionFactory().openSession();
13                 Transaction txn = session.getTransaction();
14                 try {
15                         txn.begin();
16
17                         Message msg = (Message) session.get(Message.class, 2L);
18
19                         txn.commit();
20                 }       catch(Exception e) {
21                           if(txn != null) { txn.rollback(); }
22                           e.printStackTrace();
23                 }       finally {
24                                 if(session != null) { session.close(); }
25                 }
26         }
27 }
```

hello-world.message: 4 rows total (approximately)

ID	TEXT
1	Hello World with Hibernate
2	Hello World with Hibernate and JPA Annotations
3	JBOSS Logging Library with Log4j in Action
4	JBOSS Logging Library with Log4j in Action, r2

Updating Objects

```
1  package client;
2
3  import org.hibernate.Session;
4  import org.hibernate.Transaction;
5
6  import util.HibernateUtil;
7  import entity.Message;
8
9
10 public class HelloWorldClient {
11         public static void main(String[] args) {
12               Session session = HibernateUtil.getSessionFactory().openSession();
13               Transaction txn = session.getTransaction();
14               try {
15                     txn.begin();
16
17                     Message msg = (Message) session.get(Message.class, 2L);
18                     msg.setText( "Hello Automatic Dirty Checking" );
19
20                     txn.commit();
21               }     catch(Exception e) {
22                      if(txn != null) { txn.rollback(); }
23                      e.printStackTrace();
24               }     finally {
25                           if(session != null) { session.close(); }
26               }
27         }
28 }
```

- ***Automatic dirty checking***

Check for change in record, only update when there is change.

Deleting Objects

```
1  package client;
2
3  import org.hibernate.Session;
4  import org.hibernate.Transaction;
5
6  import util.HibernateUtil;
7  import entity.Message;
8
9
10 public class HelloWorldClient {
11         public static void main(String[] args) {
12               Session session = HibernateUtil.getSessionFactory().openSession();
13               Transaction txn = session.getTransaction();
14               try {
15                     txn.begin();
16
17                     Message msg = (Message) session.get(Message.class, 2L);
18                     session.delete(msg);
19
20                     txn.commit();
21               }     catch(Exception e) {
22                      if(txn != null) { txn.rollback(); }
23                      e.printStackTrace();
24               }     finally {
25                           if(session != null) { session.close(); }
26               }
27         }
28 }
```

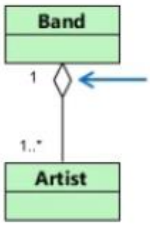
- ***Aggregation and composition.***

Aggregation

Aggregation indicates a relationship between a whole and its parts

A music band is made up of artists

If the band is broken, artists are *not* broken
(i.e. artists could still join some other bands)



Music Band

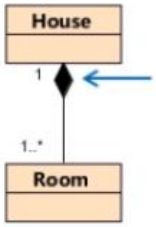
In an aggregation relationship, when the whole is destroyed, its parts are *not* destroyed with it

Composition

Composition is a *strong* form of aggregation

Each part may belong to only one whole (No sharing)

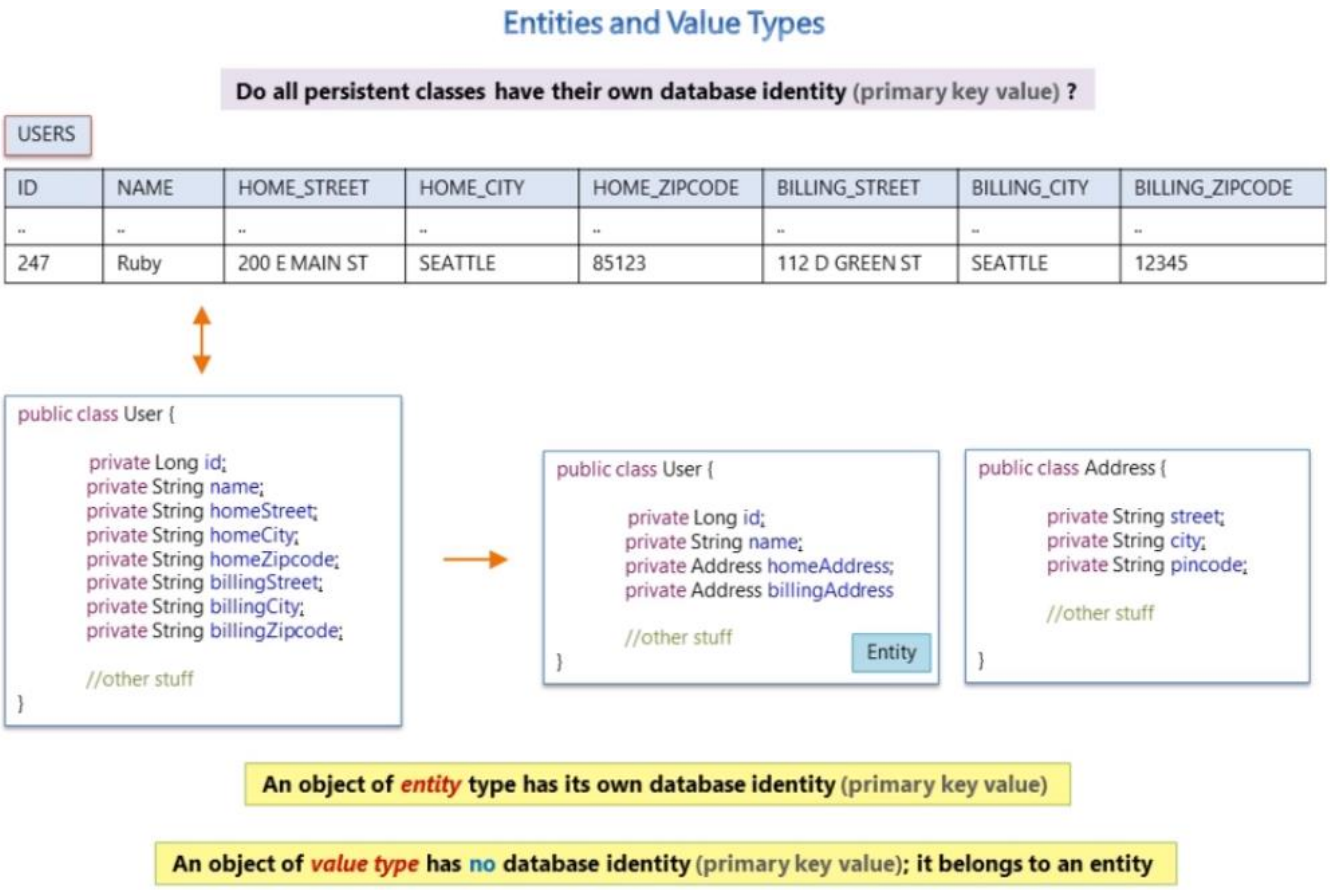
E.g. your room in your house
(it's not part of your neighbors' house)

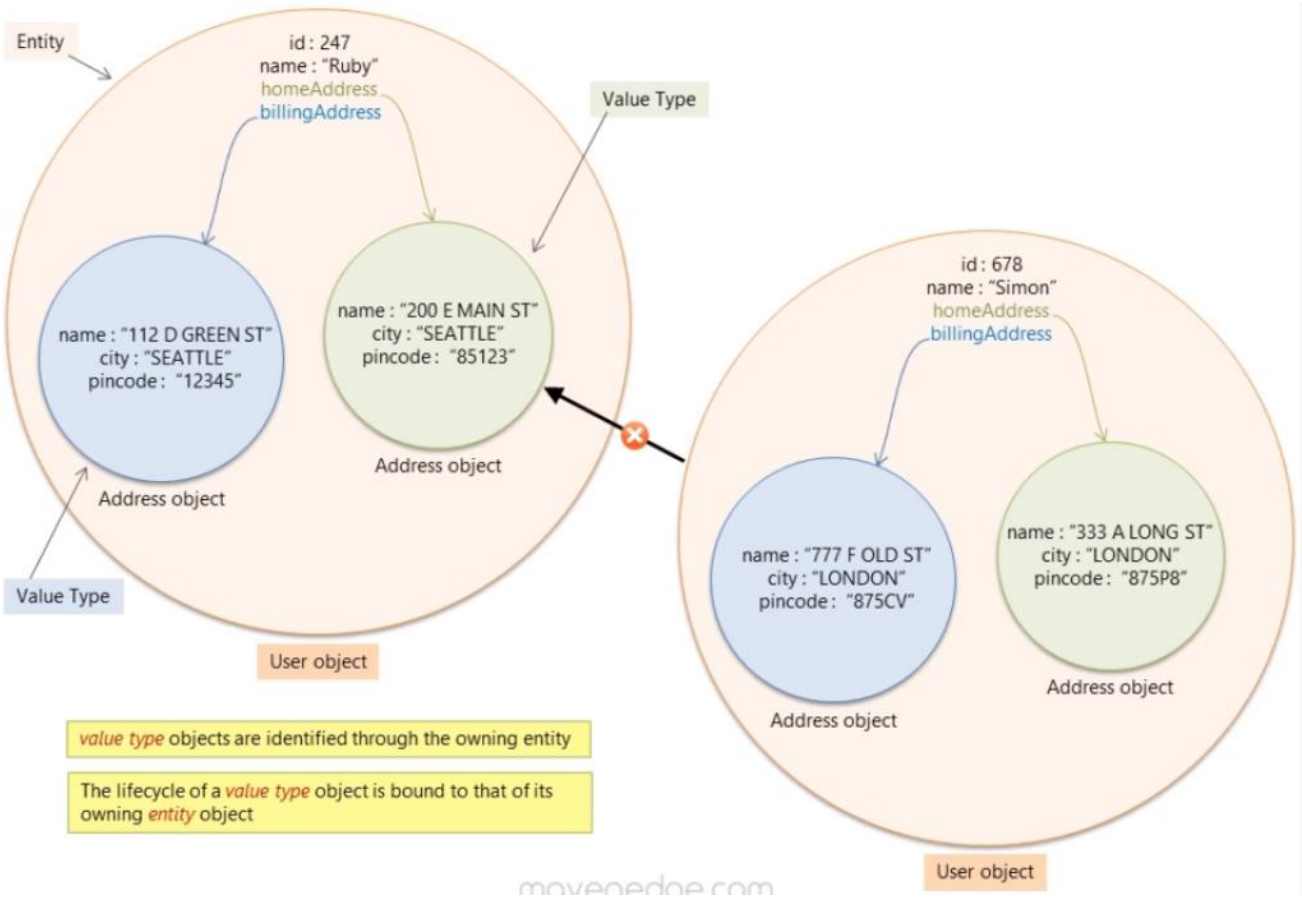


House

In a composition relationship, when the whole is destroyed its parts are *also* destroyed with it

- ***Entities and values type.***



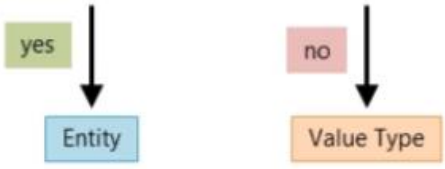


Classes like String and Integer are most simple *value type* classes

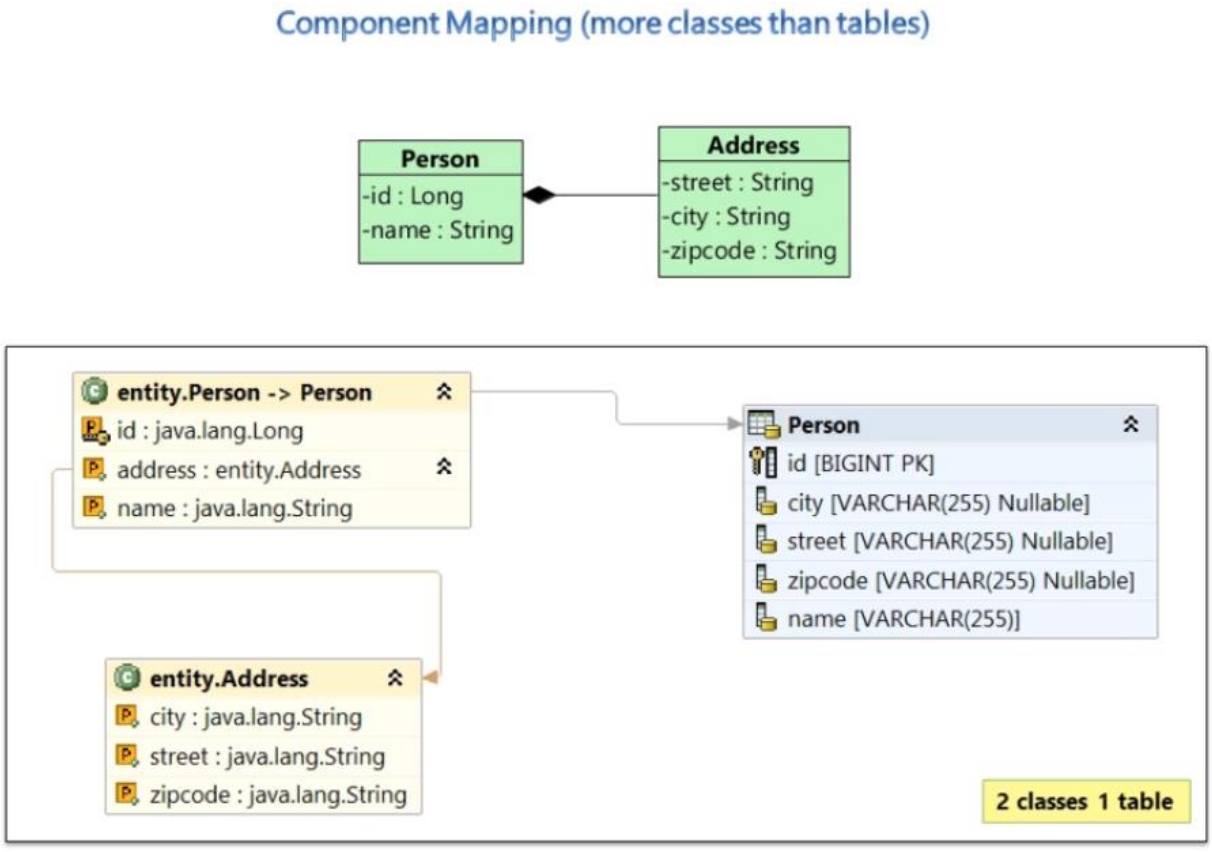
```
public class Address {

        private Long id;
        private String street;
        private String city;
        private String pincode;

        //other stuff

}
```

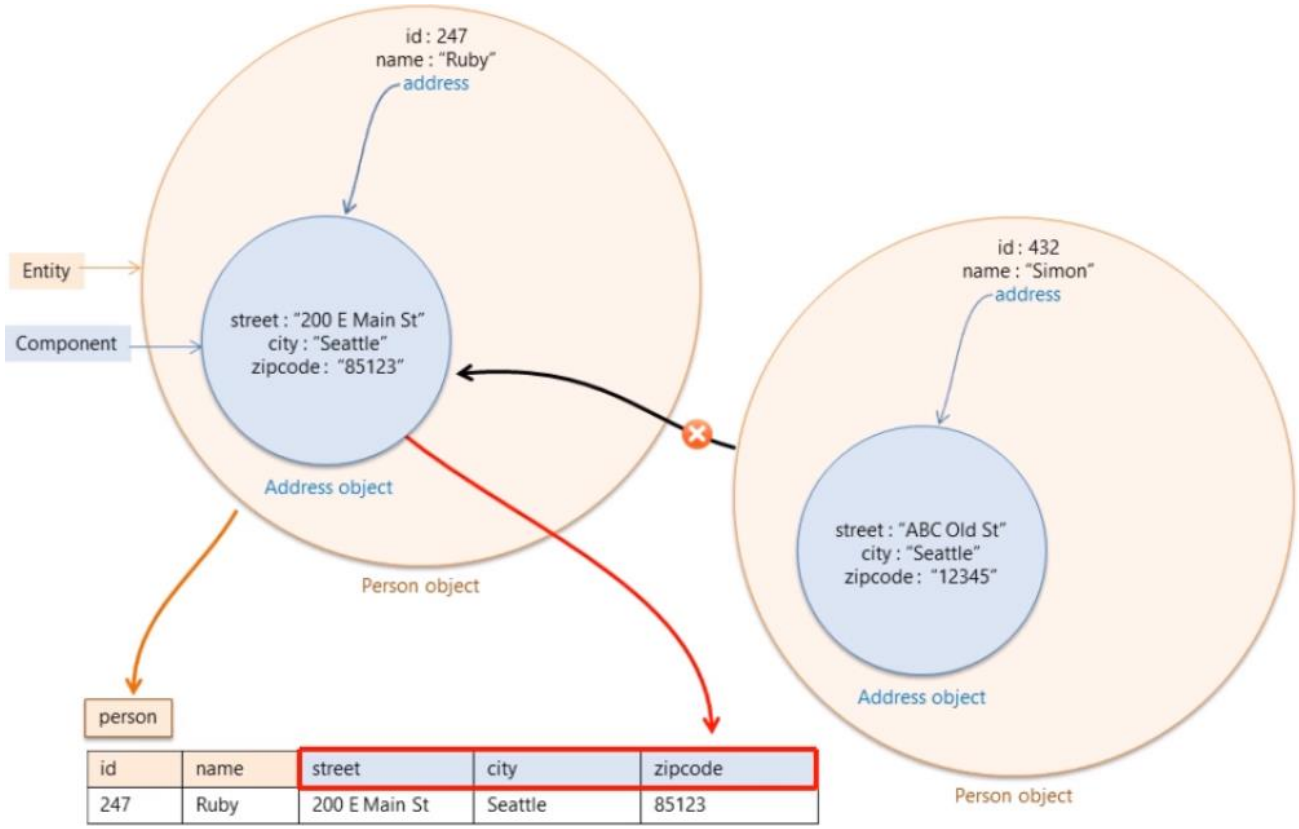
Entity

Does the database identity of an object matters ?



- ***Component mapping.***





A component has no individual identity

```java
package entity;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(nullable=false)
    private String name;

    @Embedded
    private Address address;

    public Person() {}
    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

}
```

```java
package entity;

import javax.persistence.Embeddable;

@Embeddable
public class Address {

    private String street;
    private String city;
    private String zipcode;

    public Address() {}
    public Address(String street, String city, String zipcode) {
        this.street = street;
        this.city = city;
        this.zipcode = zipcode;
    }

}
```

hibernate.cfg.xml

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hello-world</property>
        <property name="connection.username">root</property>
        <property name="connection.password">password</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Create/update database tables automatically using mapping metadata -->
        <property name="hbm2ddl.auto">update</property>

        <mapping class="entity.Person" />

    </session-factory>
</hibernate-configuration>
```

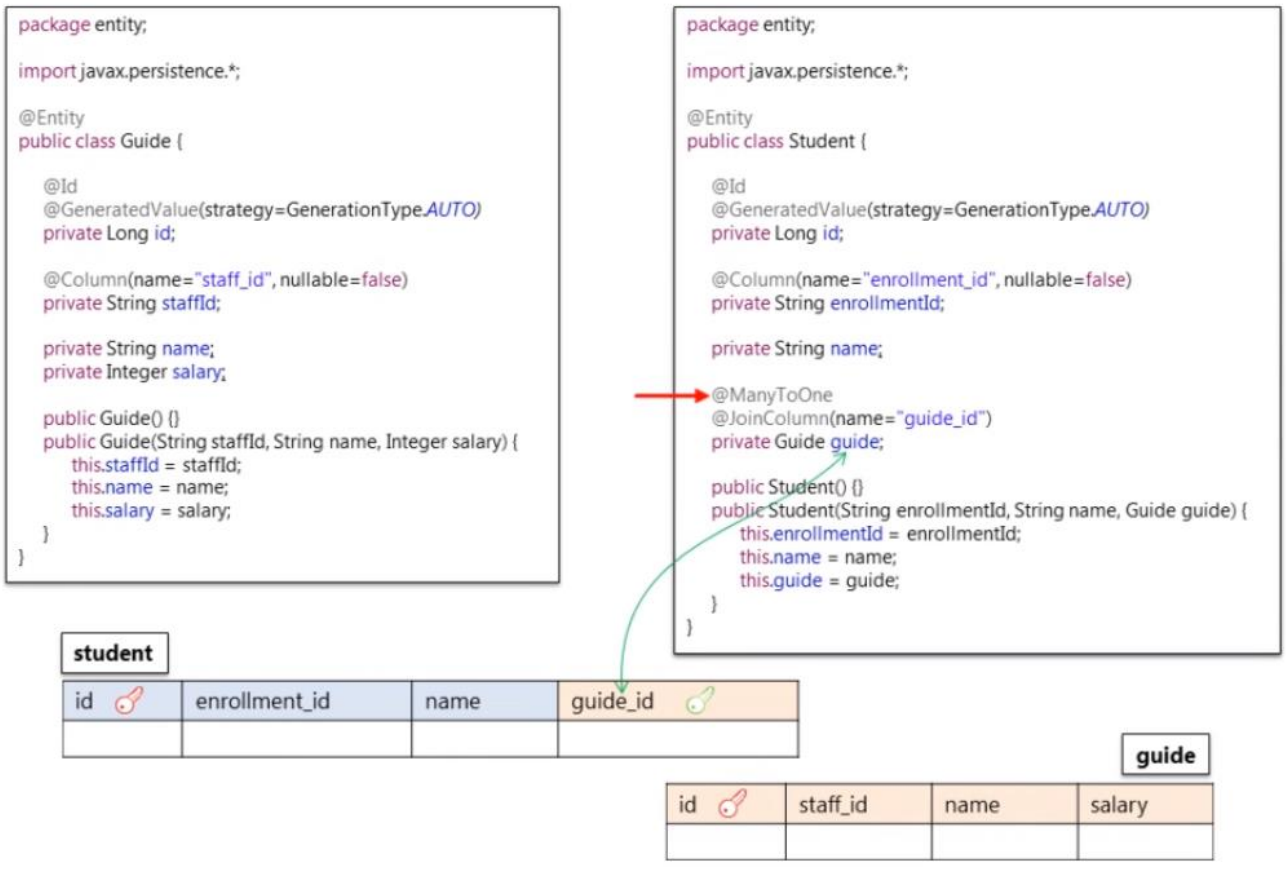
person

id	name	street	city	zipcode

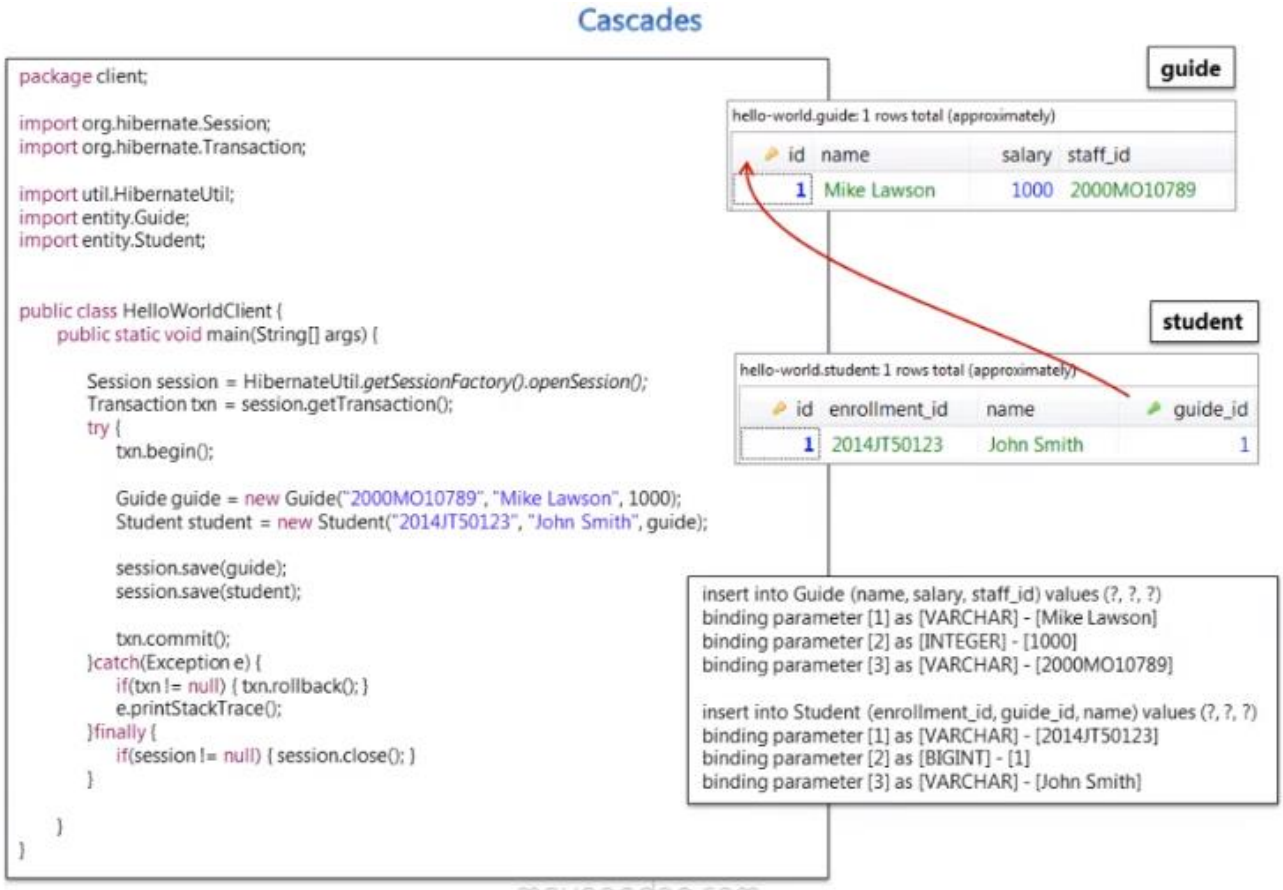
```java
package entity;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(nullable=false)
    private String name;

    @Embedded
    private Address address;

    public Person() {}
    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

}
```

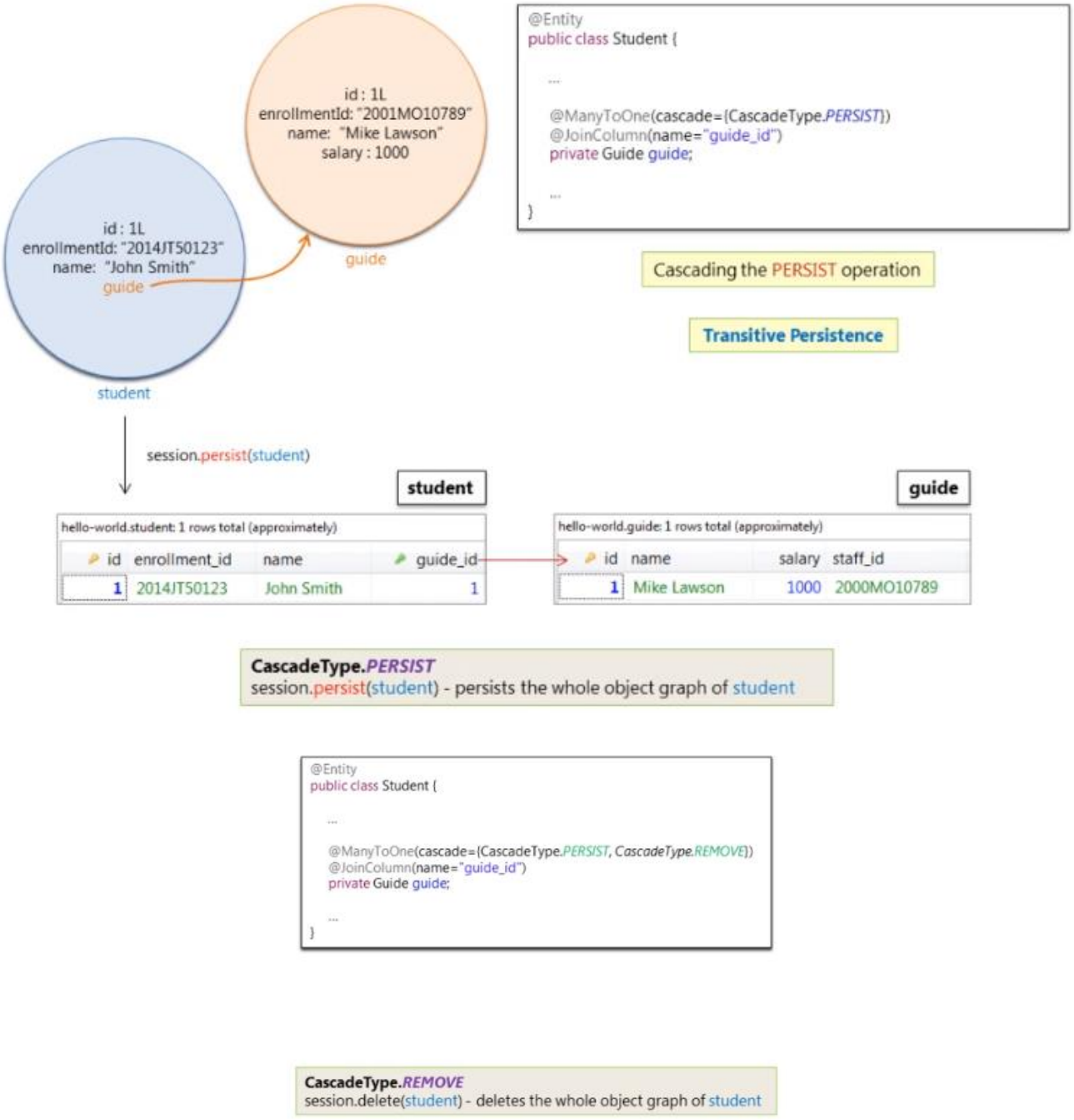
```java
package entity;

import javax.persistence.Embeddable;

@Embeddable
public class Address {

    private String street;
    private String city;
    private String zipcode;

    public Address() {}
    public Address(String street, String city, String zipcode) {
        this.street = street;
        this.city = city;
        this.zipcode = zipcode;
    }

}
```

▶ *Mapping association*



► *Cascade*


Cascades
package client;
import org.hibernate.Session;
import org.hibernate.Transaction;
import util.HibernateUtil;
import entity.Guide;
import entity.Student;
public class HelloWorldClient {
public static void main(String[] args) {
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction txn = session.getTransaction();
try {
txn.begin();
Guide guide = new Guide("2000MO10789", "Mike Lawson", 1000);
Student student = new Student("2014JT50123", "John Smith", guide);
session.save(guide);
session.save(student);
txn.commit();
}catch(Exception e) {
if(txn != null) { txn.rollback(); }
e.printStackTrace();
}finally {
if(session != null) { session.close(); }
}
}
}
guide
hello-world.guide: 1 rows total (approximately)
id name salary staff_id
1 Mike Lawson 1000 2000MO10789
student
hello-world.student: 1 rows total (approximately)
id enrollment_id name guide_id
1 2014JT50123 John Smith 1
insert into Guide (name, salary, staff_id) values (?, ?, ?)
binding parameter [1] as [VARCHAR] - [Mike Lawson]
binding parameter [2] as [INTEGER] - [1000]
binding parameter [3] as [VARCHAR] - [2000MO10789]
insert into Student (enrollment_id, guide_id, name) values (?, ?, ?)
binding parameter [1] as [VARCHAR] - [2014JT50123]
binding parameter [2] as [BIGINT] - [1]
binding parameter [3] as [VARCHAR] - [John Smith]




```
@ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
@JoinColumn(name = "guide_id")
private Guide guide;
```

```
transaction.begin();
Guide guide = Guide.builder().name("History Teacher").salary(1500).staffId("402").build();
Student student = Student.builder().guide(guide).enrollmentId("4110").name("Xuan").build();
session.persist(student);
transaction.commit();
```

Cascade delete:

```
package client;

import org.hibernate.Session;
import org.hibernate.Transaction;

import util.HibernateUtil;
import entity.Guide;
import entity.Student;


public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction txn = session.getTransaction();
        try {
            txn.begin();

            Student student = (Student) session.get(Student.class, 2L);
            session.delete(student);

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(session != null) { session.close(); }
        }

    }
}
```

```
package entity;

import javax.persistence.*;

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="enrollment_id", nullable=false)
    private String enrollmentId;

    private String name;

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name="guide_id")
    private Guide guide;

    public Student() {}
    public Student(String enrollmentId, String name, Guide guide) {
        this.enrollmentId = enrollmentId;
        this.name = name;
        this.guide = guide;
    }
}
```

student

hello-world.student: 1 rows total (approximately)

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1

guide

hello-world.guide: 1 rows total (approximately)

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789

Error Foreign key constraint:

Lab Exercise: Cascades contd.

```
8   Session session = HibernateUtil.getSessionFactory().openSession();
9   session.beginTransaction();
10
11  Student student = (Student) session.get(Student.class, 2L);
12  session.delete( student );
13
14  session.getTransaction().commit();
15  session.close();
```

```
delete from Guide where id=?

binding parameter [1] as [BIGINT] - [2]
```

```
@Entity
public class Student {
    //...

    @ManyToOne( cascade={ CascadeType.PERSIST, CascadeType.REMOVE } )
    @JoinColumn(name="guide_id")
    private Guide guide;
}
```

```
@Entity
public class Guide {
    //...

}
```

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
2	2014AL50456	Amy Gill	2
3	2014JO10333	Jane Roe	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

Figure 1

```
org.hibernate.exception.ConstraintViolationException: could not execute statement

Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
Cannot delete or update a parent row: a foreign key constraint fails
(`hello-world-labex`.`student`, CONSTRAINT `FK_kf6egxhj81a1pp6596ox4c0ul` FOREIGN KEY (`guide_id`) REFERENCES `guide` (`id`))
```

Lab Exercise: Cascades contd.

```
8   Session session = HibernateUtil.getSessionFactory().openSession();
9   session.beginTransaction();
10
11  Student student = (Student) session.get(Student.class, 2L);
12  student.setGuide( null );
13  session.delete( student );
14
15  session.getTransaction().commit();
16  session.close();
```

```
@Entity
public class Student {
   //...

   @ManyToOne( cascade={ CascadeType.PERSIST, CascadeType.REMOVE } )
   @JoinColumn(name="guide_id")
   private Guide guide;
}
```

```
@Entity
public class Guide {
   //...

}
```

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
3	2014JO10333	Jane Roe	2

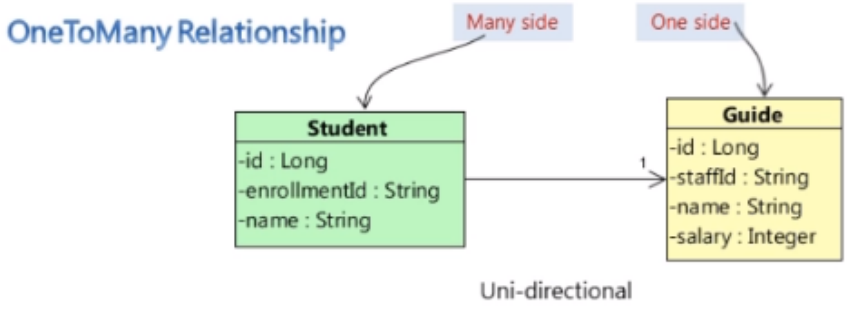
guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

Figure 3

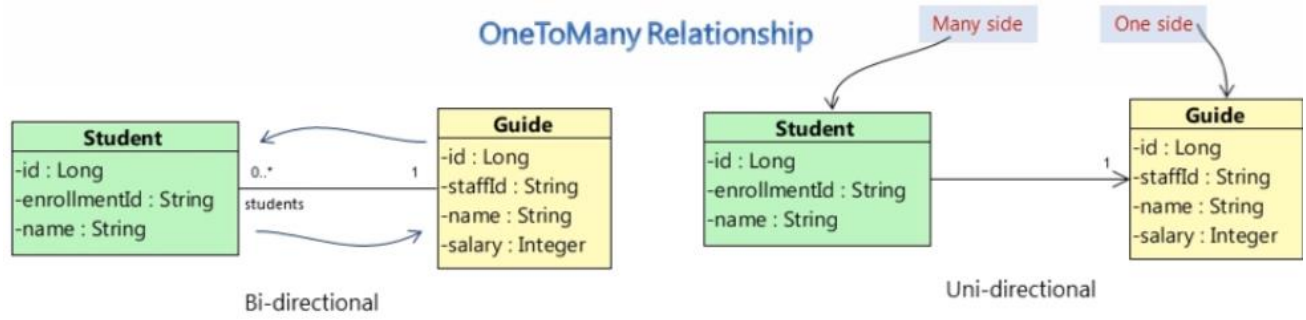
- ***One to Many relationship***

Current many to one mapping:




```
@Entity
public class Student {
   ...

   @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
   @JoinColumn(name="guide_id")
   private Guide guide;

   public Guide getGuide() { return guide; }
   ...
}
```

```
@Entity
public class Guide {
   ...

}
```




```java
@Entity
public class Student {
    ...

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name="guide_id")
    private Guide guide;

    public Guide getGuide() { return guide; }
    ...
}
```

```java
@Entity
public class Guide {
    ...

    @OneToMany(mappedBy="guide")
    private Set<Student> students = new Set<Student>();

    public Set<Student> getStudents() { return students; }
    ...
}
```

```java
@Entity
public class Student {
    ...

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @JoinColumn(name="guide_id")
    private Guide guide;

    public Guide getGuide() { return guide; }
    ...
}
```

```java
@Entity
public class Guide {
    ...

    @OneToMany(mappedBy="guide")
    private Set<Student> students = new Set<Student>();

    public Set<Student> getStudents() { return students; }
    ...
}
```

```java
public void saveGuide(){
    Transaction transaction = null;
    try(Session session = HibernateUtil.getSessionFactory().openSession()){
        try {
            transaction = session.getTransaction();
            transaction.begin();

            Guide guide1 = Guide.builder().name("Eng Teacher").salary(1200).staffId("501").students(new HashSet<>()).build();
            Guide guide2 = Guide.builder().name("Math Teacher").salary(1600).staffId("503").students(new HashSet<>()).build();

            Student student1 = Student.builder().guide(guide1).enrollmentId("4510").name("Nam").build();
            Student student2 = Student.builder().guide(guide2).enrollmentId("4511").name("Nam").build();

            guide1.getStudents().add(student1);
            guide2.getStudents().add(student2);

            session.persist(guide1);
            session.persist(guide2);

            transaction.commit();
        } catch (HibernateException exception){
            if (transaction != null) {
                transaction.rollback();
            }
            exception.printStackTrace();
        }

    }
}
```

```java
package entity;

import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="enrollment_id", nullable=false)
  private String enrollmentId;

  private String name;

  @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  @JoinColumn(name="guide_id")
  private Guide guide;

  public Student() {}
  public Student(String enrollmentId, String name, Guide guide) {
    this.enrollmentId = enrollmentId;
    this.name = name;
    this.guide = guide;
  }

  public Guide getGuide() { return guide; }
  public void setGuide(Guide guide) { this.guide = guide; }
}
```

owner

```java
package entity;

import javax.persistence.*;

@Entity
public class Guide {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="staff_id", nullable=false)
  private String staffId;

  private String name;
  private Integer salary;

  @OneToMany(mappedBy="guide",cascade={CascadeType.PERSIST})
  private Set<Student> students = new Set<Student>();

  public Guide() {}
  public Guide(String staffId, String name, Integer salary) {
    this.staffId = staffId;
    this.name = name;
    this.salary = salary;
  }

  public Set<Student> getStudents() { return students; }
}
```

inverse end

```java
package client;

import org.hibernate.Session;
import org.hibernate.Transaction;

import util.HibernateUtil;
import entity.Guide;
import entity.Student;

public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction txn = session.getTransaction();
        try {
            txn.begin();

            Guide guide1 = new Guide("2000MO10789", "Mike Lawson", 1000);
            Guide guide2 = new Guide("2000IM10901", "Ian Lamb", 2000);

            Student student1 = new Student("2014JT50123", "John Smith", guide1);
            Student student2 = new Student("2014AL50456", "Amy Gill", guide1);

            guide1.getStudents().add(student1);
            guide1.getStudents().add(student2);

            session.persist(guide1);
            session.persist(guide2);

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(session != null) { session.close(); }
        }
    }
}
```


student

hello-world.student: 2 rows total (approximately)

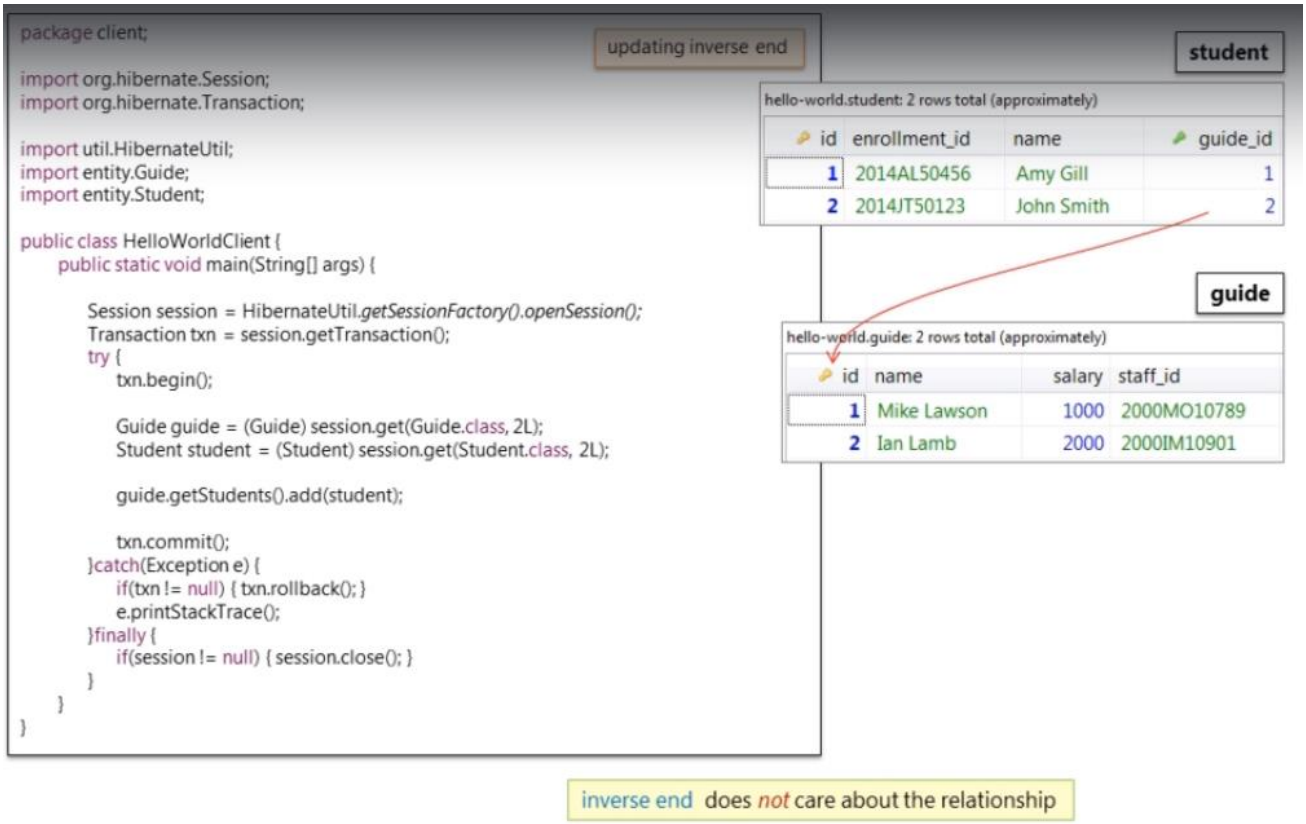
id	enrollment_id	name	guide_id
1	2014AL50456	Amy Gill	1
2	2014JT50123	John Smith	1

guide

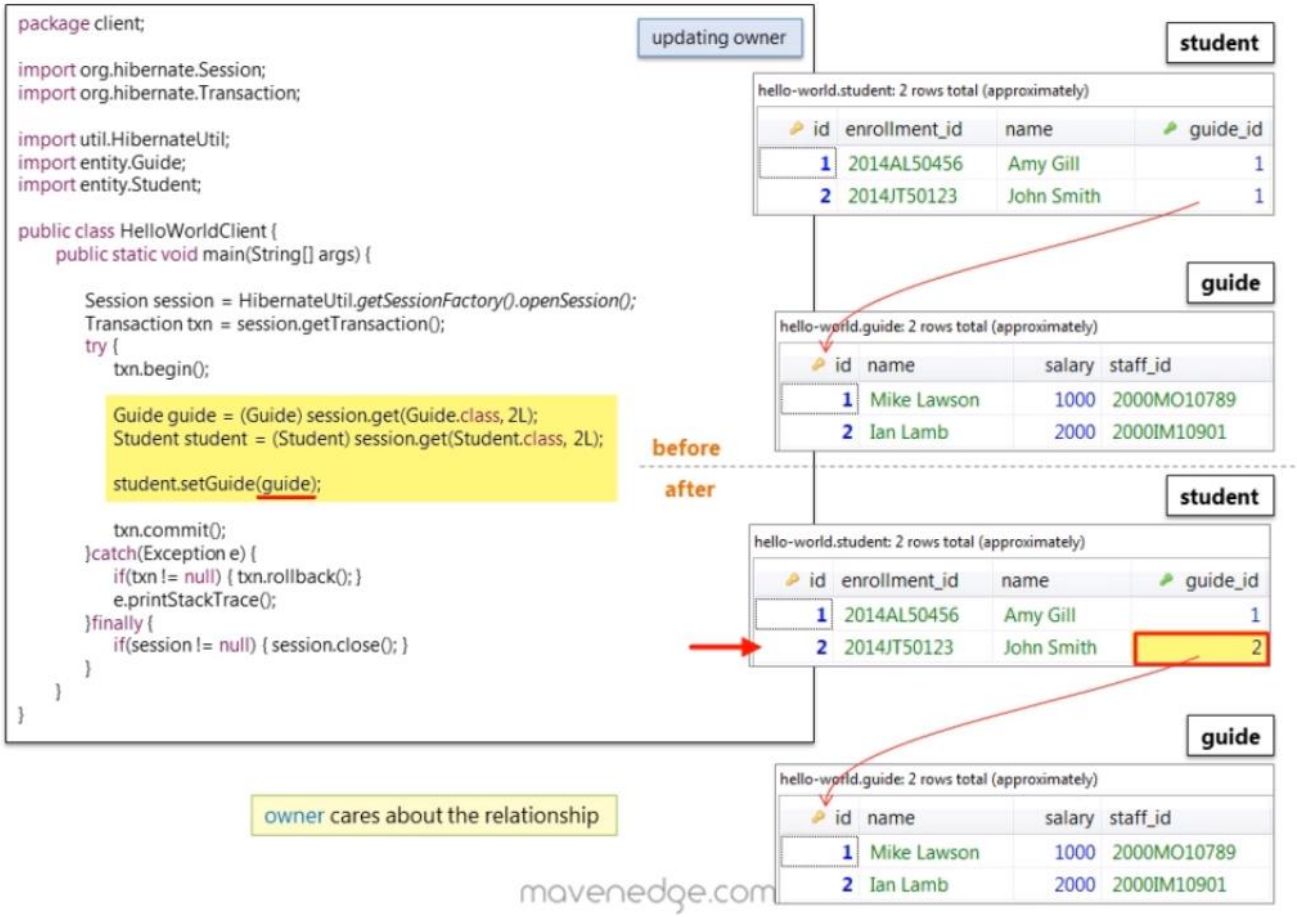
hello-world.guide: 2 rows total (approximately)

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

```java
public void saveGuide(){
    Transaction transaction = null;
    try(Session session = HibernateUtil.getSessionFactory().openSession()){
        try {
            transaction = session.getTransaction();
            transaction.begin();

            Guide guide1 = Guide.builder().name("Eng Teacher").salary(1200).staffId("501").students(new HashSet<>()).build();
            Guide guide2 = Guide.builder().name("Math Teacher").salary(1600).staffId("503").students(new HashSet<>()).build();

            Student student1 = Student.builder().guide(guide1).enrollmentId("4510").name("Nam").build();
            Student student2 = Student.builder().guide(guide2).enrollmentId("4511").name("Nam").build();

            guide1.getStudents().add(student1);
            guide2.getStudents().add(student2);

            session.persist(guide1);
            session.persist(guide2);

            transaction.commit();
        } catch (HibernateException exception){
            if (transaction != null) {
                transaction.rollback();
            }
            exception.printStackTrace();
```




```java
public void saveGuideFromStudent(){
    Transaction transaction = null;
    try(Session session = HibernateUtil.getSessionFactory().openSession()){
        try {
            transaction = session.getTransaction();
            transaction.begin();
            Guide guide2 = session.get(Guide.class, serializable: 2L);
            Student student1 = session.get(Student.class, serializable: 1L);
            student1.setGuide(guide2);
            transaction.commit();
        } catch (HibernateException exception){
            if (transaction != null) {
                transaction.rollback();
            }
            exception.printStackTrace();
```


package client;
import org.hibernate.Session;
import org.hibernate.Transaction;
import util.HibernateUtil;
import entity.Guide;
import entity.Student;
public class HelloWorldClient {
public static void main(String[] args) {
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction txn = session.getTransaction();
try {
txn.begin();
Guide guide = (Guide) session.get(Guide.class, 2L);
Student student = (Student) session.get(Student.class, 2L);
student.setGuide(guide);
txn.commit();
}catch(Exception e) {
if(txn != null) { txn.rollback(); }
e.printStackTrace();
}finally {
if(session != null) { session.close(); }
}
}
}
updating owner
owner cares about the relationship
before
after
student
hello-world.student: 2 rows total (approximately)
id enrollment_id name guide_id
1 2014AL50456 Amy Gill 1
2 2014JT50123 John Smith 1
guide
hello-world.guide: 2 rows total (approximately)
id name salary staff_id
1 Mike Lawson 1000 2000MO10789
2 Ian Lamb 2000 2000IM10901
student
hello-world.student: 2 rows total (approximately)
id enrollment_id name guide_id
1 2014AL50456 Amy Gill 1
2 2014JT50123 John Smith 2
guide
hello-world.guide: 2 rows total (approximately)
id name salary staff_id
1 Mike Lawson 1000 2000MO10789
2 Ian Lamb 2000 2000IM10901
mavenedge.com

owner

```java
package entity;

import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="enrollment_id", nullable=false)
  private String enrollmentId;

  private String name;

  @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  @JoinColumn(name="guide_id")
  private Guide guide;

  public Student() {}
  public Student(String enrollmentId, String name, Guide guide) {
    this.enrollmentId = enrollmentId;
    this.name = name;
    this.guide = guide;
  }

  public Guide getGuide() { return guide; }
  public void setGuide(Guide guide) { this.guide = guide; }
}
```

inverse end

```java
package entity;

import javax.persistence.*;

@Entity
public class Guide {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="staff_id", nullable=false)
  private String staffId;

  private String name;
  private Integer salary;

  @OneToMany(mappedBy="guide",cascade={CascadeType.PERSIST})
  private Set<Student> students = new Set<Student>();

  public Guide() {}
  public Guide(String staffId, String name, Integer salary) {
    this.staffId = staffId;
    this.name = name;
    this.salary = salary;
  }

  public Set<Student> getStudents() { return students; }

  public void addStudent(Student student) {
    students.add(student);
    student.setGuide(this);
  }
}
```

Owner is the entity that is persisted to the table that has the foreign key column

```java
@Entity
public class Student {
  ...

  @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  @JoinColumn(name="guide_id")
  private Guide guide;

  public Guide getGuide() { return guide; }
  ...
}
```

```java
@Entity
public class Guide {
  ...

  @OneToMany(mappedBy="guide",cascade={CascadeType.PERSIST})
  private Set<Student> students = new Set<Student>();

  public Set<Student> getStudents() { return students; }
  ...
}
```

student

id	enrollment_id	name	guide_id

guide

id	staff_id	name	salary

The owner side responsible for the relationship.

Before:

STUDENT 1

SELECT * from STUDENT s Enter a SQL expression to filter results (u

Grid

	ID	enrollment_id	name	guide_id
1	1	4510	Nam	1
2	2	4511	Nam	2

GUIDE 1

SELECT * from GUIDE g Enter a SQL expression to filte

Grid

	ID	name	salary	staff_id
1	1	Eng Teacher	1,200	501
2	2	Math Teacher	1,600	503

After:

STUDENT 1

SELECT * from STUDENT s Enter a SQL expression to filter result

Grid

	ID	enrollment_id	name	guide_id
1	1	4510	Nam	2
2	2	4511	Nam	2

GUIDE 1

SELECT * from GUIDE g Enter a SQL expression to filt

Grid

	ID	name	salary	staff_id
1	1	Eng Teacher	1,200	501
2	2	Math Teacher	1,600	503

Add new function:

```
@OneToMany(mappedBy = "guide", cascade = CascadeType.PERSIST)
@ToString.Exclude
@EqualsAndHashCode.Exclude
private Set<Student> students = new HashSet<>() ;

public void addStudent(Student student){
    this.students.add(student);
    student.setGuide(this);
}
```

- ***Entity states***

State	
Transient	A transient object is an object that is not associated with a session and that does not have a database record. When you create a new object of a POJO class, that entity instance is in the transient state. Transient objects exist only in memory, Hibernate does not manage transient objects.
Persistent	A persistent object is an object that has a database record and that is associated with an active session. So the moment you invoke **session.save** on a transient object, it moves to the **persistent state**. Once an object is in the persistent state, hibernate tracks changes to it and automatically saves these changes to the database when the session is flushed.
Detach	An object that has a database record but is not associated with a session is a detached object. When a session is closed, all entities associated with the session become detached. Although *Detached* objects have a database record, changes to the object will not be reflected in the database, and vice versa.

- ***Automatic Dirty Checking in Hibernate.***

So basically, an object can move into the persistent state when any one of the following happens:

- When the code invokes **session.save, session.persist** or session.saveorUpdate.
- When the code invokes **session.load** or **session.get.**

- ***Save vs persist.***

Sr. No.	Key	save()	persist()
1	Basic	It stores object in database	It also stores object in database
2	Return Type	It return generated id and return type is serializable	It does not return anything. Its void return type.
3	Transaction Boundaries	It can save object within boundaries and outside boundaries	It can only save object within the transaction boundaries
4.	Detached Object	It will create a new row in the table for detached object	It will throw persistence exception for detached object
5.	Supported by	It is only supported by Hibernate	It is also supported by JPA

► *Orphan remova*

```java
@Entity
public class Student {
    //...

    @ManyToOne(cascade={ CascadeType.PERSIST, CascadeType.REMOVE })
    @JoinColumn(name="guide_id")
    private Guide guide;

    public Guide getGuide() { return guide; }
    //...
}
```

```java
@Entity
public class Guide {
    //...

    @OneToMany( mappedBy="guide" )
    private Set<Student> students = new Set<Student>();

    public Set<Student> getStudents() { return students; }
    //...
}
```

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
2	2014AL50456	Amy Gill	2
3	2014JO10333	Jane Roe	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
~~2~~	~~2014AL50456~~	~~Amy Gill~~	~~2~~
3	2014JO10333	Jane Roe	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
~~2~~	~~Ian Lamb~~	~~2000~~	~~2000IM10901~~

```
18  Student student = (Student) session.get(Student.class, 2L);
19  session.delete(student);
```

org.hibernate.exception.ConstraintViolationException: could not execute statement

Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Cannot delete or update a parent row: a foreign key constraint fails

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
~~2~~	~~2014AL50456~~	~~Amy Gill~~	~~2~~
3	2014JO10333	Jane Roe	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
~~2~~	~~Ian Lamb~~	~~2000~~	~~2000IM10901~~

orphaned record

```java
@Entity
public class Student {
    //...
    @ManyToOne(cascade={ CascadeType.PERSIST, CascadeType.REMOVE })
    @JoinColumn(name="guide_id")
    private Guide guide;

    public Guide getGuide() { return guide; }
    //...
}
```

```java
@Entity
public class Guide {
    //...
    @OneToMany( mappedBy="guide", orphanRemoval=true )
    private Set<Student> students = new Set<Student>();

    public Set<Student> getStudents() { return students; }
    //...
}
```

student

id	enrollment_id	name	guide_id
1	2014JT50123	John Smith	1
2	2014AL50456	Amy Gill	2
3	2014JO10333	Jane Roe	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

Before:

SELECT * from STUDENT s — Enter a SQL expression to filter results (use C

	ID	enrollment_id	name	guide_id
1	1	4510	A	1
2	2	4511	B	2
3	3	4512	C	2

GUIDE 1

SELECT * from GUIDE g — Enter a SQL expression to filt

	ID	name	salary	staff_id
1	1	Eng Teacher	1,200	501
2	2	Math Teacher	1,600	503

After:

SELECT * from STUDENT s — Enter a SQL expression to filter re

	ID	enrollment_id	name	guide_id
1	1	4510	A	1

SELECT * from GUIDE g Enter a SQL expression to f

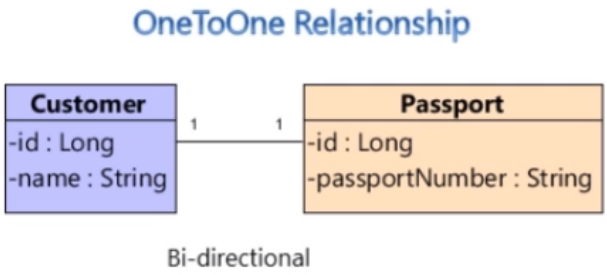
Grid

	ID	name	salary	staff_id
1	1	Eng Teacher	1,200	501

```
Hibernate: delete from STUDENT where ID=?
Hibernate: delete from STUDENT where ID=?
Hibernate: delete from GUIDE where ID=?
```

```
@OneToMany(mappedBy = "guide", cascade = CascadeType.PERSIST, orphanRemoval = true)
@ToString.Exclude
@EqualsAndHashCode.Exclude
private Set<Student> students = new HashSet<>() ;
```

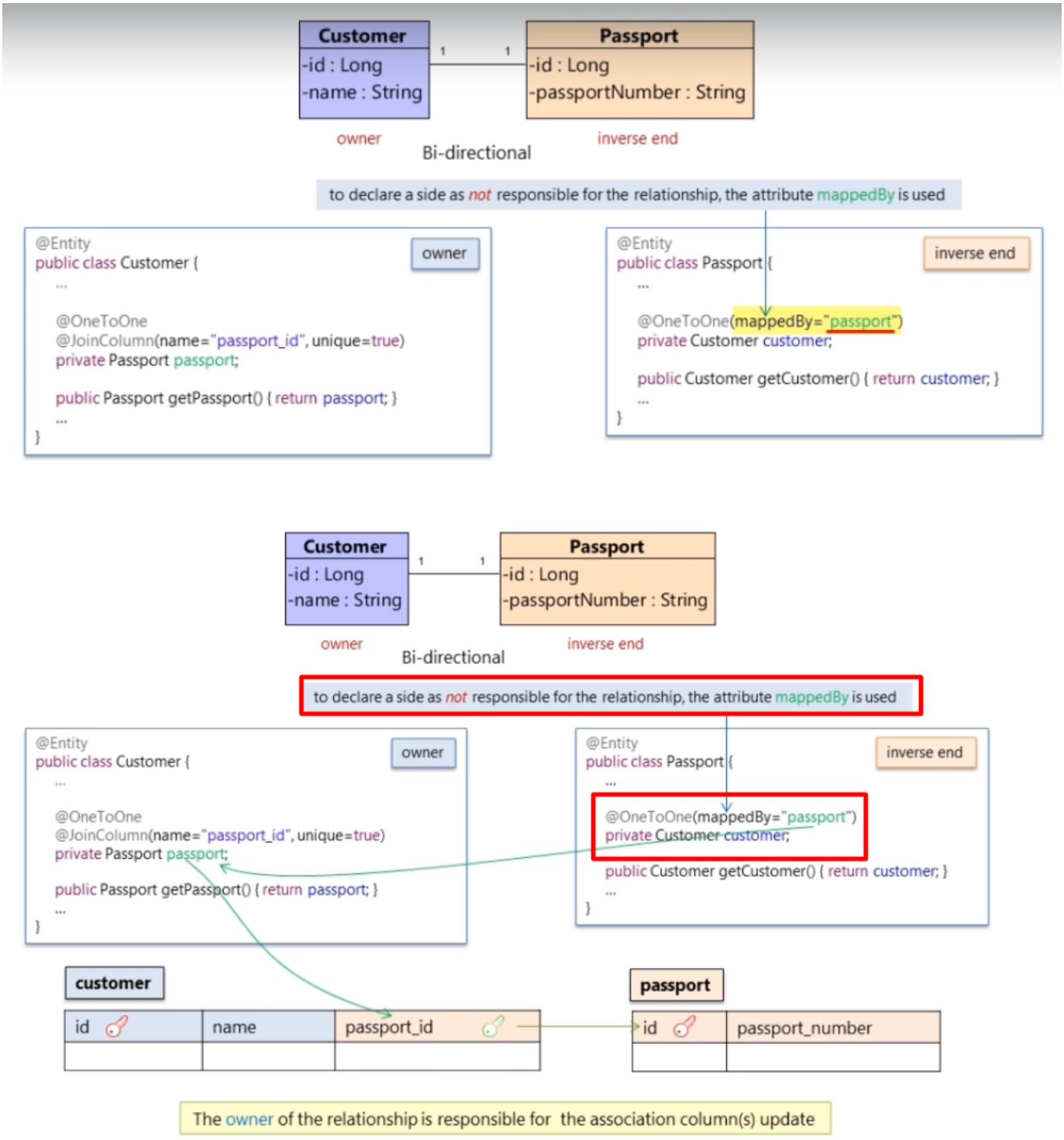
- ***One to one relationship.***




```
@Entity
public class Customer {
    ...

    @OneToOne
    @JoinColumn(name="passport_id", unique=true)
    private Passport passport;

    public Passport getPassport() { return passport; }
    ...
}
```

```
@Entity
public class Passport {
    ...

    @OneToOne(mappedBy="passport")
    private Customer customer;

    public Customer getCustomer() { return customer; }
    ...
}
```

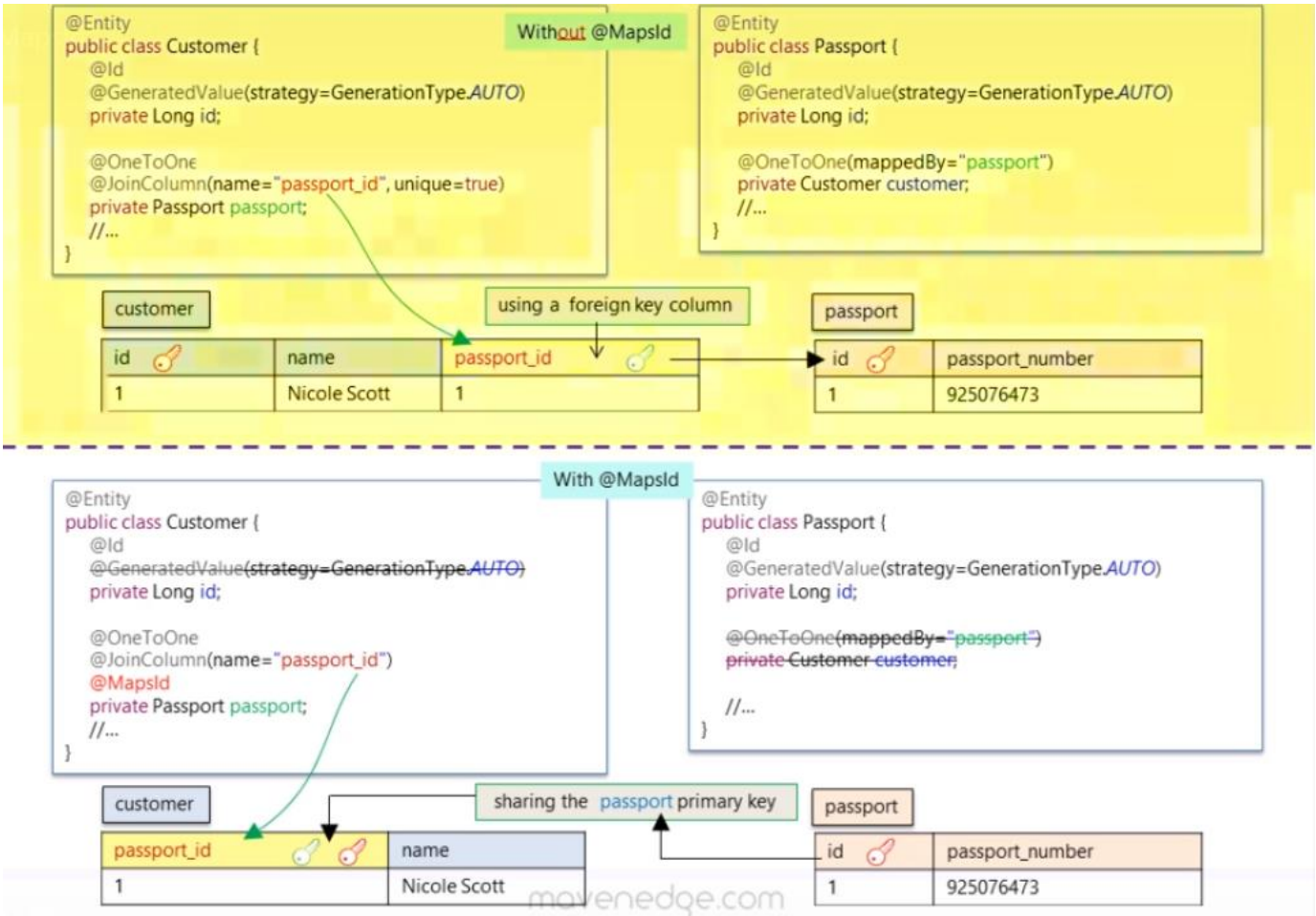
The owner of the relationship is responsible for the association column(s) update


Customer
-id : Long
-name : String
1
1
Passport
-id : Long
-passportNumber : String
owner
inverse end
Bi-directional
to declare a side as not responsible for the relationship, the attribute mappedBy is used
@Entity
public class Customer {
...
@OneToOne
@JoinColumn(name="passport_id", unique=true)
private Passport passport;
public Passport getPassport() { return passport; }
...
}
owner
@Entity
public class Passport {
...
@OneToOne(mappedBy="passport")
private Customer customer;
public Customer getCustomer() { return customer; }
...
}
inverse end
Customer
-id : Long
-name : String
1
1
Passport
-id : Long
-passportNumber : String
owner
inverse end
Bi-directional
to declare a side as not responsible for the relationship, the attribute mappedBy is used
@Entity
public class Customer {
...
@OneToOne
@JoinColumn(name="passport_id", unique=true)
private Passport passport;
public Passport getPassport() { return passport; }
...
}
owner
@Entity
public class Passport {
...
@OneToOne(mappedBy="passport")
private Customer customer;
public Customer getCustomer() { return customer; }
...
}
inverse end
customer
id
name
passport_id
passport
id
passport_number
The owner of the relationship is responsible for the association column(s) update

```java
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private String name;

    @OneToOne(cascade={CascadeType.PERSIST})
    @JoinColumn(name="passport_id", unique=true)
    private Passport passport;

    public Customer() {}
    public Customer(String name, Passport passport) {
        this.name = name;
        this.passport = passport;
    }

    public Passport getPassport() {
        return passport;
    }
}
```

```java
@Entity
public class Passport {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="passport_number")
    private String passportNumber;

    @OneToOne(mappedBy="passport")
    private Customer customer;

    public Passport() {}
    public Passport(String passportNumber) {
        this.passportNumber = passportNumber;
    }

    public Customer getCustomer() {
        return customer;
    }
}
```

One to one with MapsID:

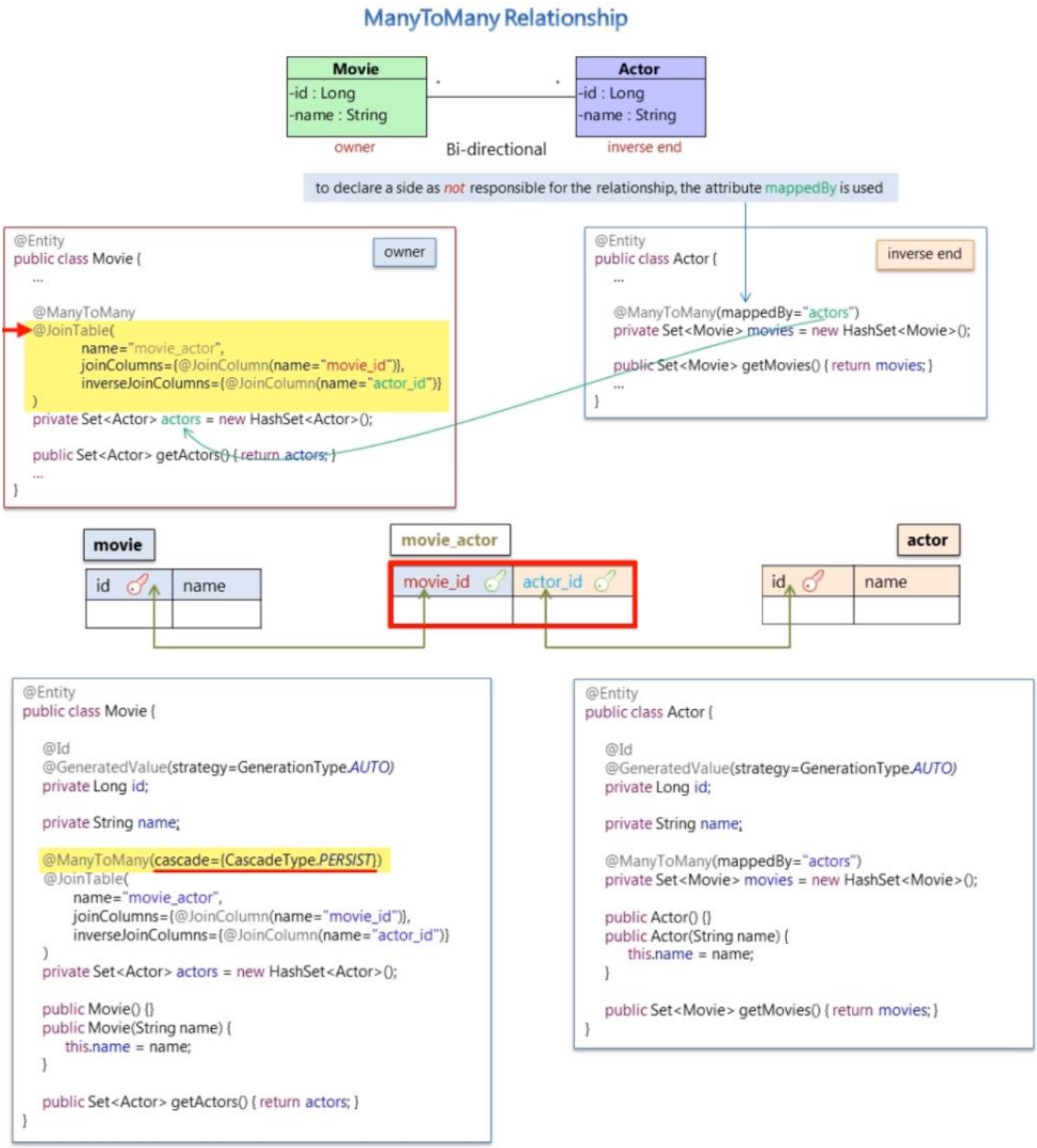


```java
public class Customer {
    @Id
    private Long ID;

    private String name;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "passport_id")
    @MapsId
    private Passport passport;

}
```

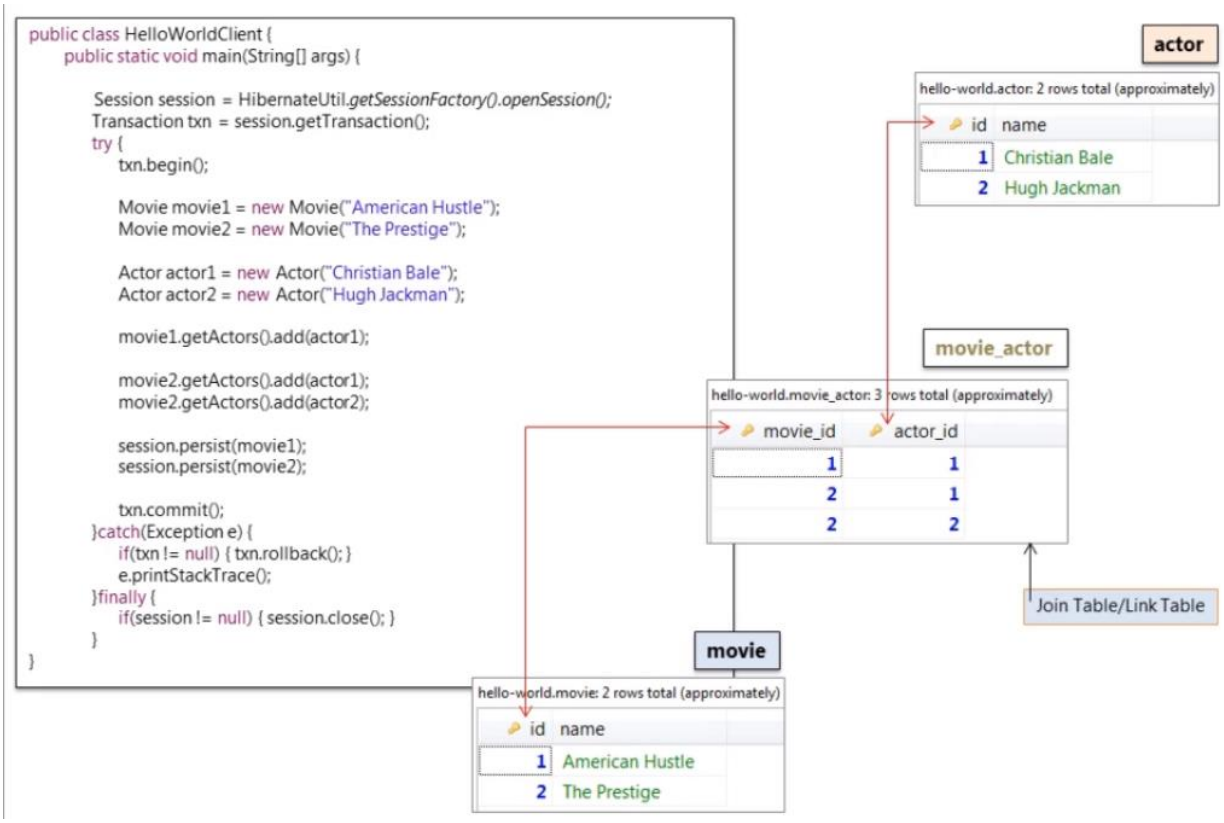
```java
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long ID;

    private String passportNumber;
}
```

```java
public void saveCustomer(){

    Transaction transaction = null;
    try (Session session = HibernateUtil.getSessionFactory().openSession()){
        try {
            transaction = session.getTransaction();
            transaction.begin();
            Passport passport = Passport.builder()
                                        .passportNumber("3312989211")
                                        .build();
            Customer customer = Customer.builder().name("Mr A")
                                        .passport(passport)
                                        .build();
            session.save(customer);
            transaction.commit();
        } catch (HibernateException hibernateException){
            if (transaction != null){
                transaction.rollback();
            }
            hibernateException.printStackTrace();
```

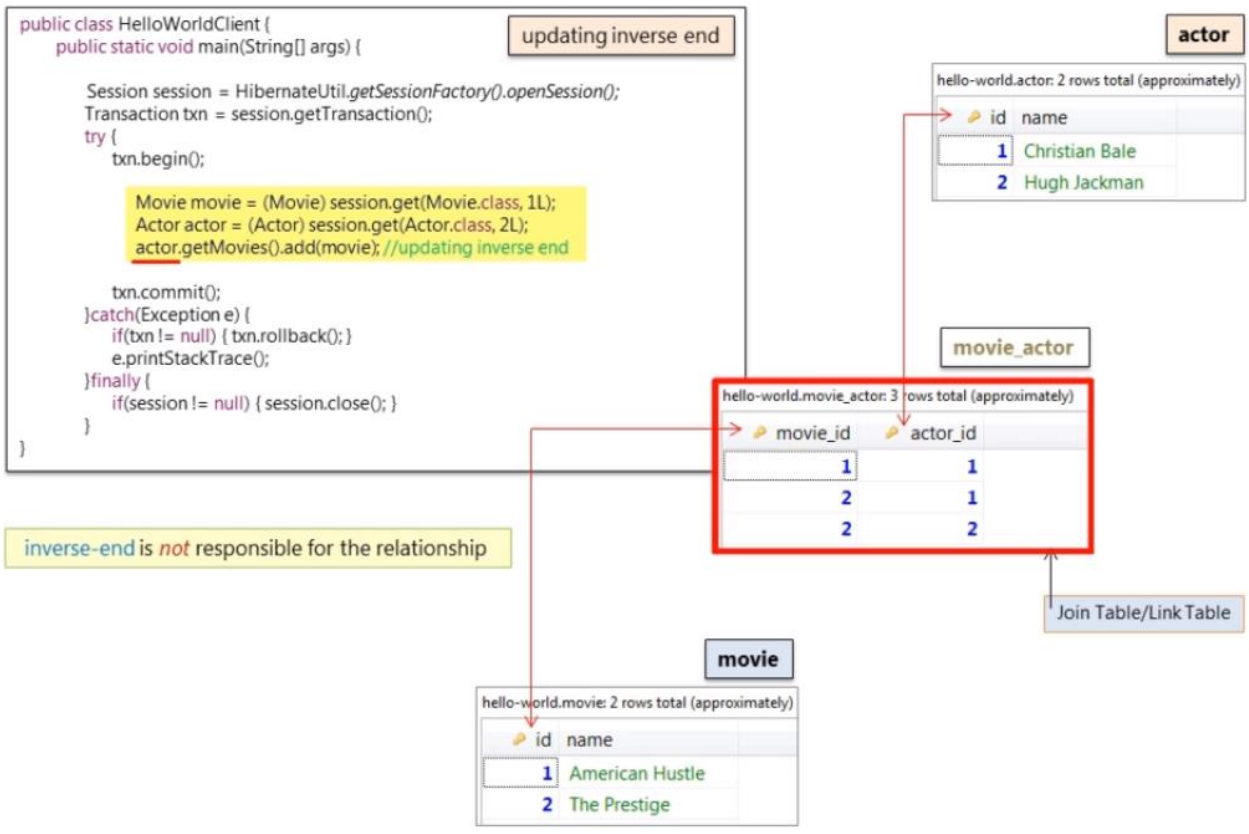
► *Many to many relationship*



Update owner side:



Update inverse side (does not affect owner side):



Update inverse end vs update owner end:

```
@Entity
public class Movie {
    //...
    @ManyToMany
    @JoinTable(
            name="movie_actor",
            joinColumns={@JoinColumn(name="movie_id")},
            inverseJoinColumns={@JoinColumn(name="actor_id")}
    )
    private Set<Actor> actors = new HashSet<Actor>();
    public Set<Actor> getActors() { return actors; }
    //...
}
```

owner

```
@Entity
public class Actor {
    //...
    @ManyToMany(mappedBy="actors")
    private Set<Movie> movies = new HashSet<Movie>();
    public Set<Movie> getMovies() { return movies; }
    //...
}
```

inverse end

```
18  Movie movie = (Movie) session.get(Movie.class, 1L);
19  Actor actor = (Actor) session.get(Actor.class, 2L);
20  actor.getMovies().add(movie); //updating inverse end – did not work
```

inverse-end is *not* responsible for the relationship

movie

id	name
1	Christian Bale
2	Hugh Jackman

movie_actor

movie_id	actor_id
1	1
2	1
2	2
1	2 → DID NOT HAPPEN

actor

id	name
1	American Hustle
2	The Prestige

1. Update inverse end

```
@Entity
public class Movie {
    //...
    @ManyToMany
    @JoinTable(
            name="movie_actor",
            joinColumns={@JoinColumn(name="movie_id")},
            inverseJoinColumns={@JoinColumn(name="actor_id")}
    )
    private Set<Actor> actors = new HashSet<Actor>();
    public Set<Actor> getActors() { return actors; }
    //...
}
```

owner

```
@Entity
public class Actor {
    //...
    @ManyToMany(mappedBy="actors")
    private Set<Movie> movies = new HashSet<Movie>();
    public Set<Movie> getMovies() { return movies; }
    //...
}
```

inverse end

```
18  Movie movie = (Movie) session.get(Movie.class, 1L);
19  Actor actor = (Actor) session.get(Actor.class, 2L);
20  movie.getActors().add(actor); //updating owner – did work
```

inverse-end is *not* responsible for the relationship

movie

id	name
1	Christian Bale
2	Hugh Jackman

movie_actor → relationship

movie_id	actor_id
1	1
2	1
2	2
1	2 → DID HAPPEN

actor

id	name
1	American Hustle
2	The Prestige

2. Update owner end

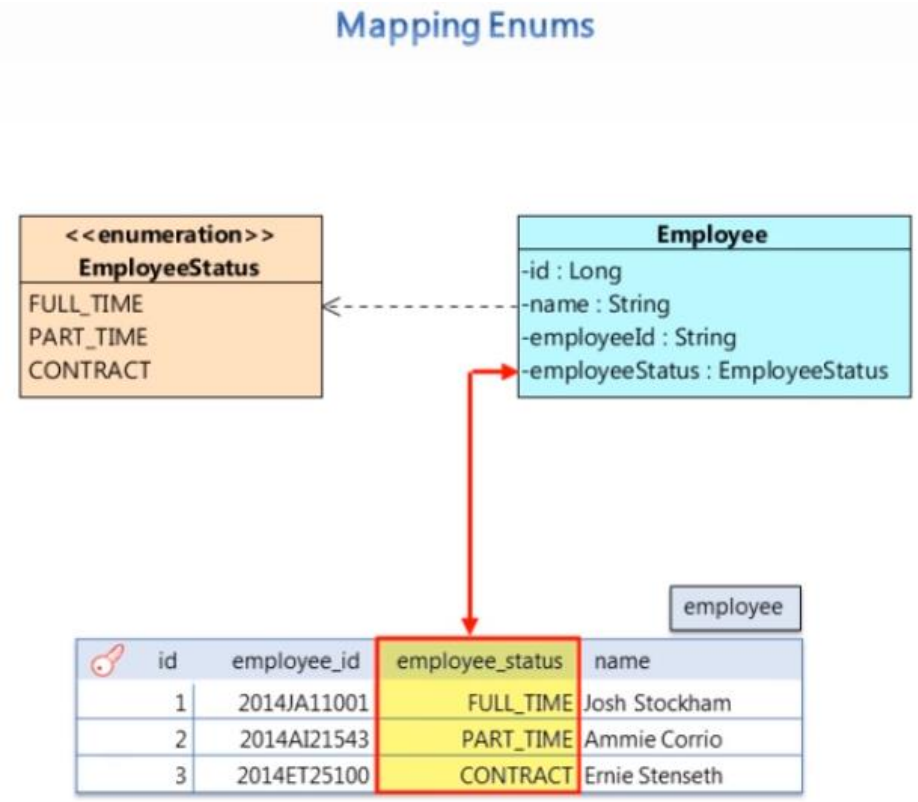
Code for update in inverse end:

```
//...
public void addMovie(Movie movie) {
    this.movies.add(movie);
    movie.getActors().add(this);
}
}
```

```java
public class Movie {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long ID;

    private String name;

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(
            name = "MOVIE_ACTOR",
            joinColumns = @JoinColumn(name = "MOVIE_ID"),
            inverseJoinColumns = @JoinColumn(name = "ACTOR_ID")
    )
    private Set<Actor> actors = new HashSet<>();
```

```java
public class Actor {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long ID;

    private String name;

    @ManyToMany(mappedBy = "actors")
    private Set<Movie> movies = new HashSet<>();
}
```

- ***Mapping Enum***



```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private String name;

    @Column(name="employee_id", unique=true)
    private String employeeId;


    @Column(name="employee_status")
    private EmployeeStatus employeeStatus;

    public Employee() {}
    public Employee(String name, String employeeId, EmployeeStatus employeeStatus) {
        this.name = name;
        this.employeeId = employeeId;
        this.employeeStatus = employeeStatus;
    }
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", employeeId=" + employeeId + ",
        employeeStatus=" + employeeStatus + "]";
    }
}
```

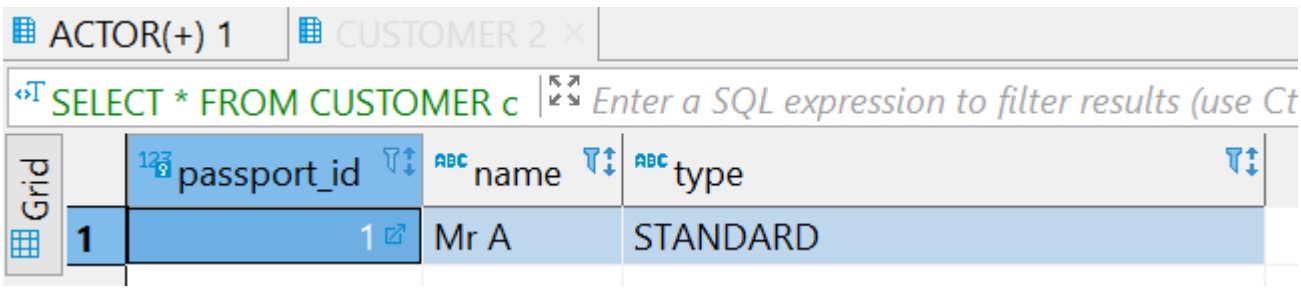
```java
public enum EmployeeStatus {
    FULL_TIME,
    PART_TIME,
    CONTRACT
}
```

```java
nport org.hibernate.Session;
nport entity.Employee;
nport entity.EmployeeStatus;
nport util.HibernateUtil;

ublic class HelloWorldClient {

ublic static void main(String[] args) {

  Session session = HibernateUtil.getSessionFactory().openSession();
  session.beginTransaction();

  //persisting
  Employee employee1 = new Employee("Josh Stockham", "2014JA11001", EmployeeStatus.FULL_TIME);
  Employee employee2 = new Employee("Ammie Corrio", "2014AI21543", EmployeeStatus.PART_TIME);
  Employee employee3 = new Employee("Ernie Stenseth", "2014ET25100", EmployeeStatus.CONTRACT);

  session.persist(employee1);
  session.persist(employee2);
  session.persist(employee3);

  session.getTransaction().commit();
  session.close();

  }
```

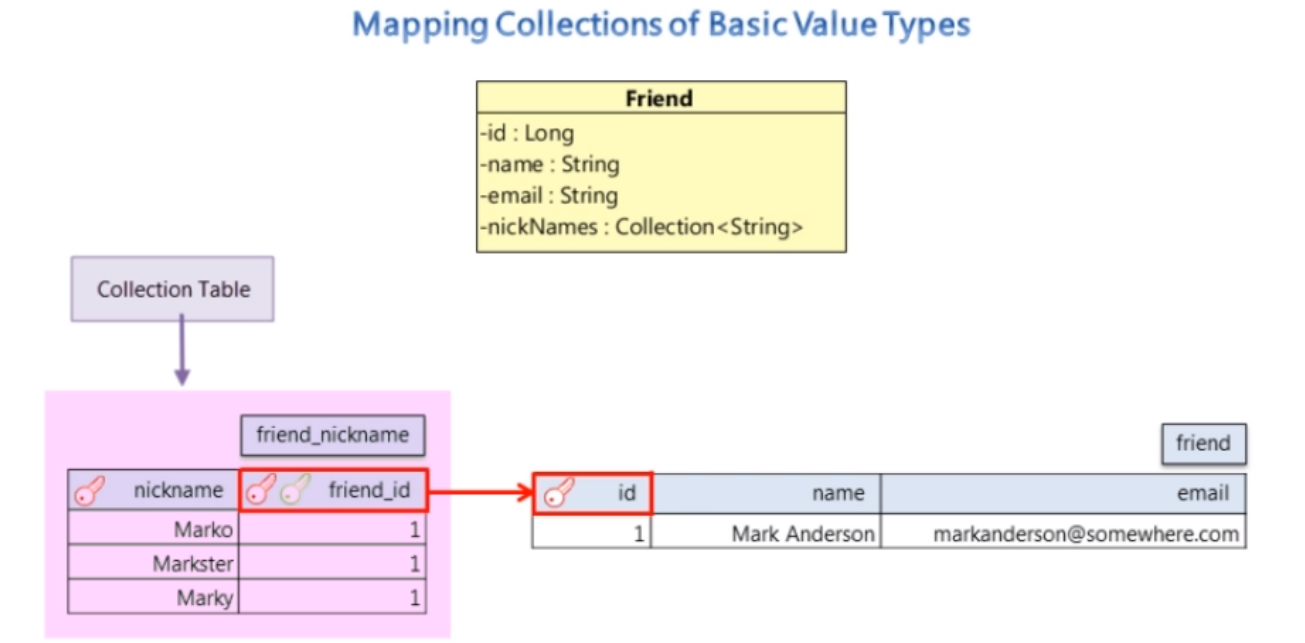
Code:

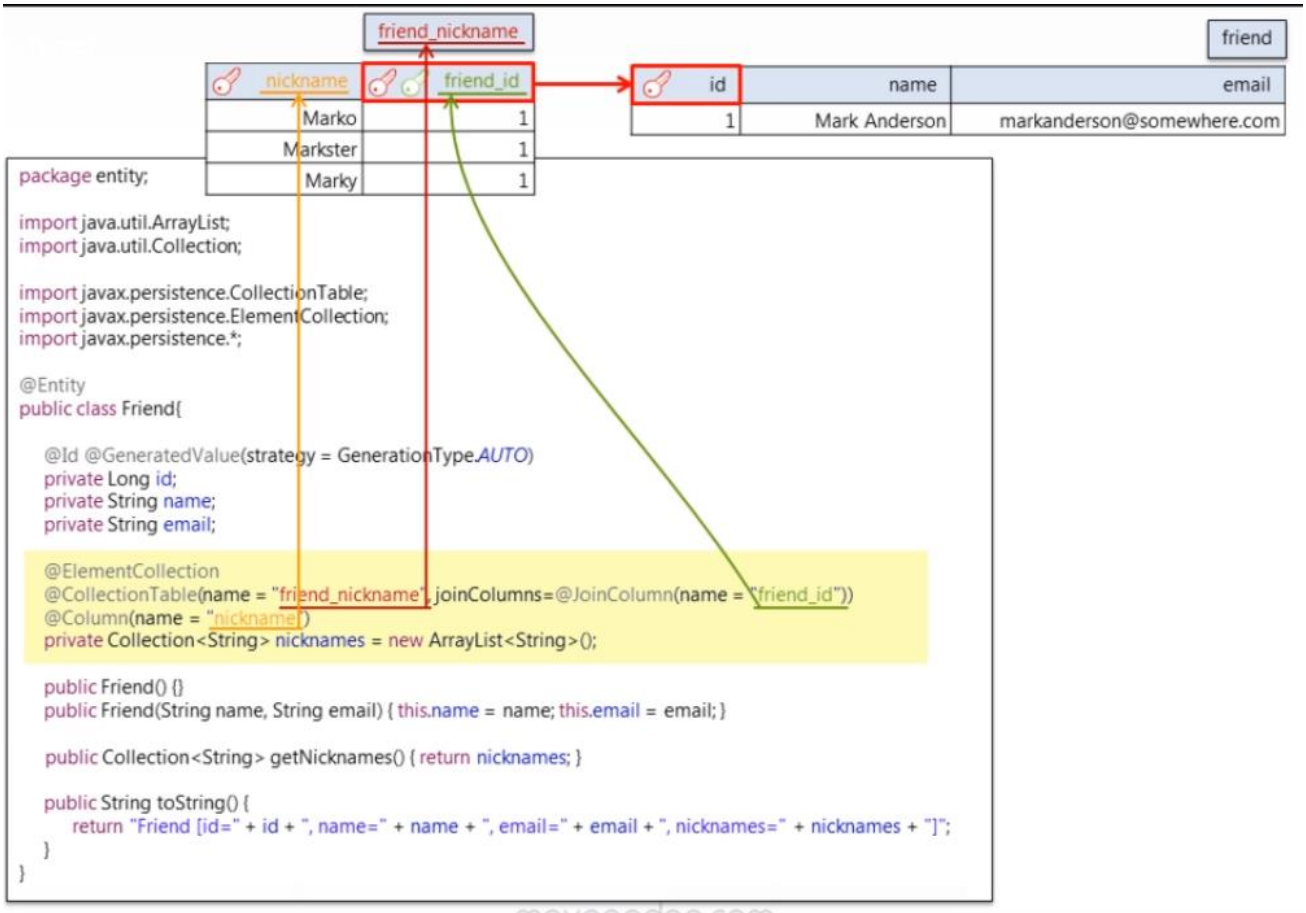
```java
public class Customer {
    @Id
    private Long ID;

    private String name;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "passport_id")
    @MapsId
    private Passport passport;

    @Enumerated(EnumType.STRING)
    @Column(name = "type")
    private CustomerType type;

    public enum CustomerType {
        STANDARD,
        VIP
    }
```

```java
Transaction transaction = null;
try (Session session = HibernateUtil.getSessionFactory().openSession()){
    try {
        transaction = session.getTransaction();
        transaction.begin();
        Passport passport = Passport.builder()
                                    .passportNumber("3312989211")
                                    .build();
        Customer customer = Customer.builder().name("Mr A")
                                    .passport(passport)
                                    .type(Customer.CustomerType.STANDARD)
                                    .build();
        session.save(customer);
        transaction.commit();
    } catch (HibernateException hibernateException){
        if (transaction != null){
            transaction.rollback();
        }
        hibernateException.printStackTrace();
```



- ***Mapping collection of basic value type.***






```
package entity;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.CollectionTable;
import javax.persistence.ElementCollection;
import javax.persistence.*;

@Entity
public class Friend{

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;

    @ElementCollection
    @CollectionTable(name = "friend_nickname", joinColumns=@JoinColumn(name = "friend_id"))
    @Column(name = "nickname")
    private Collection<String> nicknames = new ArrayList<String>();

    public Friend() {}
    public Friend(String name, String email) { this.name = name; this.email = email; }

    public Collection<String> getNicknames() { return nicknames; }

    public String toString() {
        return "Friend [id=" + id + ", name=" + name + ", email=" + email + ", nicknames=" + nicknames + "]";
    }
}
```

```java
package client;

import org.hibernate.Session;

import util.HibernateUtil;
import entity.Friend;

public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //persisting
        Friend friend = new Friend("Mark Anderson", "markanderson@pluswhere.com");

        friend.getNicknames().add("Marky");
        friend.getNicknames().add("Marco");
        friend.getNicknames().add("Markster");

        session.persist(friend);

        session.getTransaction().commit();
        session.close();

    }
}
```

hello-world.friend: 1 rows total (approximately)

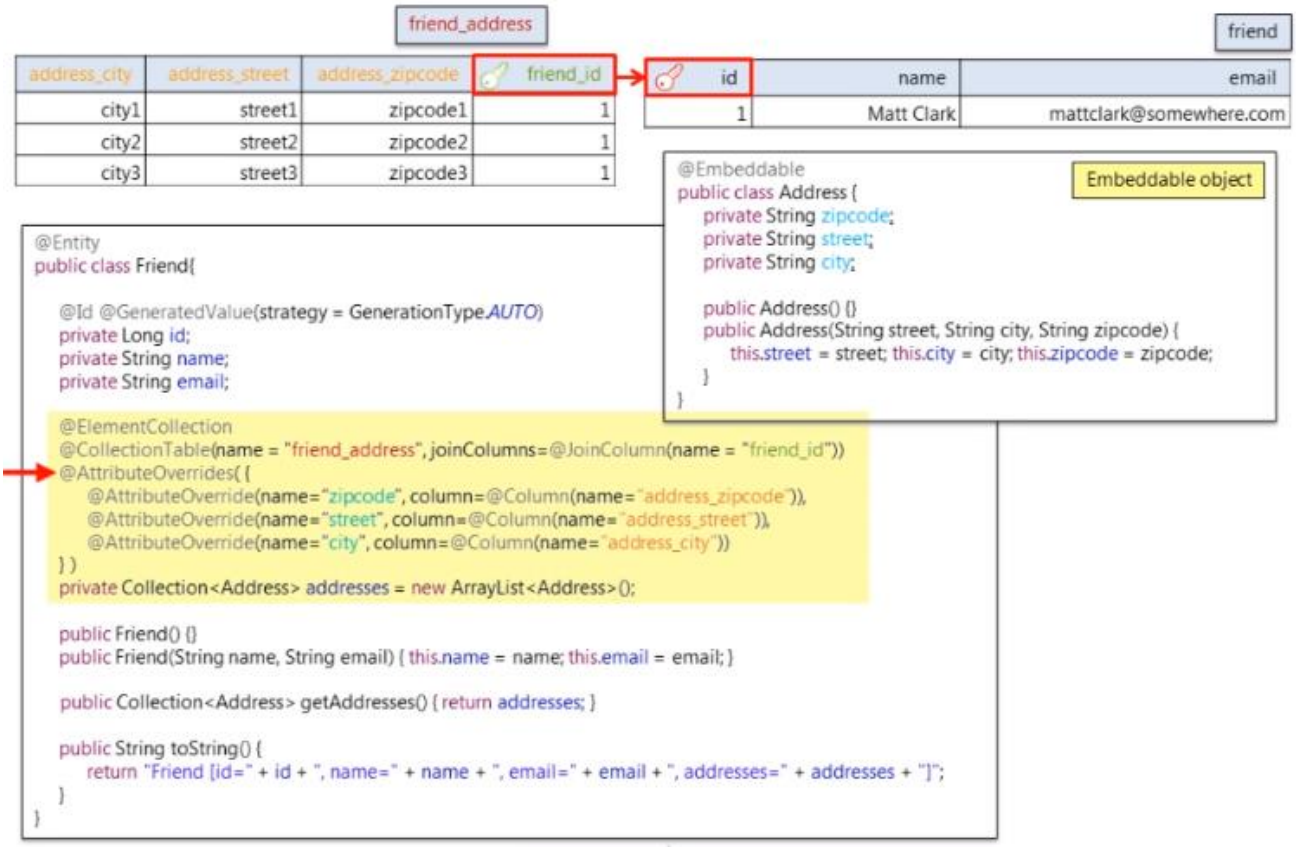
id	email	name
1	markanderson@pluswhere.com	Mark Anderson

hello-world.friend_nickname: 3 rows total (approximately)

friend_id	nickname
1	Marky
1	Marco
1	Markster

```java
package client;

import org.hibernate.Session;

import util.HibernateUtil;
import entity.Friend;

public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //persisting
        /*
        Friend friend = new Friend("Mark Anderson", "markanderson@pluswhere.com");

        friend.getNicknames().add("Marky");
        friend.getNicknames().add("Marco");
        friend.getNicknames().add("Markster");

        session.save(friend);
        */

        //retrieving
        Friend friend = (Friend) session.get(Friend.class, 1L);
        System.out.println(friend);

        session.getTransaction().commit();
        session.close();

    }
}
```

```
Friend [id=1, name=Mark Anderson, email=markanderson@pluswhere.com, nicknames=[Marco, Markster, Marky]]
```



Collection primitive type:

```
@ElementCollection
@CollectionTable(name = "PAYMENT_SCHEME",
                 joinColumns = @JoinColumn(name = "customer_id"))
@Column(name = "ips")
private Set<String> payment;
```

```
Customer customer = Customer.builder().name("Mr A")
                            .passport(passport)
                            .payment(new HashSet<>())
                            .address(new HashSet<>())
                            .type(Customer.CustomerType.STANDARD)
                            .build();
customer.getPayment().add("VISA");
customer.getPayment().add("MC");
customer.getPayment().add("NAPAS");
```

Hibernate: [GetCustomer] : Customer(ID=1, name=Mr A, passport=Passport(ID=1, passportNumber=3312989211), payment=[MC, NAPAS, VISA], type=STANDARD, address=[Address(street=Wall st, city=NY, zipcode=70000), Address(street=To Hien Thanh st, city=HCM, zipcode=51000)])

Collection value type:

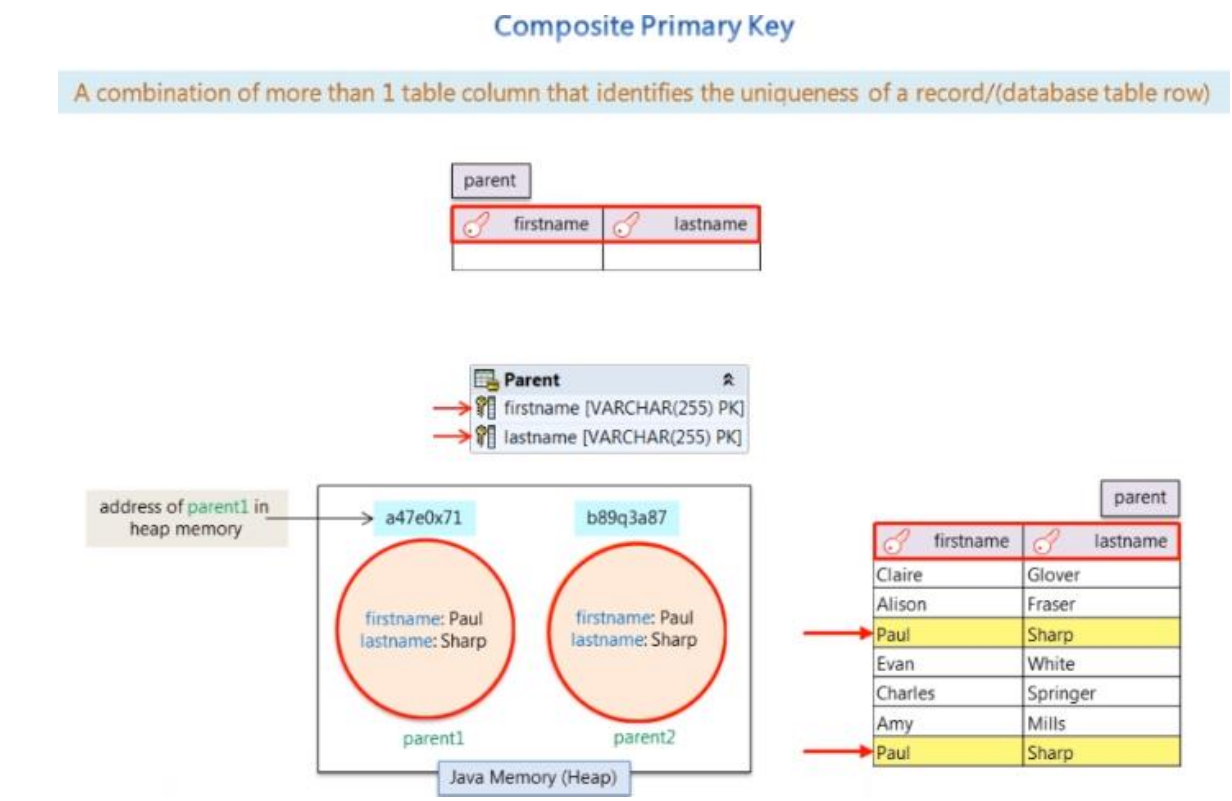
```
@ElementCollection
@CollectionTable(name = "CUSTOMER_ADDR", joinColumns = @JoinColumn(name = "customer_id"))
@AttributeOverrides({
        @AttributeOverride(name = "zipcode", column = @Column(name = "addr_zipcode")),
        @AttributeOverride(name = "city", column = @Column(name = "addr_city")),
        @AttributeOverride(name = "street", column = @Column(name = "addr_street"))
})
private Set<Address> address;
```

```
@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Address {

    private String street;
    private String city;
    private String zipcode;

}
```

```
Address address1 = Address.builder()
                        .zipcode("70000")
                        .street("Wall st")
                        .city("NY")
                        .build();

Address address2 = Address.builder()
        .zipcode("51000")
        .street("To Hien Thanh st")
        .city("HCM")
        .build();

customer.getAddress().add(address1);
customer.getAddress().add(address2);
```

Hibernate: [GetCustomer] : Customer(ID=1, name=Mr A, passport=Passport(ID=1, passportNumber=3312989211), payment=[MC, NAPAS, VISA], type=STANDARD, address=[Address(street=Wall st, city=NY, zipcode=70000), Address(street=To Hien Thanh st, city=HCM, zipcode=51000)])

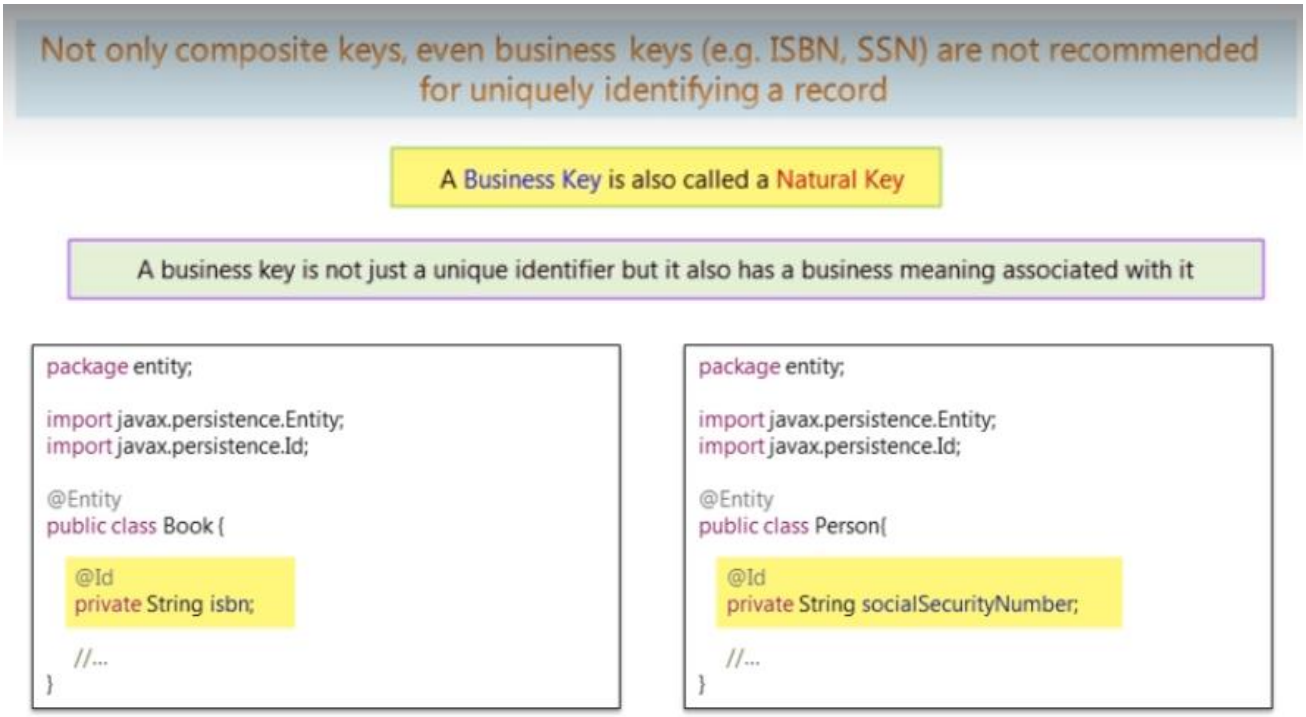
► *Composites Primary key*




```java
package entity;

import javax.persistence.Entity;
import javax.persistence.EmbeddedId;

@Entity
public class Parent {

    @EmbeddedId
    private ParentPrimaryKey parentPrimaryKey;

    public Parent() { }
    public Parent(ParentPrimaryKey parentPrimaryKey) {
        this.parentPrimaryKey = parentPrimaryKey;
    }
    //...
}
```

```java
package entity;

import java.io.Serializable;

import javax.persistence.Embeddable;

@Embeddable
public class ParentPrimaryKey implements Serializable {

    private String firstname;
    private String lastname;

    public ParentPrimaryKey() {}
    public ParentPrimaryKey(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    @Override
    public int hashCode() {
        int result = firstname.hashCode();
        result = 31 * result + lastname.hashCode();
        return result;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        ParentPrimaryKey parentPrimaryKey = (ParentPrimaryKey) o;
        if (!firstname.equals(parentPrimaryKey.firstname)) return false;
        if (!lastname.equals(parentPrimaryKey.lastname)) return false;
        return true;
    }
    //...
}
```

```java
package client;

import org.hibernate.Session;

import util.HibernateUtil;
import entity.Parent;
import entity.ParentPrimaryKey;

public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //persisting
        ParentPrimaryKey parentPrimaryKey = new ParentPrimaryKey("Gavin", "King");
        Parent parent = new Parent(parentPrimaryKey);
        session.persist(parent);

        session.getTransaction().commit();
        session.close();

    }
}
```

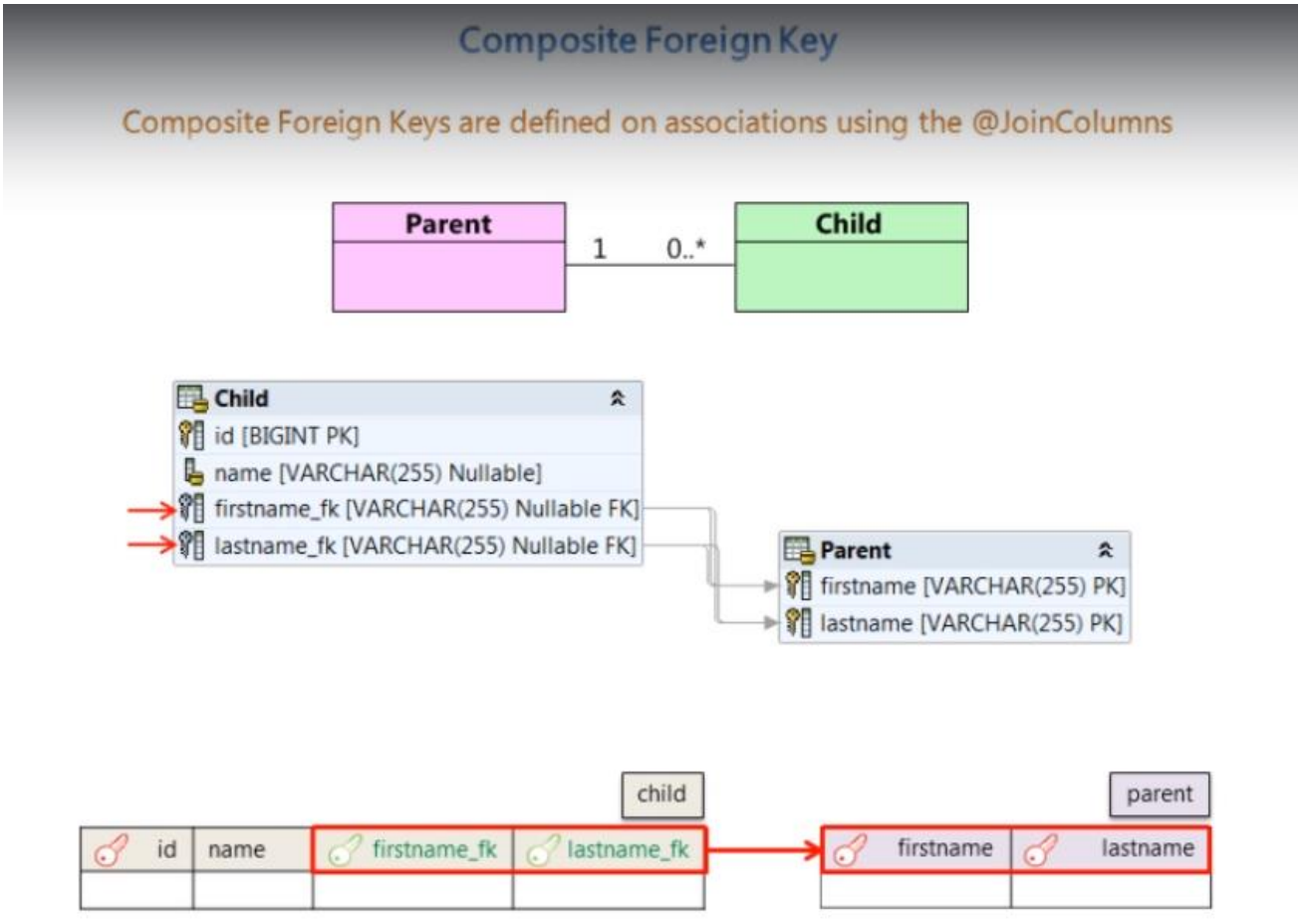


3. Not recommend

```java
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;

@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;

    private String isbn;

    //...
}
```

```java
package entity;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;

@Entity
public class Person{

    @Id @GeneratedValue
    private Long id;

    private String socialSecurityNumber;

    //...
}
```

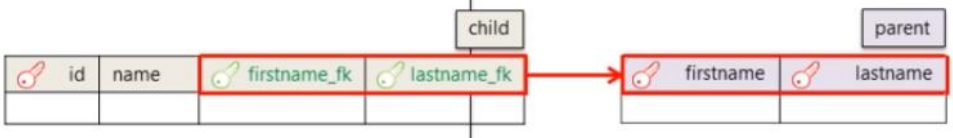
4. Recommend



```java
@Entity
public class Parent {
    @EmbeddedId
    private ParentPrimaryKey parentPrimaryKey;

    @OneToMany(mappedBy="parent", cascade={CascadeType.PERSIST})
    private Set<Child>children = new HashSet<Child>();

    public Parent() { }
    public Parent(ParentPrimaryKey parentPrimaryKey) {
        this.parentPrimaryKey = parentPrimaryKey;
    }
    public void addChild(Child child) {
        child.setParent(this);
        this.children.add(child);
    }
    //...
}
```

```java
@Embeddable
public class ParentPrimaryKey implements Serializable {
    private String firstname;
    private String lastname;

    public ParentPrimaryKey() {}
    public ParentPrimaryKey(String firstname, String lastname) {
        this.firstname = firstname; this.lastname = lastname;
    }
    @Override
    public int hashCode() {
        int result = firstname.hashCode();
        result = 31 * result + lastname.hashCode();
        return result;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        ParentPrimaryKey parentPrimaryKey = (ParentPrimaryKey) o;
        if (!firstname.equals(parentPrimaryKey.firstname)) return false;
        if (!lastname.equals(parentPrimaryKey.lastname)) return false;
        return true;
    }
    //...
}
```

```java
@Entity
public class Child {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="firstname_fk", referencedColumnName="firstname"),
        @JoinColumn(name="lastname_fk", referencedColumnName="lastname")
    })
    private Parent parent;

    public Child() {}
    public Child(String name) {
        this.name = name;
    }
    //...
}
```

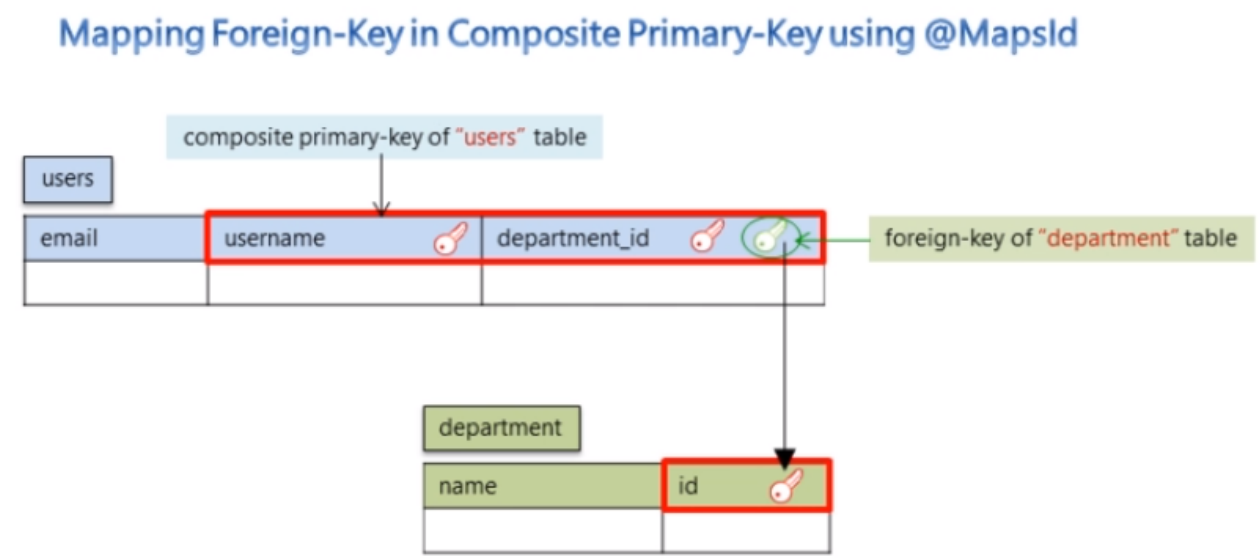



```java
package client;

import org.hibernate.Session;

import util.HibernateUtil;
import entity.Parent;
import entity.ParentPrimaryKey;

public class HelloWorldClient {
    public static void main(String[] args) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //persisting
        ParentPrimaryKey parentPrimaryKey = new ParentPrimaryKey("Charlotte", "Crawford");
        Parent parent = new Parent(parentPrimaryKey);

        Child child1 = new Child("Ruby");
        Child child2 = new Child("Groovy");

        parent.addChild(child1);
        parent.addChild(child2);

        session.persist(parent);

        session.getTransaction().commit();
        session.close();

    }
}
```

hibernate.cfg.xml

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- Database connection and other settings -->
    ... ... ...
    <mapping class="entity.Parent" />
    <mapping class="entity.Child" />

  </session-factory>
</hibernate-configuration>
```

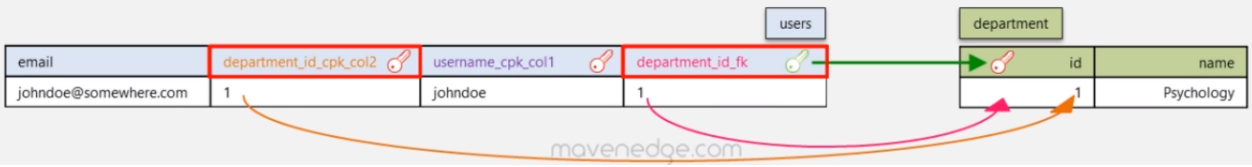
- ***Foreign key in composite primary key using MapsId***




```java
package entity;
import javax.persistence.*;

@Entity
@Table(name = "users")
public class User {

   @EmbeddedId
   private UserId userId;

   @Column(nullable=false)
   private String email;

   @ManyToOne
   @JoinColumn(name="department_id_fk")
   private Department department;

   public User() {}
   public User(UserId userId, String email) {
      this.userId = userId;
      this.email = email;
   }

   public void setDepartment(Department department) {
      this.department = department;
   }
   //set, get and toString methods
}
```

```java
package entity;
import javax.persistence.*;

@Embeddable
public class UserId implements Serializable {

   @Column(name="username_cpk_col1")
   private String username;

   @Column(name="department_id_cpk_col2")
   private Long departmentId;

   public UserId() {}
   public UserId(String username, Long departmentId) {
      this.username = username;
      this.departmentId = departmentId;
   }

   @Override
   public boolean equals(Object o) {
      //method implementation
   }
   @Override
   public int hashCode() {
      //method implementation
   }
   //set, get and toString methods
}
```

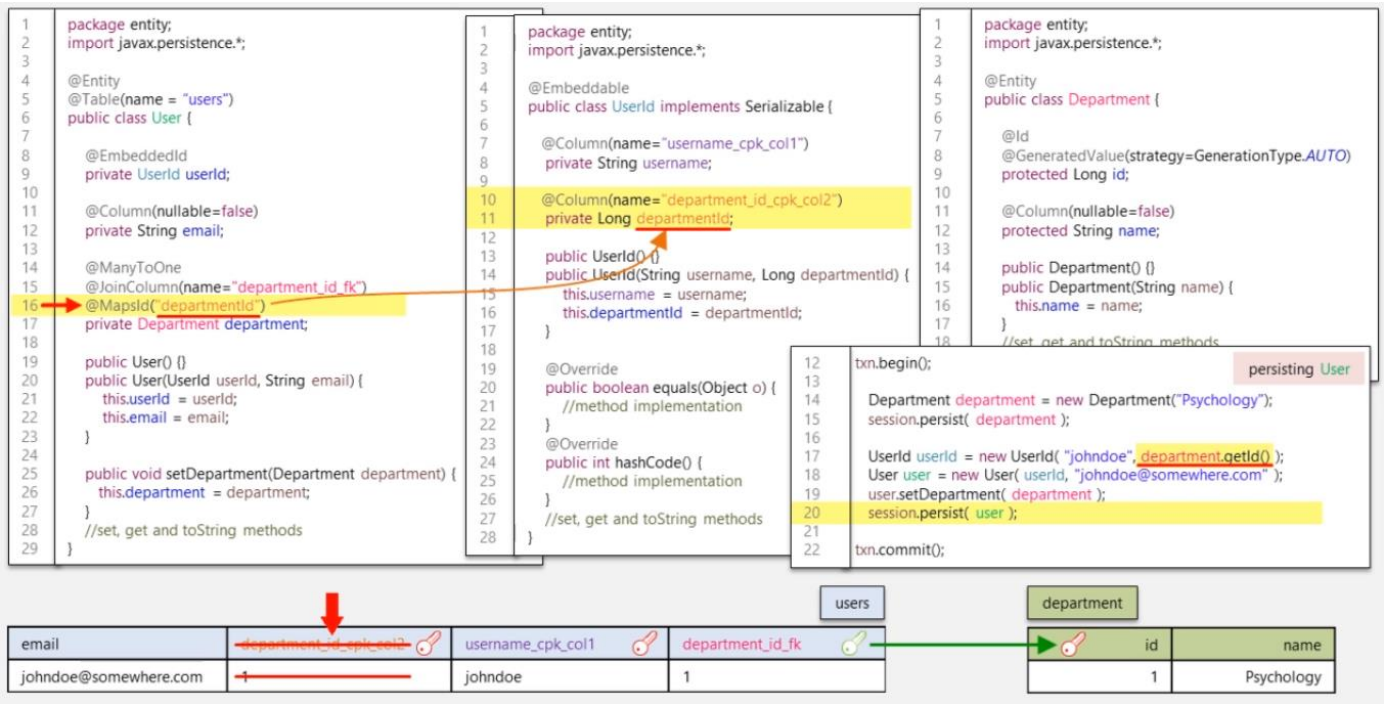
```java
package entity;
import javax.persistence.*;

@Entity
public class Department {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   protected Long id;

   @Column(nullable=false)
   protected String name;

   public Department() {}
   public Department(String name) {
      this.name = name;
   }
   //set, get and toString methods
}
```



5. Duplication problems

```
12  txn.begin();                                          persisting User
13
14  Department department = new Department("Psychology");
15  session.persist( department );
16
17  UserId userId = new UserId( "johndoe", department.getId() );
18  User user = new User( userId, "johndoe@somewhere.com" );
19  user.setDepartment( department );
20  session.persist( user );                    department_id_fk
21
22  txn.commit();
```

6. Duplication Dept ID



7. Use mapsId to avoid duplication

```java
package client;

import org.hibernate.*;
import entity.*;
import util.HibernateUtil;

public class HelloWorldClient {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //persisting User
        /*
        Department department = new Department("Psychology");
        session.persist(department);

        UserId id = new UserId("johndoe", null); //with @MapsId("departmentId"), even if you pass "null", it won't matter!
        User user = new User(id, "johndoe@somewhere.com");
        user.setDepartment(department); // Required!
        session.persist(user);
        */

        //retrieving User
        UserId userId = new UserId("johndoe", 1L);
        User user = (User) session.get(User.class, userId);
        System.out.println( user );

        session.getTransaction().commit();
        session.close();
    }
}
```

Result

```
User [userId=UserId [username=johndoe, departmentId=1],
      email=johndoe@somewhere.com,
      department=Department [id=1, name=Psychology]
]
```

users

email	username_cpk_col1	department_id_fk
johndoe@somewhere.com	johndoe	1

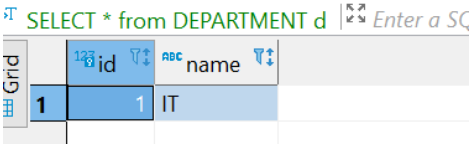
department

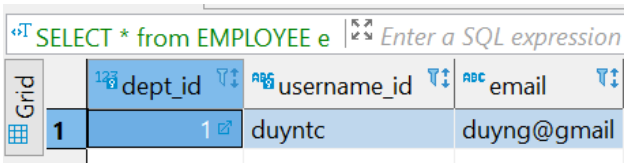
id	name
1	Psychology

```java
public class Employee {
    @EmbeddedId
    private EmployeeID ID;

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

    @ManyToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "dept_id") // use dept_id as fk and composite key too !
    @MapsId("departmentID")
    private Department department;
```

```java
public class EmployeeID implements Serializable {

    @Column(name = "username_id")
    private String userName;

    @Column(name = "department_id")
    private Long departmentID;
```

```java
public class Department {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department")
    private Set<Employee> employees;
```



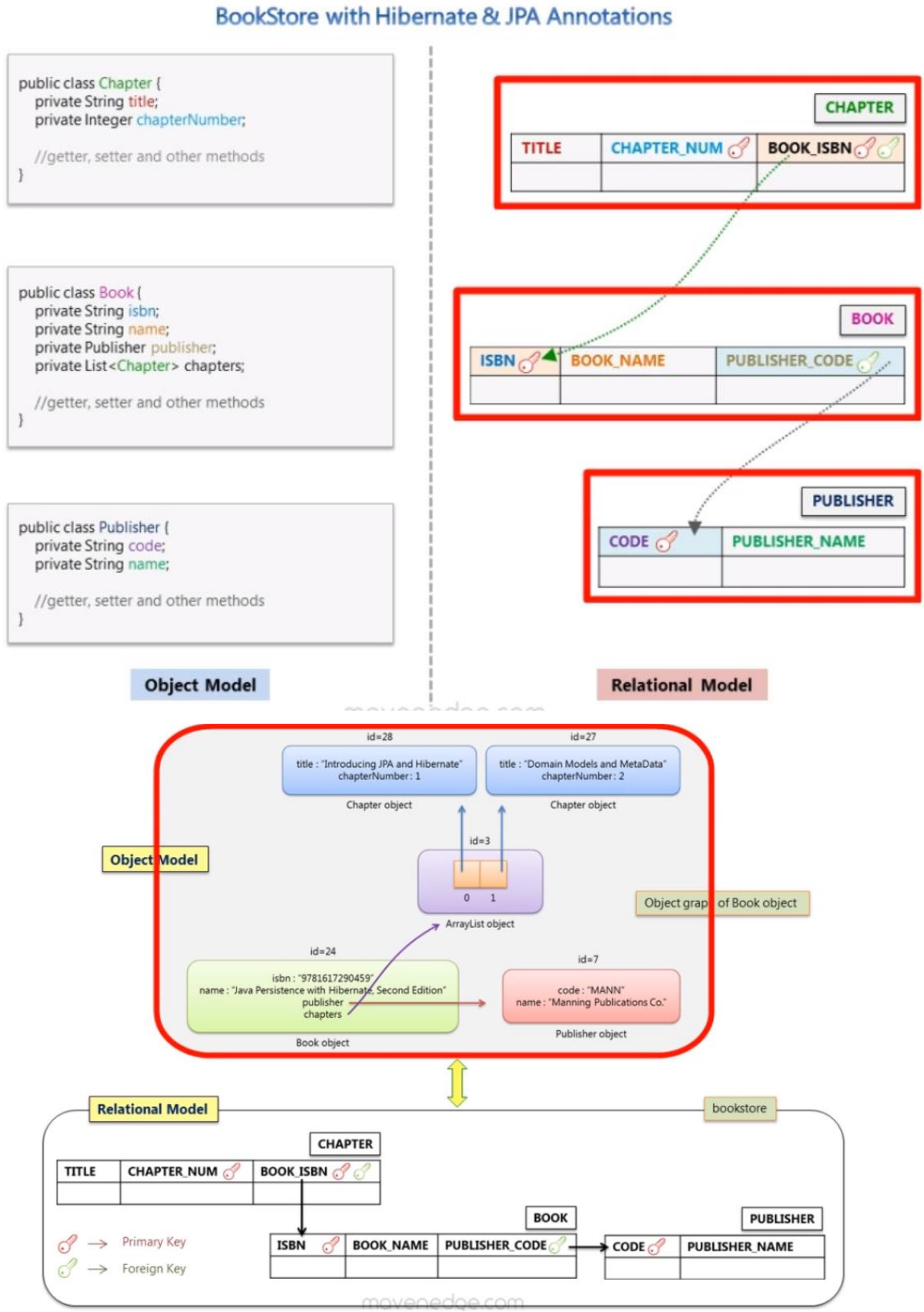



```java
Transaction transaction = null;
try (Session session = HibernateUtil.getSessionFactory().openSession()){
    try {
        transaction = session.getTransaction();
        transaction.begin();

        Department department = Department.builder().name("IT").build();
        session.persist(department);

        EmployeeID employeeID = EmployeeID.builder()
                                .departmentID(department.getId())
                                .userName("duyntc")
                                .build();

        Employee employee = Employee.builder()
                                    .email("duyng@gmail")
                                    .department(department)
                                    .ID(employeeID)
                                    .build();
        session.persist(employee);

        transaction.commit();
```

- ***Generation type:***

Generation Type	Description
AUTO	The **GenerationType.AUTO** is the default generation type and lets the persistence provider choose the generation strategy. If you use Hibernate as your persistence provider, it selects a generation strategy based on the **database specific dialect**. For most popular databases, it selects **GenerationType.SEQUENCE** which I will explain later.
IDENTITY	The **GenerationType.IDENTITY** is the easiest to use but not the best one from a performance point of view. It relies on an **auto-incremented** database column and lets the database generate a new value with each insert operation. This approach has a significant drawback if you use **Hibernate**. Hibernate requires a primary key value for each managed entity and therefore has to perform the insert statement immediately. This prevents it from using different optimization techniques like **JDBC batching**.
SEQUENCE	The **GenerationType.SEQUENCE** is my preferred way to generate primary key values and uses a database sequence to generate unique values.
TABLE	The **GenerationType.TABLE** gets only rarely used nowadays. It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order. This slows down your application, and you should, therefore

- ***Practice with hibernate and JPA annotation.***



- ***What is JPA***

JPA: java specification for mapping between java objects and SQL objects

JPA provider: Hibnerate, esclipseLink.

- ***Hibernate as JPA provider***

8. Refactor Hibnerate specific code

Hibernate

```
package client;

import org.hibernate.Session;
import org.hibernate.Transaction;

import util.HibernateUtil;
import entity.Message;

public class HelloWorldClient {
  public static void main(String[] args) {

      SessionFactory sf = HibernateUtil.getSessionFactory();
      Session session = sf.openSession();
      Transaction txn = session.getTransaction();
      try {
        txn.begin();

        Message msg= new Message("Hello World with Hibernate");
        session.persist(msg);

        txn.commit();
      }catch(Exception e) {
        if(txn != null) { txn.rollback(); }
```

hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
      <!-- Database connection settings --> ...
      <!-- SQL dialect --> ...
       <!-- Create/update database tables automatically using mapping metadata --> ...
       <!-- Persistent entity classes --> ...
   </session-factory>
</hibernate-configuration>
```

JPA

```
package client;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

import entity.Message;

public class HelloWorldClient {
  public static void main(String[] args) {

      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
      EntityManager entityManager = emf.createEntityManager();
      EntityTransaction txn = em.getTransaction();
     try {
         txn.begin();

         Message msg= new Message("Hello World with Hibernate as JPA Provider");
         entityManager.persist(msg);

         txn.commit();
      }catch(Exception e) {
```

persistence.xml

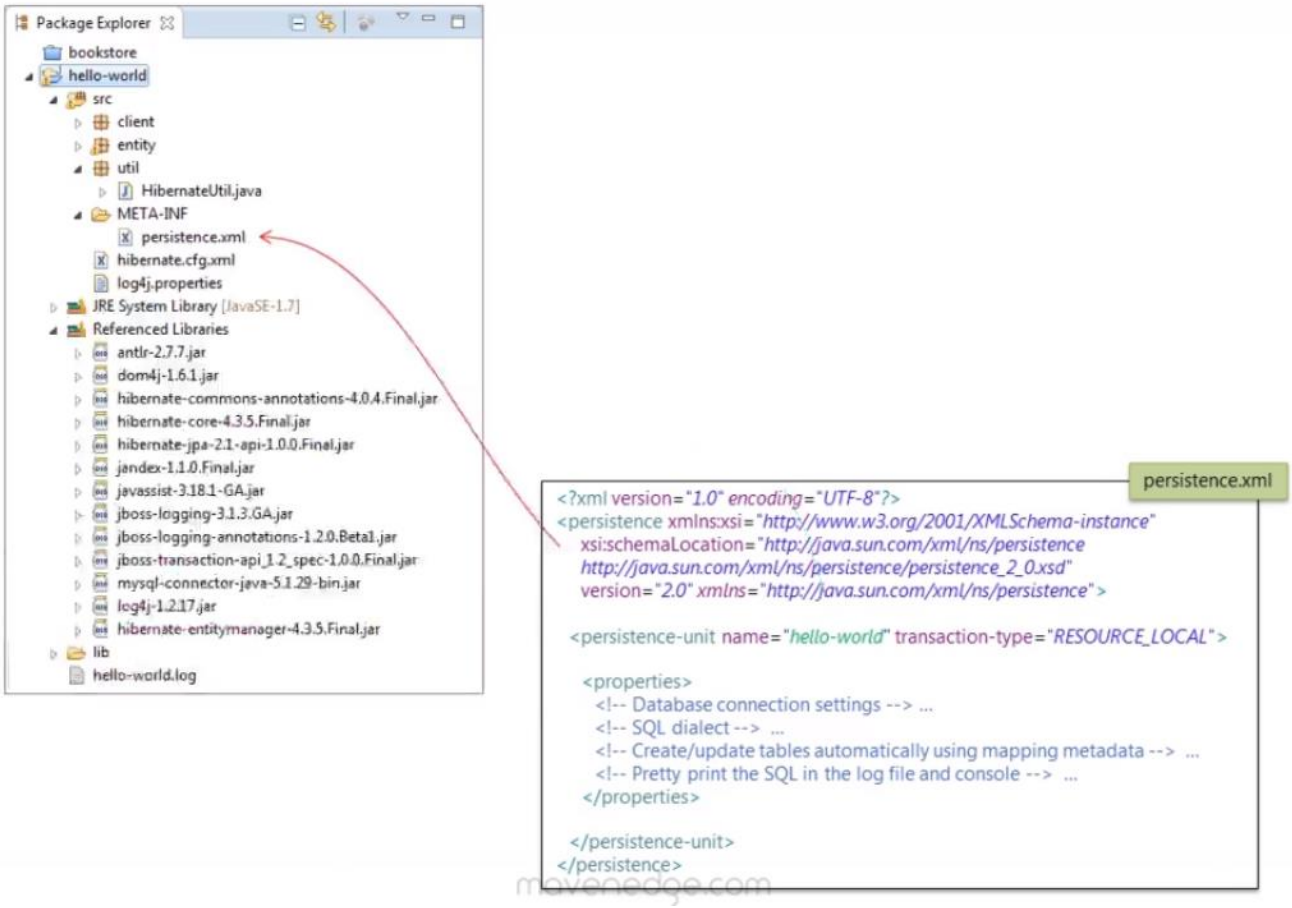
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="hello-world" transaction-type="RESOURCE_LOCAL">
    <properties>
      <!-- Database connection settings --> ...
      <!-- SQL dialect --> ...
      <!-- Create/update tables automatically using mapping metadata --> ...
    </properties>
  </persistence-unit>
</persistence>
```


JPA

```
package client;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

import entity.Message;

public class HelloWorldClient {
  public static void main(String[] args) {

      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
      EntityManager entityManager = emf.createEntityManager();
      EntityTransaction txn = em.getTransaction();
      try {
         txn.begin();

         Message msg= new Message("Hello World with Hibernate as JPA Provider");
         entityManager.persist(msg);

         txn.commit();
      }catch(Exception e) {
         if(txn != null) { txn.rollback(); }
         e.printStackTrace();
      }finally {
         if(entityManager != null) { entityManager.close(); }
      }
  }
}
```

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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_2_0.xsd"
   version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="hello-world" transaction-type="RESOURCE_LOCAL">

    <properties>
      <!-- Database connection settings --> ...
      <!-- SQL dialect --> ...
      <!-- Create/update tables automatically using mapping metadata --> ...
      <!-- Pretty print the SQL in the log file and console --> ...
    </properties>

  </persistence-unit>
</persistence>
```



Question 1: How can we avoid persisting the age field in the Person entity in Figure 1, while persisting an instance of it?

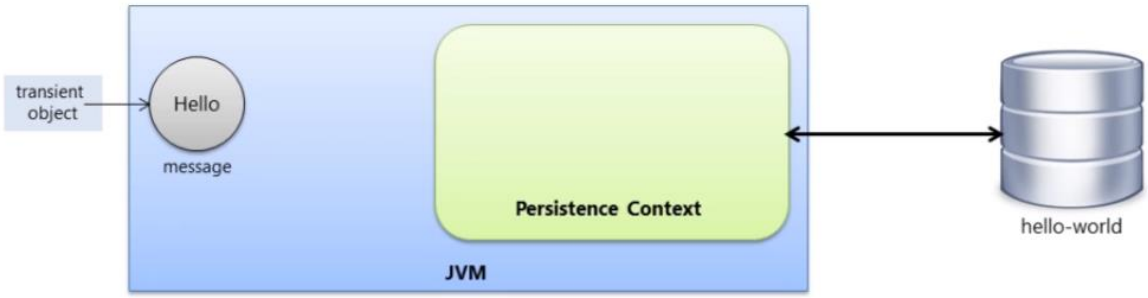
```
@Entity
public static class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private String name;

    private Date dateOfBirth;

    @Transient
    private Integer age;

    public Integer calculateAge() {
        //calculate and return age
    }
    //....
}
```

Figure 1

9. Using @Transient keyword to avoid persist age to DB

► *Objects life cycle*

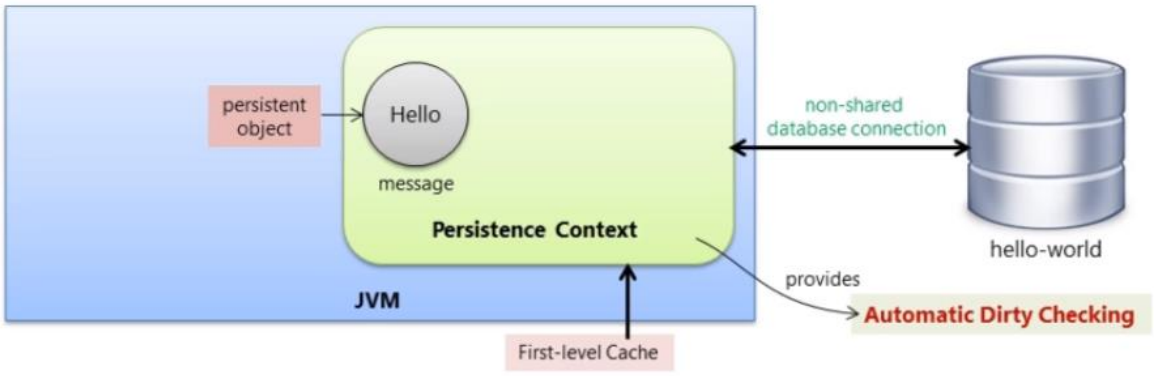
```
1  public class HelloWorldClient {
2      public static void main(String[] args) {
3
4          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
5          EntityManager em = emf.createEntityManager();
6          em.getTransaction().begin();
7
8          Message message = new Message("Hello"); //transient state
9
10         em.persist(message); //persistent state
11
12         em.getTransaction().commit();
13         em.close(); //once em is closed the message object becomes detached
14
15         message.setText("Hi"); //detached state
16
17     }
18 }
```



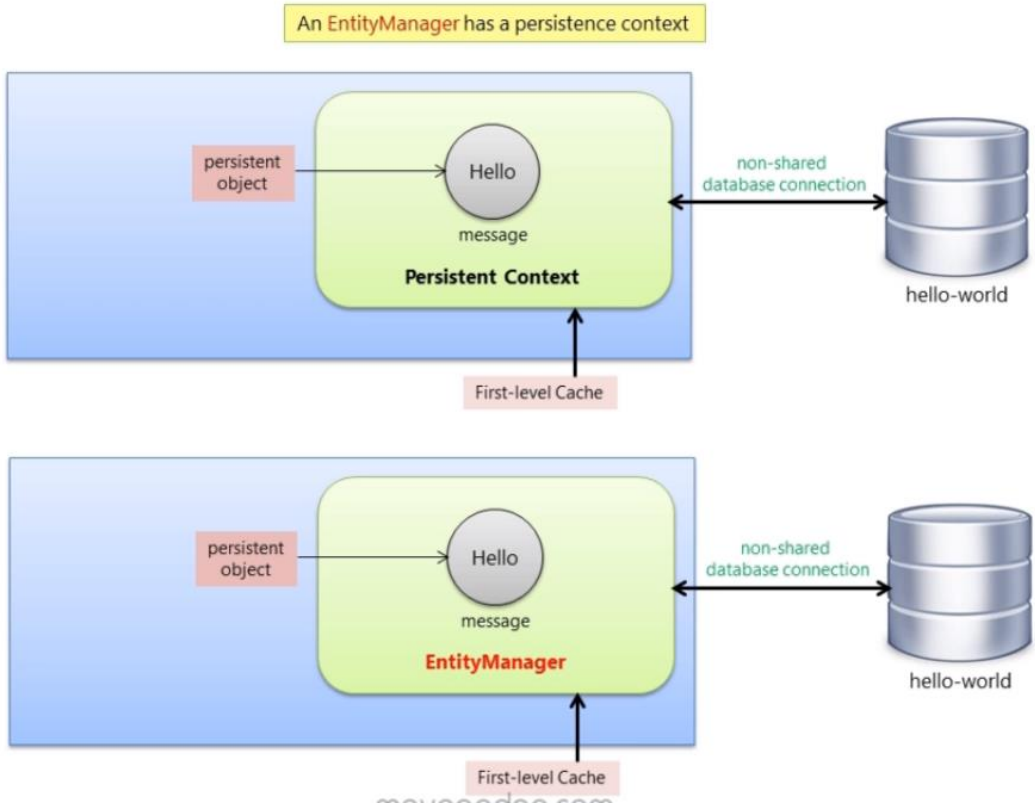
10. Transient Object State

```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Message message = new Message("Hello"); //transient state

        em.persist(message); //persistent state

        em.getTransaction().commit();
        em.close(); //once em is closed the message object becomes detached

        message.setText("Hi"); //detached state

    }
}
```

```java
Message msg = em.find(Message.class, 2L);
```



11. Persistence Object State



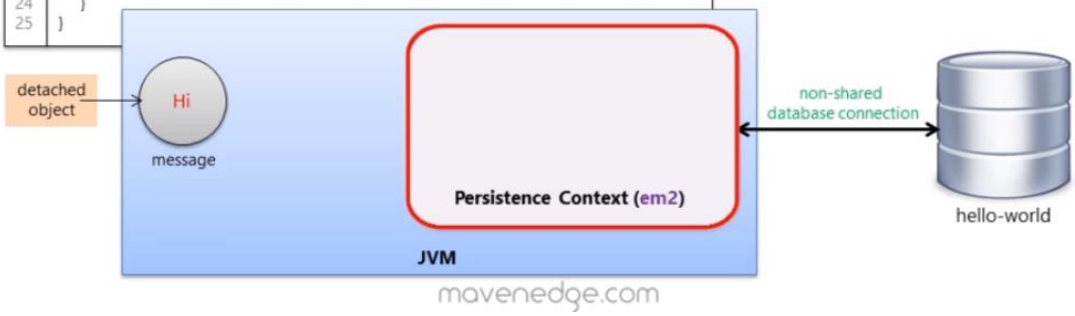
```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Message message = new Message("Hello"); //transient state

        em.persist(message); //persistent state

        em.getTransaction().commit();
        em.close(); //once em is closed the message object becomes detached

        message.setText("Hi"); //detached state

    }
}
```

12. Commit state, db connection close

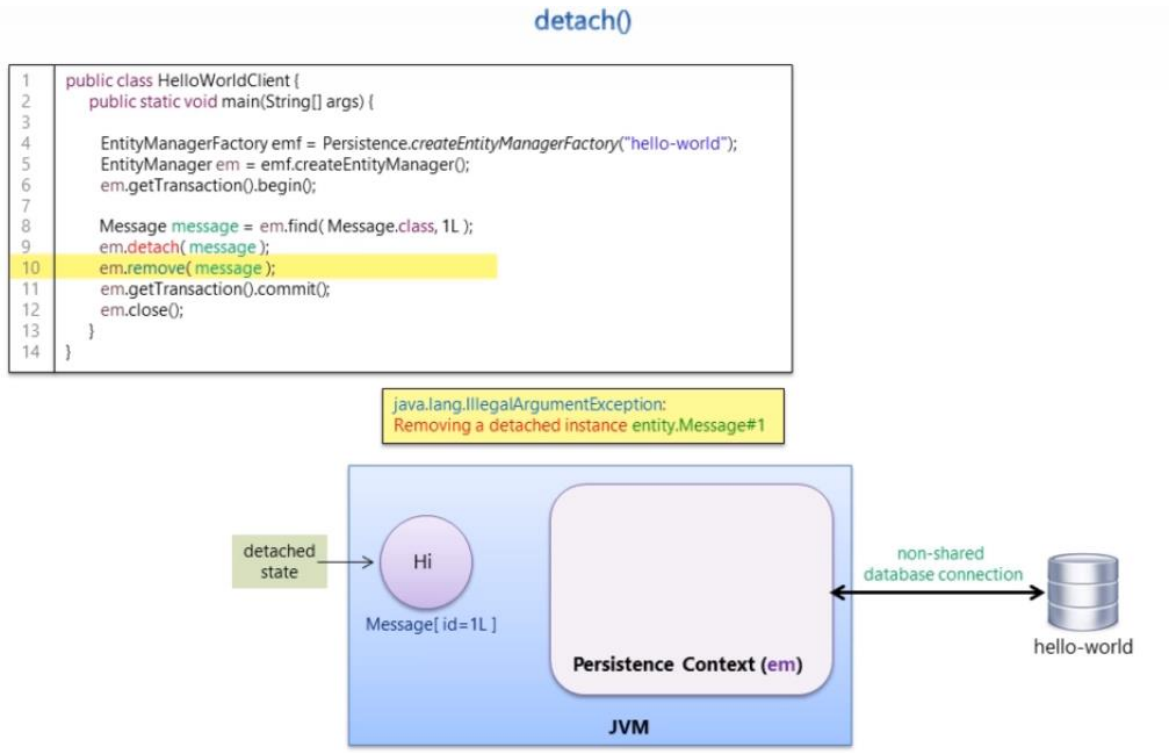
```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Message message = new Message("Hello"); //transient state

        em.persist(message); //persistent state

        em.getTransaction().commit();
        em.close(); //once em is closed the message object becomes detached

        message.setText("Hi"); //detached state

    }
}
```

13. Detach state, close entity manager, change object do not reflect

```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Message message = new Message("Hello"); //transient state

        em.persist(message); //persistent state

        em.getTransaction().commit();
        em.close(); //once em is closed the message object becomes detached

        message.setText("Hi"); //detached state

        EntityManager em2 = emf.createEntityManager();
        em2.getTransaction().begin();

        Message mergedMessage = em2.merge(message);

        em2.getTransaction().commit();
        em2.close();
    }
}
```



14. Reattach detach object using merge

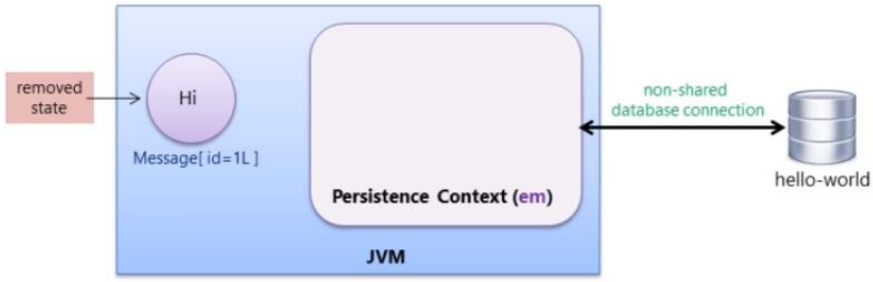


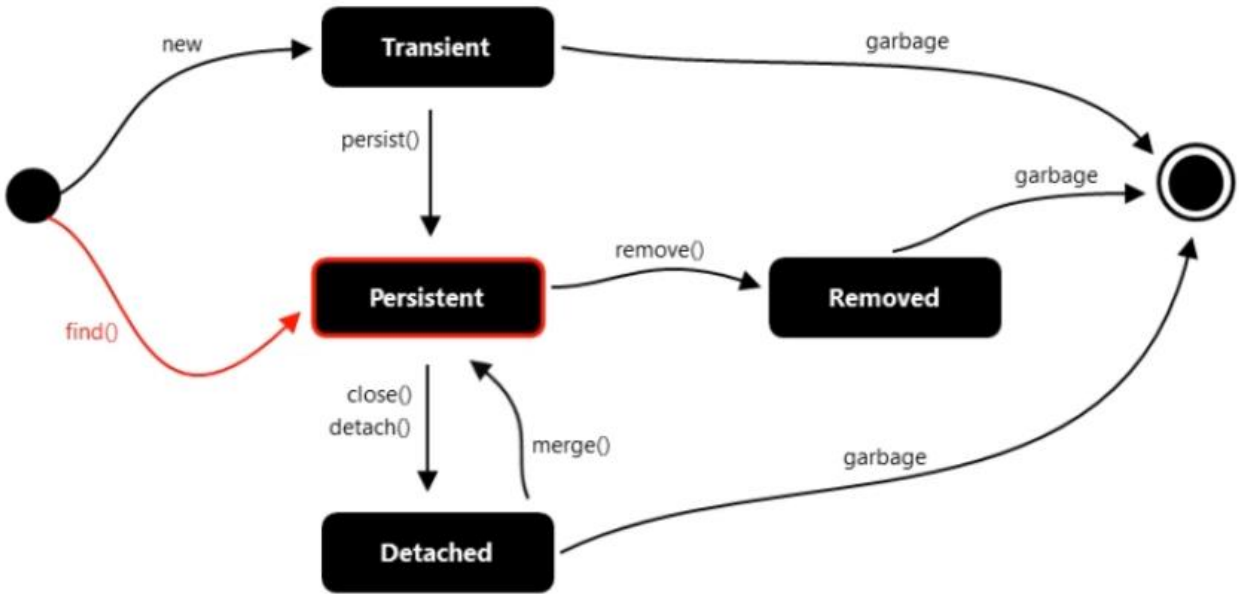
remove()

```
1  public class HelloWorldClient {
2      public static void main(String[] args) {
3
4          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
5          EntityManager em = emf.createEntityManager();
6          em.getTransaction().begin();
7
8          Message message = em.find( Message.class, 1L );
9          em.remove( message );
10
11         em.getTransaction().commit();
12         em.close();
13     }
14 }
```

```
delete from Message where id=?

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

message	
ID	TEXT
~~1~~	~~Hi~~





Task 0: Download the source-code, run it, make yourself comfortable with the concepts covered in it

Question 1: In the Question1Client given below, when the line 21 is executed, will the message object persisted to the database or not?

```
...
8   public class Question1Client {
9       public static void main(String[] args) {
10          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
11
12          EntityManager em = emf.createEntityManager();
13          EntityTransaction txn = em.getTransaction();
14          txn.begin();
15
16          Message message = new Message("Hello");
17          em.persist( message );
18
19          em.detach( message );
20
21          txn.commit();
22          em.close();
23      }
24  }
```

Question 1: until **commit**, database do not change.

Lab Exercise: Working with Objects contd.

Question 2: Considering the bi-directional One-To-Many Guide-To-Student relationship as shown below, what would be the result of executing the line 28 in Question2Client?

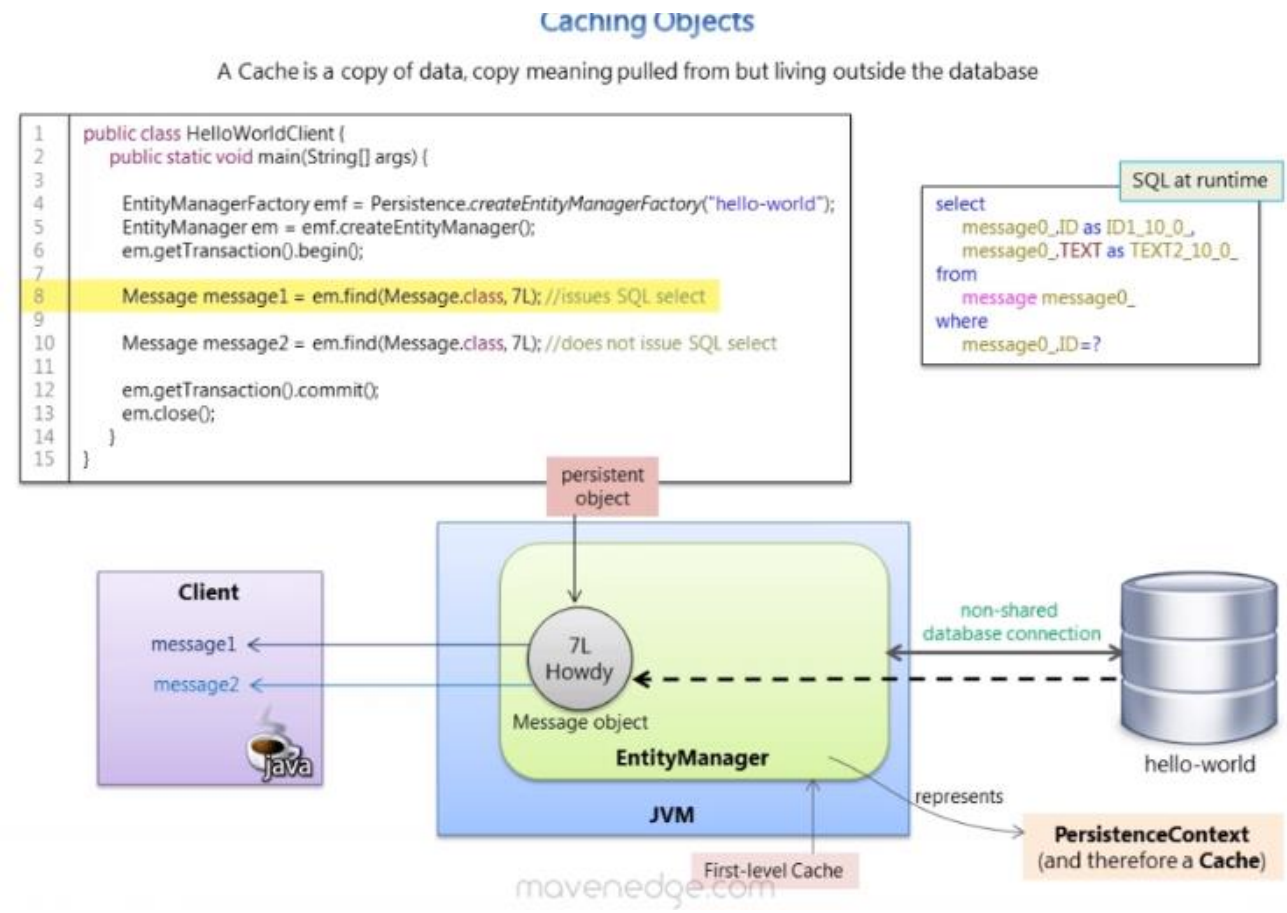
```
8   public class Question2Client {
9       public static void main(String[] args) {
10          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
11
12          EntityManager em1 = emf.createEntityManager();
13          em1.getTransaction().begin();
14
15          Guide guide = new Guide("2000RN10349", "Rose Ann", 4000);
16          em1.persist(guide);
17
18          em1.getTransaction().commit();
19          em1.close();
20
21          EntityManager em2 = emf.createEntityManager();
22          em2.getTransaction().begin();
23
24          Guide mergedGuide = em2.merge(guide);
25          mergedGuide.addStudent(new Student("2015JR50244", "Jason Bird"));
26          mergedGuide.addStudent(new Student("2015LK50878", "Lisa Mizuki"));
27
28          em2.getTransaction().commit();
29          em2.close();
30      }
31  }
```

```
@Entity
public class Student {
    //...
    @ManyToOne(cascade={CascadeType.PERSIST})
    @JoinColumn(name="guide_id")
    private Guide guide;
    //...
}
```

```
@Entity
public class Guide {
    //...
    @OneToMany(mappedBy = "guide", cascade = {CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();

    public void addStudent(Student student) {
        students.add(student);
        student.setGuide(this);
    }
    //...
}
```

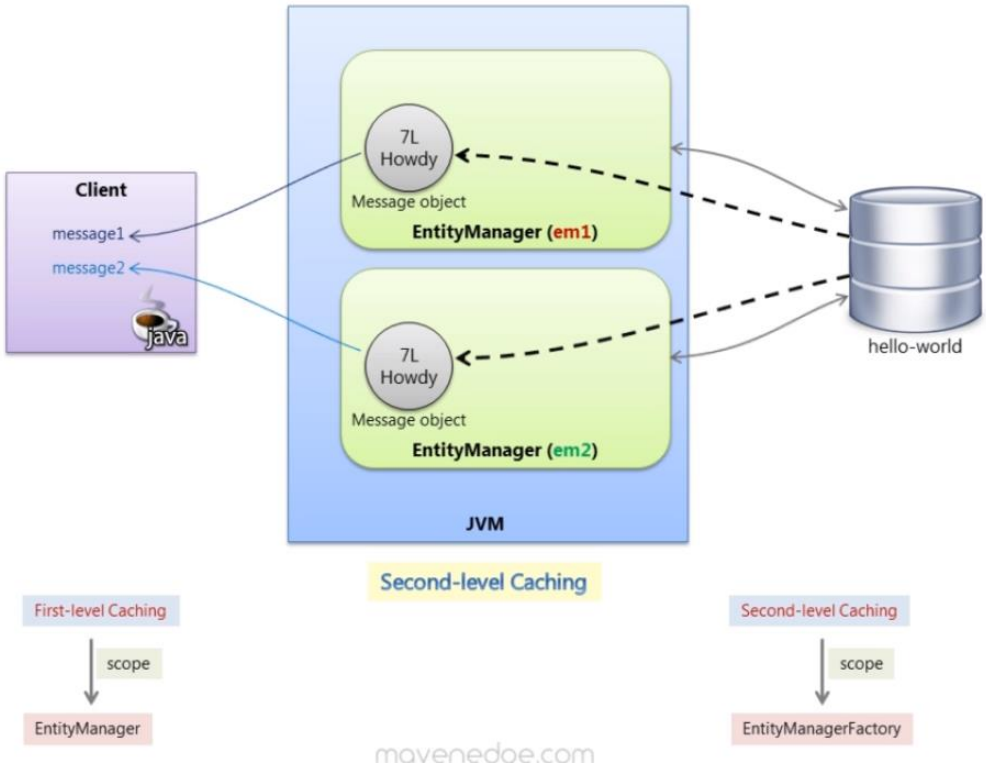

Question 2: in second transaction, guide in persistence state.

► *Caching objects*

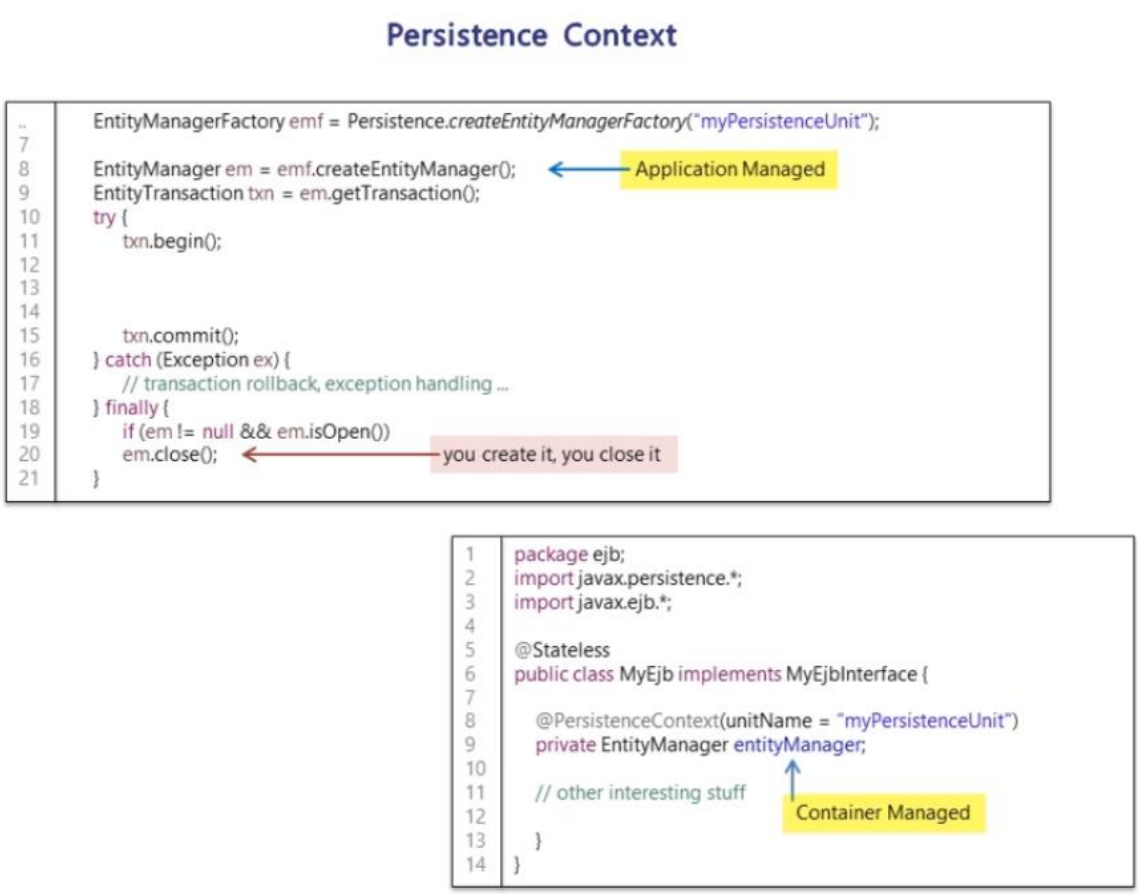



```
1  public class HelloWorldClient {
2      public static void main(String[] args) {
3
4          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
5          EntityManager em1 = emf.createEntityManager();
6          em1.getTransaction().begin();
7
8          Message message1 = em1.find(Message.class, 7L); //issues SQL select
9
10         em1.getTransaction().commit();
11         em1.close();
12
13         EntityManager em2 = emf.createEntityManager();
14         em2.getTransaction().begin();
15
16         Message message2 = em2.find(Message.class, 7L); //issues SQL select
17
18         em2.getTransaction().commit();
19         em2.close();
20     }
21 }
```

```
SQL at runtime

select
    message0_.ID as ID1_10_0_,
    message0_.TEXT as TEXT2_10_0_
from
    message message0_
where
    message0_.ID=?

select
    message0_.ID as ID1_10_0_,
    message0_.TEXT as TEXT2_10_0_
from
    message message0_
where
    message0_.ID=?
```



- ***Persistence context***



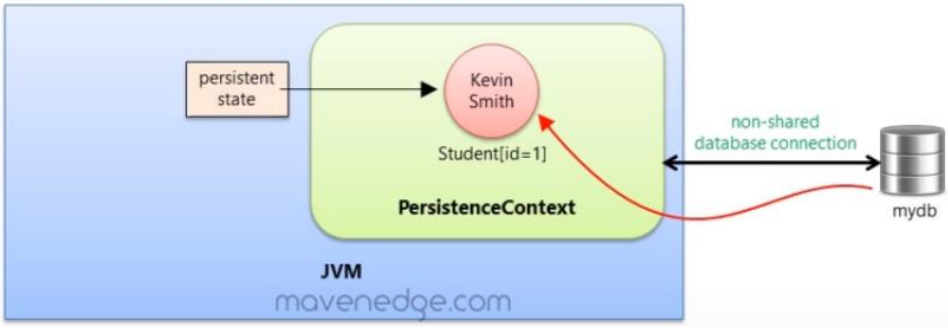
Automatic Dirty Checking

```
..  package client;
7
8   import javax.persistence.*;
9   import entity.Student;
10
11  public class AutomaticDirtyCheckingClient {
12      public static void main(String[] args) {
13          EntityManagerFactory emf = Persistence.createEnti
14
15          EntityManager em = emf.createEntityManager();
16          em.getTransaction().begin();
17
18          Student student = em.find(Student.class, 1L);
19
20
21          em.getTransaction().commit();
22          em.close();
23      }
24  }
```

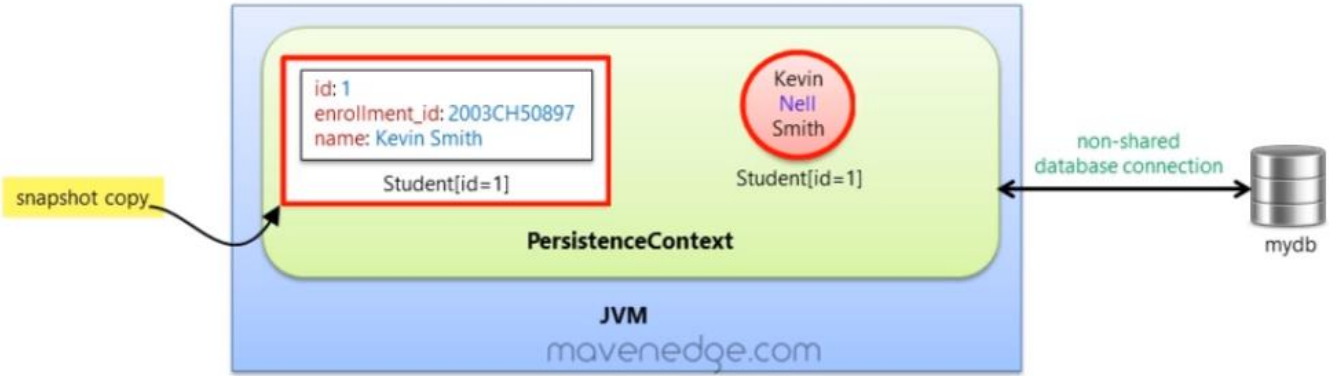
```
@Entity
public class Student {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(name="enrollment_id", nullable = false)
    private String enrollmentId;
    //set, get and other methods
}
```

student

ID	enrollment_id	name
1	2003CH50897	Kevin Smith
2	2008EE10999	Sherry Morgan
3	1987CS10007	Sara Shield




```
..  package client;
7
8   import javax.persistence.*;
9   import entity.Student;
10
11  public class AutomaticDirtyCheckingClient {
12      public static void main(String[] args) {
13          EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
14
15          EntityManager em = emf.createEntityManager();
16          em.getTransaction().begin();
17
18          Student student = em.find(Student.class, 1L);
19          student.setName("Kevin Nell Smith");
20
21          em.getTransaction().commit();
22          em.close();
23      }
24  }
```



MyCustomEntityDirtinessStrategy.java

```java
package custom;

import org.hibernate.CustomEntityDirtinessStrategy;
import org.hibernate.Session;
import org.hibernate.persister.entity.EntityPersister;

public class MyCustomEntityDirtinessStrategy implements CustomEntityDirtinessStrategy {
    @Override
    public boolean canDirtyCheck(Object entity, EntityPersister persister, Session session) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public boolean isDirty(Object entity, EntityPersister persister, Session session) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public void resetDirty(Object entity, EntityPersister persister, Session session) {
        // TODO Auto-generated method stub
    }
    @Override
    public void findDirty(Object entity, EntityPersister persister, Session session, DirtyCheckContext dirtyCheckContext) {
        // TODO Auto-generated method stub
    }
}
```

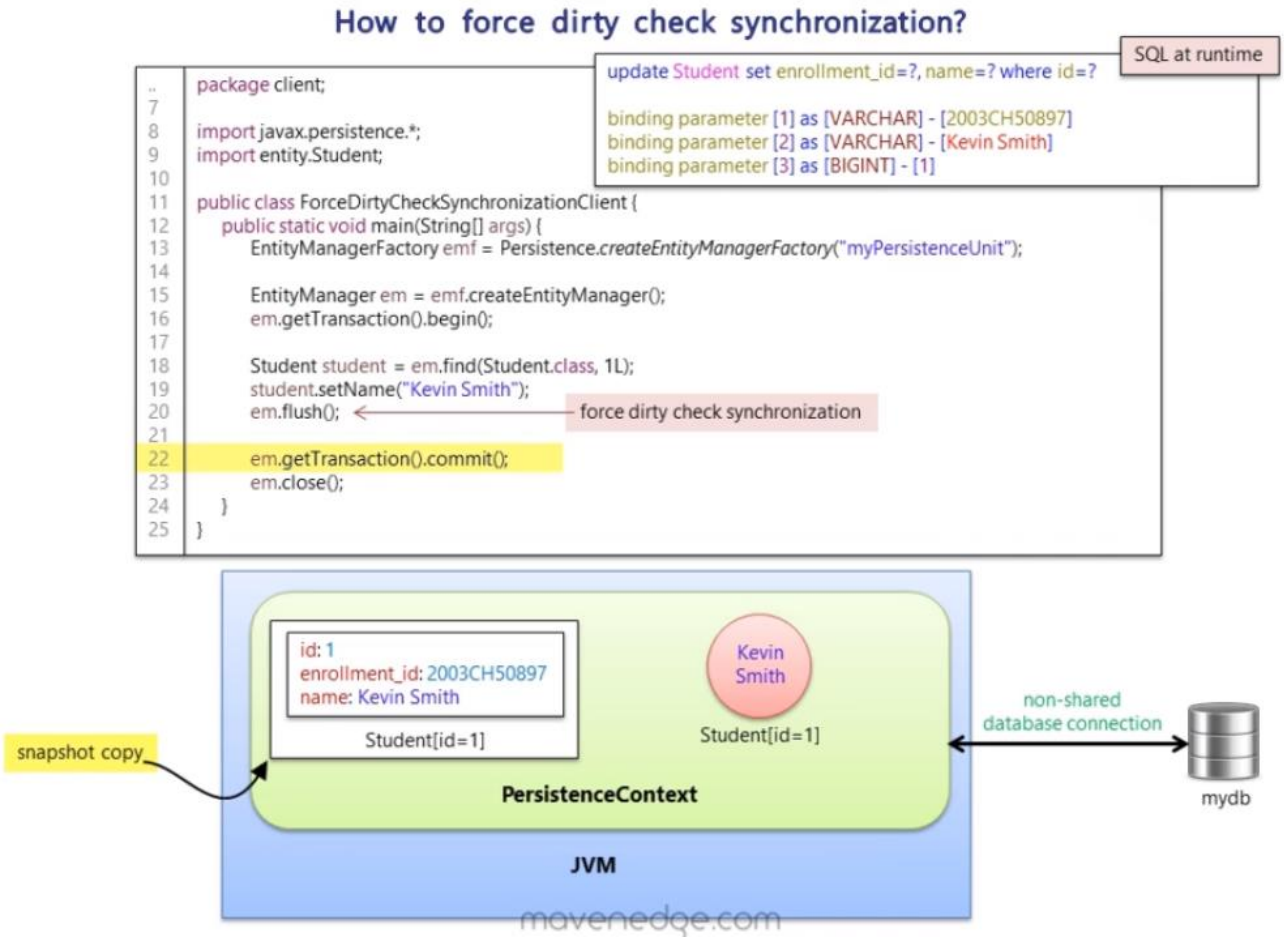
persistence.xml

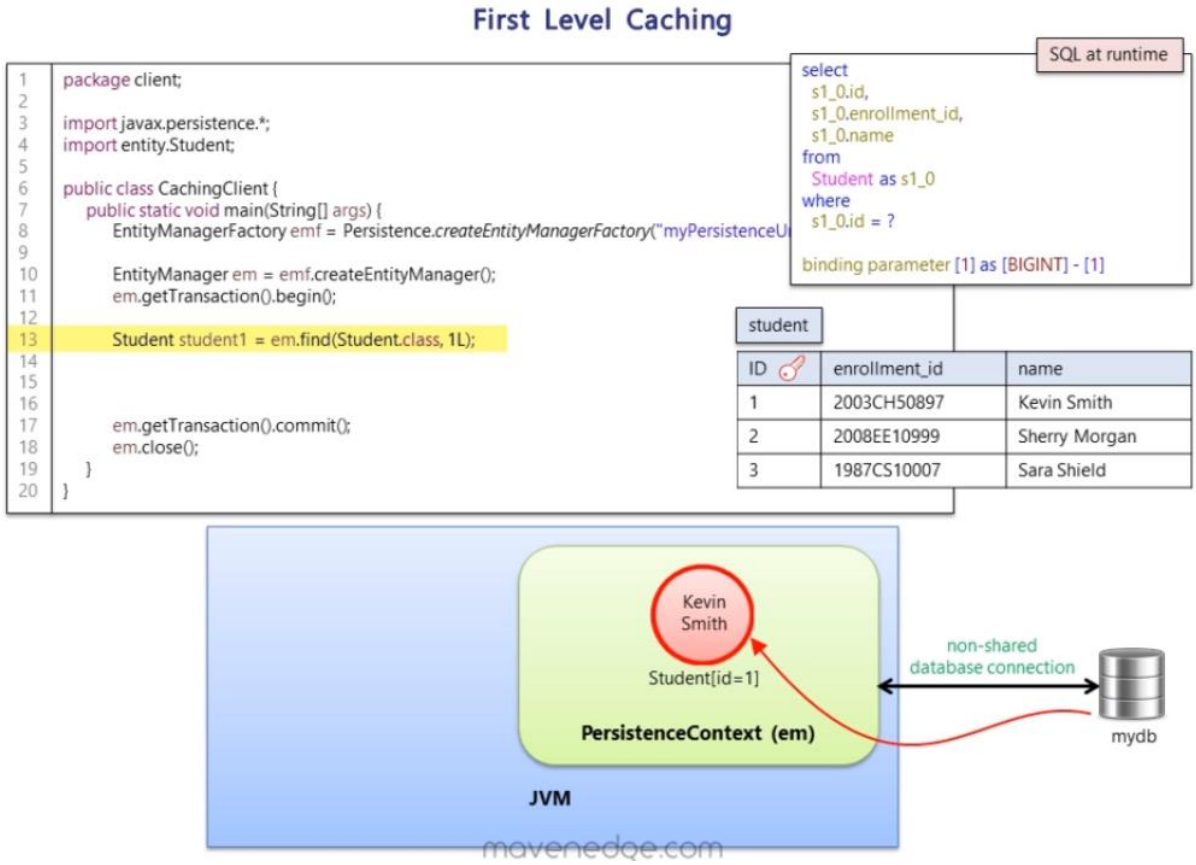
```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">

    <persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <properties>

        <!-- Other properties-->

        <!-- Configure your custom entity dirtiness strategy -->
        <property name="hibernate.entity_dirtiness_strategy" value="custom.MyCustomEntityDirtinessStrategy" />

        </properties>
    </persistence-unit>
</persistence>
```

mavenedge.com

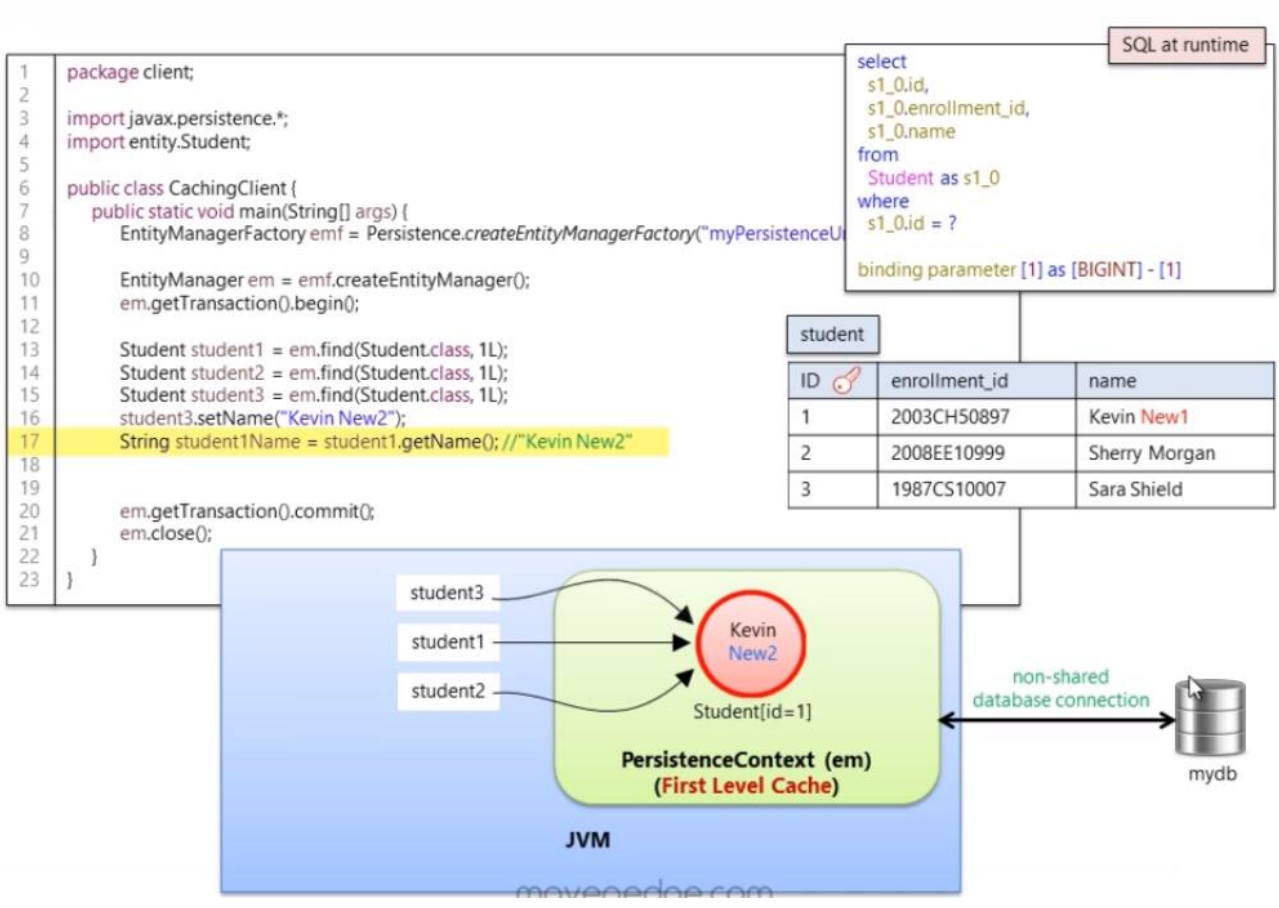
How to force dirty check synchronization?

```java
package client;

import javax.persistence.*;
import entity.Student;

public class ForceDirtyCheckSynchronizationClient {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");

        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Student student = em.find(Student.class, 1L);
        student.setName("Kevin Smith");
        em.flush();

        em.getTransaction().commit();
        em.close();
    }
}
```


How to force dirty check synchronization?
SQL at runtime
update Student set enrollment_id=?, name=? where id=?
binding parameter [1] as [VARCHAR] - [2003CH50897]
binding parameter [2] as [VARCHAR] - [Kevin Smith]
binding parameter [3] as [BIGINT] - [1]
package client;
import javax.persistence.*;
import entity.Student;
public class ForceDirtyCheckSynchronizationClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.find(Student.class, 1L);
student.setName("Kevin Smith");
em.flush();
force dirty check synchronization
em.getTransaction().commit();
em.close();
}
}
id: 1
enrollment_id: 2003CH50897
name: Kevin Smith
Student[id=1]
Kevin Smith
Student[id=1]
snapshot copy
PersistenceContext
non-shared database connection
mydb
JVM
mavenedge.com



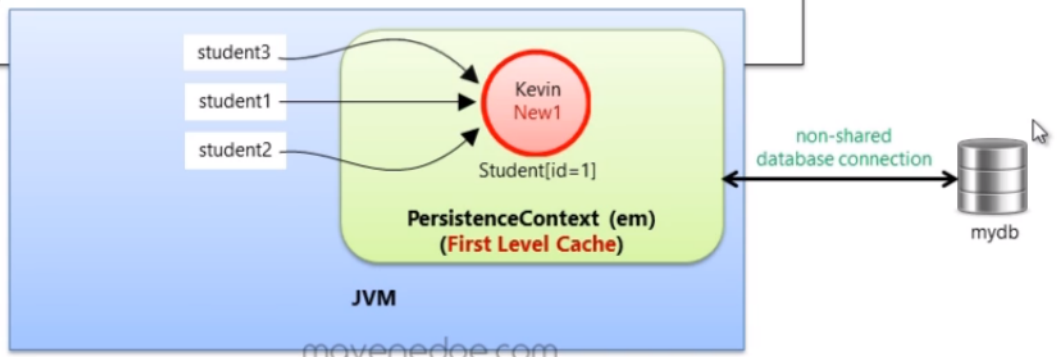
15. First level cache



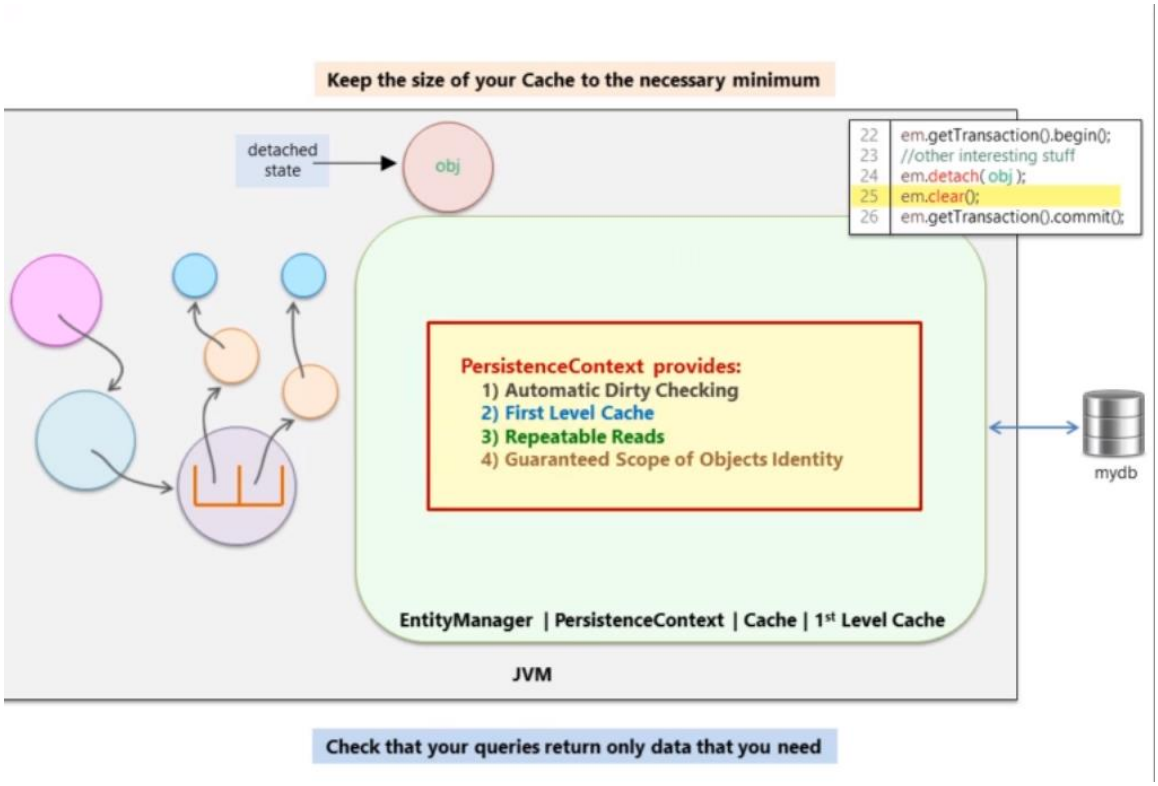
```java
package client;

import javax.persistence.*;
import entity.Student;

public class CachingClient {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");

        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        Student student1 = em.find(Student.class, 1L);
        Student student2 = em.find(Student.class, 1L);
        Student student3 = em.find(Student.class, 1L);
        student3.setName("Kevin New2");
        String student1Name = student1.getName(); //"Kevin New2"
        em.refresh(student3);

        em.getTransaction().commit();
        em.close();
    }
}
```

student

ID	enrollment_id	name
1	2003CH50897	Kevin New1
2	2008EE10999	Sherry Morgan
3	1987CS10007	Sara Shield



16. Refresh issue a select statement to database and retrieve a fresh object



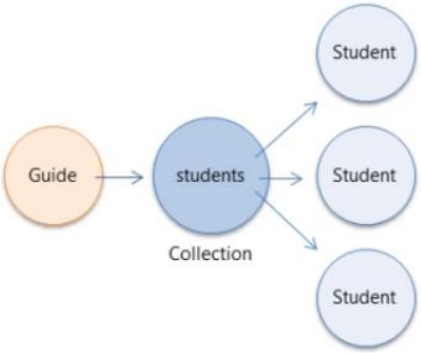
17. Keep cache to minimum and clear () detach all objects from persistence conttext

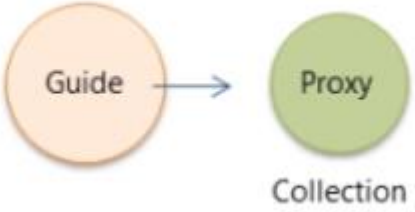
- ***Lazy fetching***

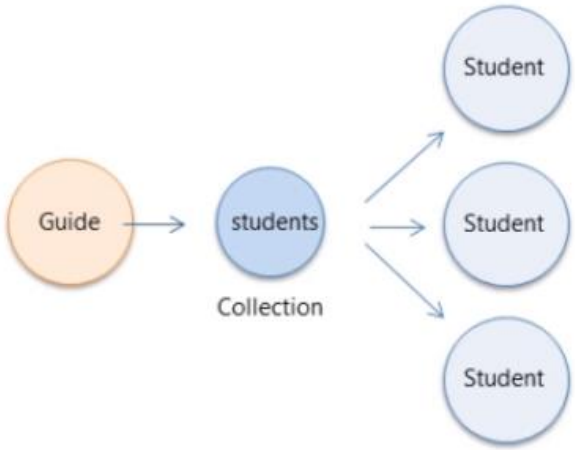
```
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();
        try {
            txn.begin();

            Guide guide = em.find(Guide.class, 2L);
            Set<Student> students = guide.getStudents();
            int numberOfStudents = students.size();

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(em != null) { em.close(); }
        }

    }
}
```

```
select
    guide0_.id as id1_2_0_,
    guide0_.name as name2_2_0_,
    guide0_.salary as salary3_2_0_,
    guide0_.staff_id as staff_id4_2_0_
from
    Guide guide0_
where
    guide0_.id=?

select
    students0_.guide_id as guide_id4_2_0_,
    students0_.id as id1_6_0_,
    students0_.id as id1_6_1_,
    students0_.enrollment_id as enrollme2_6_1_,
    students0_.guide_id as guide_id4_6_1_,
    students0_.name as name3_6_1_
from
    Student students0_
where
    students0_.guide_id=?
```

18. Only the size function is called, the student object is select



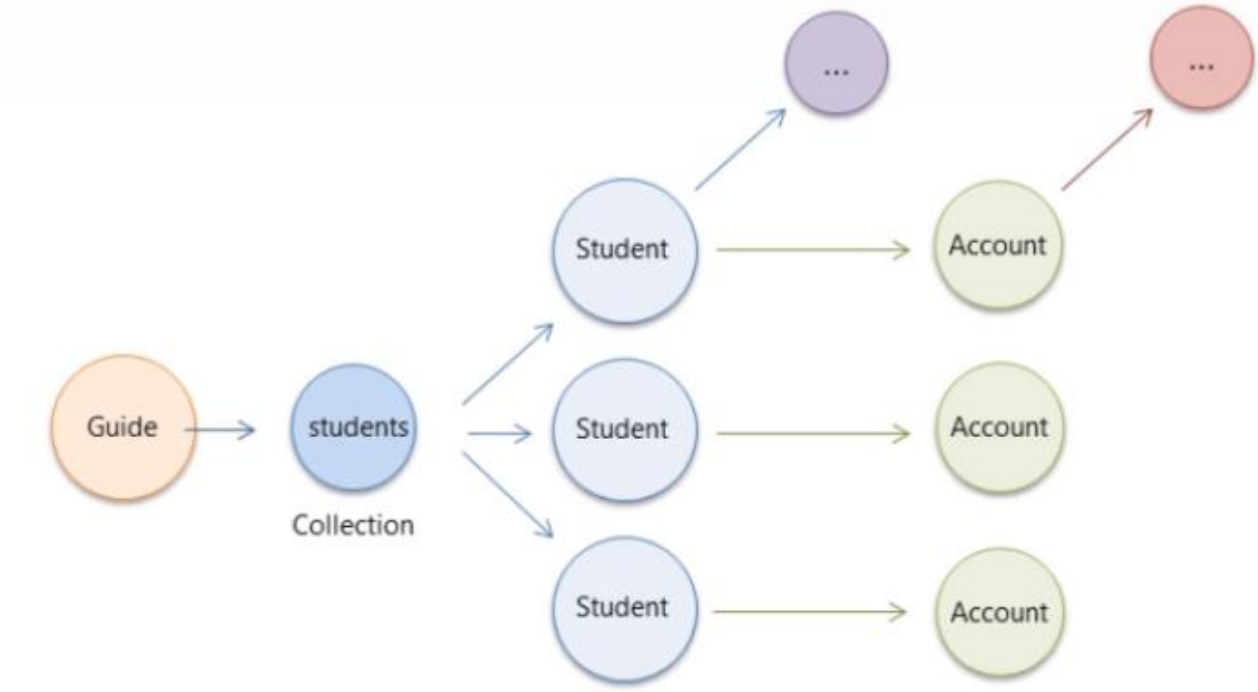




Lazy Collection Fetching

A collection is fetched when the application invokes an operation upon that collection

By default, collection associations (@OneToMany and @ManyToMany) are *lazily* fetched



```
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();

        Guide guide = null;
        Set<Student> students = null;

        try {
            txn.begin();

            guide = em.find(Guide.class, 2L);
            students = guide.getStudents();

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(em != null) { em.close(); }
        }

        int numberOfStudents = students.size(); //bad

    }
}
```

```
select
        guide0_.id as id1_2_0_,
        guide0_.name as name2_2_0_,
        guide0_.salary as salary3_2_0_,
        guide0_.staff_id as staff_id4_2_0_
from
        Guide guide0_
where
        guide0_.id=?

Exception in thread "main" org.hibernate.LazyInitializationException:
                failed to lazily initialize a collection of role: entity.Guide.students,
                could not initialize proxy - no Session
```

```
package entity;

import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="enrollment_id", nullable=false)
  private String enrollmentId;

  private String name;

  @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  @JoinColumn(name="guide_id")
  private Guide guide;

  public Student() {}
  public Student(String enrollmentId, String name, Guide guide) {
    this.enrollmentId = enrollmentId;
    this.name = name;
    this.guide = guide;
  }

  public Guide getGuide() { return guide; }
  public void setGuide(Guide guide) { this.guide = guide; }
}
```

```
package entity;

import javax.persistence.*;

@Entity
public class Guide {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="staff_id", nullable=false)
  private String staffId;

  private String name;
  private Integer salary;

  @OneToMany(mappedBy="guide",
             cascade={CascadeType.PERSIST},
             fetch=FetchType.EAGER)
  private Set<Student> students = new Set<Student>();

  public Guide() {}
  public Guide(String staffId, String name, Integer salary) {
    this.staffId = staffId;
    this.name = name;
    this.salary = salary;
  }

  public Set<Student> getStudents() { return students; }

  public void addStudent(Student student) {
    students.add(student);
    student.setGuide(this);
  }
}
```

```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();
        try {
            txn.begin();

            Guide guide = em.find(Guide.class, 2L);

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(em != null) { em.close(); }
        }

    }
}
```

```sql
select
    guide0_.id as id1_2_0_,
    guide0_.name as name2_2_0_,
    guide0_.salary as salary3_2_0_,
    guide0_.staff_id as staff_id4_2_0_,
    students1_.guide_id as guide_id4_2_1_,
    students1_.id as id1_6_1_,
    students1_.id as id1_6_2_,
    students1_.enrollment_id as enrollme2_6_2_,
    students1_.guide_id as guide_id4_6_2_,
    students1_.name as name3_6_2_
from
    Guide guide0_
left outer join
    Student students1_
        on guide0_.id=students1_.guide_id
where
    guide0_.id=?
```

```java
public class HelloWorldClient {
    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();
        try {
            txn.begin();

            Student student = em.find(Student.class, 2L);

            txn.commit();
        }catch(Exception e) {
            if(txn != null) { txn.rollback(); }
            e.printStackTrace();
        }finally {
            if(em != null) { em.close(); }
        }

    }
}
```

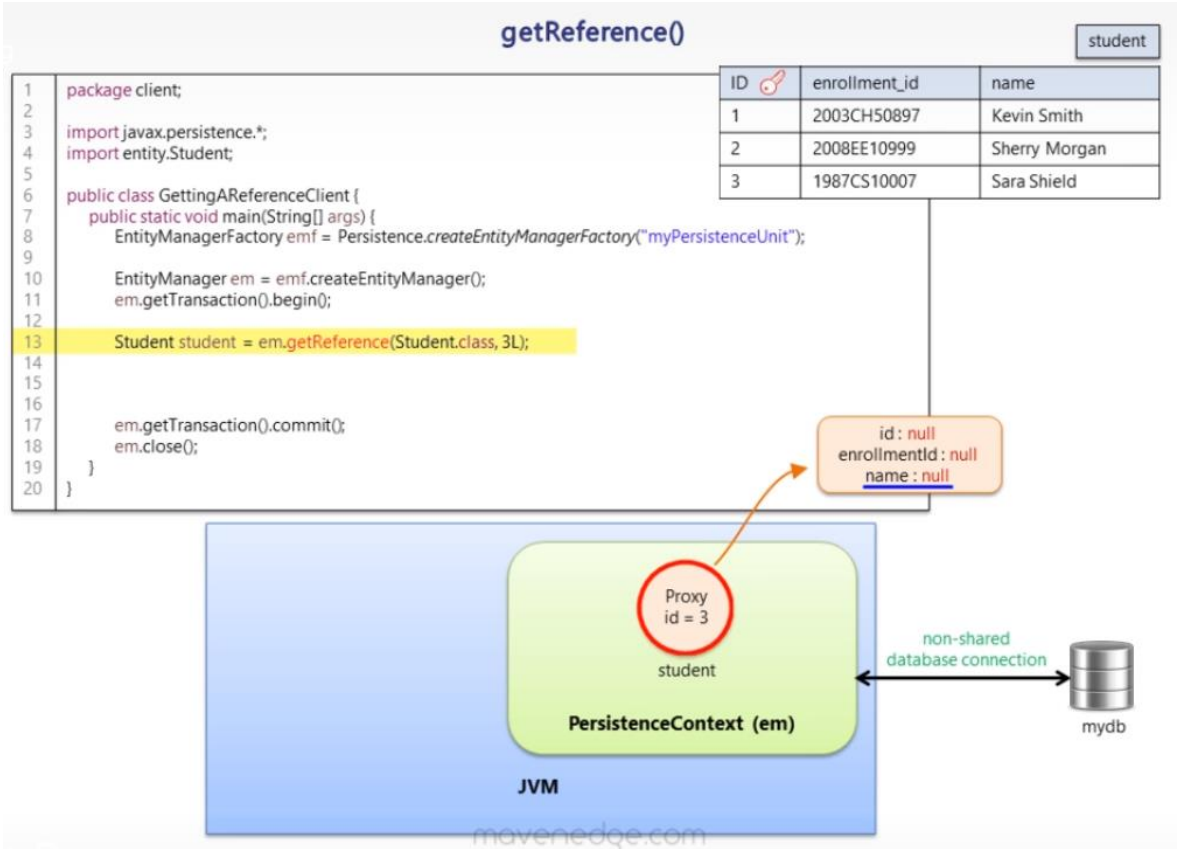
By default, single point associations (@OneToOne and @ManyToOne) are *eagerly* fetched

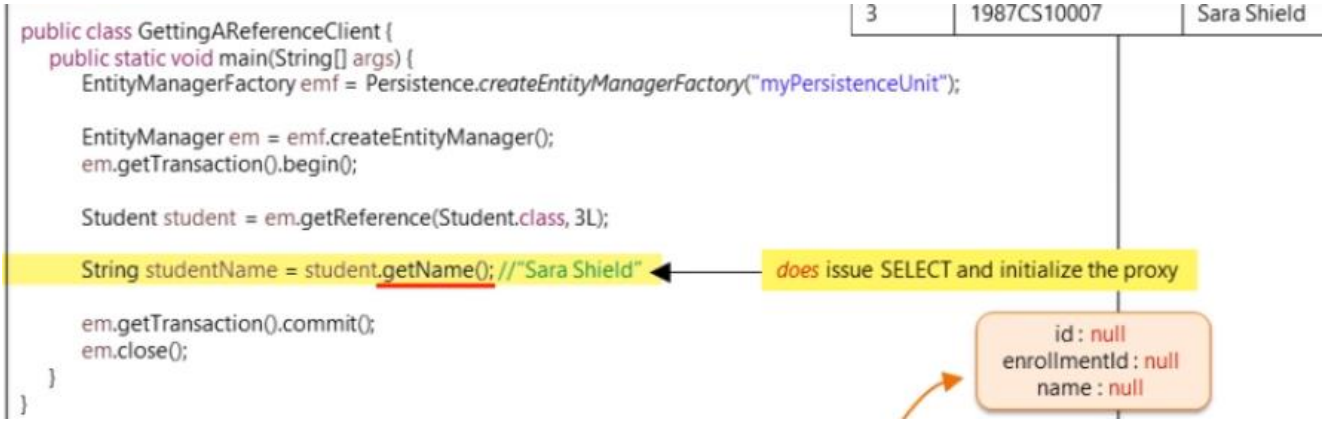
default=FetchType.*EAGER*

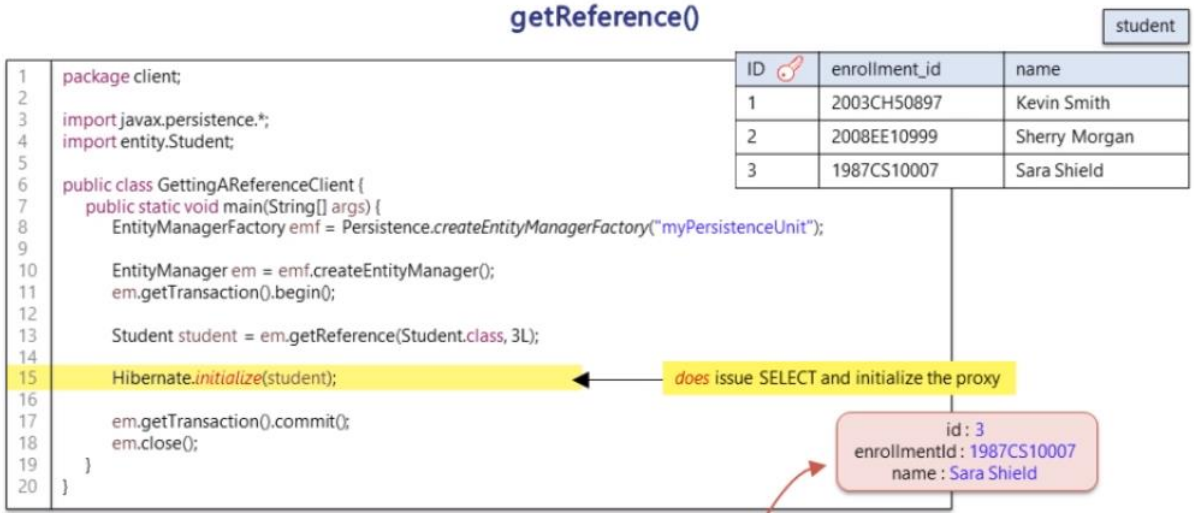
```
package entity;

import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="enrollment_id", nullable=false)
  private String enrollmentId;

  private String name;

  @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  @JoinColumn(name="guide_id")
  private Guide guide;

  public Student() {}
  public Student(String enrollmentId, String name, Guide guide) {
    this.enrollmentId = enrollmentId;
    this.name = name;
    this.guide = guide;
  }

  public Guide getGuide() { return guide; }
  public void setGuide(Guide guide) { this.guide = guide; }
}
```

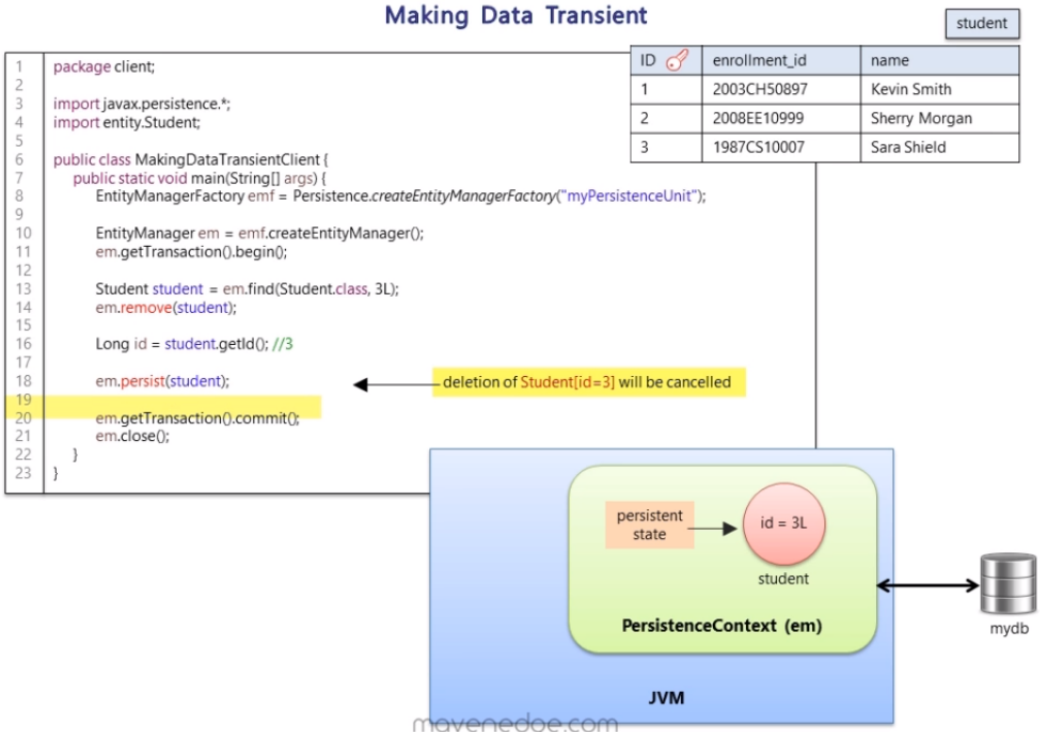
default=FetchType.*LAZY*

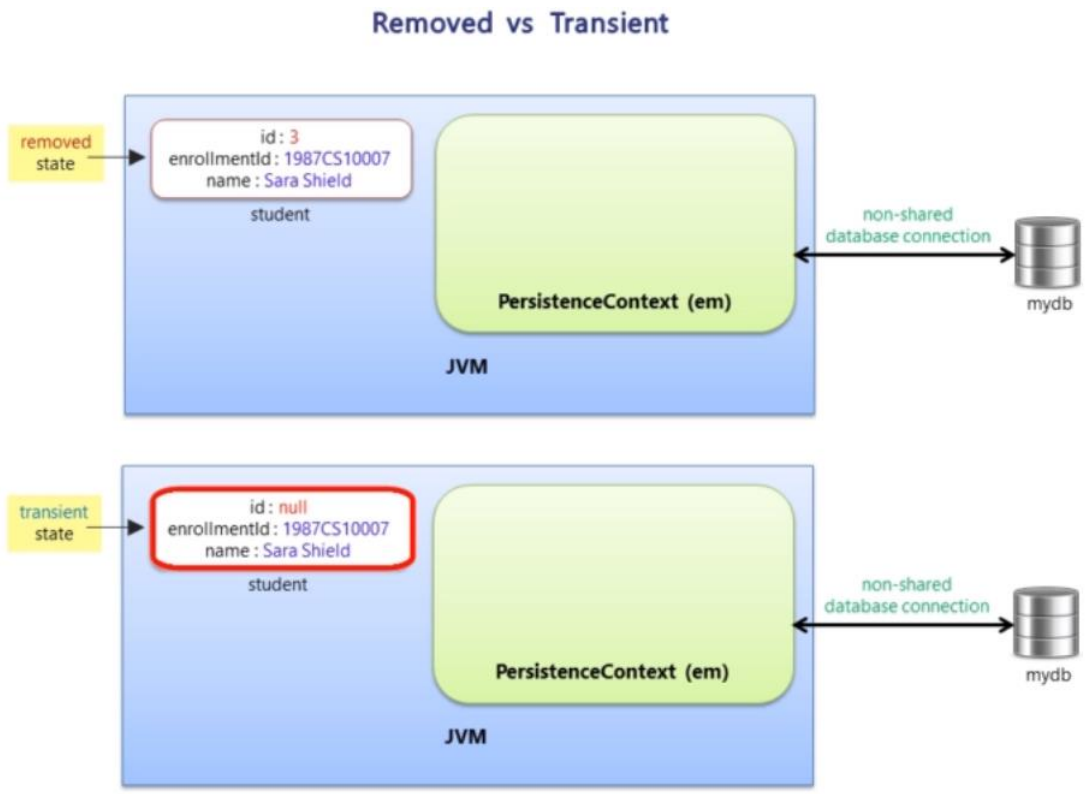
```
package entity;

import javax.persistence.*;

@Entity
public class Guide {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="staff_id", nullable=false)
  private String staffId;

  private String name;
  private Integer salary;

  @OneToMany(mappedBy="guide",
             cascade={CascadeType.PERSIST},
             fetch=FetchType.EAGER)
  private Set<Student> students = new Set<Student>();

  public Guide() {}
  public Guide(String staffId, String name, Integer salary) {
    this.staffId = staffId;
    this.name = name;
    this.salary = salary;
  }

  public Set<Student> getStudents() { return students; }

  public void addStudent(Student student) {
    students.add(student);
    student.setGuide(this);
  }
}
```

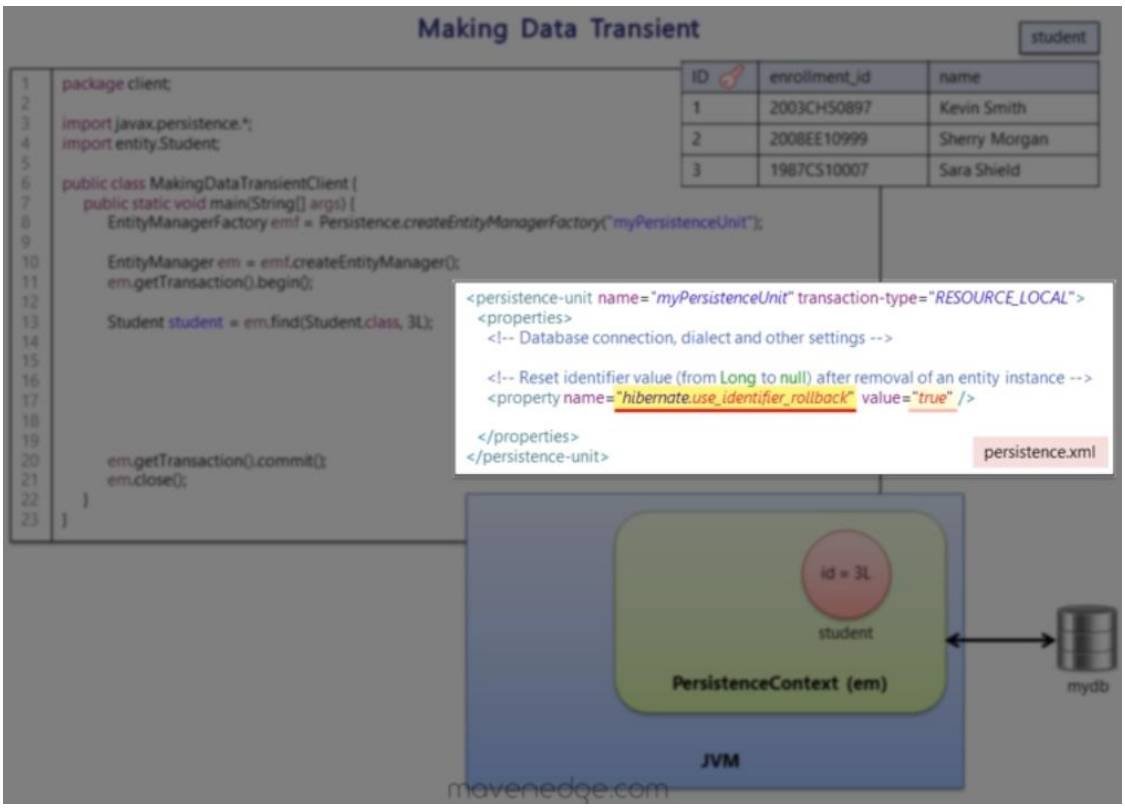


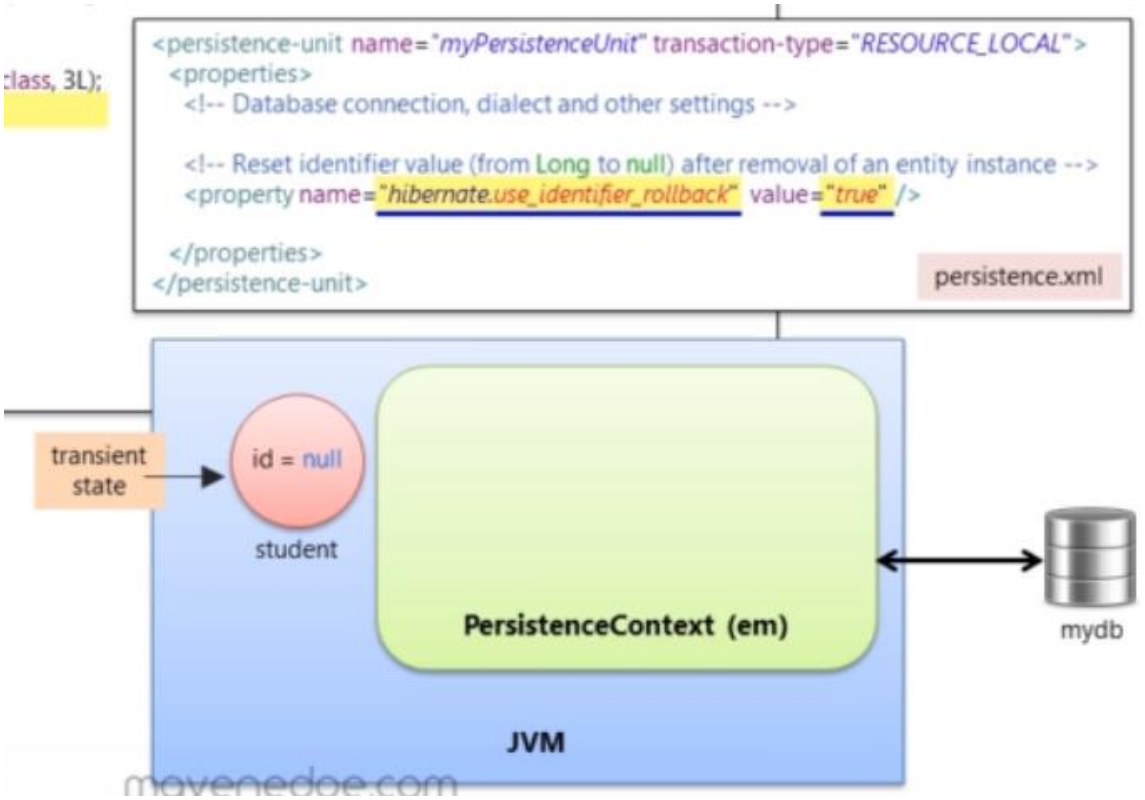

3 | 1987CS10007 | Sara Shield
public class GettingAReferenceClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.getReference(Student.class, 3L);
String studentName = student.getName(); //"Sara Shield"
does issue SELECT and initialize the proxy
em.getTransaction().commit();
em.close();
}
}
id : null
enrollmentId : null
name : null

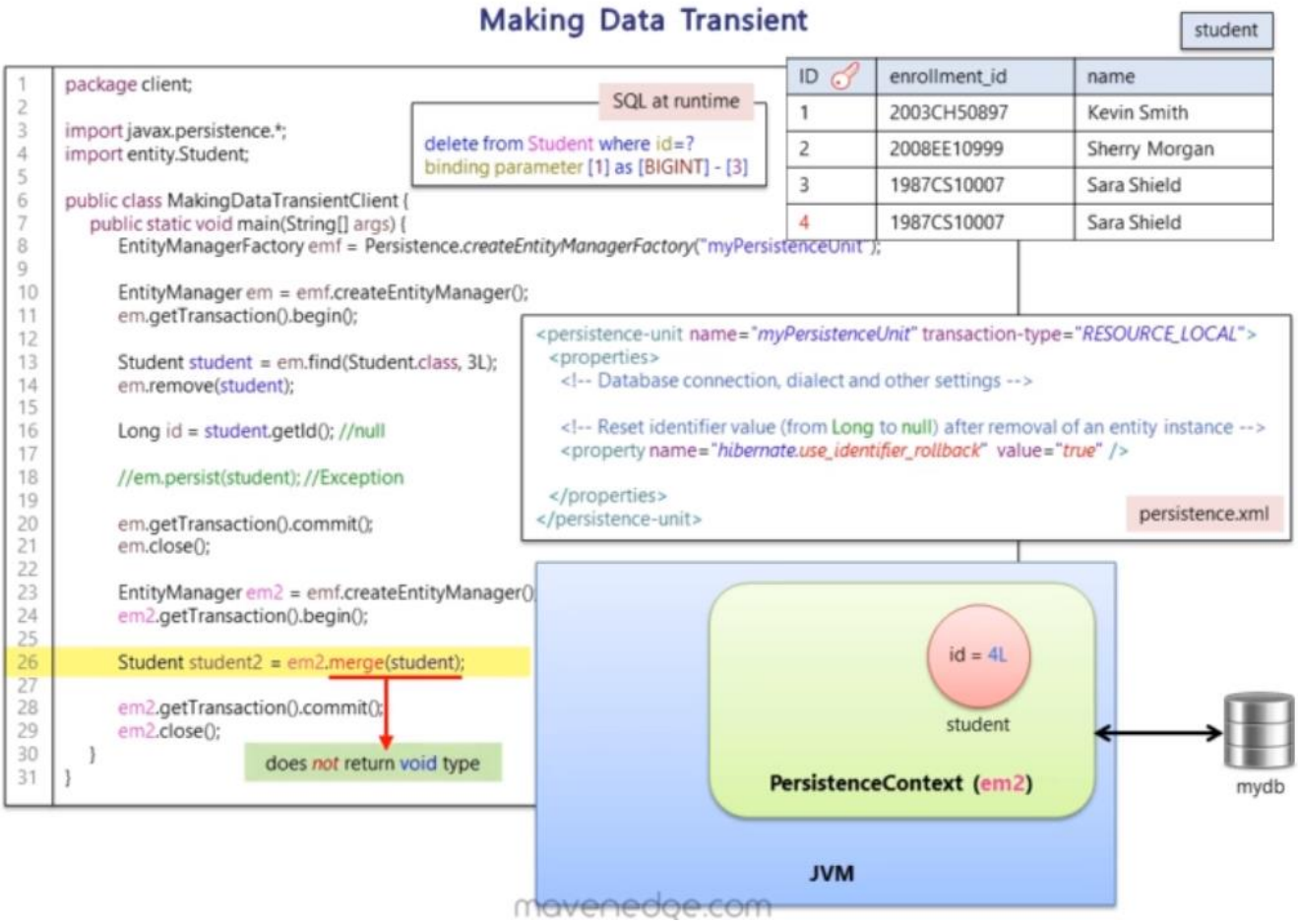

getReference()
student
ID | enrollment_id | name
1 | 2003CH50897 | Kevin Smith
2 | 2008EE10999 | Sherry Morgan
3 | 1987CS10007 | Sara Shield
package client;
import javax.persistence.*;
import entity.Student;
public class GettingAReferenceClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.getReference(Student.class, 3L);
Hibernate.initialize(student);
does issue SELECT and initialize the proxy
em.getTransaction().commit();
em.close();
}
}
id : 3
enrollmentId : 1987CS10007
name : Sara Shield

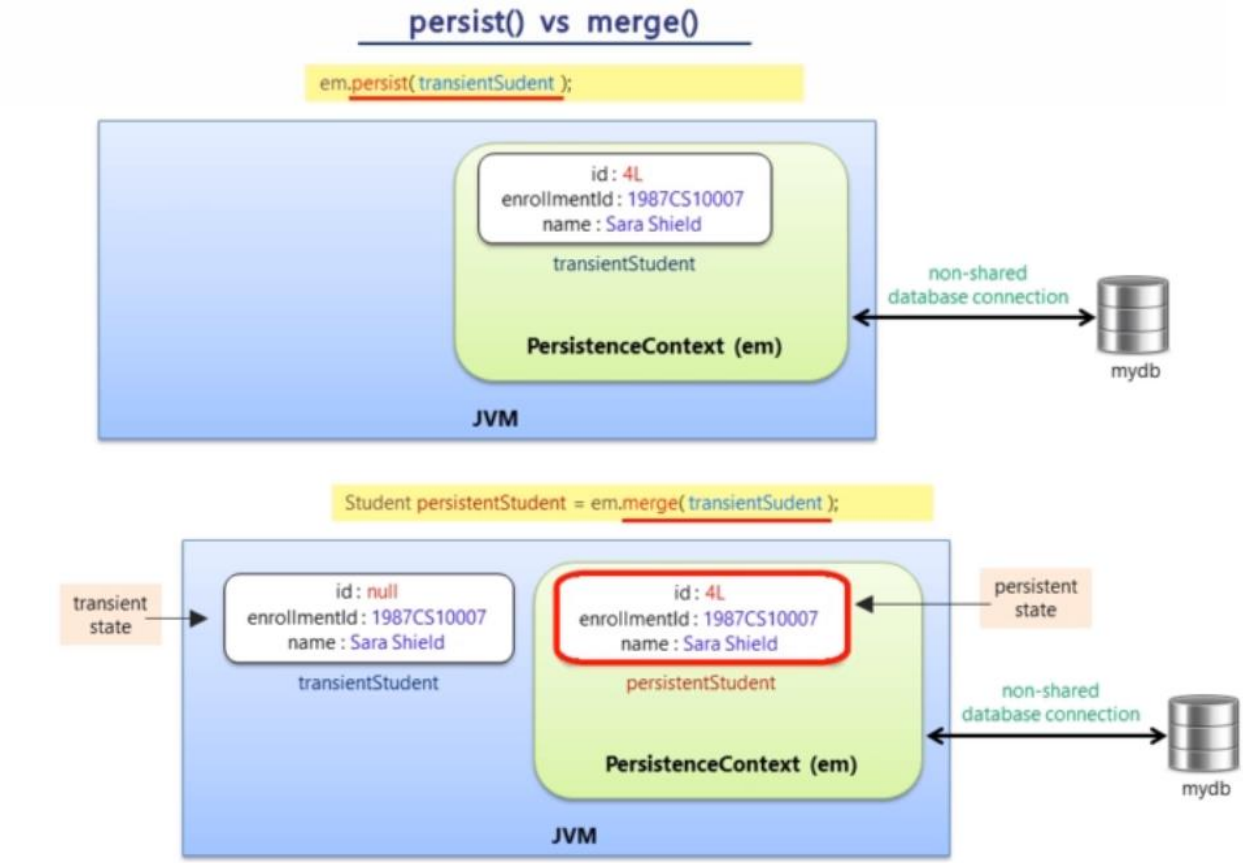

Making Data Transient
student
ID | enrollment_id | name
1 | 2003CH50897 | Kevin Smith
2 | 2008EE10999 | Sherry Morgan
3 | 1987CS10007 | Sara Shield
package client;
import javax.persistence.*;
import entity.Student;
public class MakingDataTransientClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.find(Student.class, 3L);
em.remove(student);
Long id = student.getId(); //3
em.persist(student);
deletion of Student[id=3] will be cancelled
em.getTransaction().commit();
em.close();
}
}
persistent state
id = 3L
student
PersistenceContext (em)
JVM
mydb
mavenedoe.com


Removed vs Transient
removed state
id : 3
enrollmentId : 1987CS10007
name : Sara Shield
student
PersistenceContext (em)
JVM
non-shared database connection
mydb
transient state
id : null
enrollmentId : 1987CS10007
name : Sara Shield
student
PersistenceContext (em)
JVM
non-shared database connection
mydb


Making Data Transient
student
ID
enrollment_id
name
1
2003CH50897
Kevin Smith
2
2008EE10999
Sherry Morgan
3
1987CS10007
Sara Shield
package client;
import javax.persistence.*;
import entity.Student;
public class MakingDataTransientClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.find(Student.class, 3L);
em.getTransaction().commit();
em.close();
}
}
<persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
<properties>
<!-- Database connection, dialect and other settings -->
<!-- Reset identifier value (from Long to null) after removal of an entity instance -->
<property name="hibernate.use_identifier_rollback" value="true" />
</properties>
</persistence-unit>
persistence.xml
id = 3L
student
PersistenceContext (em)
JVM
mydb
mavenedge.com


:lass, 3L);
<persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
<properties>
<!-- Database connection, dialect and other settings -->
<!-- Reset identifier value (from Long to null) after removal of an entity instance -->
<property name="hibernate.use_identifier_rollback" value="true" />
</properties>
</persistence-unit>
persistence.xml
transient state
id = null
student
PersistenceContext (em)
mydb
JVM
mavenedge.com


Making Data Transient
student
ID
enrollment_id
name
1 2003CH50897 Kevin Smith
2 2008EE10999 Sherry Morgan
3 1987CS10007 Sara Shield
4 1987CS10007 Sara Shield
SQL at runtime
delete from Student where id=?
binding parameter [1] as [BIGINT] - [3]
package client;
import javax.persistence.*;
import entity.Student;
public class MakingDataTransientClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = em.find(Student.class, 3L);
em.remove(student);
Long id = student.getId(); //null
//em.persist(student); //Exception
em.getTransaction().commit();
em.close();
EntityManager em2 = emf.createEntityManager();
em2.getTransaction().begin();
Student student2 = em2.merge(student);
em2.getTransaction().commit();
em2.close();
}
}
does not return void type
<persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
<properties>
<!-- Database connection, dialect and other settings -->
<!-- Reset identifier value (from Long to null) after removal of an entity instance -->
<property name="hibernate.use_identifier_rollback" value="true" />
</properties>
</persistence-unit>
persistence.xml
id = 4L
student
PersistenceContext (em2)
mydb
JVM
mavenedge.com

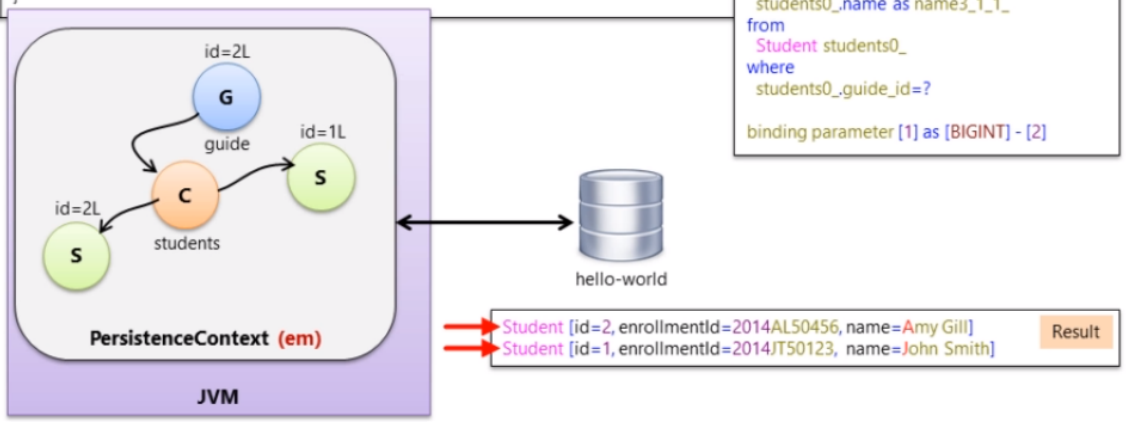


▶ *Order by*

```
..  public class OrderByClient {
7       public static void main(String[] args) {
8
9           EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
10
11          EntityManager em = emf.createEntityManager();
12          em.getTransaction().begin();
13
14          Guide guide = em.find(Guide.class, 2L);
15          Set<Student> students = guide.getStudents();
16
17          for (Student student : students) {
18              System.out.println(student);
19          }
20
21          em.getTransaction().commit();
22          em.close();
23      }
24  }
```

SQL at runtime

```
select
  guide0_.id as id1_0_0_,
  guide0_.name as name2_0_0_,
  guide0_.salary as salary3_0_0_,
  guide0_.staff_id as staff_id4_0_0_
from
  Guide guide0_
where
  guide0_.id=?

binding parameter [1] as [BIGINT] - [2]

select
  students0_.guide_id as guide_id4_0_0_,
  students0_.id as id1_1_0_,
  students0_.id as id1_1_1_,
  students0_.enrollment_id as enrollme2_1_1_,
  students0_.guide_id as guide_id4_1_1_,
  students0_.name as name3_1_1_
from
  Student students0_
where
  students0_.guide_id=?

binding parameter [1] as [BIGINT] - [2]
```



```java
@Entity
public class Student {
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="enrollment_id", nullable=false)
  private String enrollmentId;

  private String name;

  @ManyToOne(fetch=FetchType.LAZY)
  @JoinColumn(name="guide_id")
  private Guide guide;

  //constructors, set/get, toString() and other methods
}
```

fetch=FetchType.LAZY

```java
@Entity
public class Guide {
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @Column(name="staff_id", nullable=false)
  private String staffId;

  private String name;
  private Integer salary;

  @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
  @OrderBy("name DESC")
  private Set<Student> students = new HashSet<Student>();

  public void addStudent(Student student) {
    students.add(student);
    student.setGuide(this);
  }

  //constructors, set/get, toString() and other methods
}
```

fetch=FetchType.LAZY

student

id	enrollment_id	name	guide_id
1	2014AL50456	Amy Gill	2
2	2014JT50123	John Smith	2

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901

```java
..
7   public class OrderByClient {
8       public static void main(String[] args) {
9
10          EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
11
12          EntityManager em = emf.createEntityManager();
13          em.getTransaction().begin();
14
15          Guide guide = em.find(Guide.class, 2L);
16          Set<Student> students = guide.getStudents();
17
18          for (Student student : students) {
19              System.out.println(student);
20          }
21
22          em.getTransaction().commit();
23          em.close();
24      }
    }
```

w/ @OrderBy("name DESC")

SQL at runtime

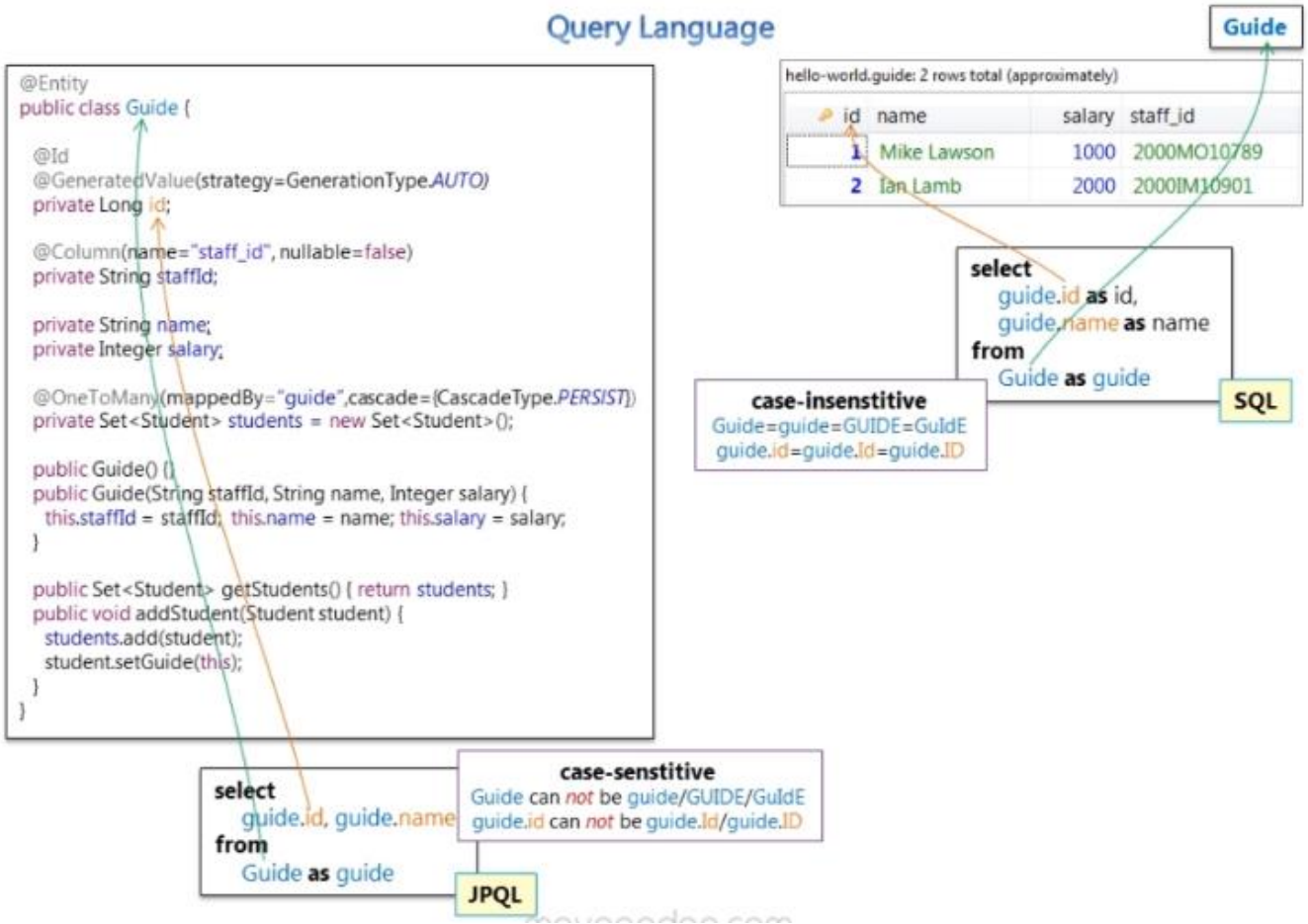
```
select
  guide0_.id as id1_0_0_,
  guide0_.name as name2_0_0_,
  guide0_.salary as salary3_0_0_,
  guide0_.staff_id as staff_id4_0_0_
from
  Guide guide0_
where
  guide0_.id=?

binding parameter [1] as [BIGINT] - [2]

select
  students0_.guide_id as guide_id4_0_0_,
  students0_.id as id1_1_0_,
  students0_.id as id1_1_1_,
  students0_.enrollment_id as enrollme2_1_1_,
  students0_.guide_id as guide_id4_1_1_,
  students0_.name as name3_1_1_
from
  Student students0_
where
  students0_.guide_id=?
order by
  students0_.name desc

binding parameter [1] as [BIGINT] - [2]
```

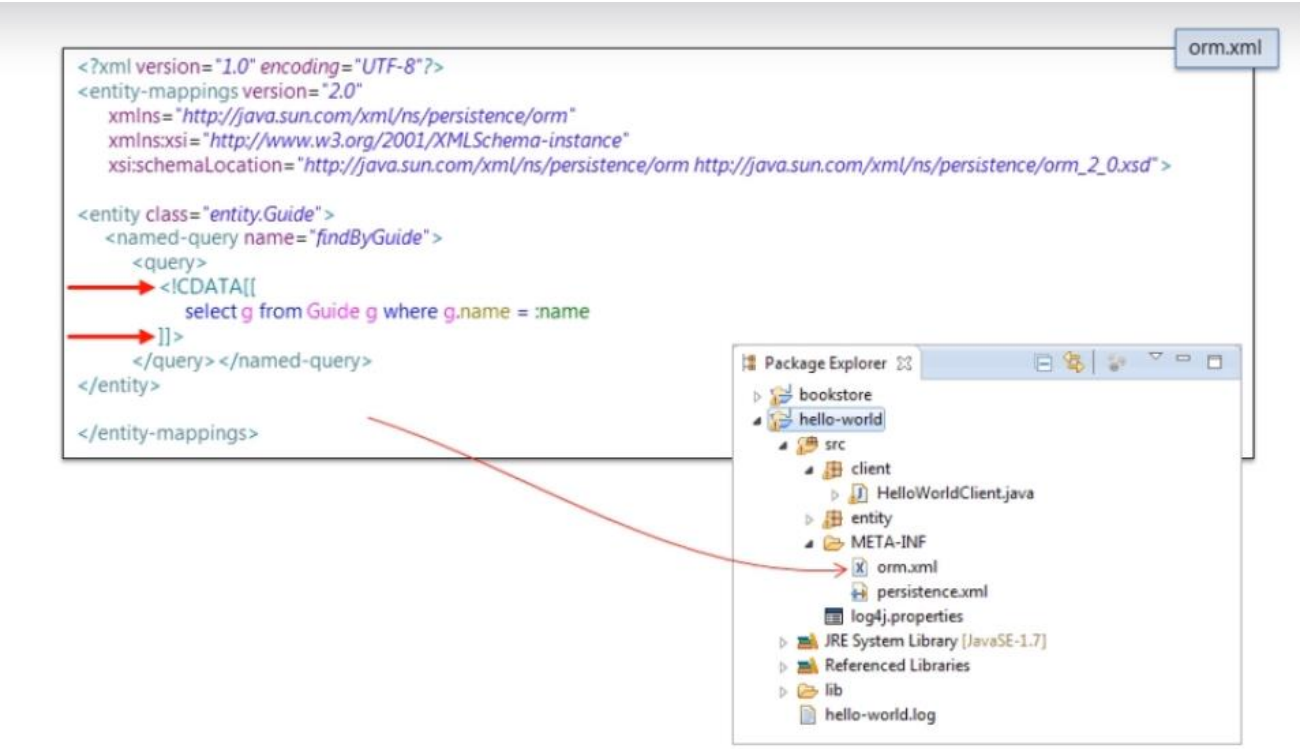
Result

```
Student [id=1, enrollmentId=2014JT50123, name=John Smith]
Student [id=2, enrollmentId=2014AL50456, name=Amy Gill]
```

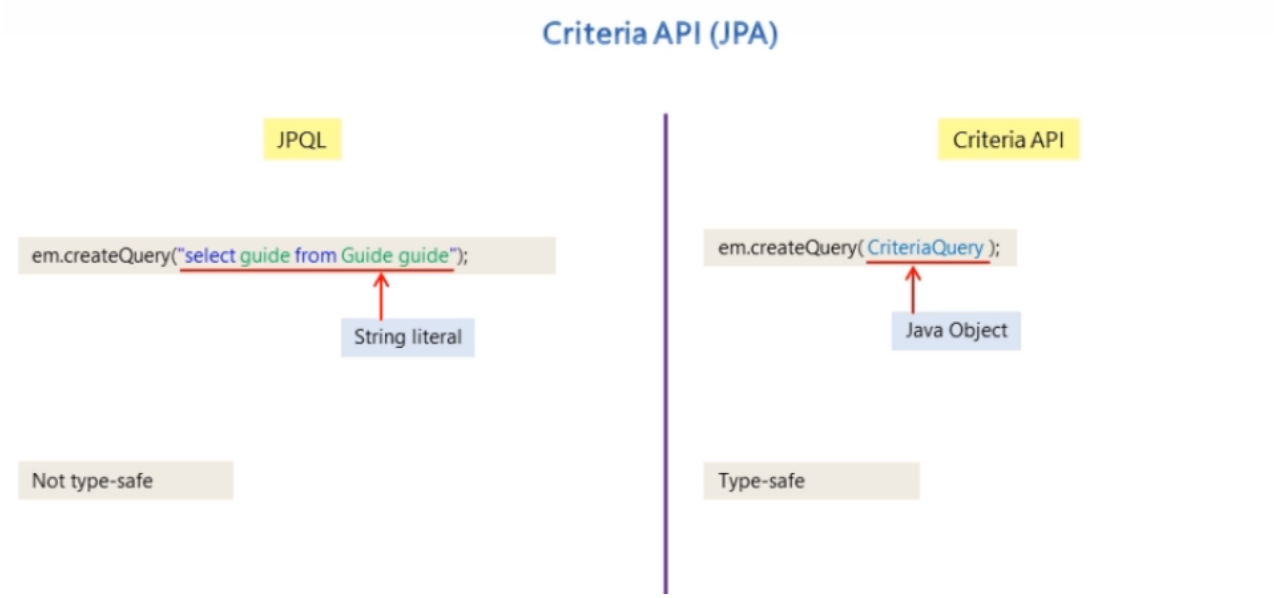
```
@OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
@OrderBy("name DESC, enrollmentId")
private Set<Student> students = new HashSet<Student>();
```

- ***Query language***





▶ *Criteria API*



Criteria API (JPA)

1. Programmatic query-construction

```
select g from Guide g join fetch g.students student
```

syntactically correct

2. Type-safety

```
TypedQuery<T> typedQuery = em.createQuery( CriteriaQuery );
List<T> result = typedQuery.getResultList()
```

result type T

CriteriaQuery<T> is generically typed

```
TypedQuery<String> typedQuery = em.createQuery("select g.name from Guide g", String.class);
List<String> names = typedQuery.getResultList();
```

```
Guide_.name
```

metamodel of Guide entity

public and static

Querying Entities

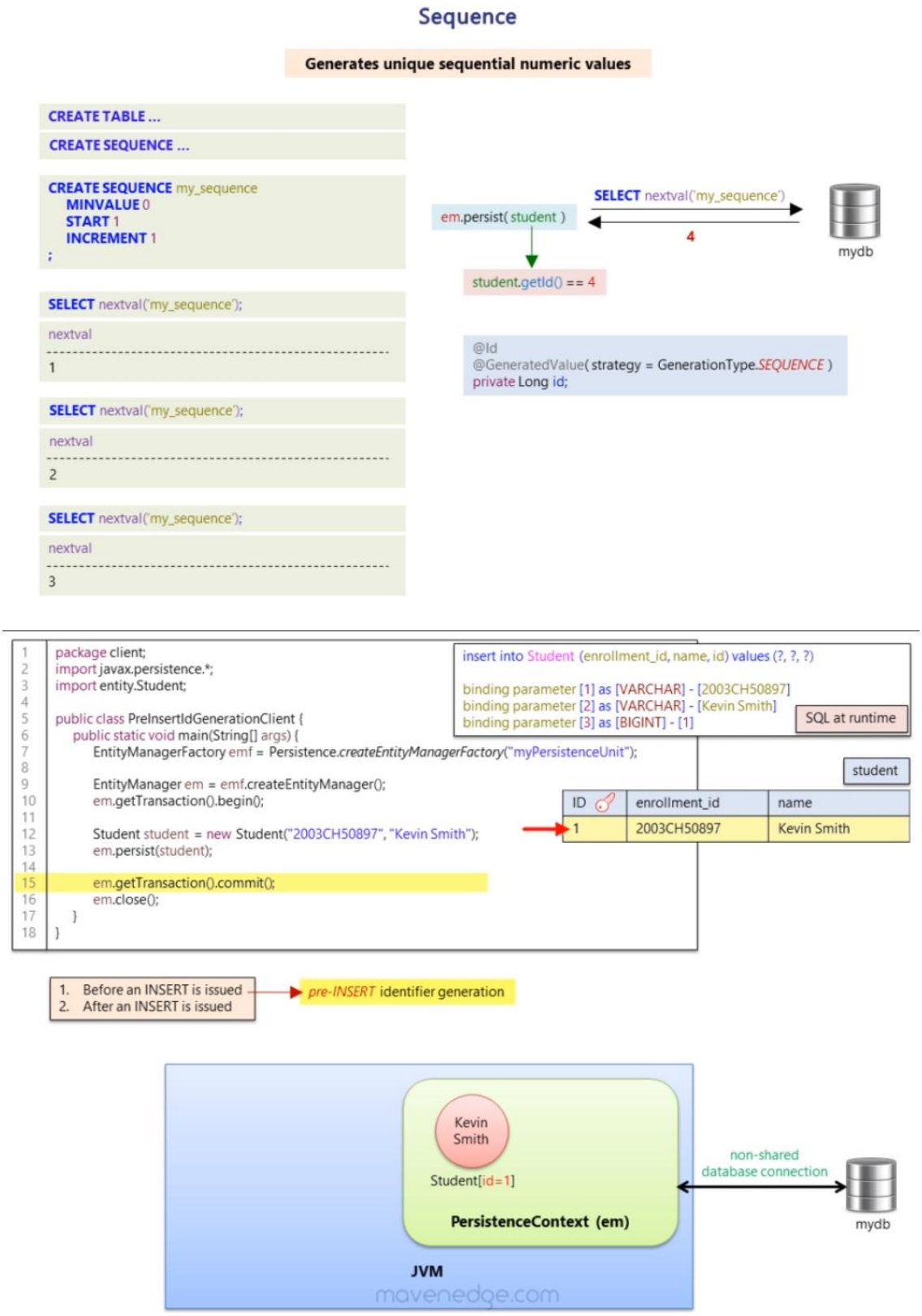
select guide from Guide guide

```
package client;

import java.util.List;
import javax.persistence.*;
import entity.Guide;

public class HelloWorldClient {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        CriteriaBuilder builder = em.getCriteriaBuilder();

        CriteriaQuery<Guide> criteria = builder.createQuery(Guide.class);
        Root<Guide> root = criteria.from( Guide.class );
        criteria.select(root);

        TypedQuery<Guide> query = em.createQuery(criteria);

        List<Guide> guides = query.getResultList();
        for (Guide guide : guides) {
            System.out.println(guide);
        }

        em.getTransaction().commit();
        em.close();
    }
}
```

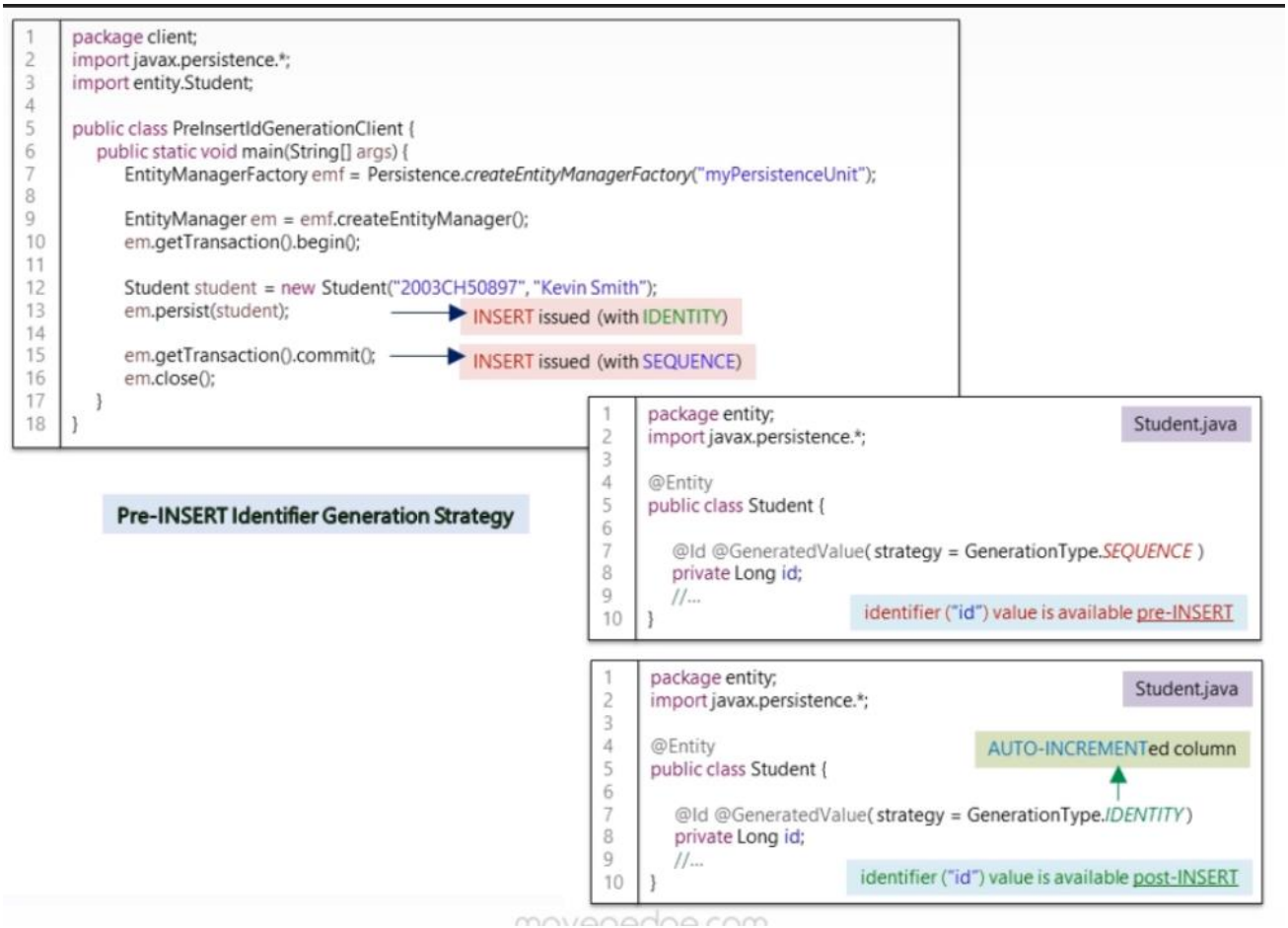
- ***Pre-insert identifier generation***

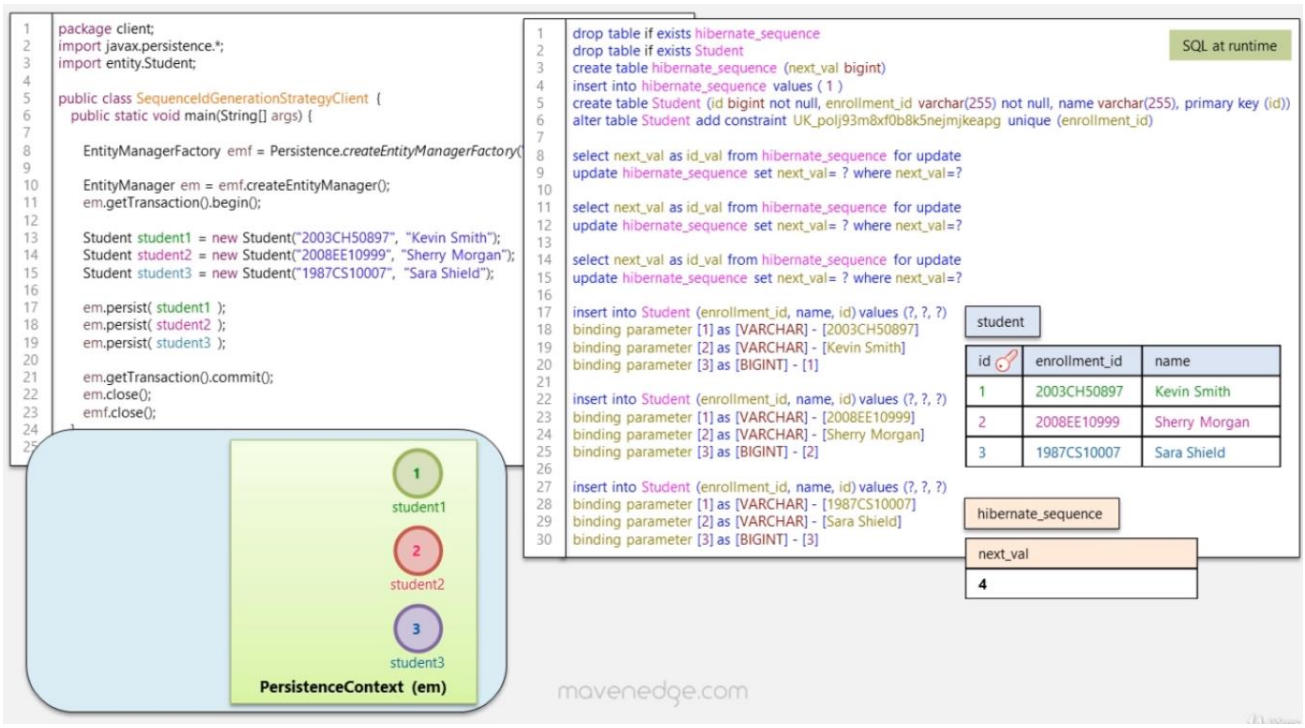
Pre-INSERT Identifier Generation

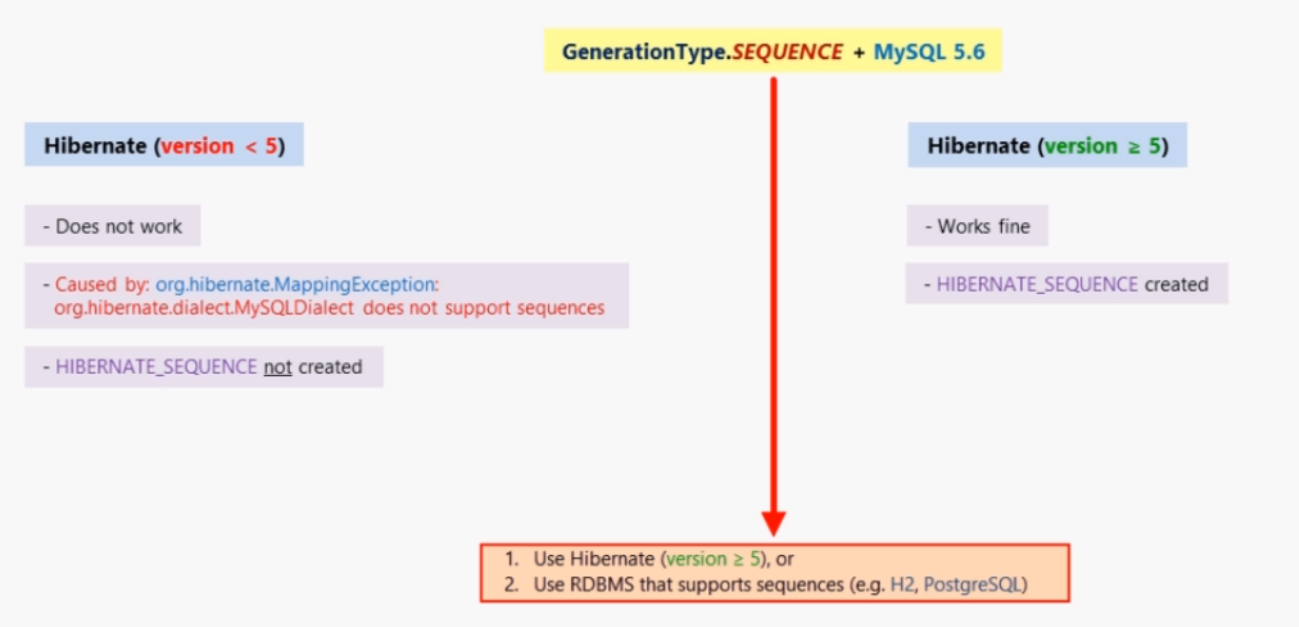
```
1  package entity;
2  import javax.persistence.*;
3
4  @Entity
5  public class Student {
6
7      @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
8      private Long id;
9
10     @Column(nullable = false)
11     private String name;
12
13     @Column(name="enrollment_id", nullable = false, unique=true)
14     private String enrollmentId;
15
16     public Student() {}
17     public Student(String enrollmentId, String name) {
18         this.enrollmentId = enrollmentId;
19         this.name = name;
20     }
21
22     //set and get methods
23
24     @Override
25     public String toString() {
26         return "Student [id=" + id + ", enrollmentId=" + enrollmentId + ", name=" + name + "]";
27     }
28 }
```

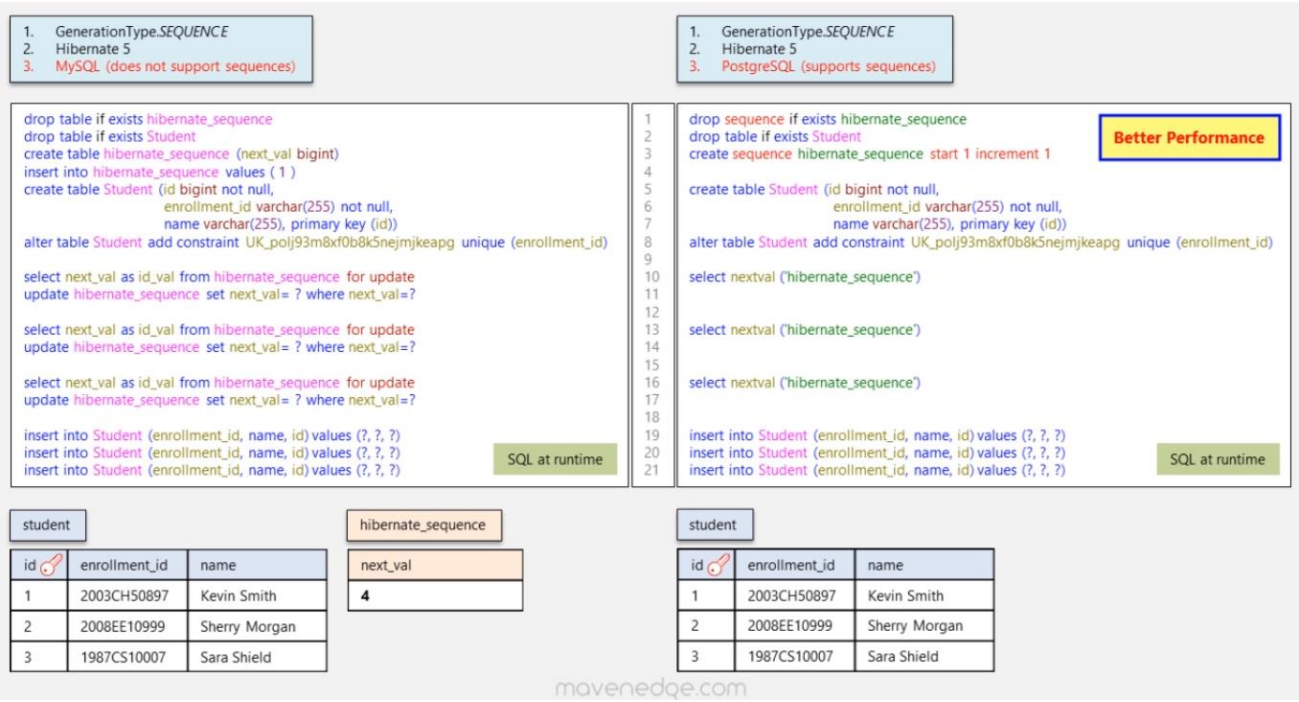


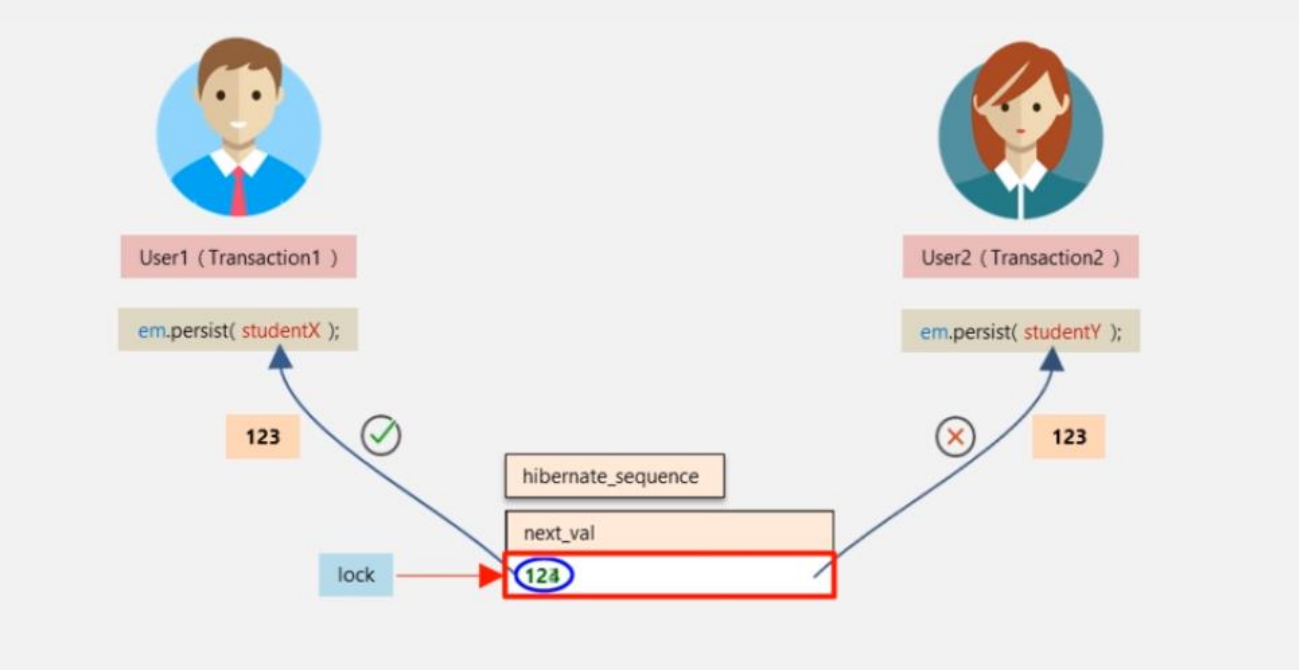
19. New ID issued in persist function, before INSERT query is created


package client;
import javax.persistence.*;
import entity.Student;
public class PreInsertIdGenerationClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student = new Student("2003CH50897", "Kevin Smith");
em.persist(student);
INSERT issued (with IDENTITY)
em.getTransaction().commit();
INSERT issued (with SEQUENCE)
em.close();
}
}
Pre-INSERT Identifier Generation Strategy
Student.java
package entity;
import javax.persistence.*;
@Entity
public class Student {
@Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
private Long id;
//...
}
identifier ("id") value is available pre-INSERT
Student.java
package entity;
import javax.persistence.*;
@Entity
public class Student {
AUTO-INCREMENTed column
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
//...
}
identifier ("id") value is available post-INSERT

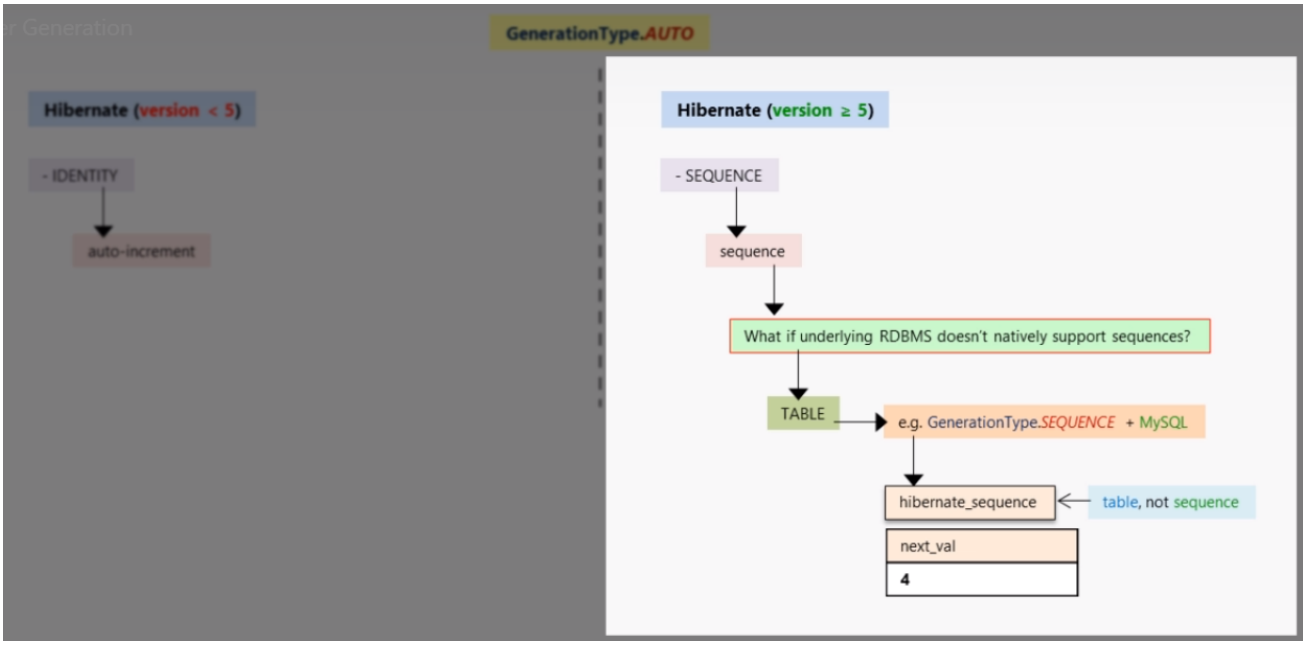

package client;
import javax.persistence.*;
import entity.Student;
public class SequenceIdGenerationStrategyClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Student student1 = new Student("2003CH50897", "Kevin Smith");
Student student2 = new Student("2008EE10999", "Sherry Morgan");
Student student3 = new Student("1987CS10007", "Sara Shield");
em.persist(student1);
em.persist(student2);
em.persist(student3);
em.getTransaction().commit();
em.close();
emf.close();
1
student1
2
student2
3
student3
PersistenceContext (em)
SQL at runtime
drop table if exists hibernate_sequence
drop table if exists Student
create table hibernate_sequence (next_val bigint)
insert into hibernate_sequence values (1)
create table Student (id bigint not null, enrollment_id varchar(255) not null, name varchar(255), primary key (id))
alter table Student add constraint UK_polj93m8xf0b8k5nejmjkeapg unique (enrollment_id)
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
insert into Student (enrollment_id, name, id) values (?, ?, ?)
binding parameter [1] as [VARCHAR] - [2003CH50897]
binding parameter [2] as [VARCHAR] - [Kevin Smith]
binding parameter [3] as [BIGINT] - [1]
insert into Student (enrollment_id, name, id) values (?, ?, ?)
binding parameter [1] as [VARCHAR] - [2008EE10999]
binding parameter [2] as [VARCHAR] - [Sherry Morgan]
binding parameter [3] as [BIGINT] - [2]
insert into Student (enrollment_id, name, id) values (?, ?, ?)
binding parameter [1] as [VARCHAR] - [1987CS10007]
binding parameter [2] as [VARCHAR] - [Sara Shield]
binding parameter [3] as [BIGINT] - [3]
student
id
enrollment_id
name
1
2003CH50897
Kevin Smith
2
2008EE10999
Sherry Morgan
3
1987CS10007
Sara Shield
hibernate_sequence
next_val
4


GenerationType.SEQUENCE + MySQL 5.6
Hibernate (version < 5)
- Does not work
- Caused by: org.hibernate.MappingException:
org.hibernate.dialect.MySQLDialect does not support sequences
- HIBERNATE_SEQUENCE not created
Hibernate (version ≥ 5)
- Works fine
- HIBERNATE_SEQUENCE created
1. Use Hibernate (version ≥ 5), or
2. Use RDBMS that supports sequences (e.g. H2, PostgreSQL)

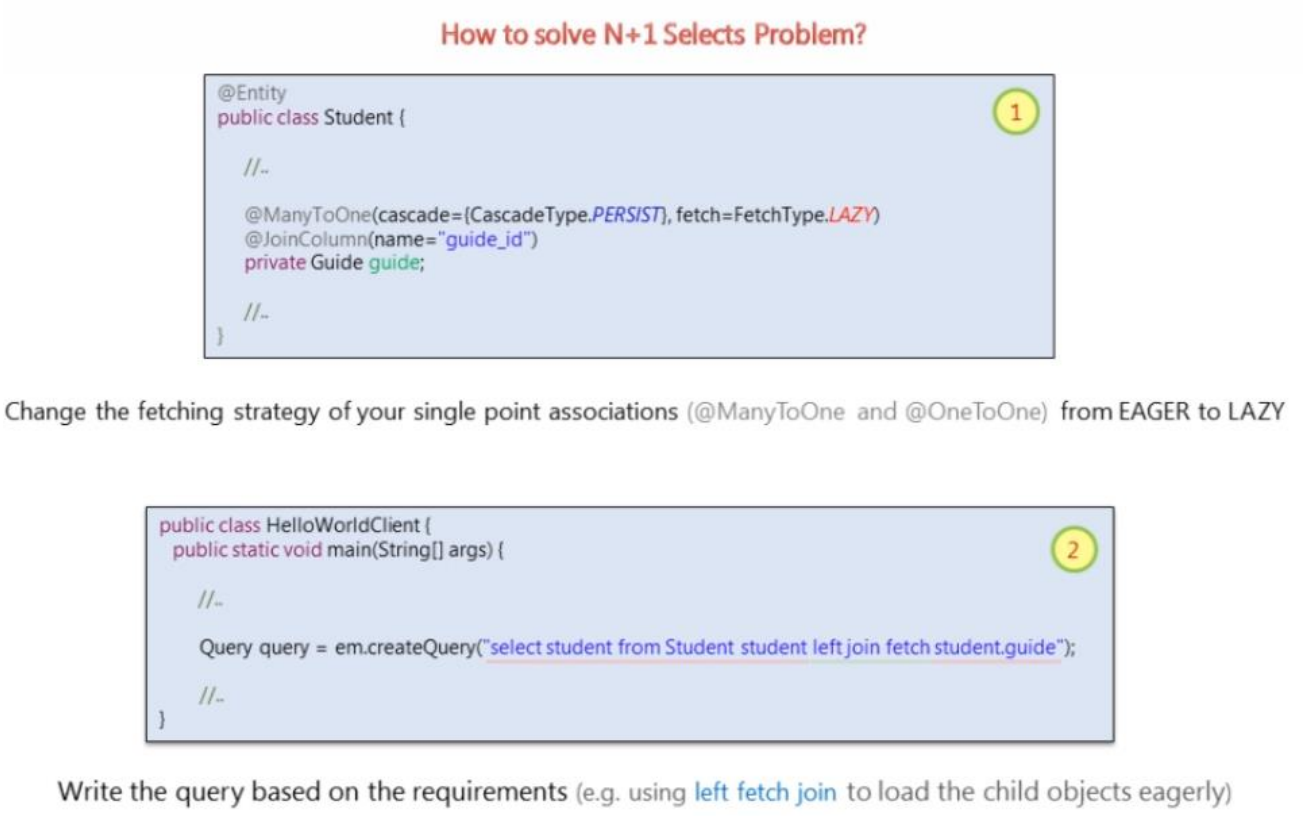

1. GenerationType.SEQUENCE
2. Hibernate 5
3. MySQL (does not support sequences)
drop table if exists hibernate_sequence
drop table if exists Student
create table hibernate_sequence (next_val bigint)
insert into hibernate_sequence values (1)
create table Student (id bigint not null,
enrollment_id varchar(255) not null,
name varchar(255), primary key (id))
alter table Student add constraint UK_polj93m8xf0b8k5nejmjkeapg unique (enrollment_id)
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
select next_val as id_val from hibernate_sequence for update
update hibernate_sequence set next_val= ? where next_val=?
insert into Student (enrollment_id, name, id) values (?, ?, ?)
insert into Student (enrollment_id, name, id) values (?, ?, ?)
insert into Student (enrollment_id, name, id) values (?, ?, ?)
SQL at runtime
student
id
enrollment_id
name
1
2003CH50897
Kevin Smith
2
2008EE10999
Sherry Morgan
3
1987CS10007
Sara Shield
hibernate_sequence
next_val
4
1. GenerationType.SEQUENCE
2. Hibernate 5
3. PostgreSQL (supports sequences)
drop sequence if exists hibernate_sequence
drop table if exists Student
create sequence hibernate_sequence start 1 increment 1
create table Student (id bigint not null,
enrollment_id varchar(255) not null,
name varchar(255), primary key (id))
alter table Student add constraint UK_polj93m8xf0b8k5nejmjkeapg unique (enrollment_id)
select nextval ('hibernate_sequence')
select nextval ('hibernate_sequence')
select nextval ('hibernate_sequence')
insert into Student (enrollment_id, name, id) values (?, ?, ?)
insert into Student (enrollment_id, name, id) values (?, ?, ?)
insert into Student (enrollment_id, name, id) values (?, ?, ?)
Better Performance
SQL at runtime
student
id
enrollment_id
name
1
2003CH50897
Kevin Smith
2
2008EE10999
Sherry Morgan
3
1987CS10007
Sara Shield
mavenedge.com



20. TABLE strategy always creates a table for id increment



- ***N + 1 select problems***



Lab Exercise: N+1 Selects Problem

The required jar files for the *entire course* could be downloaded from Lecture 10

Task 0: Download the source-code, run it, make yourself comfortable with the concepts covered in it

Question 1: Considering the following Guide-To-Student is a bi-directional One-To-Many relationship and the GUIDE and STUDENT tables, choose the correct option(s) from the options given below.

```
@Entity
public class Student {
    //...
    @ManyToOne(cascade={CascadeType.PERSIST})
    @JoinColumn(name="guide_id")
    private Guide guide;
    //...
}
```

```
@Entity
public class Guide {
    //...
    @OneToMany(mappedBy = "guide", cascade = {CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();
    //...
}
```

student

id	enrollment_id	name	guide_id
1	2014AL50456	Amy Gill	2
2	2014JT50123	John Smith	2
3	2014BE50789	Bruce Lee	NULL
4	2014RG50347	Rahul Singh	3

guide

id	name	salary	staff_id
1	Mike Lawson	1000	2000MO10789
2	Ian Lamb	2000	2000IM10901
3	David Crow	3000	2000DO10777

(A) "select student from Student student" will result in N+1 Selects Problem in the given mapping
(B) N+1 Selects Problem in the given mapping could be avoided by using FetchType.*LAZY* in @ManyToOne mapping in Student entity
(C) N+1 Selects Problem in the given mapping could be avoided by using JOIN FETCH whenever associated Guide's data also needs to be accessed along with Student data
(D) Using FetchType.*EAGER* is bad for JPA associations as it fetches more data than needed

► *Batch fetching*

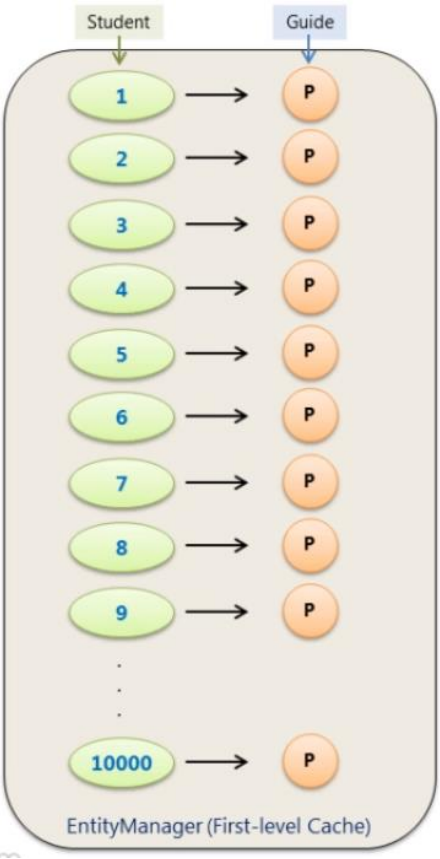
SQL at runtime

```
select
    student0_.id as id1_1_,
    student0_.enrollment_id as enrollme2_1_,
    student0_.guide_id as guide_id4_1_,
    student0_.name as name3_1_
from
    Student student0_
```

```
    //...
}
```

```
@Entity
public class Guide {
    //..

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST},
                        fetch=FetchType.LAZY)
    private Set<Student> students = new HashSet<Student>();

    //...
}
```

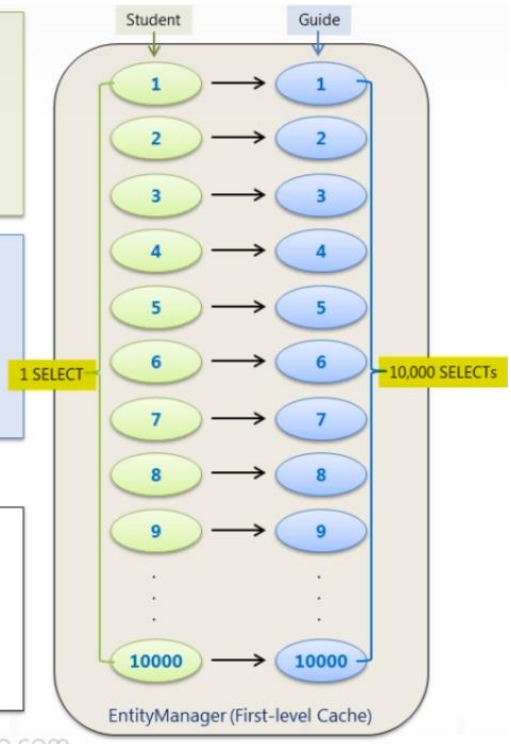
```
Query query = em.createQuery("select student from Student student");

List<Student> students = query.getResultList();

for (Student student : students) {
    System.out.println(student.getName() + ": " + student.getEnrollmentId());
}
```



```java
@Entity
public class Student {
    //..

    @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //...
}
```

```java
@Entity
public class Guide {
    //..

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST},
                    fetch=FetchType.LAZY)
    private Set<Student> students = new HashSet<Student>();

    //...
}
```

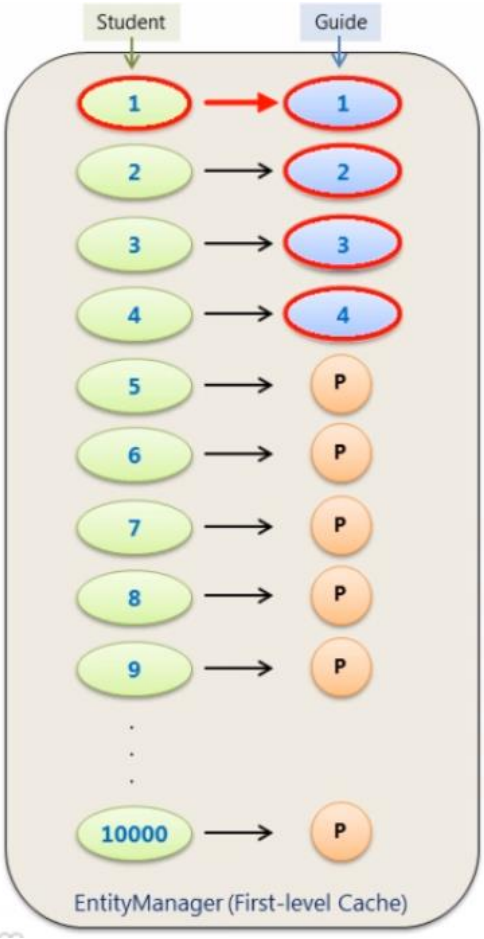
```java
Query query = em.createQuery("select student from Student student");

List<Student> students = query.getResultList();

for (Student student : students) {
    if(student.getGuide() != null) {
        System.out.println(student.getName() + ": " + student.getEnrollmentId() +
          ": " + student.getGuide().getName());
    }
}
```




```java
@Entity
public class Student {
    //..

    @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //...
}
```

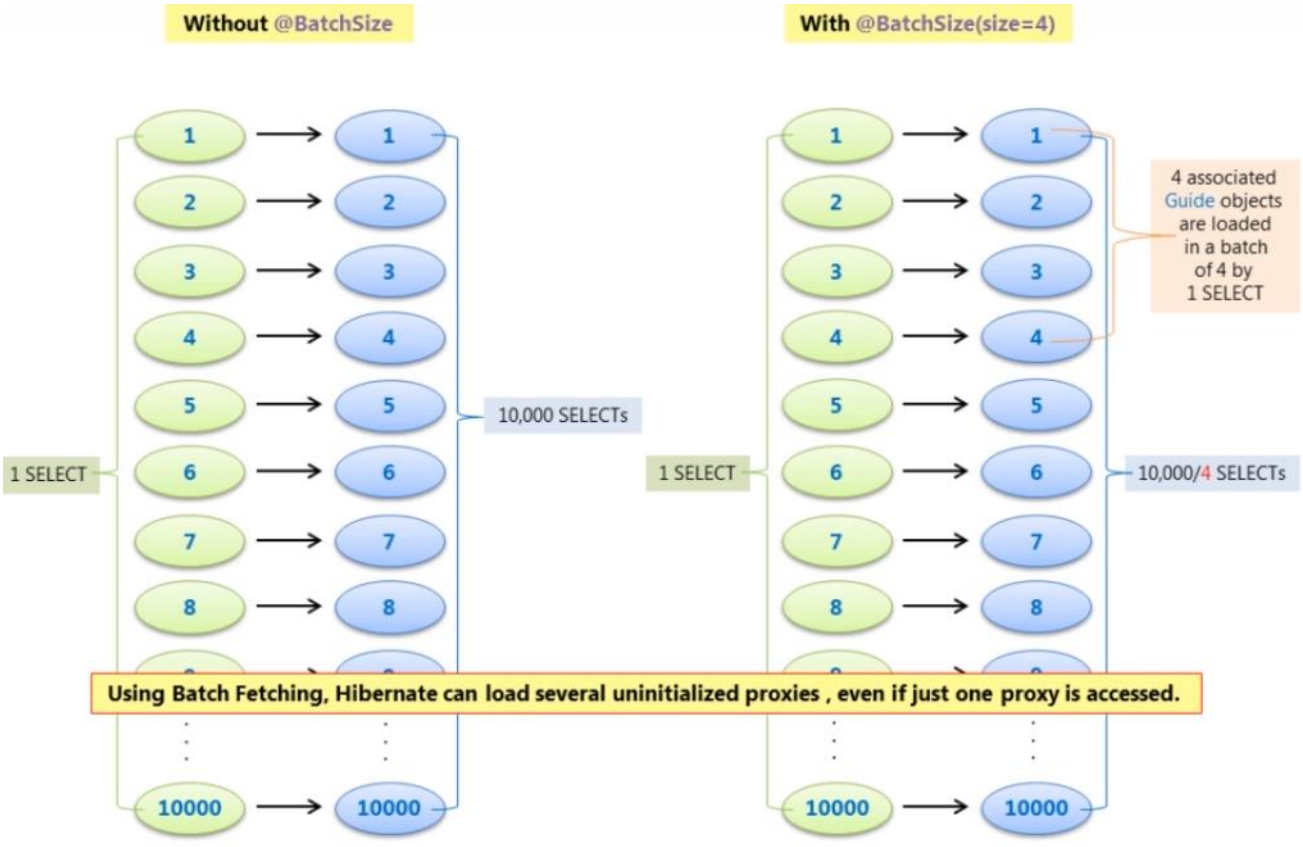
```java
//...
import org.hibernate.annotations.BatchSize;
@Entity
@BatchSize(size=4)
public class Guide {
    //..

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST},
                    fetch=FetchType.LAZY)
    private Set<Student> students = new HashSet<Student>();

    //...
}
```

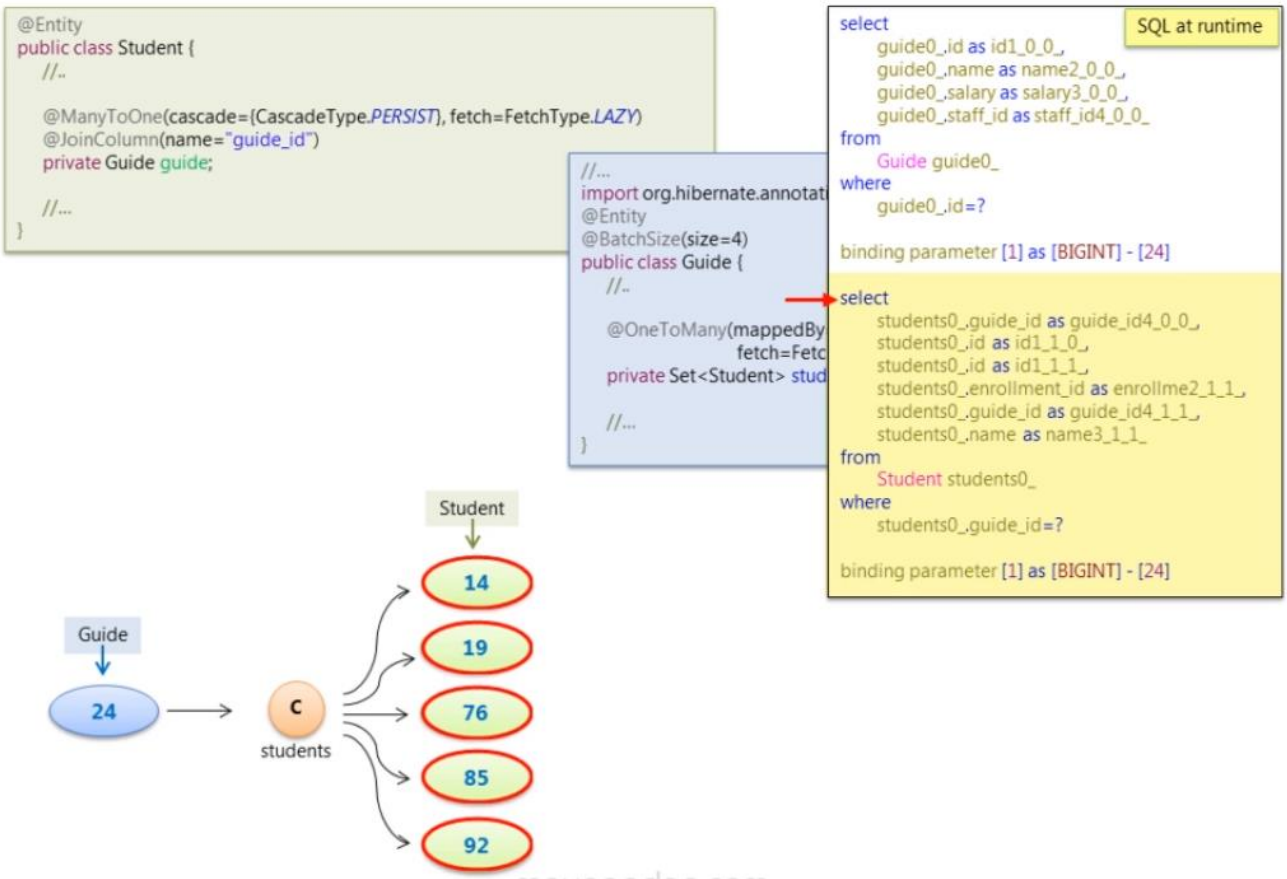
```java
Query query = em.createQuery("select student from Student student");

List<Student> students = query.getResultList();

for (Student student : students) {
    if(student.getGuide() != null) {
        System.out.println(student.getName() + ": " + student.getEnrollmentId() +
          ": " + student.getGuide().getName());
    }
}
```



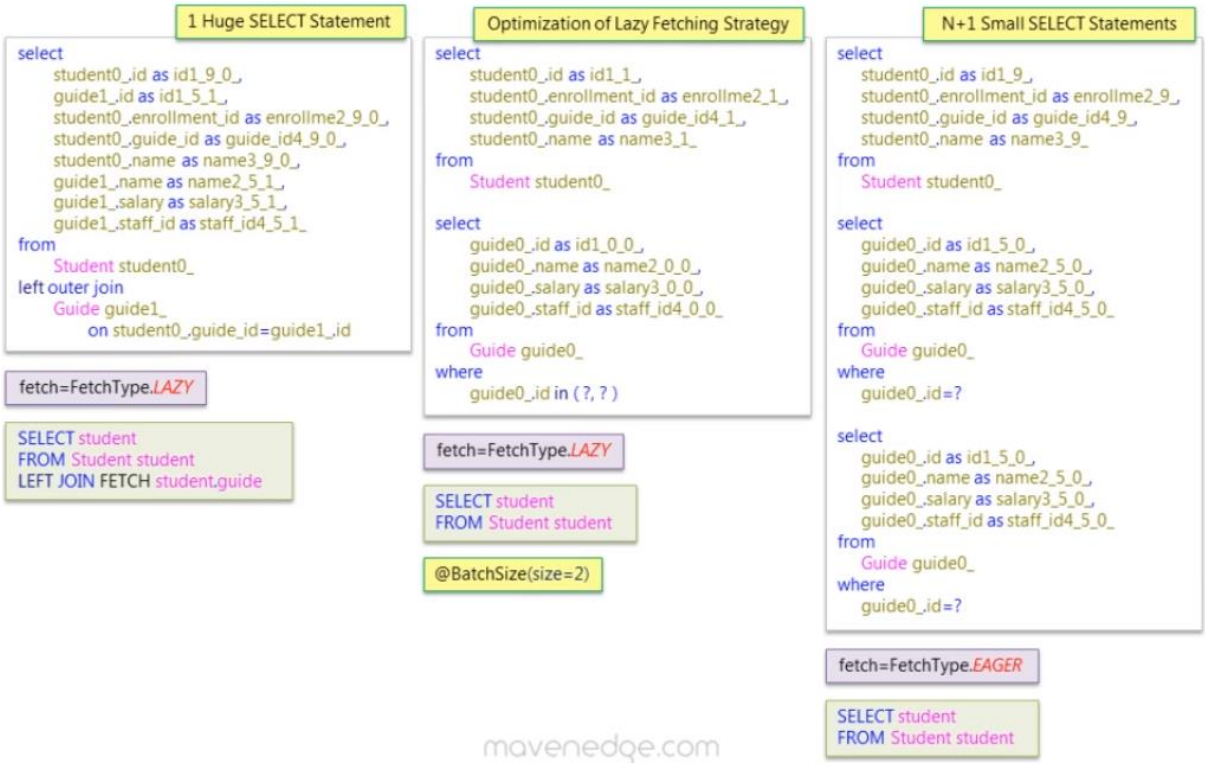
SQL at runtime

```
select
    student0_.id as id1_1_,
    student0_.enrollment_id as enrollme2_1_,
    student0_.guide_id as guide_id4_1_,
    student0_.name as name3_1_
from
    Student student0_

select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
    guide0_.staff_id as staff_id4_0_0_
from
    Guide guide0_
where
    guide0_.id in ( ?, ?, ?, ? )

binding parameter [1] as [BIGINT] - [1]
binding parameter [2] as [BIGINT] - [2]
binding parameter [3] as [BIGINT] - [3]
binding parameter [4] as [BIGINT] - [4]
```

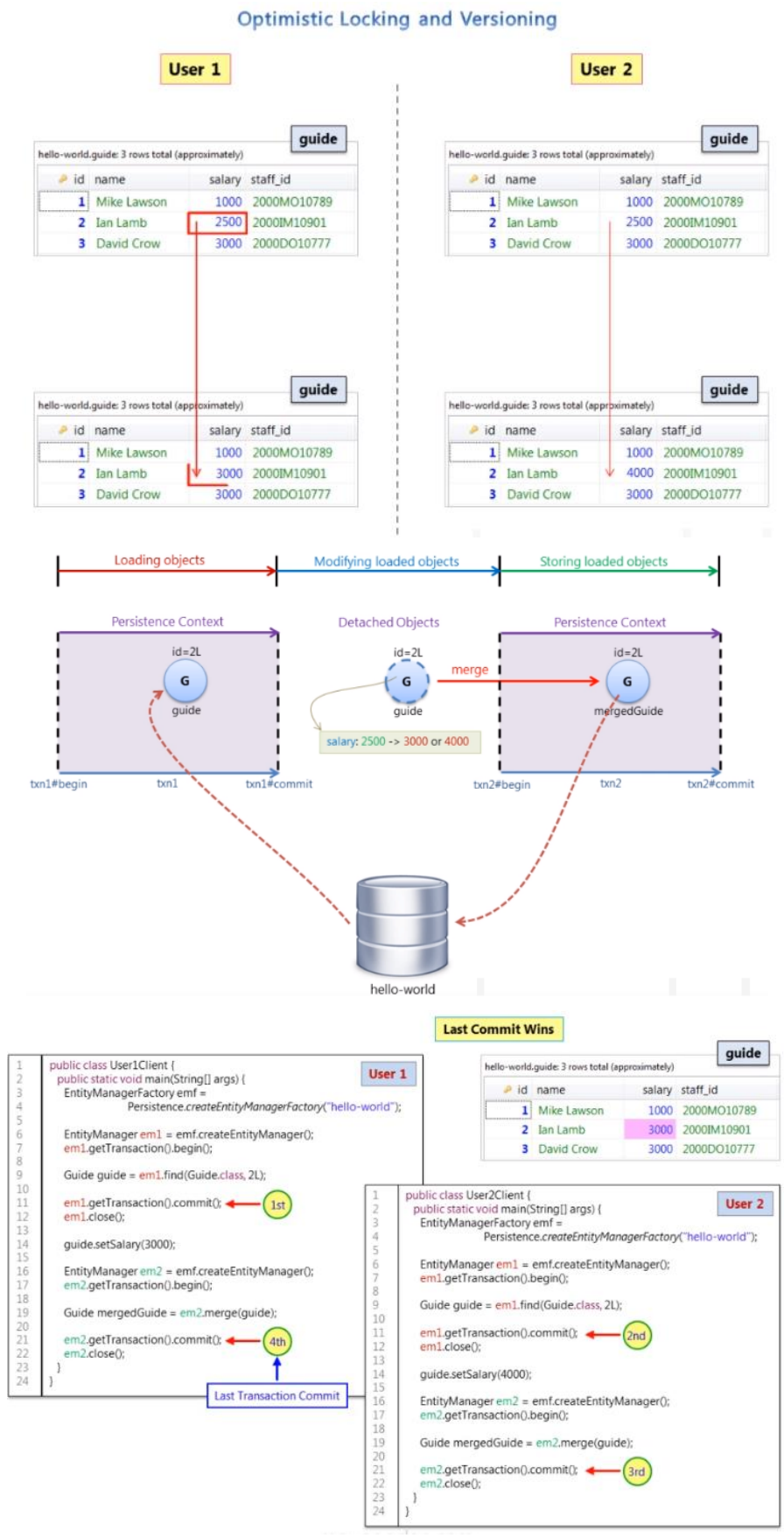




21. @OneToMany can not use batch size as the 'Many' side is fetch it's fetch the whole association in one select query

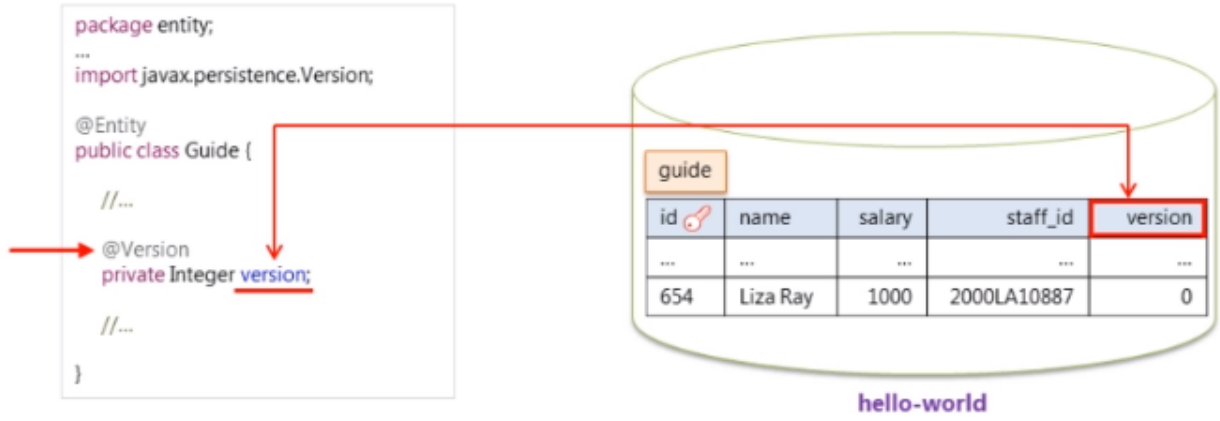


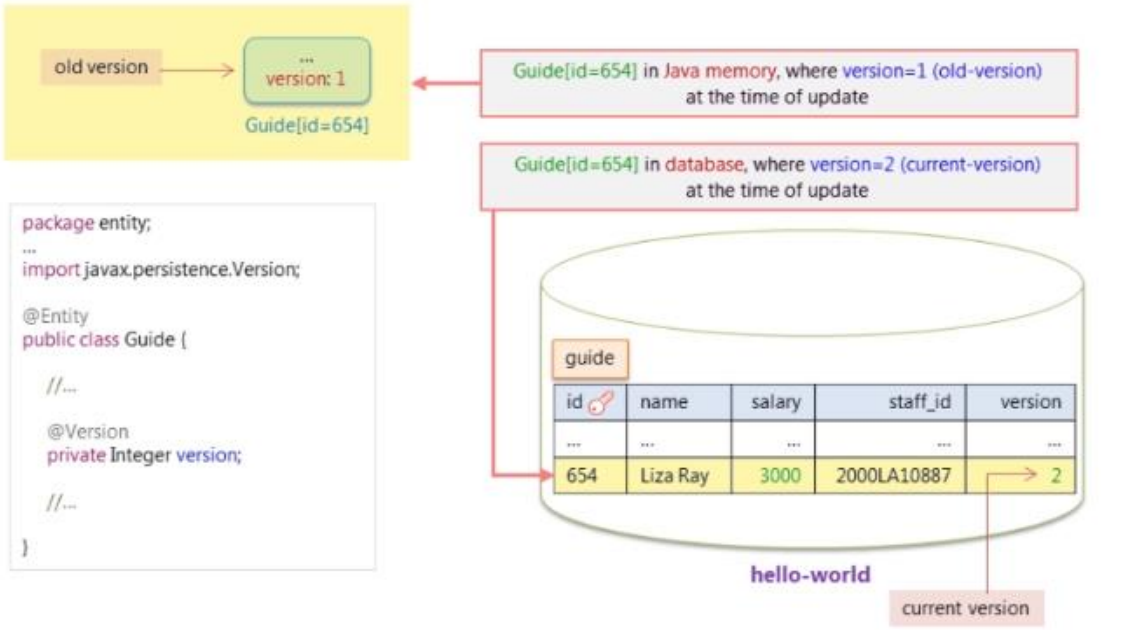
► *Optimistic locking*



Versioning

```
ALTER TABLE `guide`
        ADD `version` INT(11) NOT NULL DEFAULT '0';
```





Hibernate is going to check for the version number at each update

An exception will be thrown, to prevent a lost update, if Hibernate doesn't find the in-memory version of an entity to be same as the database version (current version)

```java
public class User1Client {
  public static void main(String[] args) {
    EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("hello-world");

    EntityManager em1 = emf.createEntityManager();
    em1.getTransaction().begin();

    Guide guide = em1.find(Guide.class, 2L);

    em1.getTransaction().commit();
    em1.close();

    guide.setSalary(3000);

    EntityManager em2 = emf.createEntityManager();
    em2.getTransaction().begin();

    Guide mergedGuide = em2.merge(guide);

    em2.getTransaction().commit();
    em2.close();
  }
}
```

User 1

...
version: 0
salary: 3000
Guide[id=2]

guide

hello-world.guide: 3 rows total (approximately)

id	name	salary	staff_id	version
1	Mike Lawson	1000	2000MO10789	0
2	Ian Lamb	4000	2000IM10901	1
3	David Crow	3000	2000DO10777	0

```
Exception in thread "main" javax.persistence.OptimisticLockException:
                Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect) : [entity.Guide#2]
```

```java
public class User1Client {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");

    EntityManager em1 = emf.createEntityManager();
    em1.getTransaction().begin();

    Guide guide = em1.find(Guide.class, 2L);

    em1.getTransaction().commit();
    em1.close();

    guide.setSalary(3000);

    EntityManager em2 = emf.createEntityManager();
    EntityTransaction txn2 = em2.getTransaction();
    try {
      txn2.begin();

      Guide mergedGuide = em2.merge(guide);

      txn2.commit();
    } catch (OptimisticLockException ole) {
        if(txn2 != null) {
          txn2.rollback();
          System.err.println("The guide was updated by some other user while you were doing interesting things.");
        }
    } finally {
        if(em2 != null) { em2.close(); }
    }
  }
}
```

User 1

Optimistic Locking → No Database Locking

Official name of the Versioning strategy to prevent Lost Updates

Pessimistic Locking → Database Locking

could be used only within a single transaction

The guide was updated by some other user while you were doing interesting things.

```java
public class UserXClient {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    List<Object[]> resultList = em.createQuery("select guide.name, guide.salary from Guide as guide")
                                  .setLockMode(LockModeType.PESSIMISTIC_READ)
                                  .getResultList();

    for (Object[] objects : resultList) {
        System.out.println("Name: " + objects[0] + ", Salary: " + objects[1]);
    }

    long sumOfSalaries = (long) em.createQuery("select sum(guide.salary) from Guide as guide").getSingleResult();
    System.out.println("The total salary of all the guides is " + sumOfSalaries + ".");

    em.getTransaction().commit();
    em.close();
  }
}
```

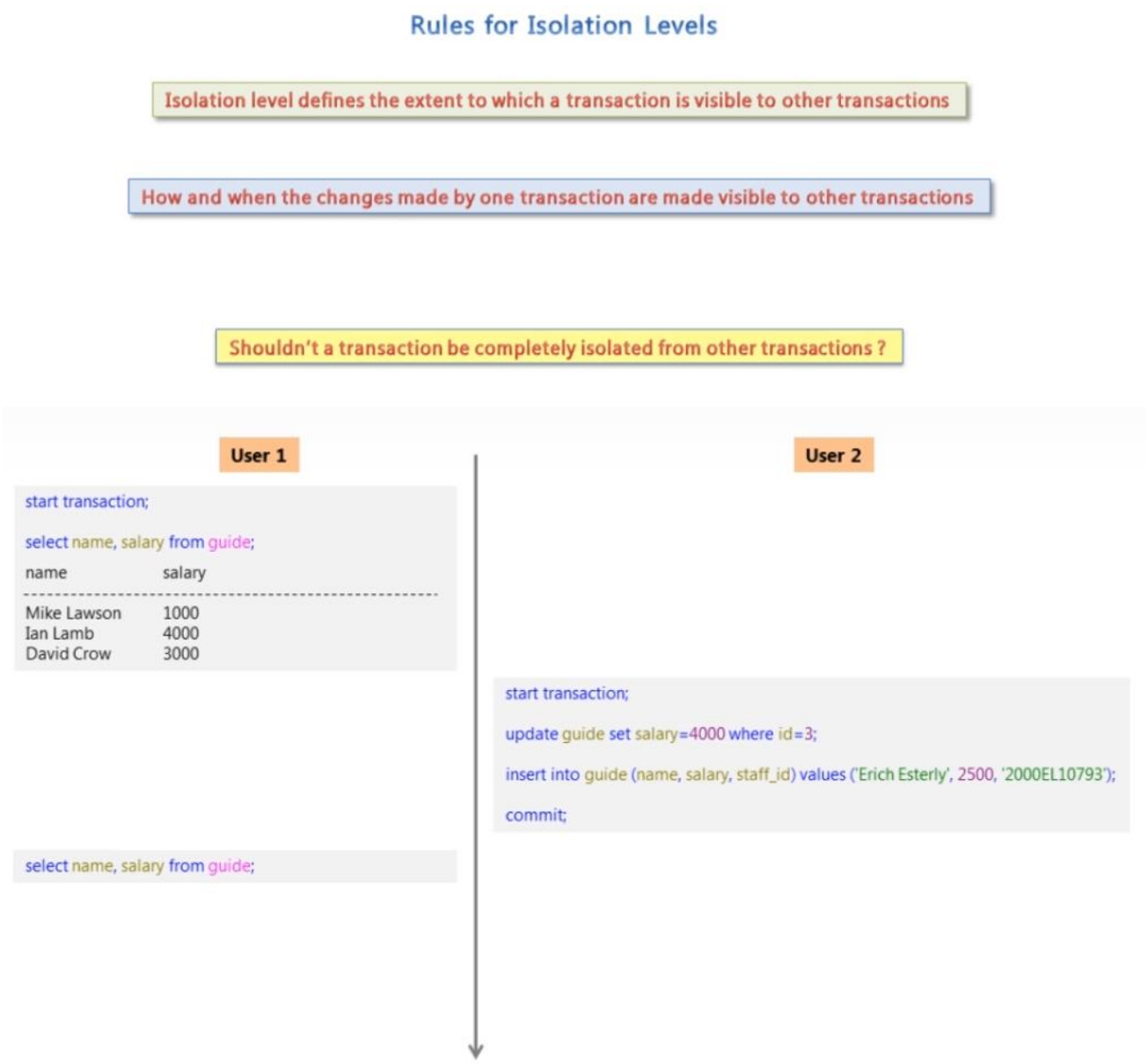
placing a READ lock on the data being read

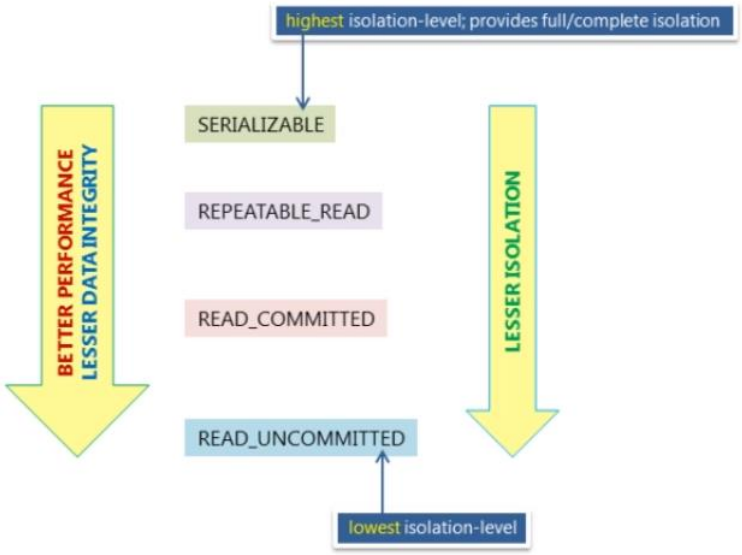
User X

```java
EntityManagerFactory factory = Persistence.createEntityManagerFactory( persistenceUnitName: "clientJPA");

EntityManager firstManager = factory.createEntityManager();
firstManager.getTransaction().begin();
Guide guide = null;
try {
    guide = firstManager.find(Guide.class, ID);
} catch (HibernateException exception){
    System.out.println("[Guide] ID " + ID + " not found" );
    return;
}
firstManager.getTransaction().commit();
firstManager.close();

guide.setSalary(salary);
```

```java
EntityManager secondManager = factory.createEntityManager();
secondManager.getTransaction().begin();

try {
    guide = secondManager.merge(guide);
    secondManager.getTransaction().commit();
} catch (OptimisticLockException exception){
    if(secondManager.getTransaction() != null){
        secondManager.getTransaction().rollback();
        System.out.println("Record have been changed please refresh before start again" );
    }
} finally {
    secondManager.close();
}
System.out.println("Final [Guide] " + guide.toString());
```

- ***Isolation rules***





Isolation Level - SERIALIZABLE

User 1

```
start transaction;

select name, salary from guide;
```

name	salary
Mike Lawson	1000
Ian Lamb	4000
David Crow	3000

```
select name, salary from guide;
```

name	salary
Mike Lawson	1000
Ian Lamb	4000
David Crow	3000

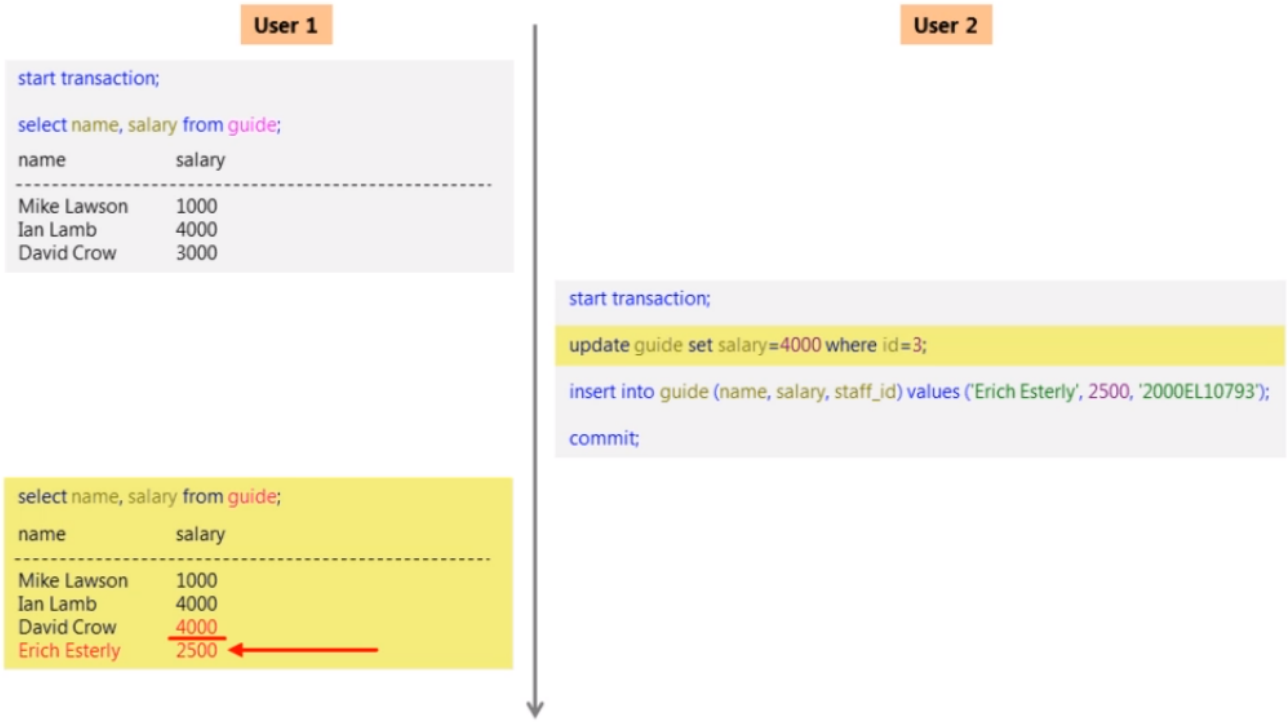
```
commit;
```

User 2

```
start transaction;

update guide set salary=4000 where id=3;

insert into guide (name, salary, staff_id) values ('Erich Esterly', 2500, '2000EL10793');

commit;
```

Isolation Level – REPEATABLE_READ

Isolation Level – READ_COMMITTED



Isolation Level - READ_UNCOMMITTED

User 1

```
start transaction;

select name, salary from guide;
```

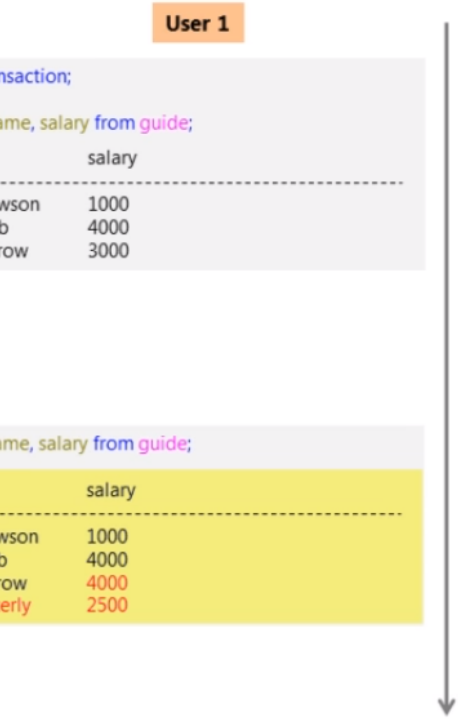
name	salary
Mike Lawson	1000
Ian Lamb	4000
David Crow	3000

User 2

```
start transaction;

update guide set salary=4000 where id=3;

insert into guide (name, salary, staff_id) values ('Erich Esterly', 2500, '2000EL10793');
```

```
select name, salary from guide;
```

name	salary
Mike Lawson	1000
Ian Lamb	4000
David Crow	4000
Erich Esterly	2500



Isolation Level - READ_UNCOMMITTED

Dirty Reads are possible

MySQL

- Supports all 4 isolation-levels
- REPEATABLE READ (default)

Oracle

- Supports SERIALIZABLE and READ COMMITTED
- READ COMMITTED (default)

► *Caching and object identity*

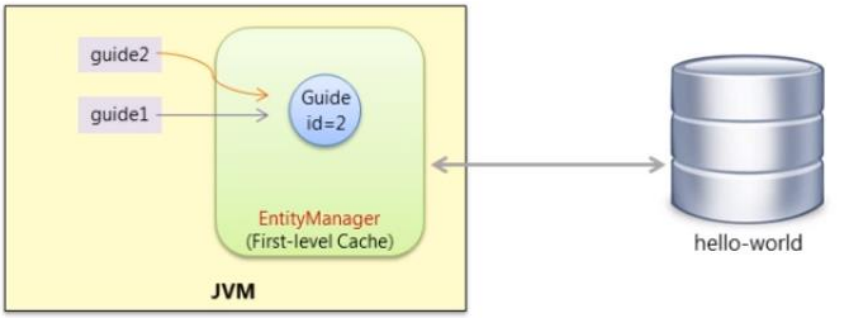
```
1  package client;
2  import javax.persistence.*;
3  import entity.Guide;
4
5  public class HelloWorldClient {
6    public static void main(String[] args) {
7
8      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
9      EntityManager em = emf.createEntityManager();
10     em.getTransaction().begin();
11
12     Guide guide1 = em.find(Guide.class, 2L);
13     Guide guide2 = em.find(Guide.class, 2L);
14     Guide guide3 = (Guide) em.createQuery("select guide from Guide guide where guide.id = :id").setParameter("id", 2L).getSingleResult();
15
16     em.getTransaction().commit();
17     em.close();
18   }
19 }
```

hello-world.guide: 3 rows total (approximately)

id	name	salary	staff_id	version
1	Mike Lawson	1000	2000MO10789	0
2	Ian Lamb	4000	2000IM10901	1
3	David Crow	3000	2000DO10777	0

SQL at runtime

```
select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
    guide0_.staff_id as staff_id4_0_0_
from
    Guide guide0_
where
    guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
```



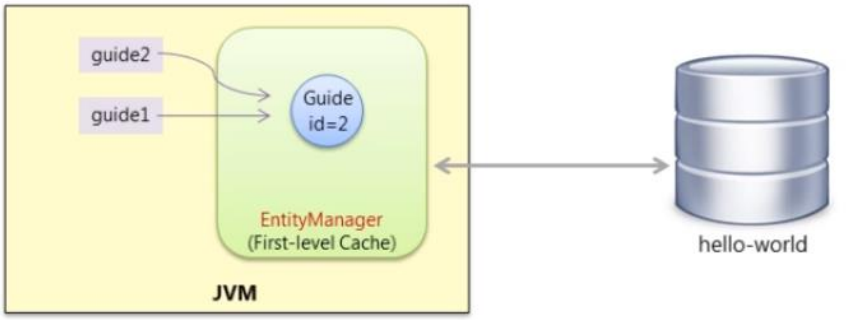

```
1  package client;
2  import javax.persistence.*;
3  import entity.Guide;
4
5  public class HelloWorldClient {
6    public static void main(String[] args) {
7
8      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
9      EntityManager em = emf.createEntityManager();
10     em.getTransaction().begin();
11
12     Guide guide1 = em.find(Guide.class, 2L);
13     Guide guide2 = em.find(Guide.class, 2L);
14     Guide guide3 = (Guide) em.createQuery("select guide from Guide guide where guide.id = :id").setParameter("id", 2L).getSingleResult();
15
16     em.getTransaction().commit();
17     em.close();
18   }
19 }
```

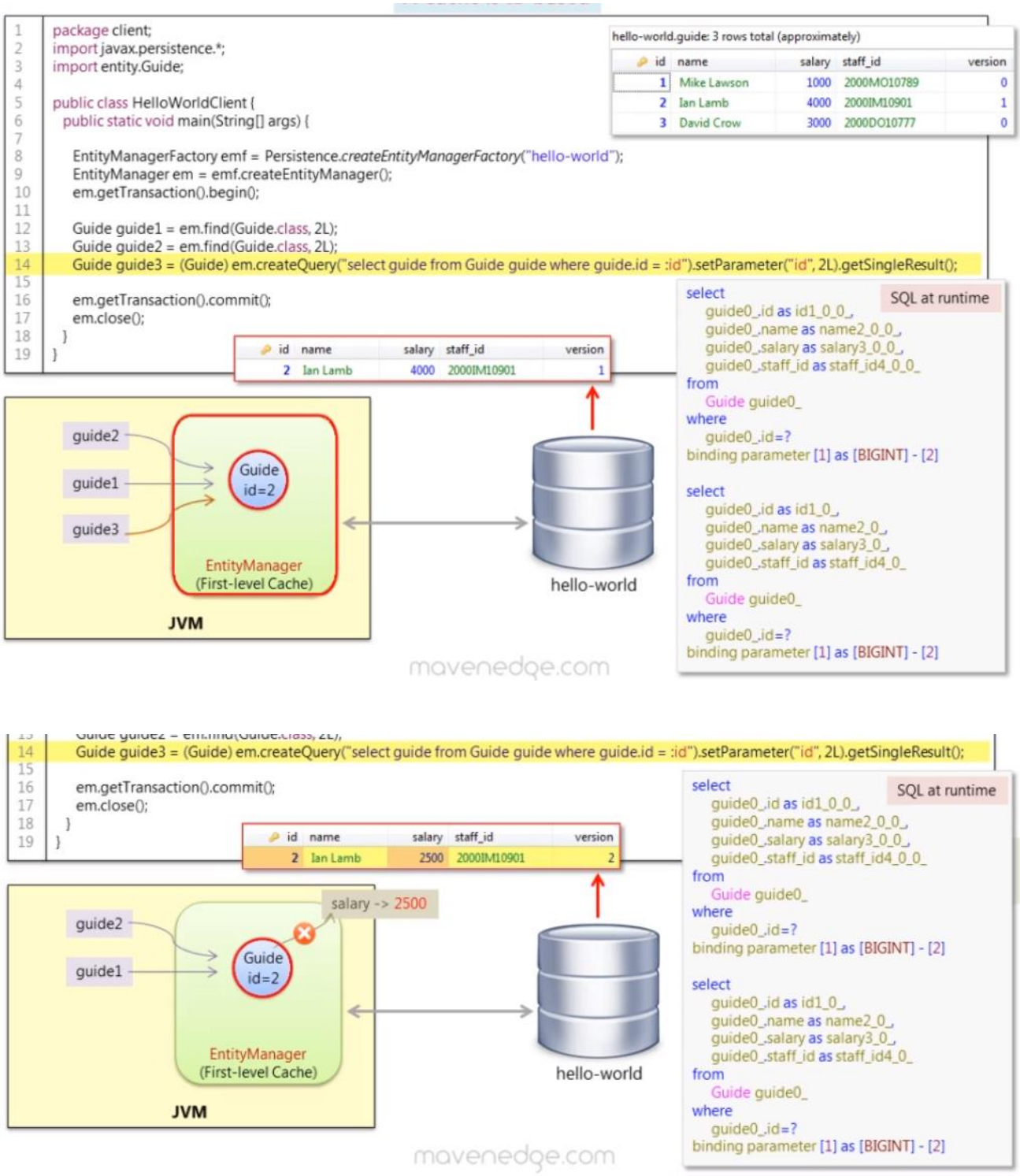
hello-world.guide: 3 rows total (approximately)

id	name	salary	staff_id	version
1	Mike Lawson	1000	2000MO10789	0
2	Ian Lamb	4000	2000IM10901	1
3	David Crow	3000	2000DO10777	0

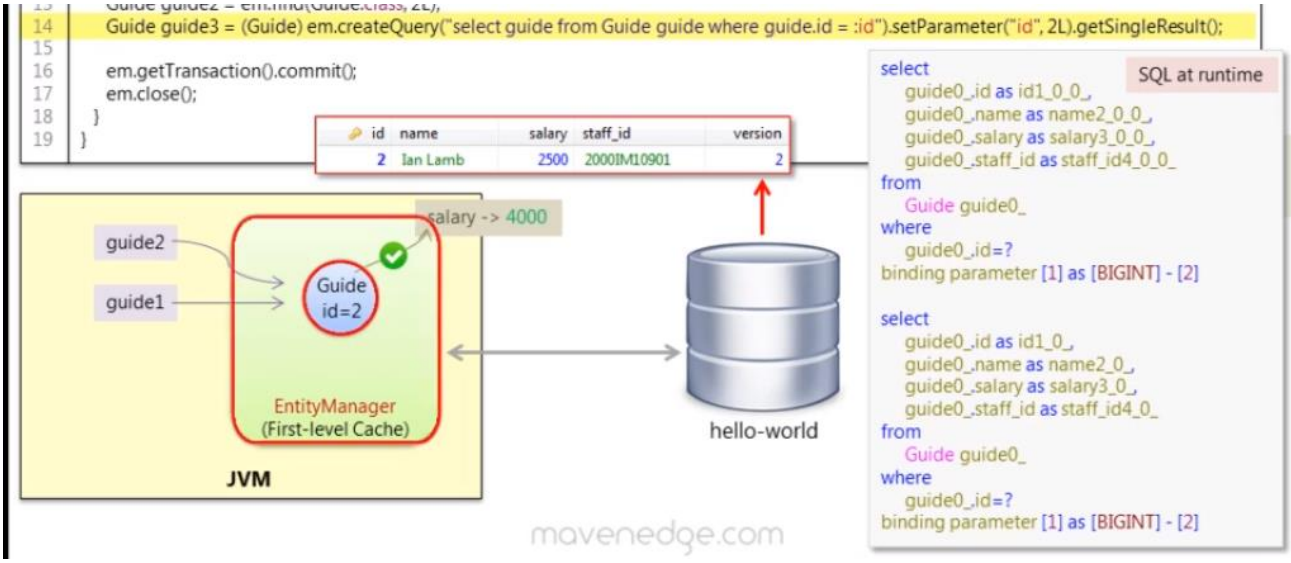
SQL at runtime

```
select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
    guide0_.staff_id as staff_id4_0_0_
from
    Guide guide0_
where
    guide0_.id=?
binding parameter [1] as [BIGINT] - [2]

select
    guide0_.id as id1_0_,
    guide0_.name as name2_0_,
    guide0_.salary as salary3_0_,
    guide0_.staff_id as staff_id4_0_
from
    Guide guide0_
where
    guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
```



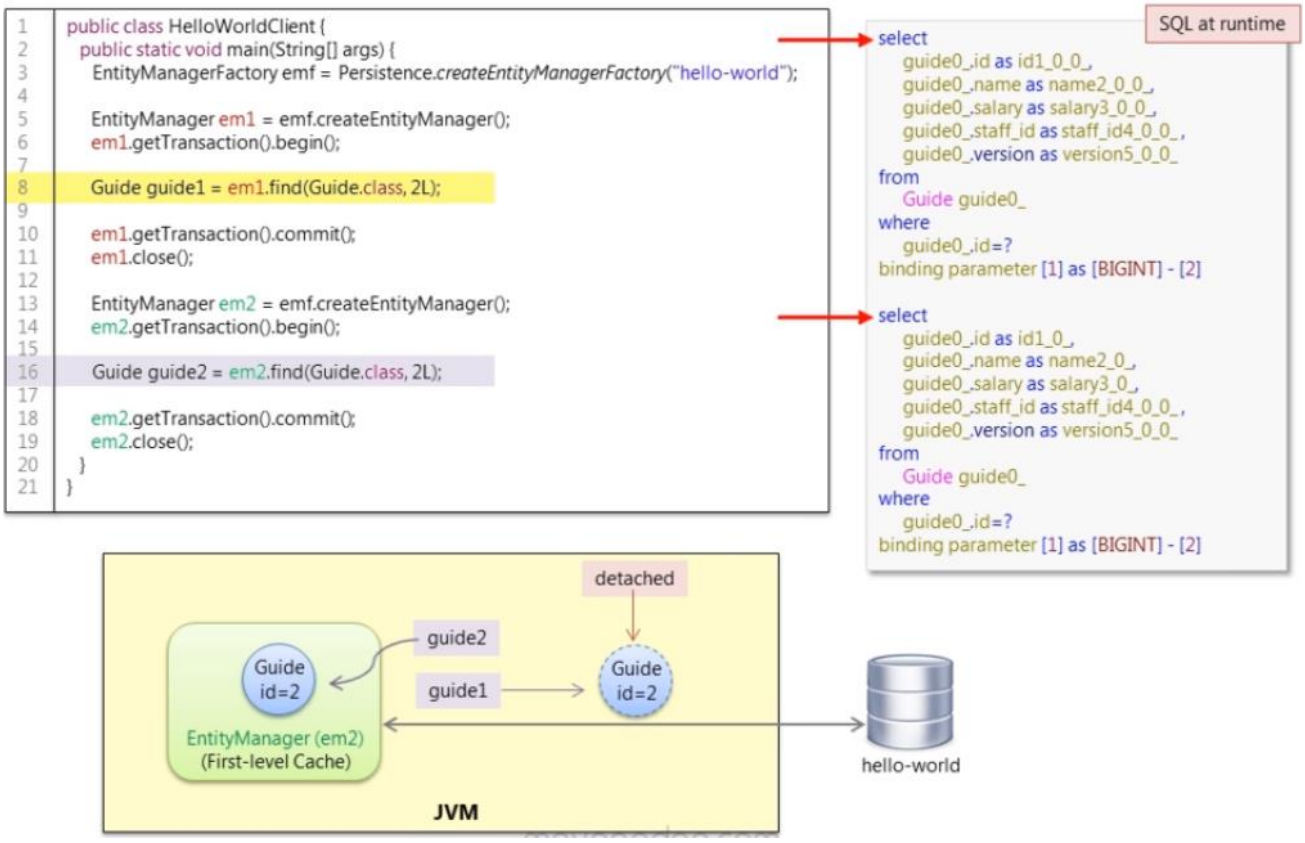

package client;
import javax.persistence.*;
import entity.Guide;
public class HelloWorldClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Guide guide1 = em.find(Guide.class, 2L);
Guide guide2 = em.find(Guide.class, 2L);
Guide guide3 = (Guide) em.createQuery("select guide from Guide guide where guide.id = :id").setParameter("id", 2L).getSingleResult();
em.getTransaction().commit();
em.close();
}
}
hello-world.guide: 3 rows total (approximately)
id name salary staff_id version
1 Mike Lawson 1000 2000MO10789 0
2 Ian Lamb 4000 2000IM10901 1
3 David Crow 3000 2000DO10777 0
id name salary staff_id version
2 Ian Lamb 4000 2000IM10901 1
guide2
guide1
guide3
Guide id=2
EntityManager (First-level Cache)
JVM
hello-world
SQL at runtime
select
guide0_.id as id1_0_0_,
guide0_.name as name2_0_0_,
guide0_.salary as salary3_0_0_,
guide0_.staff_id as staff_id4_0_0_
from
Guide guide0_
where
guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
select
guide0_.id as id1_0_,
guide0_.name as name2_0_,
guide0_.salary as salary3_0_,
guide0_.staff_id as staff_id4_0_
from
Guide guide0_
where
guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
mavenedge.com

Guide guide2 = em.find(Guide.class, 2L);
Guide guide3 = (Guide) em.createQuery("select guide from Guide guide where guide.id = :id").setParameter("id", 2L).getSingleResult();
em.getTransaction().commit();
em.close();
}
}
id name salary staff_id version
2 Ian Lamb 2500 2000IM10901 2
salary -> 2500
guide2
guide1
Guide id=2
EntityManager (First-level Cache)
JVM
hello-world
SQL at runtime
select
guide0_.id as id1_0_0_,
guide0_.name as name2_0_0_,
guide0_.salary as salary3_0_0_,
guide0_.staff_id as staff_id4_0_0_
from
Guide guide0_
where
guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
select
guide0_.id as id1_0_,
guide0_.name as name2_0_,
guide0_.salary as salary3_0_,
guide0_.staff_id as staff_id4_0_
from
Guide guide0_
where
guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
mavenedge.com

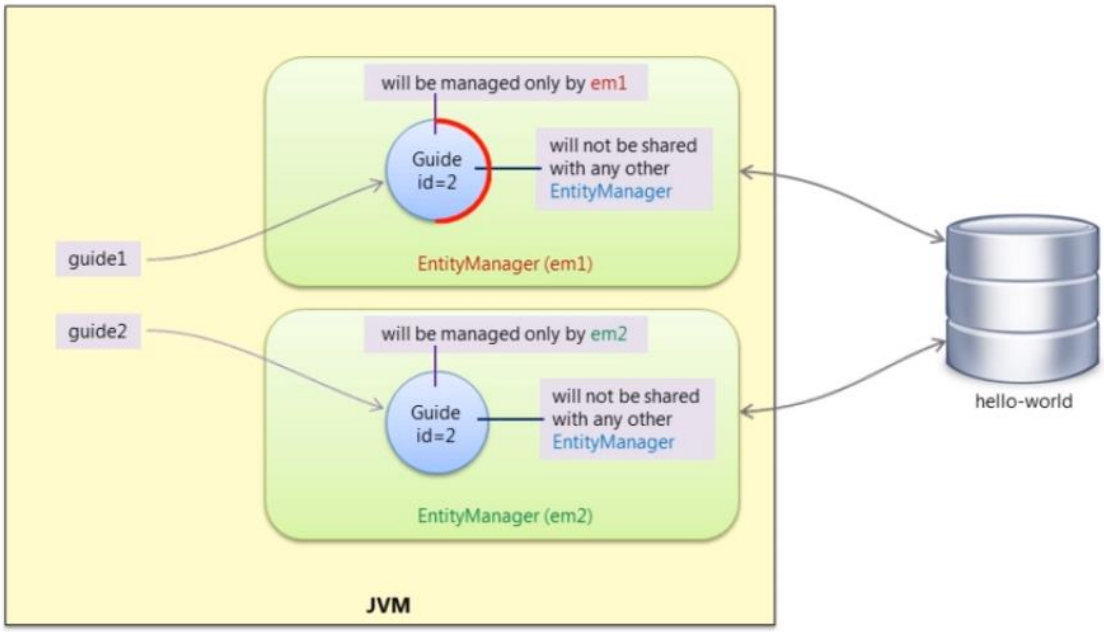


The lecture on "Isolation Rules" covers REPEATABLE_READ and other transaction isolation levels

- ***Second level caching***



22. Object outside entity manager is not cached



By default, Hibernate does not cache the persistent objects across different EntityManagers

First Level Cache — scope → EntityManager

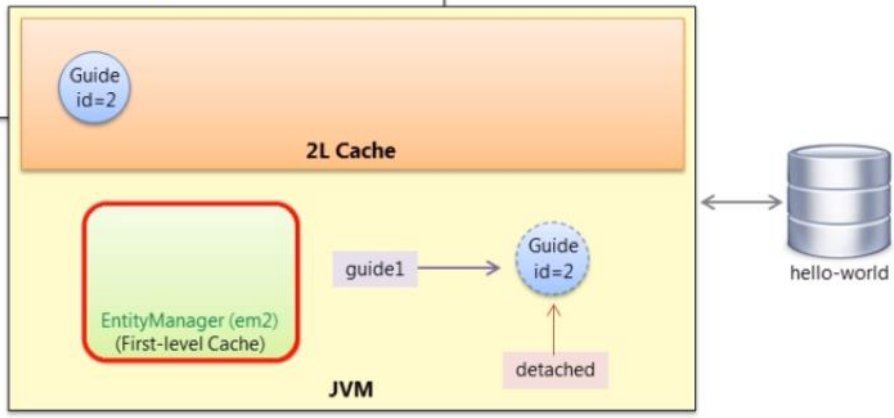
Second Level Cache — scope → EntityManagerFactory

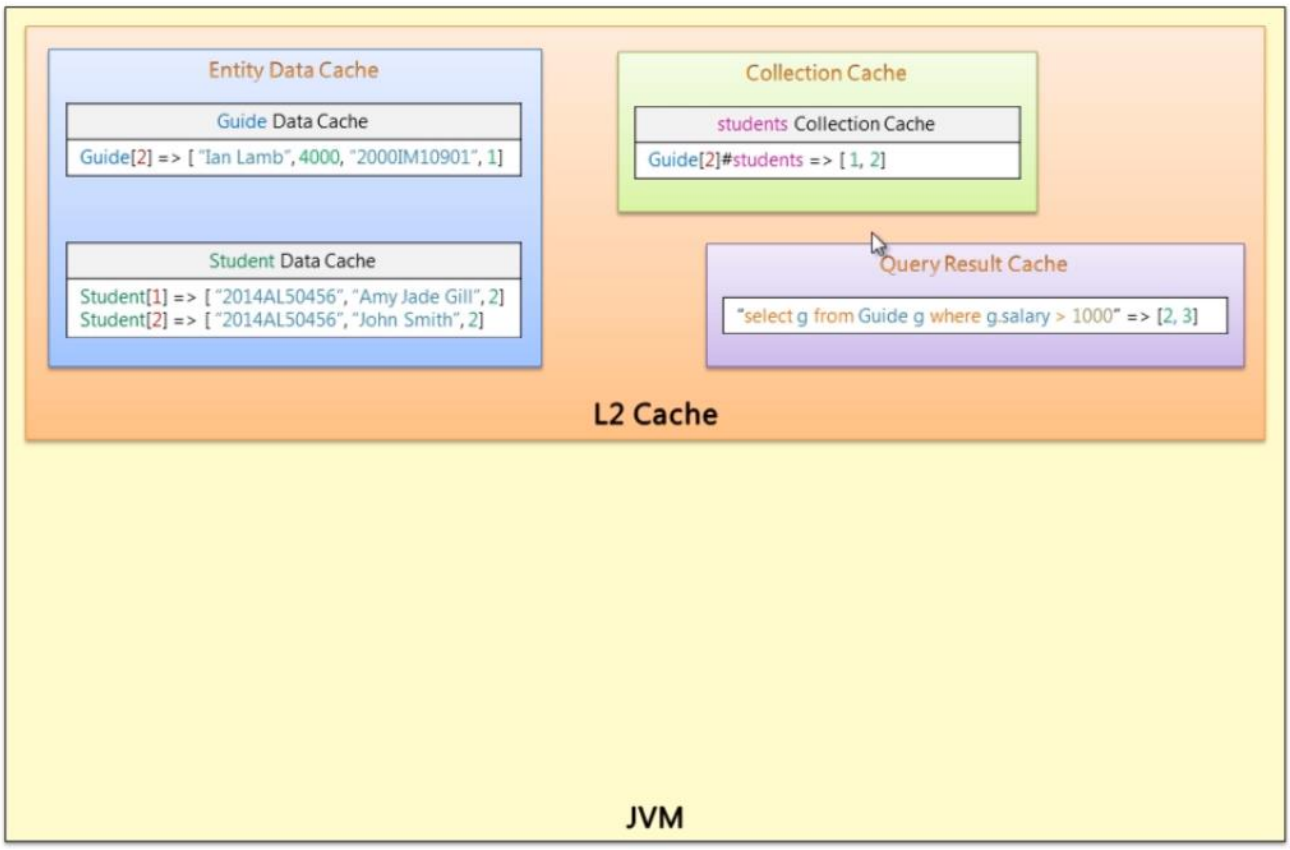
```
1  public class HelloWorldClient {
2    public static void main(String[] args) {
3      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
4
5      EntityManager em1 = emf.createEntityManager();
6      em1.getTransaction().begin();
7
8      Guide guide1 = em1.find(Guide.class, 2L);
9
10     em1.getTransaction().commit();
11     em1.close();
12
13     EntityManager em2 = emf.createEntityManager();
14     em2.getTransaction().begin();
15
16     Guide guide2 = em2.find(Guide.class, 2L);
17
18     em2.getTransaction().commit();
19     em2.close();
20   }
21 }
```

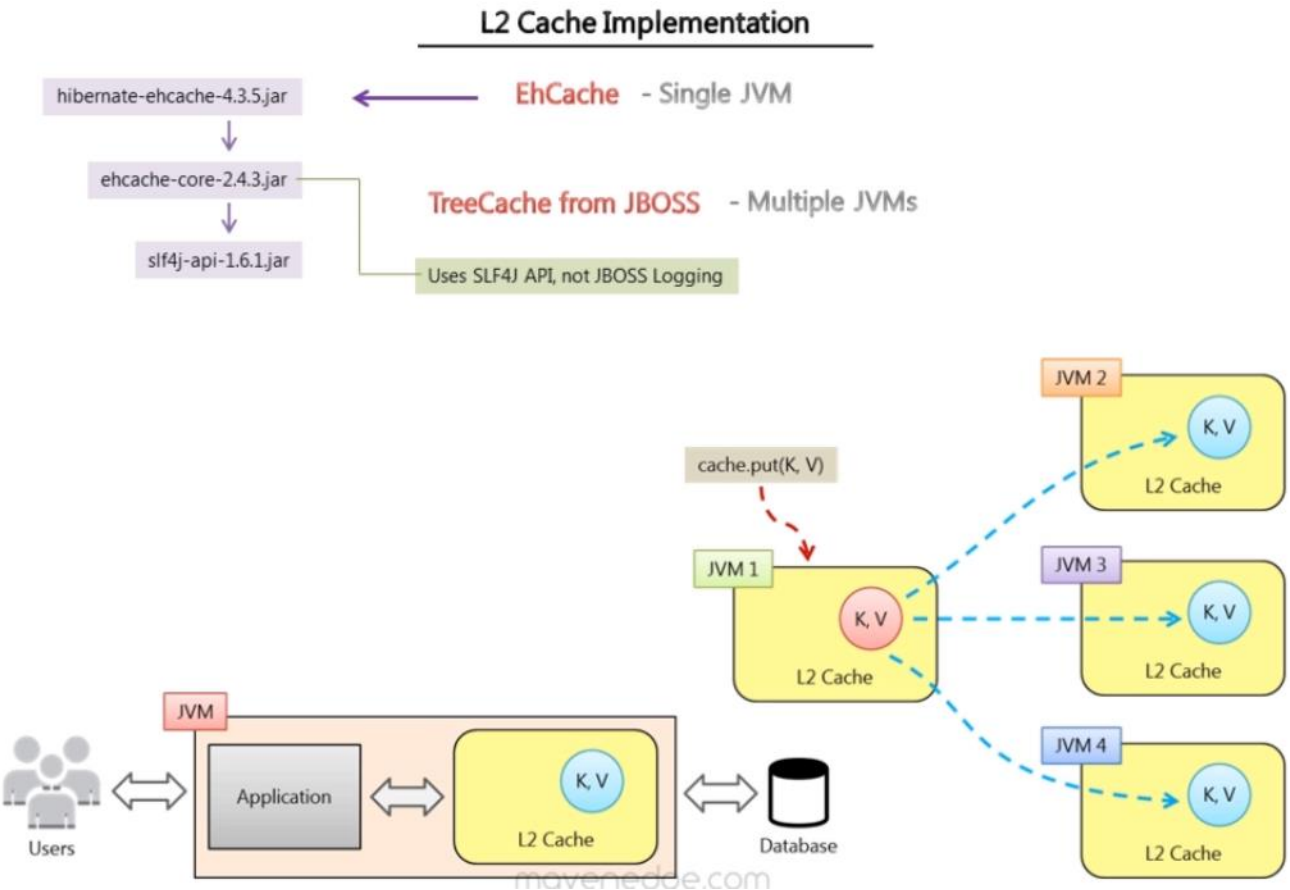
2L Cache enabled

SQL at runtime

```
select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
    guide0_.staff_id as staff_id4_0_0_,
    guide0_.version as version5_0_0_
from
    Guide guide0_
where
    guide0_.id=?
binding parameter [1] as [BIGINT] - [2]
```




Entity Data Cache
Guide Data Cache
Guide[2] => ["Ian Lamb", 4000, "2000IM10901", 1]
Student Data Cache
Student[1] => ["2014AL50456", "Amy Jade Gill", 2]
Student[2] => ["2014AL50456", "John Smith", 2]
Collection Cache
students Collection Cache
Guide[2]#students => [1, 2]
Query Result Cache
"select g from Guide g where g.salary > 1000" => [2, 3]
L2 Cache
JVM


L2 Cache Implementation
hibernate-ehcache-4.3.5.jar
EhCache - Single JVM
ehcache-core-2.4.3.jar
TreeCache from JBOSS - Multiple JVMs
slf4j-api-1.6.1.jar
Uses SLF4J API, not JBOSS Logging
cache.put(K, V)
JVM 1
K, V
L2 Cache
JVM 2
JVM 3
JVM 4
JVM
Application
Users
Database
mavenedge.com

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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_2_0.xsd"
             version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

  <persistence-unit name="hello-world" transaction-type="RESOURCE_LOCAL">
    <properties>

      <!-- Database connection, SQL Dialect, hbm2ddl.auto and other settings -->
      ...
      ...
      ...


      <property name="javax.persistence.sharedCache.mode" value="ALL" />

      <property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.EhCacheRegionFactory" />

    </properties>
  </persistence-unit>
</persistence>
```

```
package entity;

import java.util.*;
import javax.persistence.Cacheable;
import javax.persistence.*;

@Entity
@Cacheable    //or @Cacheable(true)
public class Guide {

    //all the interesting stuff

}
```

L2 Cache enabled

```
1  public class HelloWorldClient {
2    public static void main(String[] args) {
3      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
4
5      EntityManager em1 = emf.createEntityManager();
6      em1.getTransaction().begin();
7
8      Guide guide1 = em1.find(Guide.class, 2L);
9
10     em1.getTransaction().commit();
11     em1.close();
12
13     EntityManager em2 = emf.createEntityManager();
14     em2.getTransaction().begin();
15
16     Guide guide2 = em2.find(Guide.class, 2L);
17     guide2.setSalary(7000);
18
19     em2.getTransaction().commit();
20     em2.close();
21   }
22 }
```

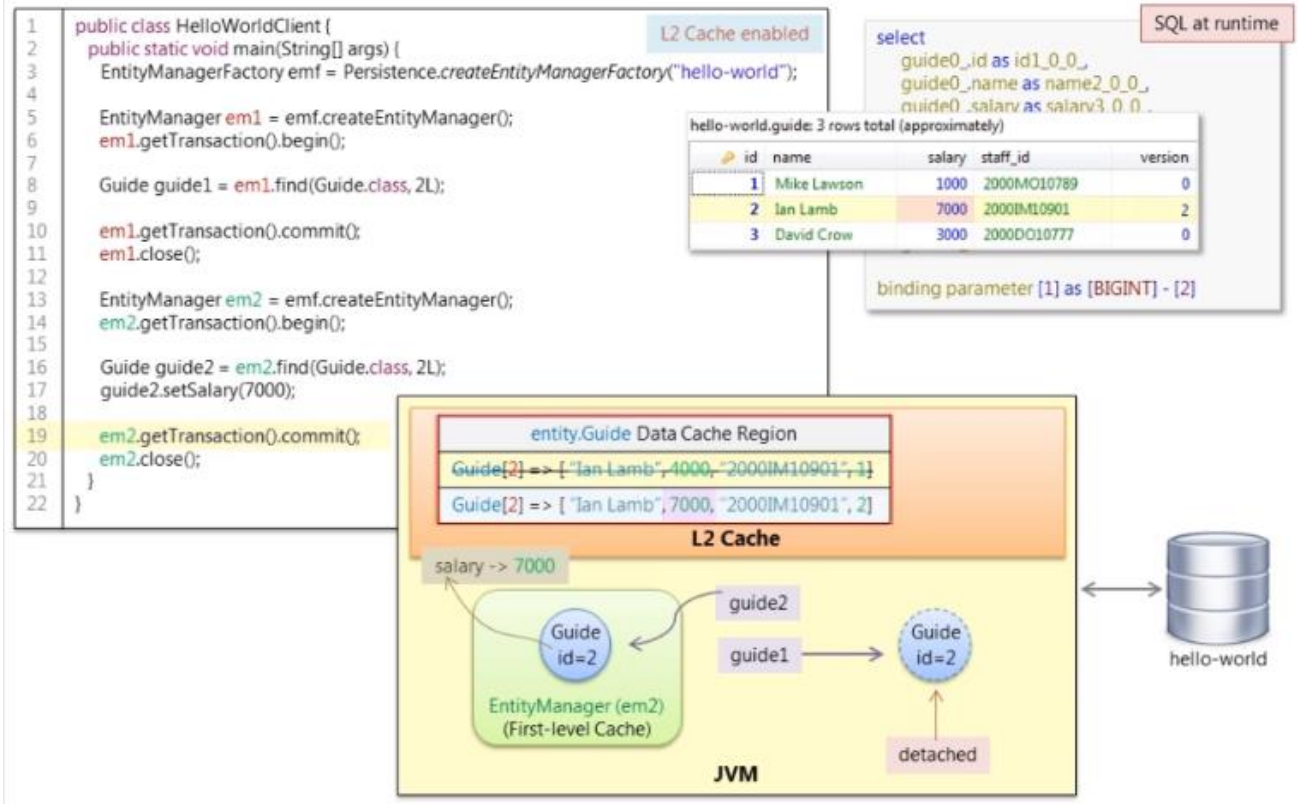
SQL at runtime

```
select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
```

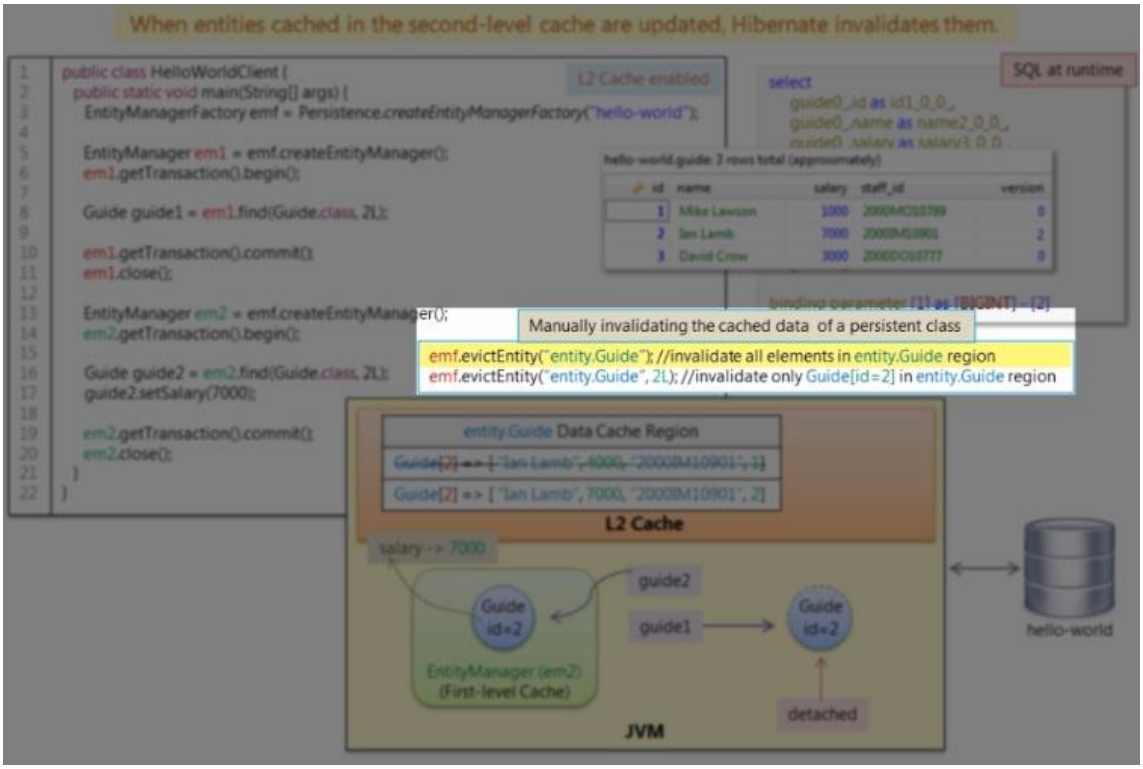
hello-world.guide: 3 rows total (approximately)

id	name	salary	staff_id	version
1	Mike Lawson	1000	2000MO10789	0
2	Ian Lamb	7000	2000IM10901	2
3	David Crow	3000	2000DO10777	0

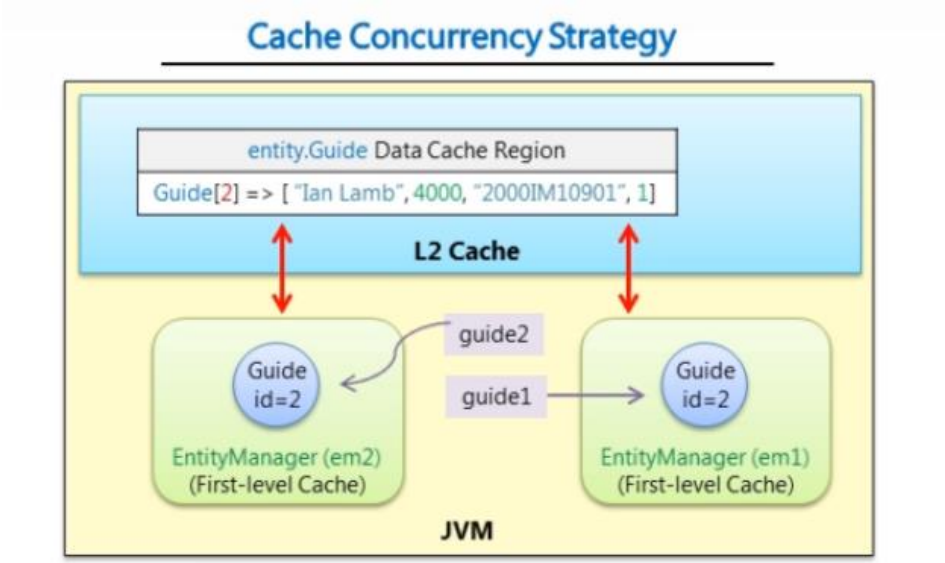
```
binding parameter [1] as [BIGINT] - [2]
```



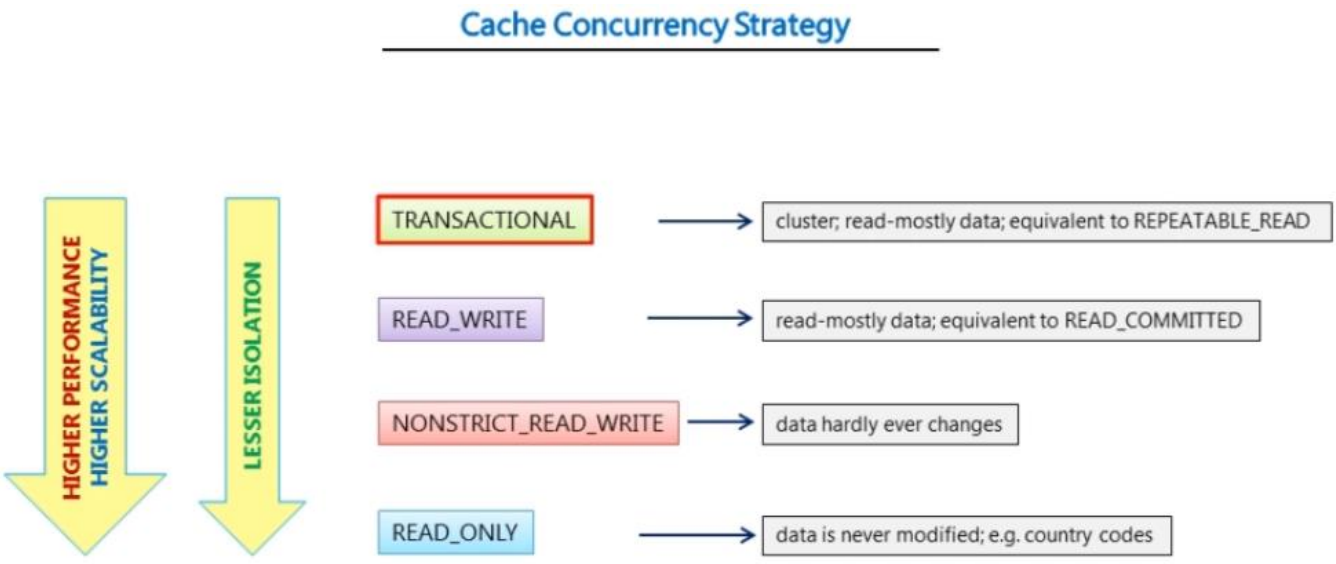
23. When Record change, cache invalidated them.

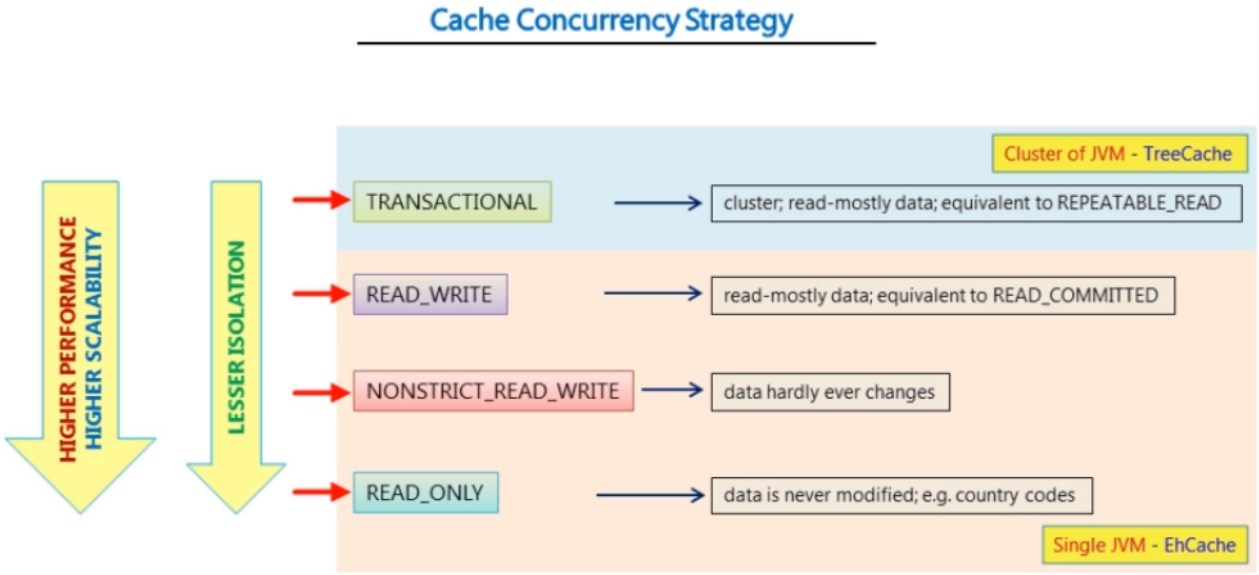



```
1   package client;
2
3   import javax.persistence.*;
4
5   import org.hibernate.SessionFactory;
6   import org.hibernate.stat.Statistics;
7
8   import entity.Guide;
9
10  public class HelloWorldClient {
11    public static void main(String[] args) {
12      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
13
14      Statistics stats = emf.unwrap(SessionFactory.class).getStatistics();
15      stats.setStatisticsEnabled(true);
16
17      EntityManager em1 = emf.createEntityManager();
18      em1.getTransaction().begin();
19
20      Guide guide1 = em1.find(Guide.class, 2L);
21
22      em1.getTransaction().commit();
23      em1.close();
24
25      EntityManager em2 = emf.createEntityManager();
26      em2.getTransaction().begin();
27
28      Guide guide2 = em2.find(Guide.class, 2L);
29
30      em2.getTransaction().commit();
31      em2.close();
32
33      System.out.println(stats.getSecondLevelCacheStatistics("entity.Guide"));
34
35    }
36  }
```

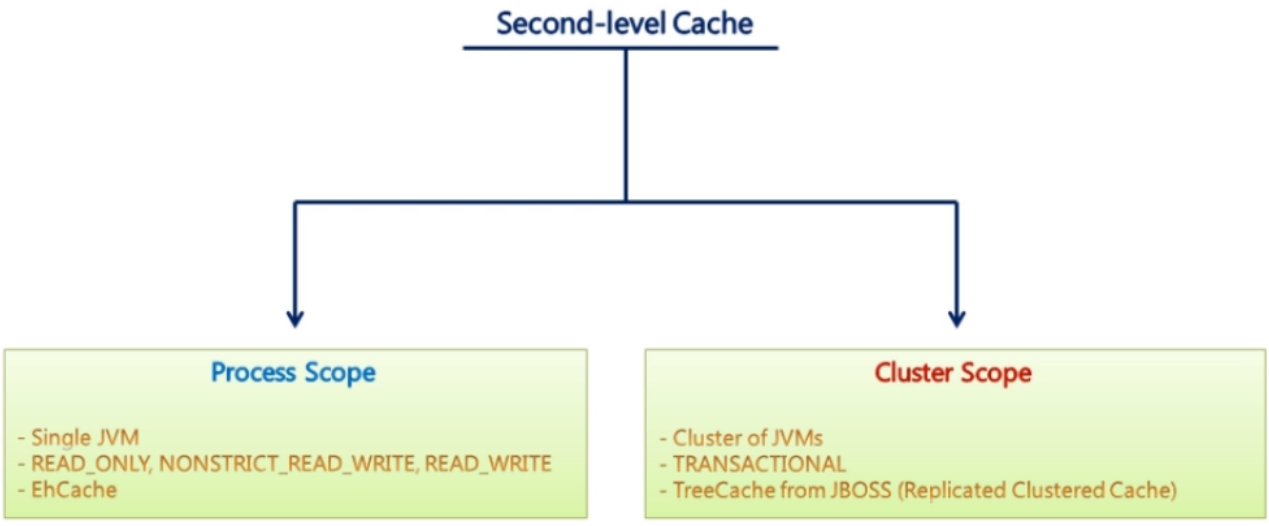



```java
package entity;

import java.util.*;
import javax.persistence.Cacheable;
import javax.persistence.*;

@Entity
@Cacheable    //default CacheConcurrencyStrategy.READ_WRITE
public class Guide {

    //all the interesting stuff

}
```



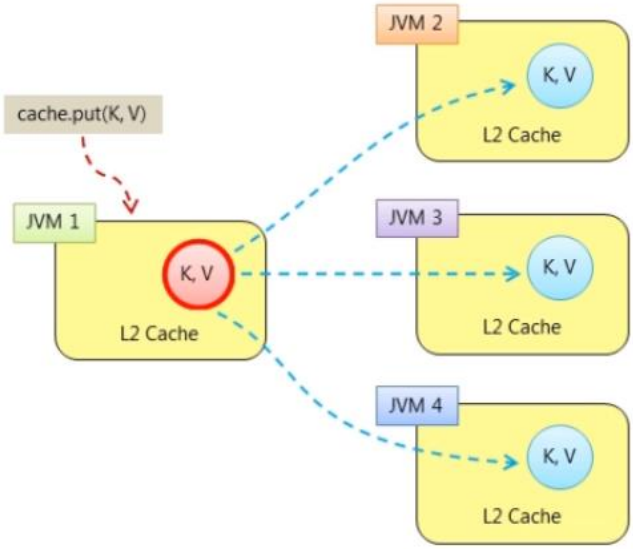



```
package entity;

import java.util.*;
import javax.persistence.Cacheable;
import javax.persistence.*;

import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Guide {

    //intesresting stuff

}
```



Replicated Clustered Cache

The cached data of your second-level cache is replicated to all the virtual machines that your application is running on



- ***Caching association***

Caching Associations (in Second-level Cache)

By default, associated objects aren't cached.

The reason to cache associations is to avoid extra calls to the database.

```
package entity;
import javax.persistence.*;

@Entity
public class Student {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //other interesting stuff

}
```

```
package entity;
import java.util.*;
import javax.persistence.*;

@Entity
@Cacheable
public class Guide {

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();

    //other interesting stuff

}
```

```java
package entity;
import javax.persistence.*;

@Entity
public class Student {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //other interesting stuff

}
```

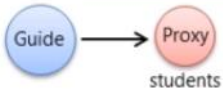
```java
package entity;
import java.util.*;
import javax.persistence.*;

@Entity
@Cacheable
public class Guide {

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();

    //other interesting stuff

}
```

Lazy Loading – default


```java
package entity;
import javax.persistence.*;

@Entity
@Cacheable
public class Student {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //other interesting stuff

}
```

```java
package entity;
import java.util.*;
import javax.persistence.*;

@Entity
@Cacheable
public class Guide {

    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();

    //other interesting stuff

}
```

Lazy Loading – default



ehcache.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd" >

    <cache name="entity.Guide"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
    />
    <cache name="entity.Student"
           maxElementsInMemory="10000"
           eternal="true"
           overflowToDisk="false"
    />

</ehcache>
```

after entity.Student L2 Cache enabled

```
1  public class HelloWorldClient {
2    public static void main(String[] args) {
3      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello-world");
4
5      Statistics stats = emf.unwrap(SessionFactory.class).getStatistics();
6      stats.setStatisticsEnabled(true);
7
8      EntityManager em1 = emf.createEntityManager();
9      em1.getTransaction().begin();
10
11     Guide guide1 = em1.find(Guide.class, 2L);
12     int size1 = guide1.getStudents().size();
13
14     em1.getTransaction().commit();
15     em1.close();
16
17     EntityManager em2 = emf.createEntityManager();
18     em2.getTransaction().begin();
19
20     Guide guide2 = em2.find(Guide.class, 2L);
21     int size2 = guide2.getStudents().size();
22
23     em2.getTransaction().commit();
24     em2.close();
25
26     System.out.println("Guide: " + stats.getSecondLevelCacheStatistics("entity.Guide"));
27     System.out.println("Student: " + stats.getSecondLevelCacheStatistics("entity.Student"));
28   }
29 }
30
```

SQL at runtime

```
select
    guide0_.id as id1_0_0_,
    guide0_.name as name2_0_0_,
    guide0_.salary as salary3_0_0_,
    guide0_.staff_id as staff_id4_0_0_,
    guide0_.version as version5_0_0_
from
    Guide guide0_
where
    guide0_.id=?
binding parameter [1] as [BIGINT] - [2]

select
    students0_.guide_id as guide_id4_0_0_,
    students0_.id as id1_1_0_,
    students0_.id as id1_1_1_,
    students0_.enrollment_id as enrollme2_1_1_,
    students0_.guide_id as guide_id4_1_1_,
    students0_.name as name3_1_1_
from
    Student students0_
where
    students0_.guide_id=?
binding parameter [1] as [BIGINT] - [2]

select
    students0_.guide_id as guide_id4_0_0_,
    students0_.id as id1_1_0_,
    students0_.id as id1_1_1_,
    students0_.enrollment_id as enrollme2_1_1_,
    students0_.guide_id as guide_id4_1_1_,
    students0_.name as name3_1_1_
from
    Student students0_
where
    students0_.guide_id=?
binding parameter [1] as [BIGINT] - [2]
```

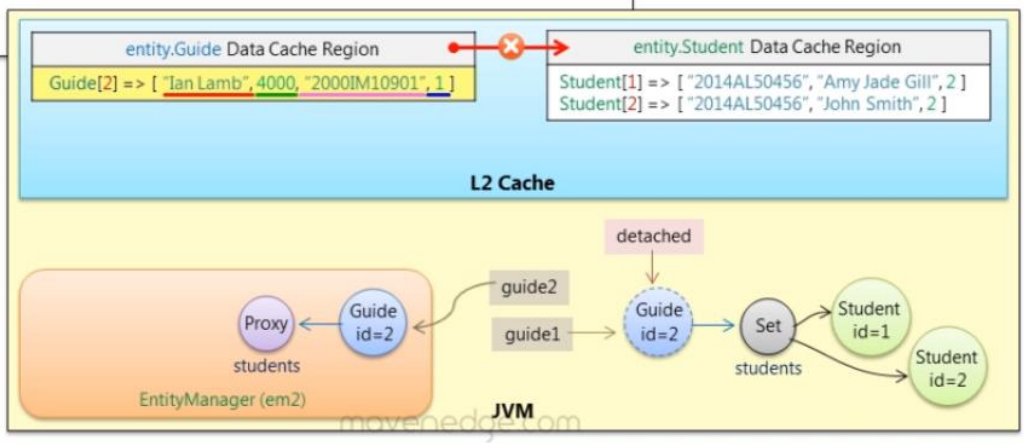
```
Guide: SecondLevelCacheStatistics[hitCount=1,missCount=1,putCount=1,elementCountInMemory=1,elementCountOnDisk=0,sizeInMemory=2188]
Student: SecondLevelCacheStatistics[hitCount=0,missCount=0,putCount=2,elementCountInMemory=2,elementCountOnDisk=0,sizeInMemory=4260]
```


after entity.Student L2 Cache enabled

```
8      EntityManager em1 = emf.createEntityManager();
9      em1.getTransaction().begin();
10
11     Guide guide1 = em1.find(Guide.class, 2L);
12     int size1 = guide1.getStudents().size();
13
14     em1.getTransaction().commit();
15     em1.close();
16
17     EntityManager em2 = emf.createEntityManager();
18     em2.getTransaction().begin();
19
20     Guide guide2 = em2.find(Guide.class, 2L);
21     int size2 = guide2.getStudents().size();
22
23     em2.getTransaction().commit();
24     em2.close();
```

SQL at runtim

```
select guide0_.id as id1_0_0_, ...
from Guide guide0_
where guide0_.id=?

select students0_.guide_id as guide_id4_0_0_, ...
from Student students0_
where students0_.guide_id=?
```

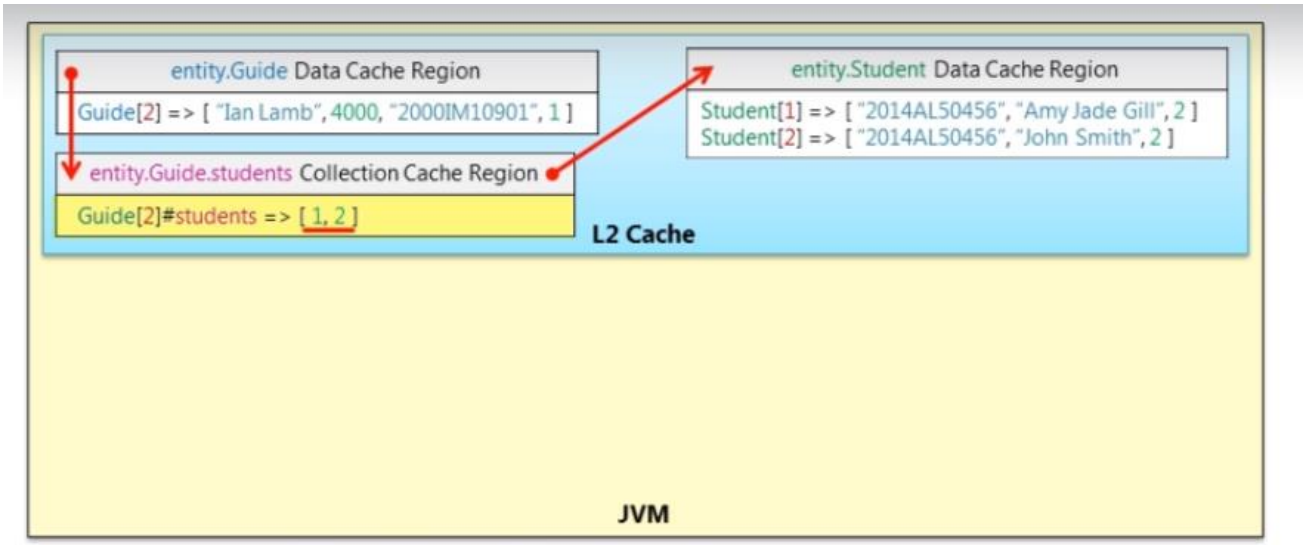


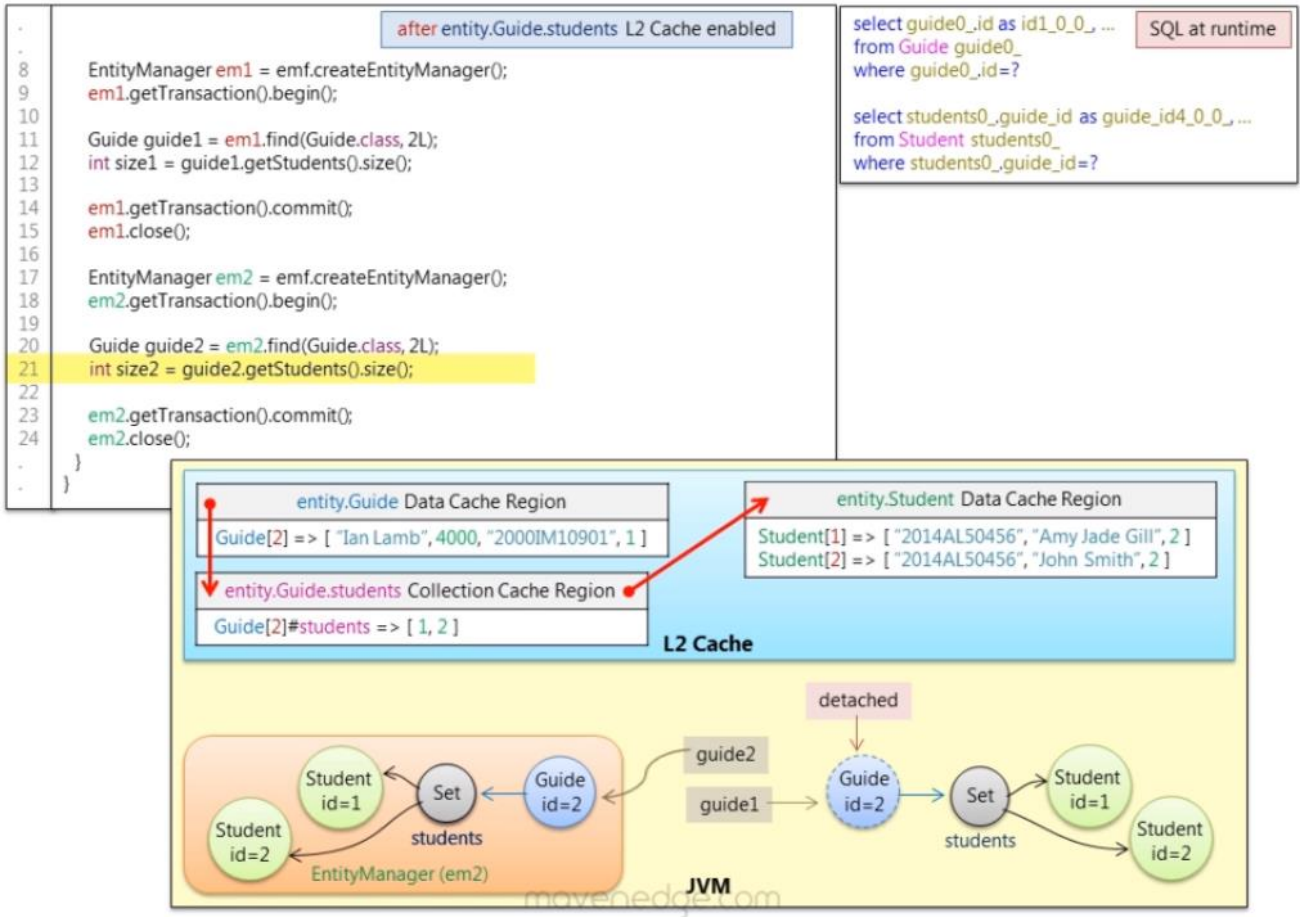
```java
package entity;
import javax.persistence.*;

@Entity
@Cacheable
public class Student {

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="guide_id")
    private Guide guide;

    //other interesting stuff

}
```

```java
package entity;
import java.util.*;
import javax.persistence.*;

import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cacheable
public class Guide {

    @Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
    private Set<Student> students = new HashSet<Student>();

    //other interesting stuff

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd">

    <cache name="entity.Guide"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
    />
    <cache name="entity.Student"
           maxElementsInMemory="10000"
           eternal="true"
           overflowToDisk="false"
    />
    <cache name="entity.Guide.students"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
    />
</ehcache>
```

ehcache.xml






```
public void checkCache(Long ID){
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("clientJPA");
        EntityManager firstManager = factory.createEntityManager();
        Statistics statistics = factory.unwrap(SessionFactory.class).getStatistics();
        statistics.setStatisticsEnabled(true);

        firstManager.getTransaction().begin();
        System.out.println("Start first Entity Manager");
        Guide guide;
        try {
            guide = firstManager.find(Guide.class, ID);
            Integer size = guide.getStudents().size();
        } catch (HibernateException exception){
            System.out.println("[Guide] ID " + ID + " not found" );
            return;
        }
        firstManager.getTransaction().commit();
        firstManager.close();


        EntityManager secondManager = factory.createEntityManager();
        System.out.println("Start second Entity Manager");
        secondManager.getTransaction().begin();
        try {
            guide = secondManager.find(Guide.class, ID);
            Integer size = guide.getStudents().size();
        } catch (HibernateException exception){
            System.out.println("[Guide] ID " + ID + " not found" );
            return;
        } finally {
            secondManager.close();
        }

        EntityManager thirdManager = factory.createEntityManager();
        System.out.println("Start third Entity Manager");
        thirdManager.getTransaction().begin();
```

```
        try {
            guide = thirdManager.find(Guide.class, ID);
            Integer size = guide.getStudents().size();
        } catch (HibernateException exception){
            System.out.println("[Guide] ID " + ID + " not found" );
            return;
        } finally {
            thirdManager.close();
        }

        System.out.println("[Guide] LOG " +
statistics.getSecondLevelCacheStatistics("client.entity.Guide"));
        System.out.println("[Student] LOG " +
statistics.getSecondLevelCacheStatistics("client.entity.Student"));
        //System.out.println("[Guide] " + statistics.getEntityStatistics("Guide"));
    }
```

Log hibernate:

```
Start first Entity Manager
Hibernate:
    select
        guide0_.id as id1_0_0_,
        guide0_.name as name2_0_0_,
        guide0_.salary as salary3_0_0_,
        guide0_.staff_id as staff_id4_0_0_,
        guide0_.version as version5_0_0_
    from
        PER_GUIDE guide0_
    where
        guide0_.id=?
Hibernate:
    select
        students0_.guide_id as guide_id5_2_0_,
        students0_.id as id1_2_0_,
        students0_.id as id1_2_1_,
        students0_.enrollment_id as enrollme2_2_1_,
        students0_.guide_id as guide_id5_2_1_,
        students0_.name as name3_2_1_,
        students0_.version as version4_2_1_
    from
        PER_STUDENT students0_
    where
        students0_.guide_id=?
    order by
        students0_.id asc
Start second Entity Manager
Start third Entity Manager
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by net.sf.ehcache.pool.sizeof.ObjectGraphWalker
(file:/C:/Users/duyng/.m2/repository/net/sf/ehcache/ehcache/2.10.3/ehcache-2.10.3.jar) to field
java.lang.String.value
WARNING: Please consider reporting this to the maintainers of
net.sf.ehcache.pool.sizeof.ObjectGraphWalker
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
[Guide] LOG
SecondLevelCacheStatistics[hitCount=2,missCount=1,putCount=1,elementCountInMemory=1,elementCountOnDis
k=0,sizeInMemory=560]
[Student] LOG
SecondLevelCacheStatistics[hitCount=2,missCount=0,putCount=1,elementCountInMemory=1,elementCountOnDis
k=0,sizeInMemory=528]

Process finished with exit code -1
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd">

    <defaultCache
            maxEntriesLocalHeap="1000"
            eternal="false"
            timeToIdleSeconds="300"
            timeToLiveSeconds="600">
    </defaultCache>

    <cache name="client.entity.Guide"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
    />
    <cache name="client.entity.Student"
           maxElementsInMemory="10000"
           eternal="true"
           overflowToDisk="false"
    />
    <cache name="client.entity.Guide.students"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
    />
```

```xml
<persistence-unit name="clientJPA" transaction-type="RESOURCE_LOCAL">
    <properties>
        <!-- Database connection settings -->
        <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:mysql://database-aws.czcwawrrzspo.us-east-2.rds.amazonaws.com:3306/orm" />
        <property name="javax.persistence.jdbc.user" value="admin" />
        <property name="javax.persistence.jdbc.password" value="admin12345" />

        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL57Dialect" />
        <property name="hibernate.hbm2ddl.auto" value="update"/>
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.show_sql" value = "true" />


        <property name="javax.persistence.sharedCache.mode" value="ENABLE_SELECTIVE" />
        <property name="hibernate.cache.region.factory_class" value="ehcache-singleton" />
        <property name="hibernate.cache.use_second_level_cache" value="true"/>
        <property name="hibernate.cache.use_query_cache" value="true"/>

    </properties>

</persistence-unit>
```

```java
@BatchSize(size = 4)
@Setter
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage= CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Guide {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name="staff_id", nullable=false)
    private String staffId;

    private String name;
    private Integer salary;

    @org.hibernate.annotations.Cache(usage= CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    @OneToMany(mappedBy="guide", cascade={CascadeType.PERSIST})
    @OrderBy("id ASC")
    private Set<Student> students = new HashSet<~>();
```

```java
@org.hibernate.annotations.Cache(usage= CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Student {

    @Id
    @GeneratedValue(strategy= GenerationType.SEQUENCE)
    private Long id;


    @Column(name="enrollment_id", nullable=false)
    private String enrollmentId;

    private String name;

    @ManyToOne(cascade={CascadeType.PERSIST})
    @JoinColumn(name="guide_id")
    private Guide guide;

    @Version
    Integer version;
```

3.3. JOOQ Object Oriented Querying

3.3.1. SQL building

- ***DSL Context***

```
+ show imports
```

```java
// Create it from a pre-existing configuration
DSLContext create = DSL.using(configuration);

// Create it from ad-hoc arguments
DSLContext create = DSL.using(connection, dialect);
```

```
+ show imports
```

```java
// Execute a statement from a single execution chain:
Result<?> result =
DSL.using(connection, dialect)
   .select()
   .from(BOOK)
   .where(BOOK.TITLE.like("Animal%"))
   .fetch();
```

- ***In-list padding***

Databases that feature a **cursor cache / statement cache** (e.g. Oracle, SQL Server, DB2, etc.) are highly optimised for **prepared statement re-use**. When a client sends a prepared statement to the server, the server will go to the cache and **look up whether there already exists a previously calculated execution plan for the statement** (i.e. the **SQL string**). This is called a "**soft-parse**" (in **Oracle**). If not, the execution plan is calculated on the fly. This is called a "**hard-parse**" (in **Oracle**).

Preventing hard-parses is extremely important in **high throughput OLTP systems** where queries are usually **not very complex** but are **run millions of times** in a short amount of time. **Using bind variables**, this is usually not a problem, with the exception of the **IN predicate,** which generates **different** SQL strings even when using bind variables:

```
-- All of these are different SQL statements:
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
```

```
// All of these are the same jOOQ statement
DSL.using(configuration)
   .select()
   .from(AUTHOR)
   .where(AUTHOR.ID.in(collection))
   .fetch();
```

```
-- Original
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?)
```

```
-- Padded
SELECT * FROM AUTHOR WHERE ID IN (?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
SELECT * FROM AUTHOR WHERE ID IN (?, ?, ?, ?, ?, ?, ?, ?)
```

24. Better SQL string with padding, result in the sample execution plan

```
import static org.jooq.impl.DSL.*;
import org.jooq.*;
import org.jooq.conf.*;
import org.jooq.impl.*;

Settings settings = new Settings()
    .withInListPadding(true) // Default to false
    .withInListPadBase(4);   // Default to 2
```

► *SQL Statements (DML)*

```
public class DataSource {
    private static final HikariConfig config = new HikariConfig();
    private static final HikariDataSource dataSource;

    static {

        Properties configProperties = new Properties();
        try {
            configProperties.load(new FileReader( fileName: System.getProperty("user.dir") + "/s
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }

        config.setJdbcUrl(configProperties.getProperty("jdbc.url"));
        config.setUsername(configProperties.getProperty("jdbc.username"));
        config.setPassword(configProperties.getProperty("jdbc.password"));

        config.addDataSourceProperty( propertyName: "cachePrepStmts" , value: "true" );
        config.addDataSourceProperty( propertyName: "prepStmtCacheSize" , value: "250" );
        config.addDataSourceProperty( propertyName: "prepStmtCacheSqlLimit" , value: "2048" );
        dataSource = new HikariDataSource( config );
    }

    private DataSource() {}

    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
}
```

25. Hikari Data connection pool

```
public void getStudents(){
    System.out.println("getStudents ");
    try {
        Connection conn = DataSource.getConnection();
        DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
        Result<Record> result = create.select().from(PerStudent.PER_STUDENT).fetch();
        System.out.println(result);
    } catch (SQLException | DataAccessException | IllegalArgumentException sqlException) {
        sqlException.printStackTrace();
    }
}
```

26. Fetch all

```java
public void getStudent(Long ID) {
    System.out.println("getStudent ID " + ID);
    try {
        Connection conn = DataSource.getConnection();
        DSLContext context = DSL.using(conn, SQLDialect.MYSQL);
        Result<Record> result = context.select().from(PER_STUDENT).where(PER_STUDENT.ID.eq(ID)).fetch();
        if (result.size() > 0){
            System.out.println(result.get(0).toString());
        } else {
            System.out.println("No record with ID " + ID + " fou" );
        }
    } catch (SQLException | DataAccessException sqlException) {
        sqlException.printStackTrace();
    }
}
```

27. Fetch One

```java
public void getStudentHavingGuide(Integer n) {
    System.out.println("get Student having at least " + n + " guide");
    try {
        Connection conn = DataSource.getConnection();
        DSLContext context = DSL.using(conn, SQLDialect.MYSQL);
        Result<Record2<String, Integer>> result = context.select(PER_STUDENT.NAME, count())
                                                  .from(PER_STUDENT)
                                                  .groupBy(PER_STUDENT.GUIDE_ID)
                                                  .having(count().ge(n))
                                                  .fetch();
        if (result.size() > 0){
            System.out.println(result);
        } else {
            System.out.println("No record found");
        }
    } catch (SQLException | DataAccessException sqlException) {
        sqlException.printStackTrace();
    }
}
```

28. Fetch conditional

4. Spring Security

4.1. Spring security overview

Support Spring security type:

- An in-memory user store
- A JDBC-based user store
- An LDAP-backed user store
- A custom user details service

- ***In-memory user store [deprecated]***

- ***JDBC-based user store:***

First inject datasource into Security config:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception{
    auth.jdbcAuthentication()
            .passwordEncoder(new BCryptPasswordEncoder())
            .dataSource(dataSource)
            .usersByUsernameQuery("select username, password, enabled from users where username=?") // duyntc / codejava
            .authoritiesByUsernameQuery("select username, role from users where username=?");
};
```

- ***Secure Request***

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
                .antMatchers("/**/*.js", "/**/*.css", "/**/*.png").permitAll()
                .antMatchers("/**")
                .access("hasRole('USER') && hasIpAddress('127.0.0.1/32') ")

                .and()
                .formLogin()
                .loginPage("/login")
                .permitAll()
                .defaultSuccessUrl("/component/list", true)
                .and()
                .logout()
                .logoutSuccessUrl("/login");
    }
```

Logout:

```
<form method="POST" th:action="@{/logout}">
  <input type="submit" value="Logout"/>
</form>
```

- ***Preventing cross-site request forgery***

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site (such as bank, credit card, …) when the user is authenticated.

A CSRF attack works because browser requests automatically include all cookies including session cookies.

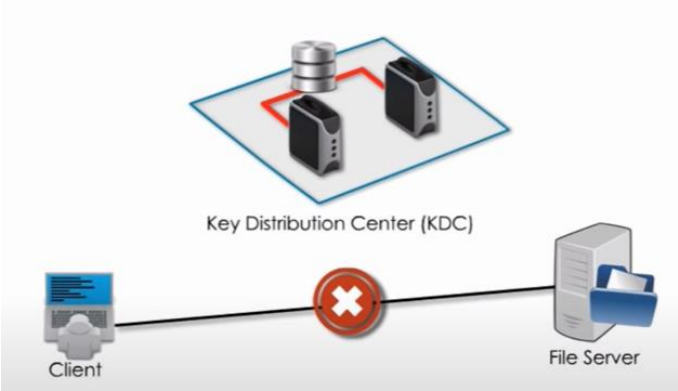
Prevent Cross-site request forgery:

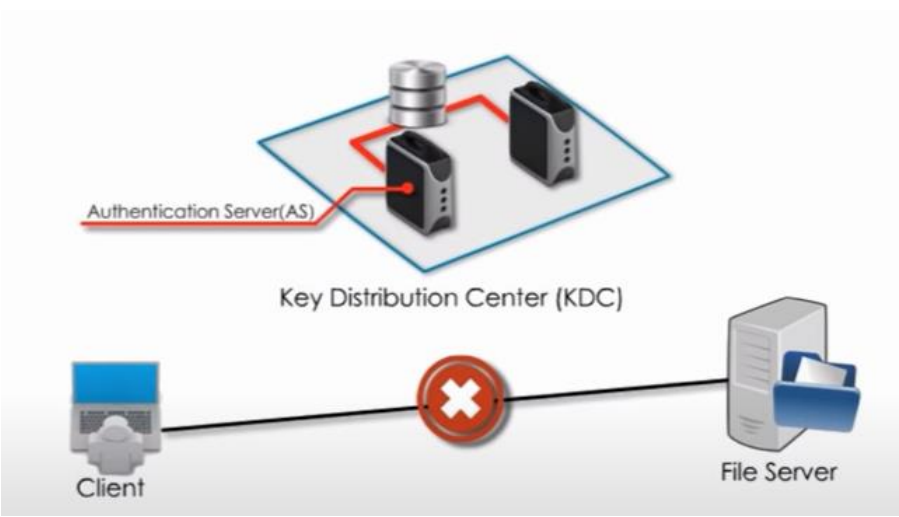
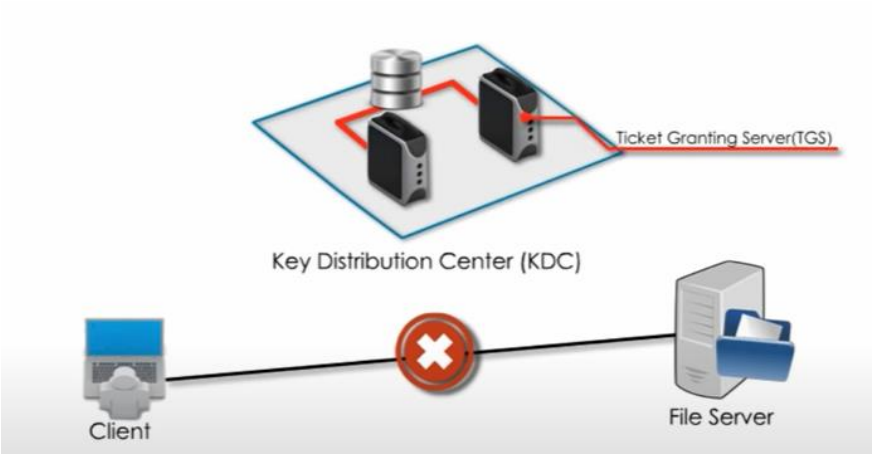
```
<form method="POST" th:action="@{/login}" id="loginForm">
    <input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

- ***Get User info***

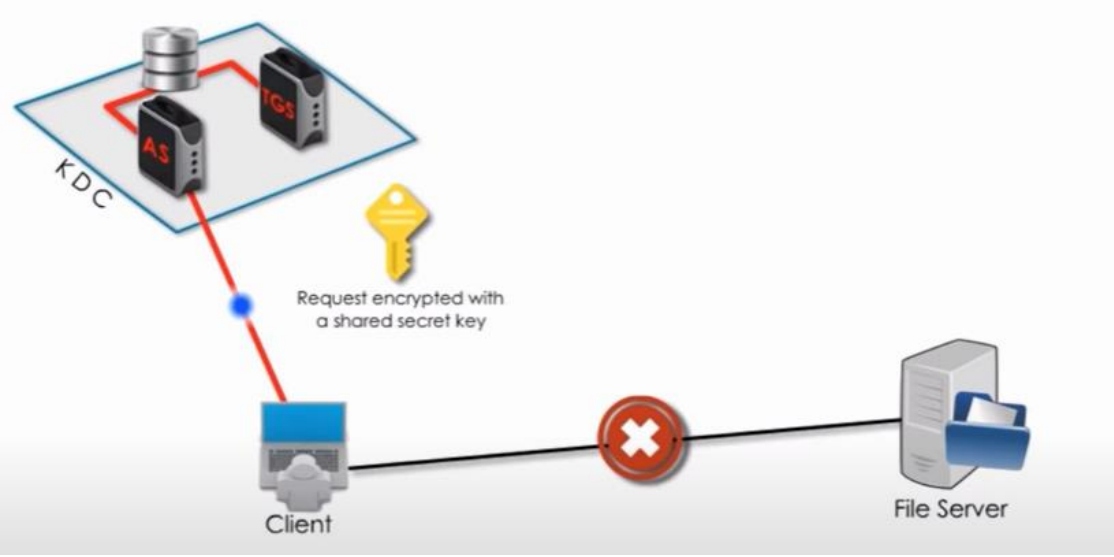
```
@RequestMapping("/list")
public String getAllComponent(Model model, Principal principal){
    model.addAttribute( attributeName: "components", repository.findAll());
    model.addAttribute( attributeName: "username", principal.getName());
    log.info("Principal {}", principal);
    return "component/component";
}
```

- ***Kerberos***

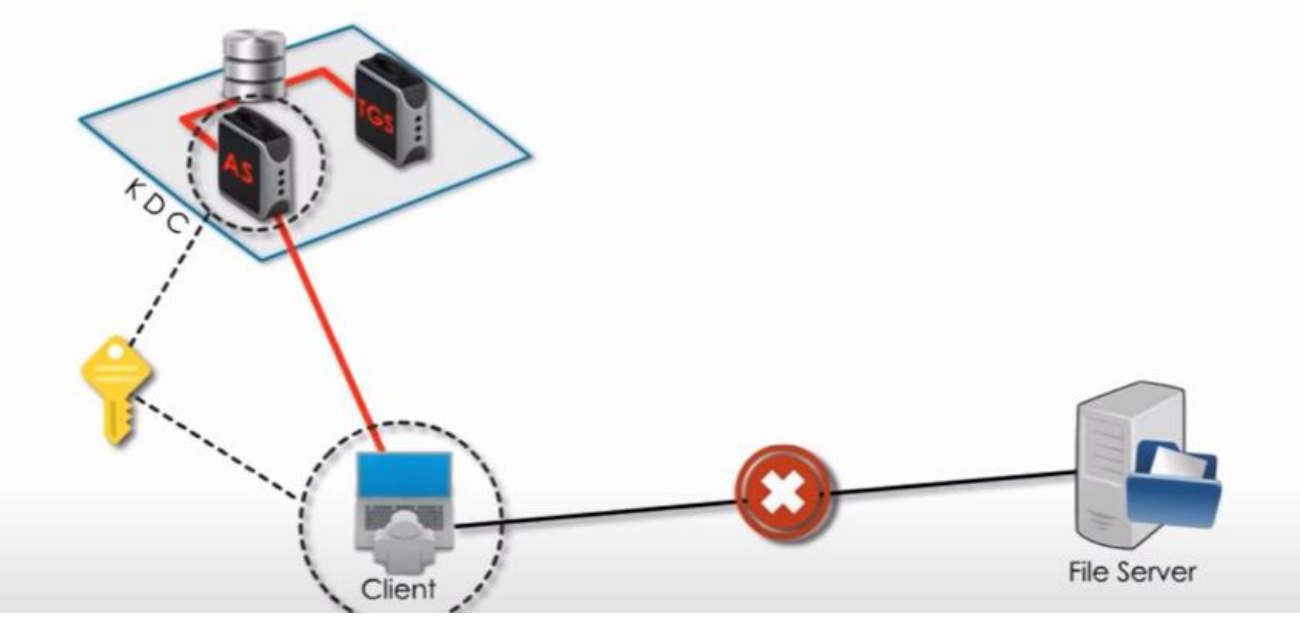
Component	Description
Key Distribution Center	 **to prevent eavesdropping.**

Authentication Server	
Ticket grating server	

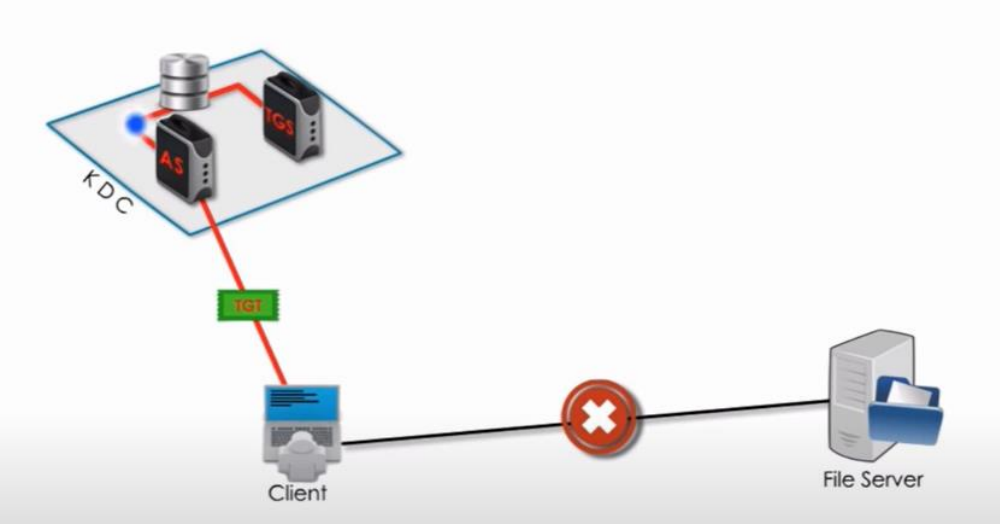
Step 1: User Authenticate

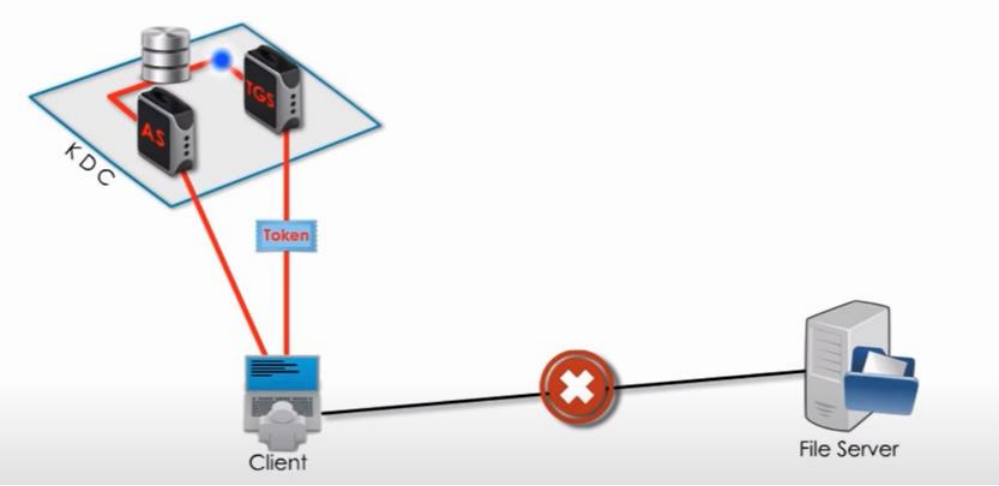


Step 2: Authentication server use password to decrypt user password:



Step 3: Ticket is grant to client





- ***Oauth***
- ***Oauth flow:***

In OAuth 2.0, the following three parties are involved:

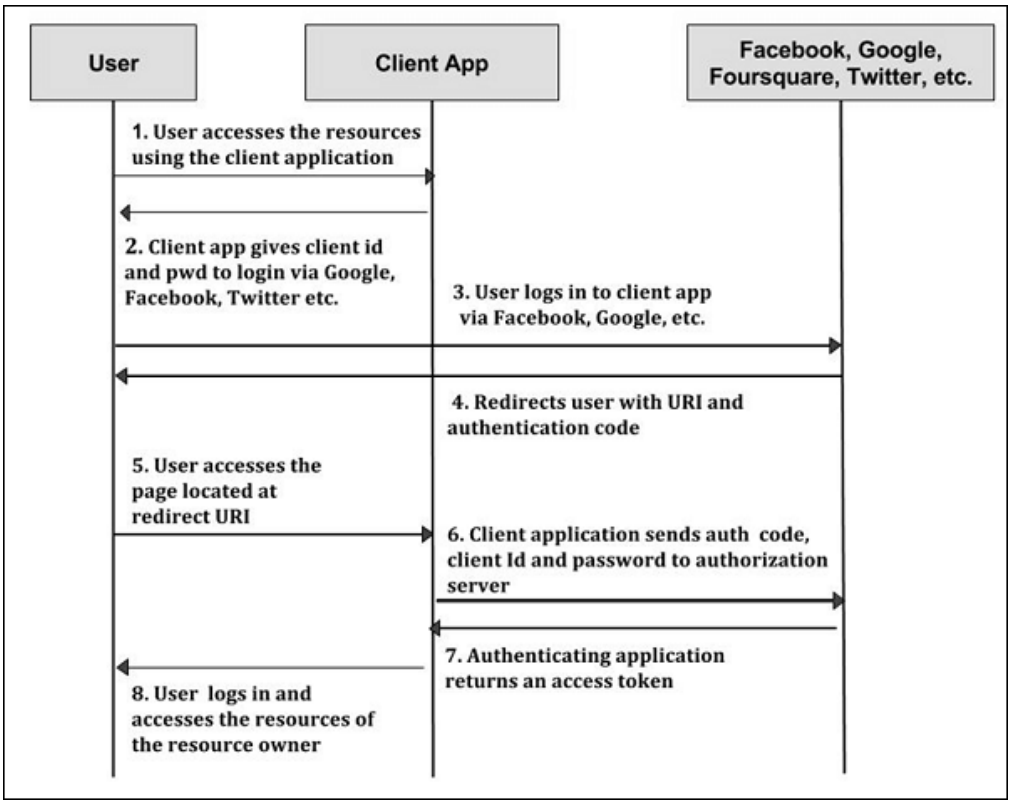
- The user, who possesses data that is accessed through the API and wants to allow the application to access it
- The application, which is to access the data through the API on the user's behalf
- The API, which controls and enables access to the user's data.

OAuth Scheme:

1. implicit flow
2. application flow
3. confidential password flow
4. public password flow
5. confidential access code flow
6. public access code flow.

Oauth definition:

OAuth is an open authorization protocol, which allows accessing the resources of the resource owner by enabling the client applications on HTTP services such as Facebook, GitHub, etc. It allows sharing of resources stored on one site to another site without using their credentials. It uses username and password tokens instead.



Client credentials:

Step 1 – The client authenticates with the authorization server and makes a request for access token from the token endpoint.

Step 2 – The authorization server authenticates the client and provides access token if it's valid and authorized.

Access token:

An access token is a string that identifies a user, an application, or a page.

The token includes information such as when the token will expire and which app created that token.

- **First, it is necessary to acquire OAuth 2.0 client credentials from API console.**
- **Then, the access token is requested from the authorization server by the client.**
- **It gets an access token from the response and sends the token to the API that you wish to access.**

Init oauth with authorization request endpoint contain client id:

```
https://publicapi.example.com/oauth2/authorize?client_id=your_client_id&redirect_uri=your_url&response_type=code
```

Sending credentials:

- **Bearer token: token stay in header:**

  ```
  Authorization: Bearer [token-value]
  ```

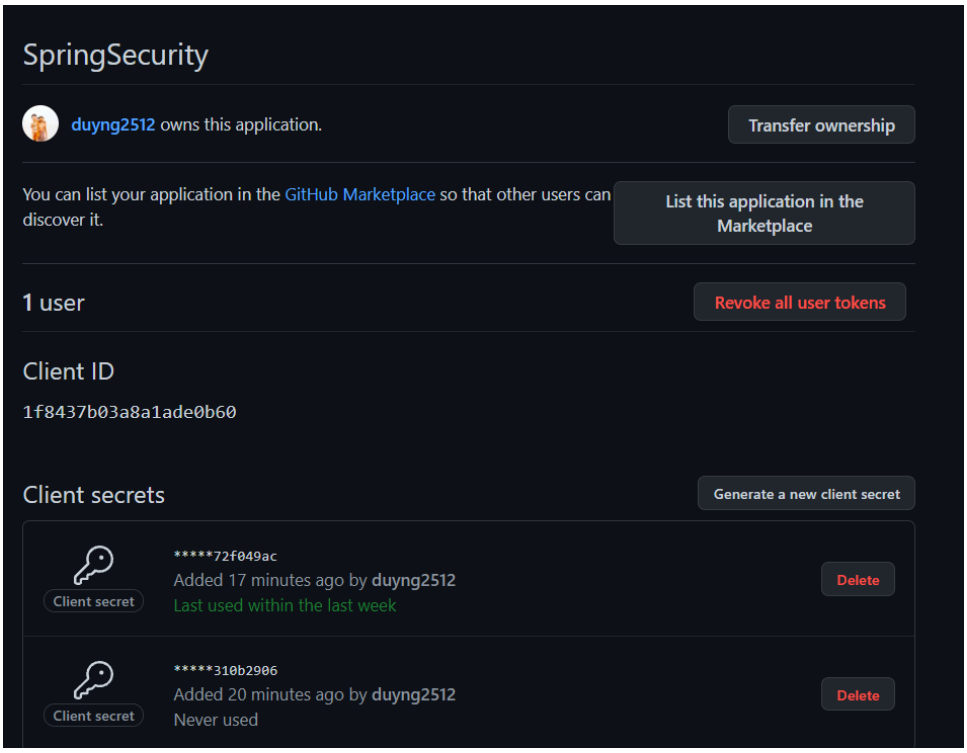
- **MAC**

Authentication scheme:

- **Authorization code – The most common flow, mostly used for server-side and mobile web applications. This flow is similar to how users sign up into a web application using their Facebook or Google account.**
- **Implicit – This flow requires the client to retrieve an access token directly. It is useful in cases when the user's credentials cannot be stored in the client code because they can be easily accessed by the third party. It is suitable for web, desktop, and mobile applications that do not include any server component.**

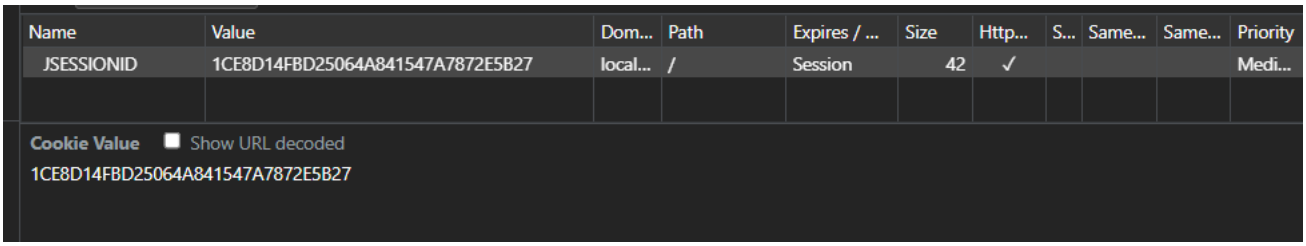
- **Resource owner password credentials (or just password) – Requires logging in with a username and password. Since in that case the credentials will be a part of the request, this flow is suitable only for trusted clients (for example, official applications released by the API provider).**
- **Client Credentials – Intended for the server-to-server authentication, this flow describes an approach when the client application acts on its own behalf rather than on behalf of any individual user. In most scenarios, this flow provides the means to allow users specify their credentials in the client application, so it can access the resources under the client's control.**

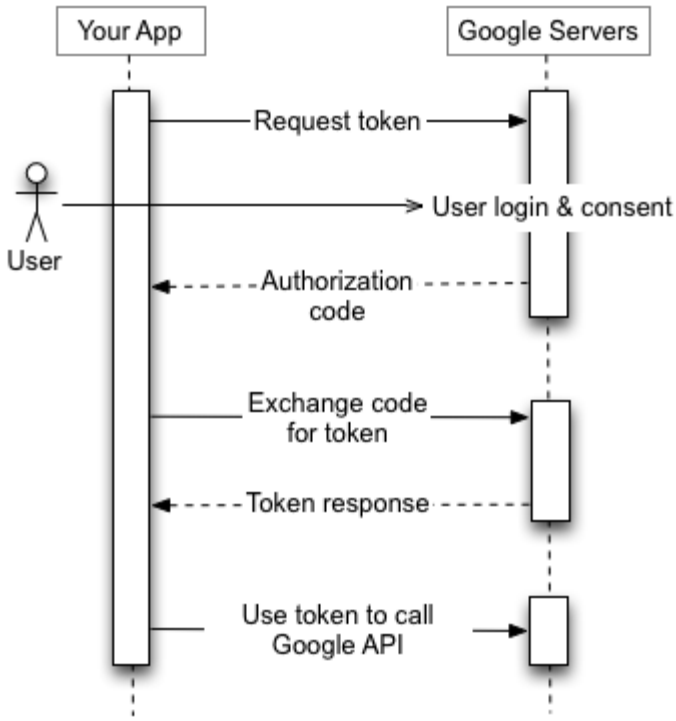
- ***Implement with github oauth:***



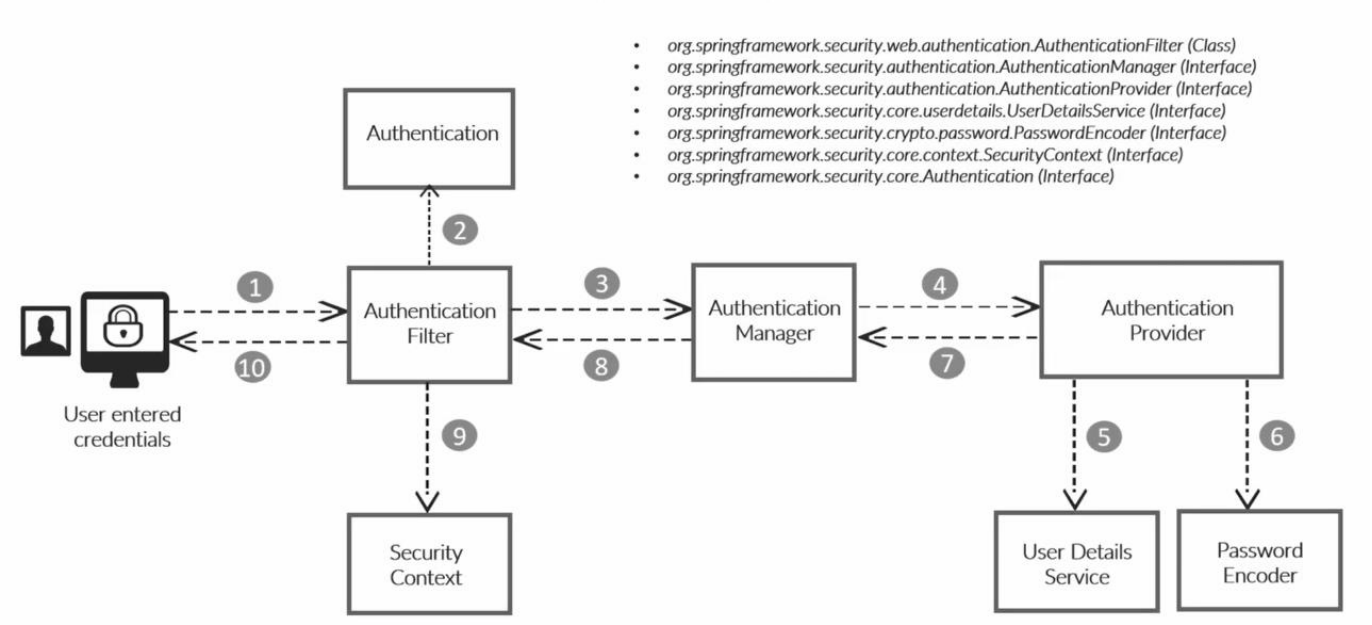
Add new github client Id and client secret.

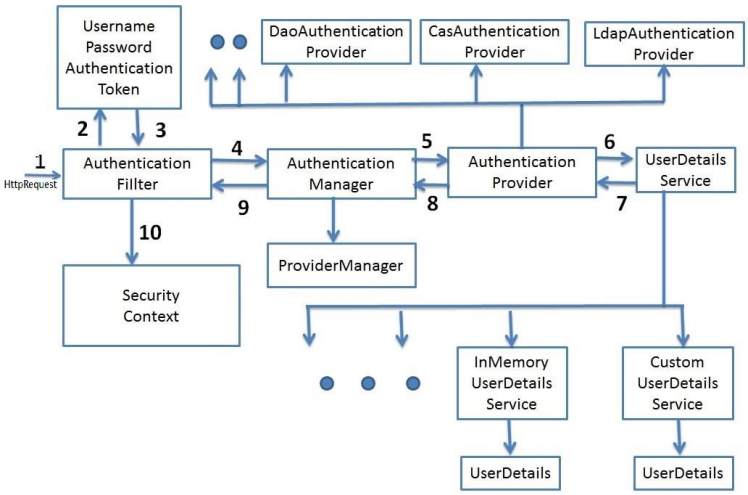
Request to github and return with token in JSEESIONID cookie.



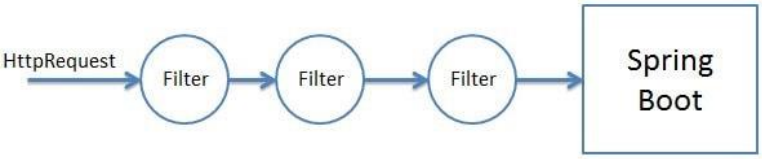


4.2. Spring security Interal

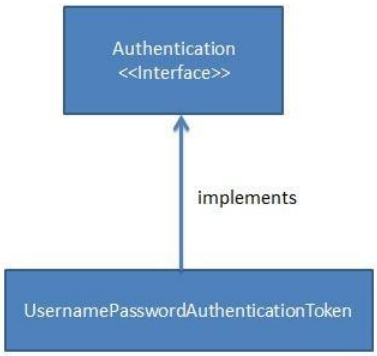




- ***Filters:*** Before the request reaches the Dispatcher Servlet, it is first intercepted by a chain of filters.



- ***Authentication Object Creation:*** When the request is intercepted by the appropriate AuthenticationFilter it retrieves the username and password from the request and creates the Authentication Object.

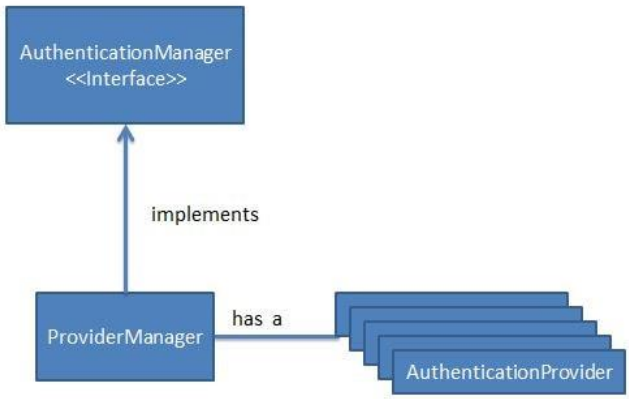


- ***AuthenicationManager -*** Using the Authentication Object created the filter will then call the authenticate method of the Authentication Manager. The Authentication Manager is only a interface and actual implementation of the authenticate method is provided by the ProviderManager.

Important point to note here is that the Authentication Manager takes an Authentication object as input and after successful authentication again returns an object of type Authentication.

Field	Authentication(User Request before Authentication)	Authentication(After Authentication)
Principal	username	User Object
Granted Authorities	Not granted any authorities	ROLE_ADMIN
Authenticated	false	true

The ProviderManager has a list of AuthenticationProviders.
From it's authenticate method it calls the authenticate method of the appropriate AuthenticateProvider. In response it gets the Principal Authentication Object if the authentication is successful.

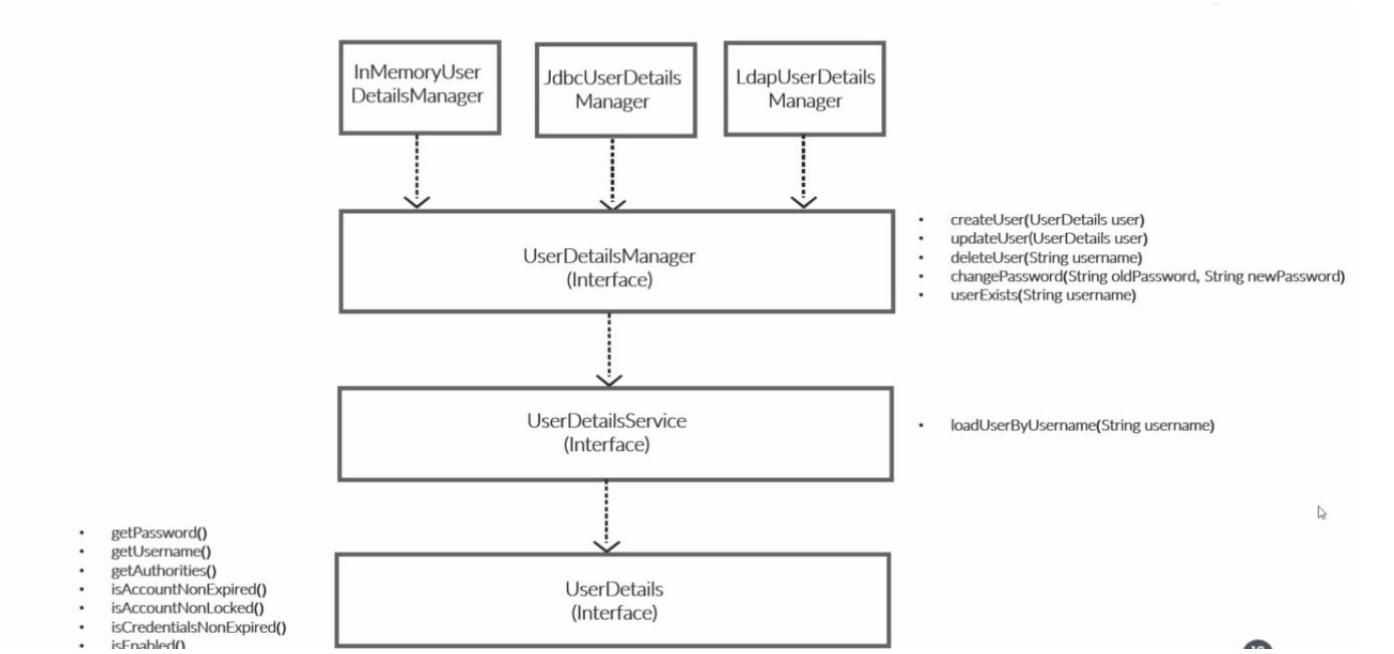


- ***AuthenticationProvider*** - The AuthenicationProvider is an interface with a single authenticate method.

4.2.1.1. *Spring WebSecurityConfigurerAdapter custom configuration.*

- ***In memory implemetation***

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    InMemoryUserDetailsManager userDetailService = new InMemoryUserDetailsManager();
    UserDetails admin = User.withUsername("admin")
                        .password("admin").authorities("admin").build();
    UserDetails user = User.withUsername("user")
                        .password("user").authorities("read").build();

    userDetailService.createUser(user);
    userDetailService.createUser(admin);
    auth.userDetailsService(userDetailService);
}
```



- ***JDBC Implementation.***

Pom.xml:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Web Security Config:

```java
@Bean
public PasswordEncoder encoder() { return NoOpPasswordEncoder.getInstance(); }

@Bean
public UserDetailsService userDetailsService(DataSource dataSource){
    return new JdbcUserDetailsManager(dataSource);
};
```

Application properties:

```
spring.security.user.name=admin
spring.security.user.password=admin12345
spring.datasource.url=jdbc:mysql://database-aws.czcwawrrzspo.us-east-2.rds.amazonaws.com/sandbox
spring.datasource.username=admin
spring.datasource.password=admin12345
spring.jpa.generate-ddl=false
spring.jpa.hibernate.ddl-auto=none
```

- ***Custom table for store credentials***

New entity objects

```java
@Entity
@Data
@NoArgsConstructor
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String email;
    private String pwd;
    private String role;
}
```

New Repositories

```java
@Repository
public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByEmail(String email);
}
```

UserDetails

```java
public class CustomerUserDetails implements UserDetails {
    private final Customer customer;
    public CustomerUserDetails(Customer customer) { this.customer = customer; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        authorities.add(new SimpleGrantedAuthority(customer.getRole()));
        return authorities;
    }

    @Override
    public String getPassword() { return customer.getPwd(); }

    @Override
    public String getUsername() { return customer.getEmail(); }
```

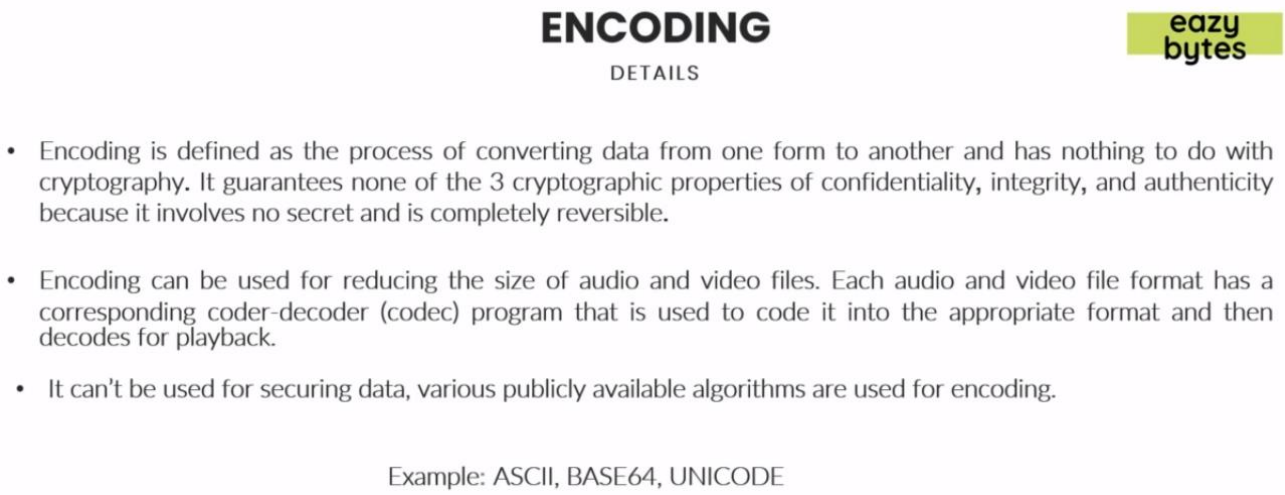
UserDetailsService:

```java
@Service
public class CustomerUserDetailsServices implements UserDetailsService {

    CustomerRepository customerRepository;
    public CustomerUserDetailsServices(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {
        List<Customer> customerList = customerRepository.findByEmail(s);
        if (customerList.size() == 0){
            throw new UsernameNotFoundException("User with email [" + s +"] not found");
        }
        return new CustomerUserDetails(customerList.get(0));
    }
}
```

Remove this bean

```java
@Bean
public UserDetailsService userDetailsService(DataSource dataSource){
    return new JdbcUserDetailsManager(dataSource);
};
```

- ***Encoding, hashing, encryption.***



ENCRYPTION

DETAILS

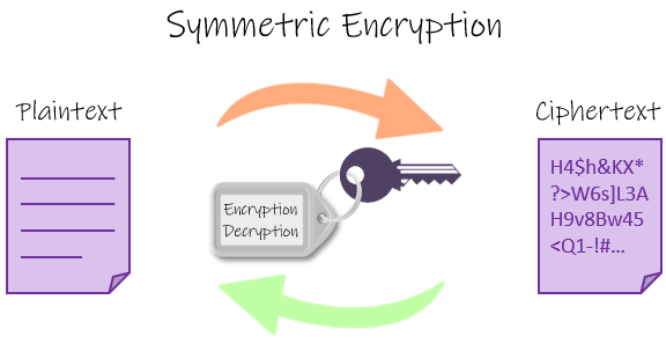
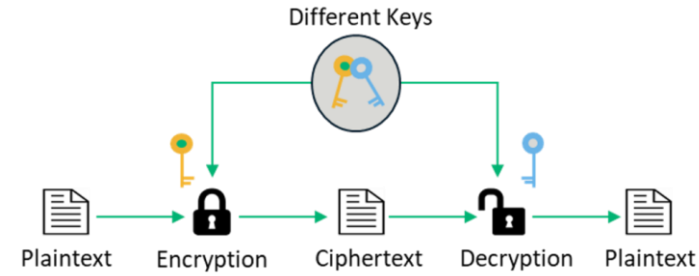
eazy bytes

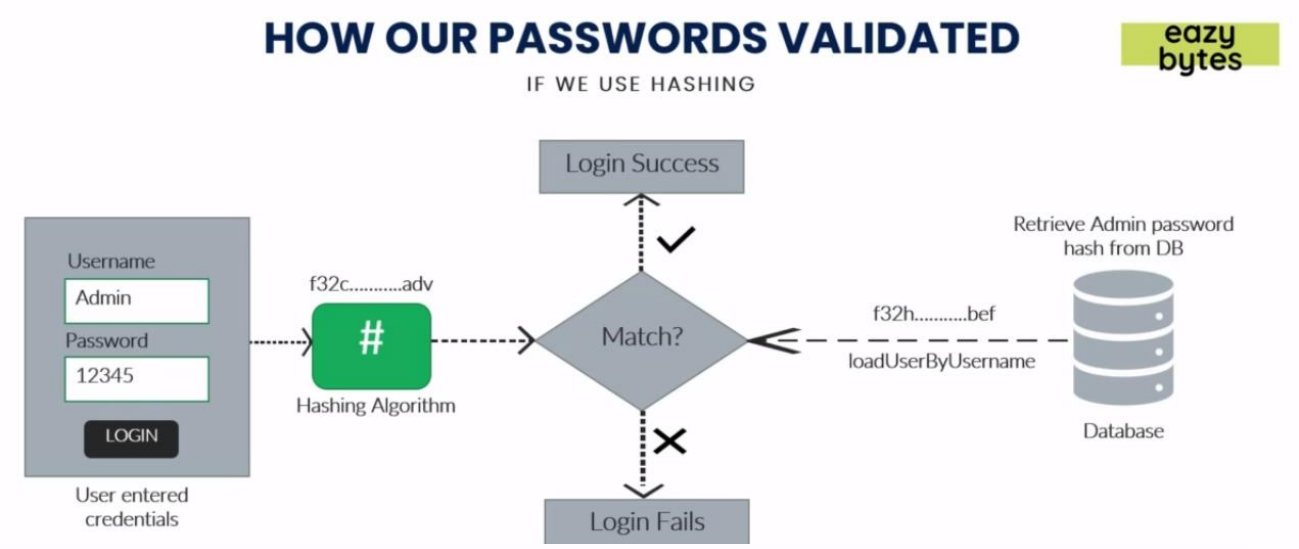
- Encryption is defined as the process of transforming data in such a way that guarantees confidentiality. To achieve that, encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- Encryption is divided into two categories: symmetric and asymmetric, where the major difference is the number of keys needed.
- In symmetric encryption algorithms, a single secret (key) is used to both encrypt and decrypt data. Only those who are authorized to access the data should have the single shared key in their possession.

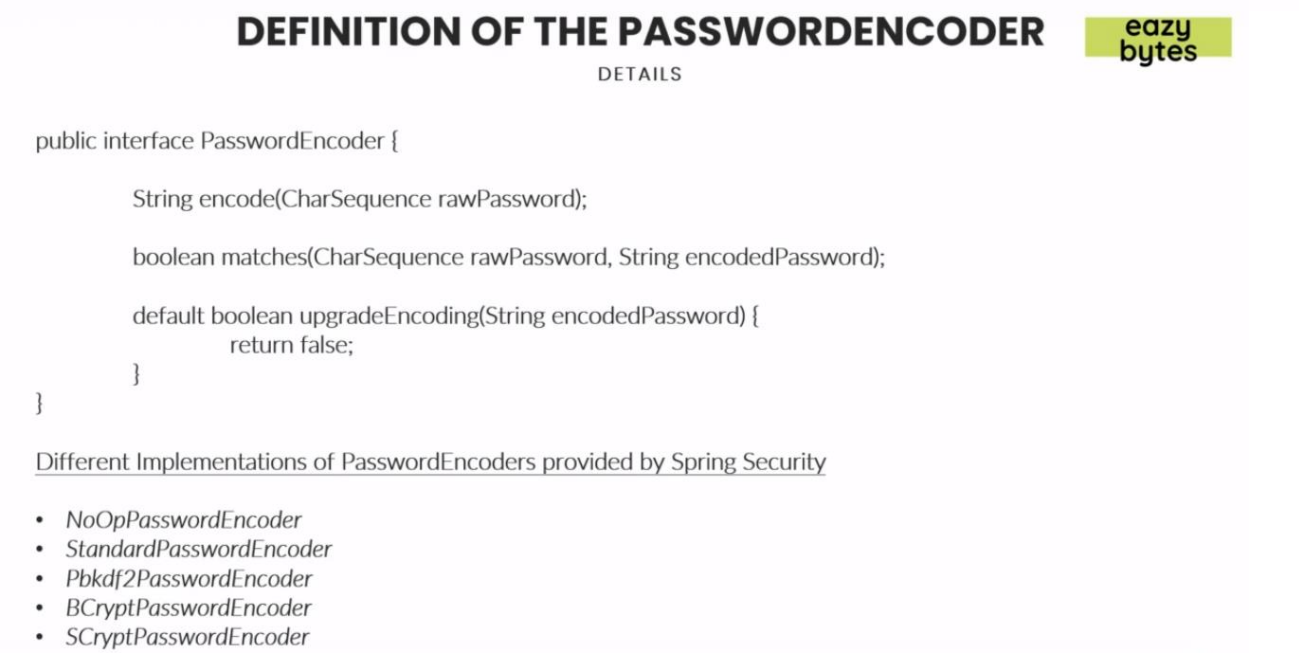
  Example: file system encryption, database encryption e.g. credit card details

- On the other hand, in asymmetric encryption algorithms, there are two keys in use: one public and one private. As their names suggest, the private key must be kept secret, whereas the public can be known to everyone. When applying encryption, the public key is used, whereas decrypting requires the private key. Anyone should be able to send us encrypted data, but only we should be able to decrypt and read it!

  Example: TLS, VPN, SSH

Symetric	Asymetric
	 **Blue Key is called private key, can be shared to anyone, to decrypt data.** **Yellow Key is called public key, need to keep secret.**





- ***Using Bcrypt for hashing password***

Password should store in DB in hash form:

customer 1 | Output

```
SELECT * from customer c
```

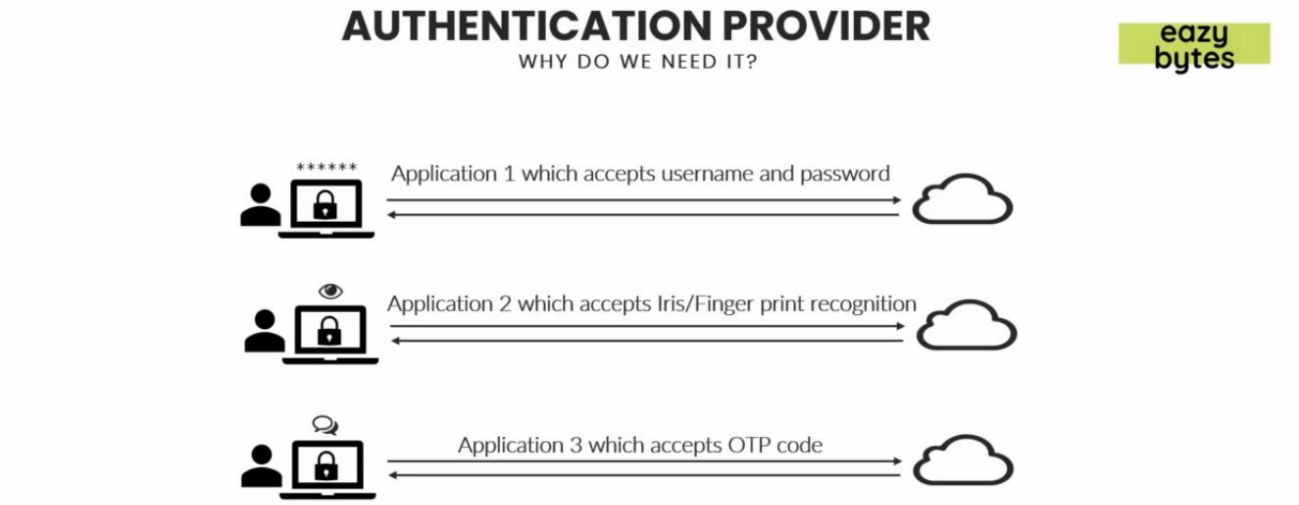
Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

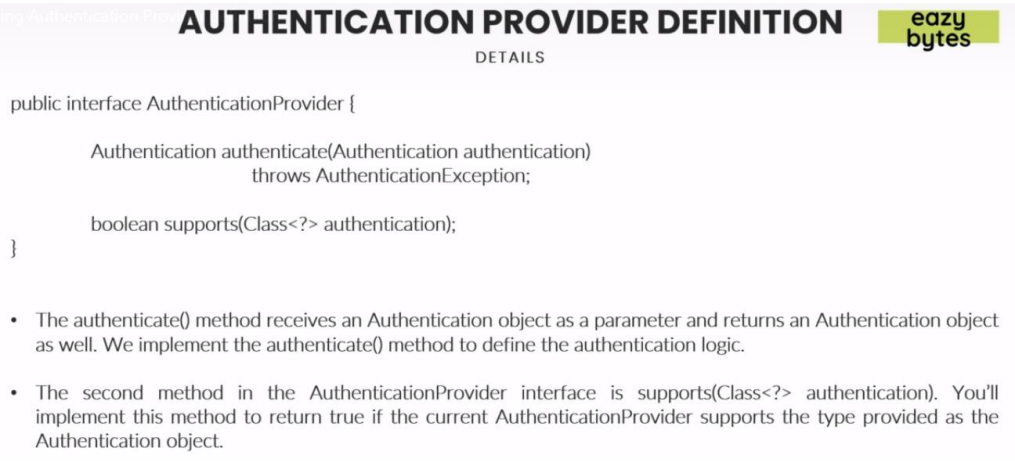
	id	email	pwd	role
1	2	user	$2a$12$tyyL36Rw.7DtabESaHUNyeFHGpFl04uNkSnE1gdgPzXKDTL7hIcLC	1
2	3	admin	$2a$12$nJ.6XqGvE.t02ceL0e1WEeX6jQQLN8ZlTg6jwE/t4QkZZ88t4J7Om	1

```java
@Bean
public PasswordEncoder encoder() { return new BCryptPasswordEncoder(); }
```

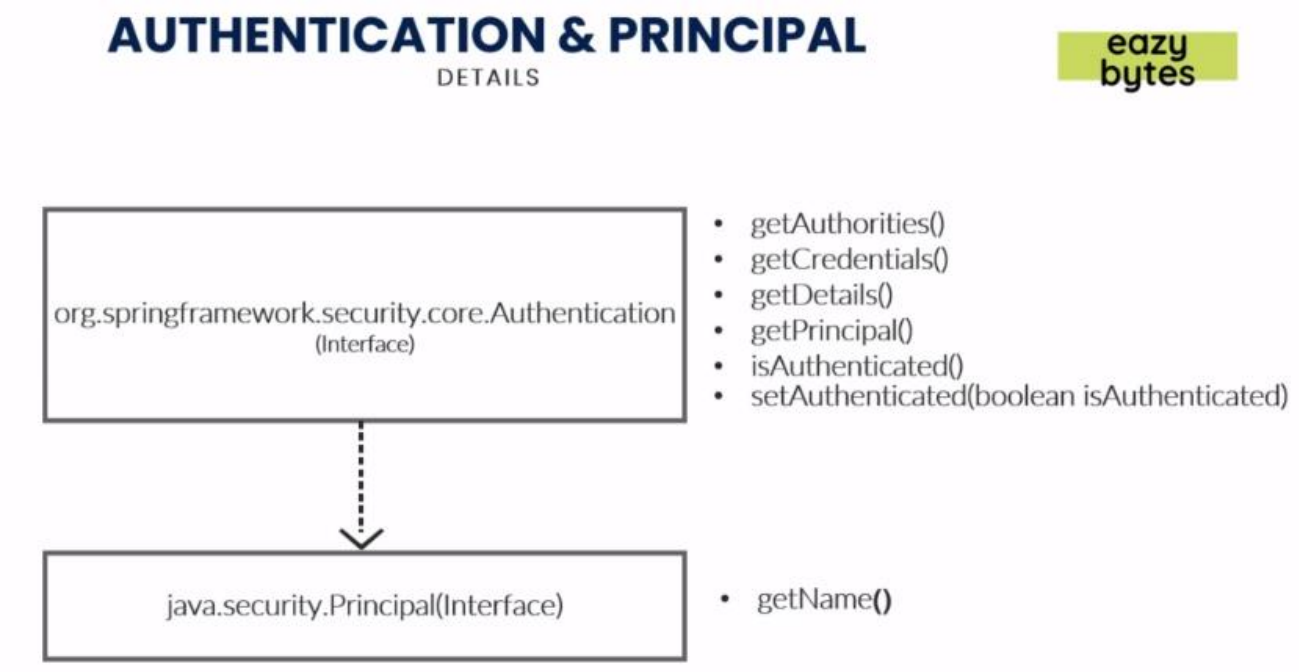
- ***Authentication Provider***



The AuthenticationProvider in Spring Security takes care of the authentication logic. The default implementation of the AuthenticationProvider delegates the responsibility of finding the user in the system to a UserDetailsService and PasswordEncoder for password validation. But if we have a custom authentication requirement that is not fulfilled by Spring Security framework then we can build our own authentication logic by implementing the AuthenticationProvider interface.



- The authenticate() method receives an Authentication object as a parameter and returns an Authentication object as well. We implement the authenticate() method to define the authentication logic.
- The second method in the AuthenticationProvider interface is supports(Class<?> authentication). You'll implement this method to return true if the current AuthenticationProvider supports the type provided as the Authentication object.



- ***CORS and CSRF***

CROSS-ORIGIN RESOURCE SHARING (CORS)

HOW TO HANDLE IT USING SPRING SECURITY

eazy bytes

1. CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin.

2. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

3. "other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

    - a different scheme (HTTP or HTTPS)
    - a different domain
    - a different port

CROSS-ORIGIN RESOURCE SHARING (CORS)

HOW TO HANDLE IT USING SPRING SECURITY

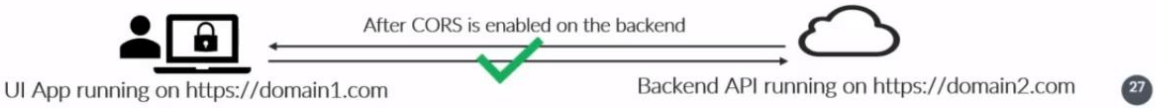
eazy bytes

4. But what if there is a legitimate scenario where cross-origin access is desirable or even necessary. For example, in our EazyBank application where the UI and backend are hosted on two different ports.

5. When a server has been configured correctly to allow cross-origin resource sharing, some special headers will be included. Their presence can be used to determine that a request supports CORS. Web browsers can use these headers to determine whether a request should continue or fail.

6. First the browser sends a pre-flight request to the backend server to determine whether it supports CORS or not. The server can then respond to the pre-flight request with a collection of headers:

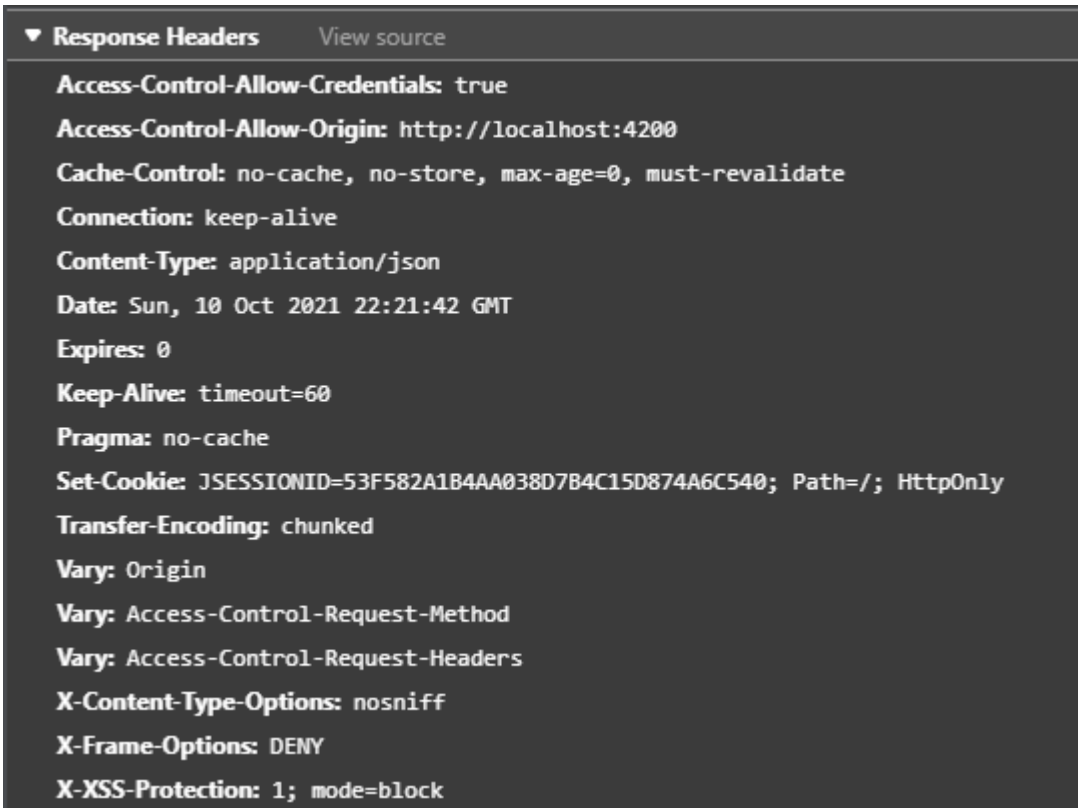
    - *Access-Control-Allow-Origin: Defines which origins may have access to the resource. A '*' represents any origin*
    - *Access-Control-Allow-Methods: Indicates the allowed HTTP methods for cross-origin requests*
    - *Access-Control-Allow-Headers: Indicates the allowed request headers for cross-origin requests*
    - *Access-Control-Allow-Credentials : Indicates whether or not the response to the request can be exposed when the credentials flag is true.*
    - *Access-Control-Max-Age: Defines the expiration time of the result of the cached preflight request*

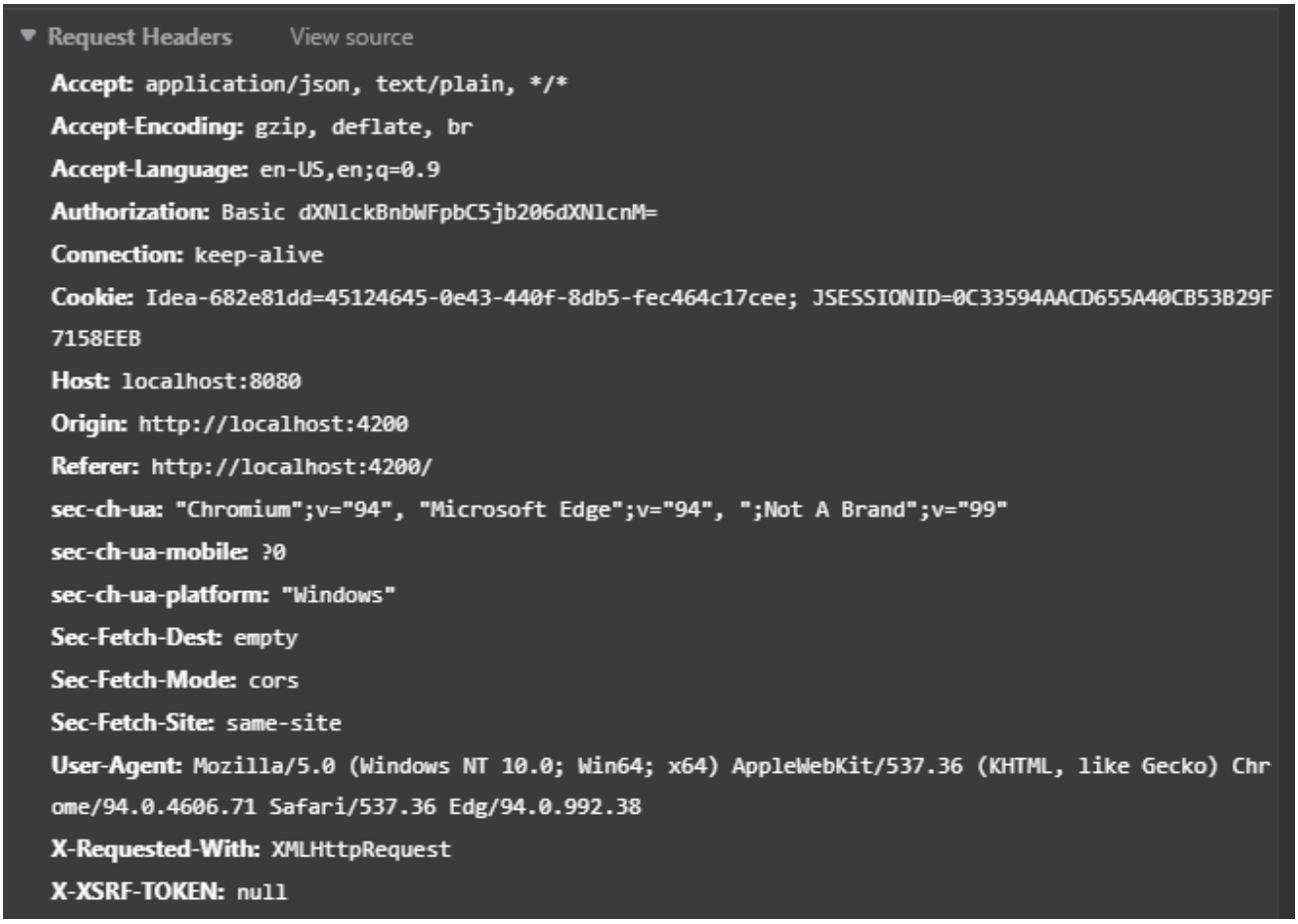


27

Spring confuguration:

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().configurationSource(request -> {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Collections.singletonList("http://localhost:4200/"));
        configuration.setAllowedMethods(Collections.singletonList("*"));
        configuration.setAllowedHeaders(Collections.singletonList("*"));
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L );
        return configuration;
    }) CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .authenticated() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .formLogin() FormLoginConfigurer<HttpSecurity>
            .and() HttpSecurity
            .httpBasic();
};
```





Type	
Request	GET /data HTTP/1.1 Host: robust-website.com **Origin**: https://normal-website.com
Response	HTTP/1.1 200 OK ... **Access-Control-Allow-Origin**: https://normal-website.com

Adding credentials	`GET /data HTTP/1.1`
	`Host: robust-website.com`
	`...`
	`Origin``: https://normal-website.com`
	`Cookie``: JSESSIONID=<value>`
	
	HTTP/1.1 200 OK
	...
	Access-Control-Allow-Origin: https://normal-website.com
	Access-Control-Allow-Credentials: true
	
	Access-Control-Allow-Origin: * wild card do not allow Access-Control-Allow-Credentials

► *CSRF Attack*

CROSS-SITE REQUEST FORGERY (CSRF)

HOW TO HANDLE IT USING SPRING SECURITY

eazy bytes

1. A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.

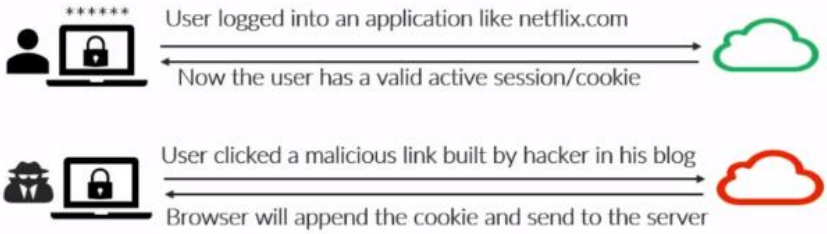
2. Consider a website netflix.com and the attacker's website travelblog.com. Also assume that the victim is logged in and his session is being maintained by cookies. The attacker will:

   - Find out what action he needs to perform on behalf of the victim and find out its endpoint (for example, to change password on netflix.com a POST request is made to the website that contains new password as the parameter)
   - Place HTML code on his website travelblog.com that will imitate a legal request to netflix.com (for example, a form with method as post and a hidden input field that contains the new password).
   - Make sure that the form is submitted by either using "autosubmit" or luring the victim to click on a submit button.

CROSS-SITE REQUEST FORGERY (CSRF)

HOW TO HANDLE IT USING SPRING SECURITY

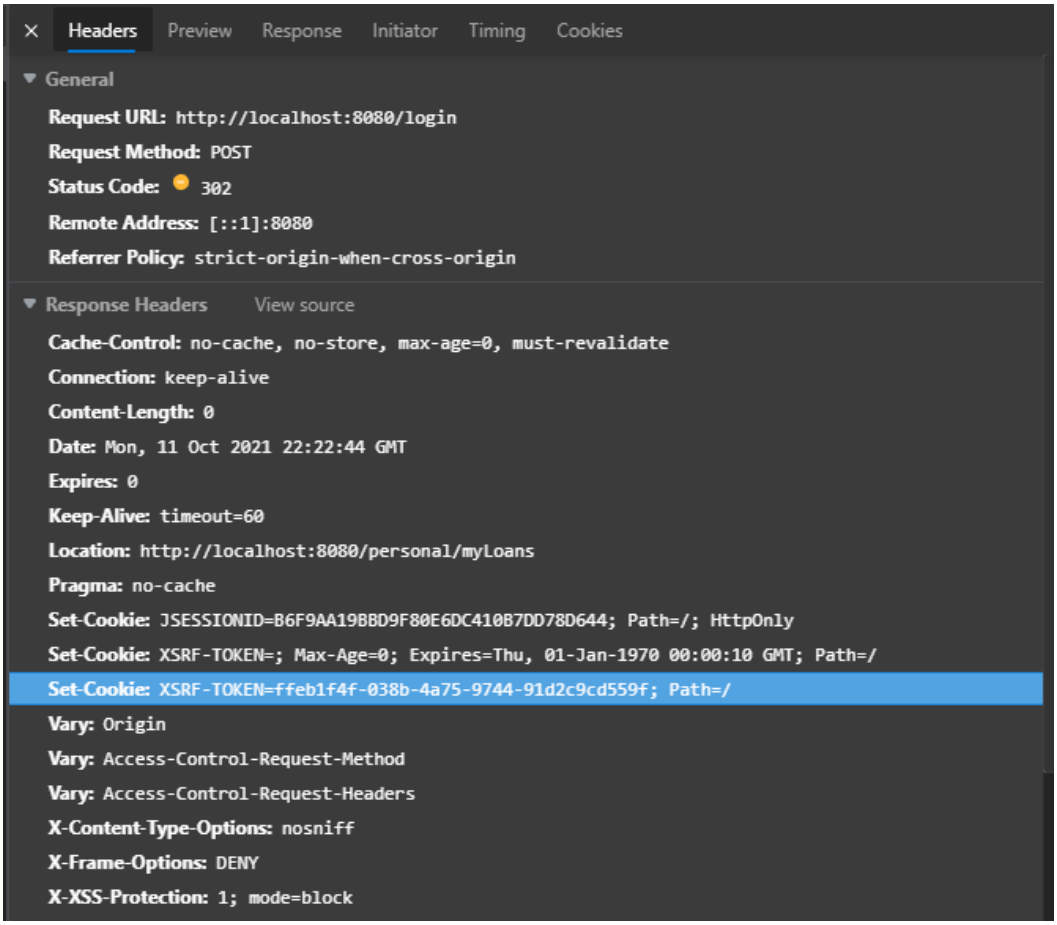
eazy bytes



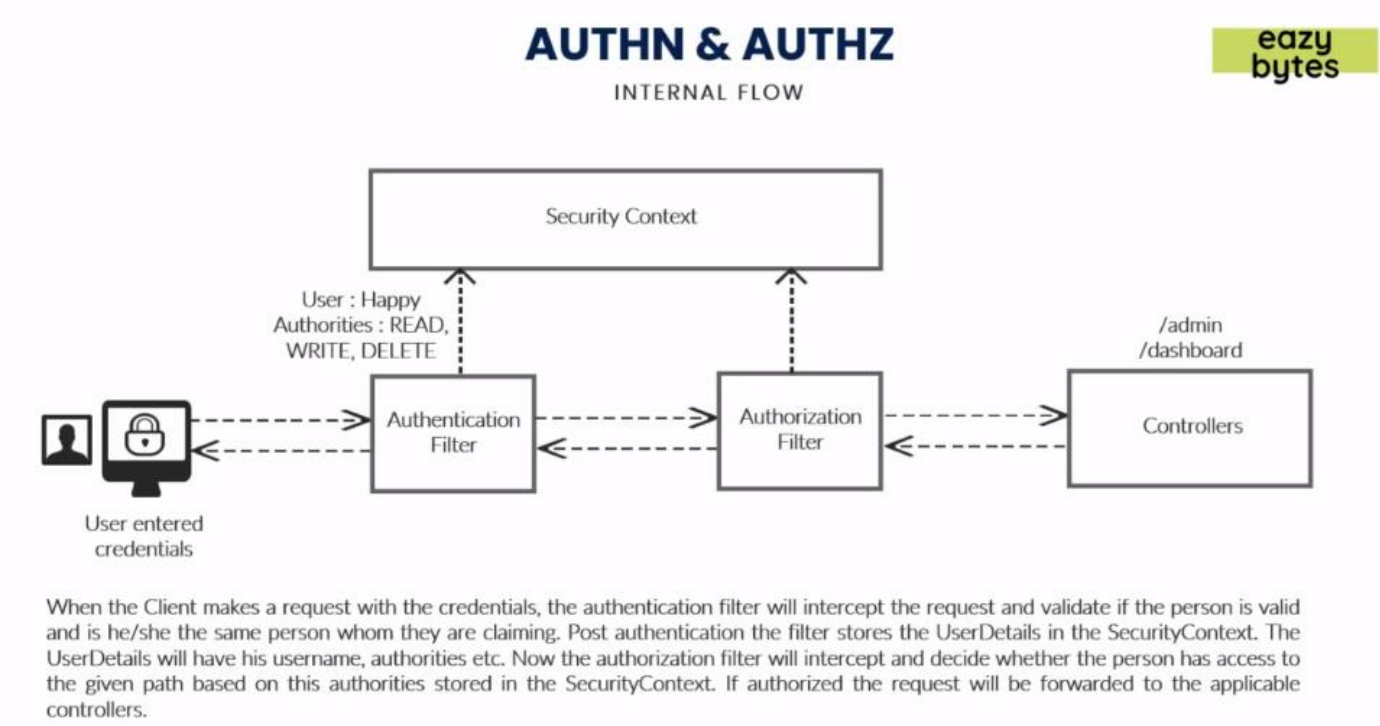
- When the victim visits travelblog.com and that form is submitted, the victim's browser makes a request to netflix.com for a password change. Also the browser appends the cookies with the request. The server treats it as a genuine request and resets the victim's password to the attacker's supplied value. This way the victim's account gets taken over by the attacker.

- There are many proposed ways to implement CSRF protection on server side, among which the use of CSRF tokens is most popular. A CSRF token is a string that is tied to a user's session but is not submitted automatically. A website proceeds only when it receives a valid CSRF token along with the cookies, since there is no way for an attacker to know a user specific token, the attacker can not perform actions on user's behalf.

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().configurationSource(request -> {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Collections.singletonList("http://localhost:4200/"));
        configuration.setAllowedMethods(Collections.singletonList("*"));
        configuration.setAllowedHeaders(Collections.singletonList("*"));
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L );
        return configuration;
    }) CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .csrf() CsrfConfigurer<HttpSecurity>
            .ignoringAntMatchers("/public/**")
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and() HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .authenticated() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .formLogin() FormLoginConfigurer<HttpSecurity>
            .and() HttpSecurity
            .httpBasic();
};
```



► *Spring Authorities and Roles.*



When the Client makes a request with the credentials, the authentication filter will intercept the request and validate if the person is valid and is he/she the same person whom they are claiming. Post authentication the filter stores the UserDetails in the SecurityContext. The UserDetails will have his username, authorities etc. Now the authorization filter will intercept and decide whether the person has access to the given path based on this authorities stored in the SecurityContext. If authorized the request will be forwarded to the applicable controllers.

Inside UserDetails which is a contract of the User inside the Spring Security, the authorities will be stored in the form of Collection of GrantedAuthority. These authorities can be fetched using the method getAuthorities()

```
public interface UserDetails {
        Collection<? extends GrantedAuthority> getAuthorities();
}
```

Inside GrantedAuthority interface we have a getAuthority() method which will return the authority/role name in the form of a string. Using this value the framework will try to validate the authorities of the user with the implementation of the application.

```
public interface GrantedAuthority {
        String getAuthority();
}
```

CONFIGURING AUTHORITIES

IN SPRING SECURITY

eazy bytes

In Spring Security the authorities of the user can be configured and validated using the following ways,

- hasAuthority() —Accepts a single authority for which the endpoint will be configured and user will be validated against the single authority mentioned. Only users having the same authority configured can call the endpoint.
- hasAnyAuthority() — Accepts multiple authorities for which the endpoint will be configured and user will be validated against the authorities mentioned. Only users having any of the authority configured can call the endpoint.
- access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring authorities which are not possible with the above methods. We can use operators like OR, AND inside access() method.

```
@Override
public Authentication authenticate(Authentication authentication) {
    String username = authentication.getName();
    String pwd = authentication.getCredentials().toString();
    List<Customer> customer = customerRepository.findByEmail(username);
    if (customer.size() > 0) {
        if (passwordEncoder.matches(pwd, customer.get(0).getPwd())) {
            return new UsernamePasswordAuthenticationToken(username, pwd,
                    getGrantedAuthorities(customer.get(0)));
        } else {
            throw new BadCredentialsException("Invalid password!");
        }
    } else {
        throw new BadCredentialsException("No user registered with this details!");
    }
}
```

```
private List<GrantedAuthority> getGrantedAuthorities(Customer customer){
    List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
    List<Authority> authorityList = authorityRepository.findByCustomer(customer);

    for (Authority authority : authorityList) {
        grantedAuthorities.add(new SimpleGrantedAuthority(authority.getName()));
    }
    return grantedAuthorities;
}
```

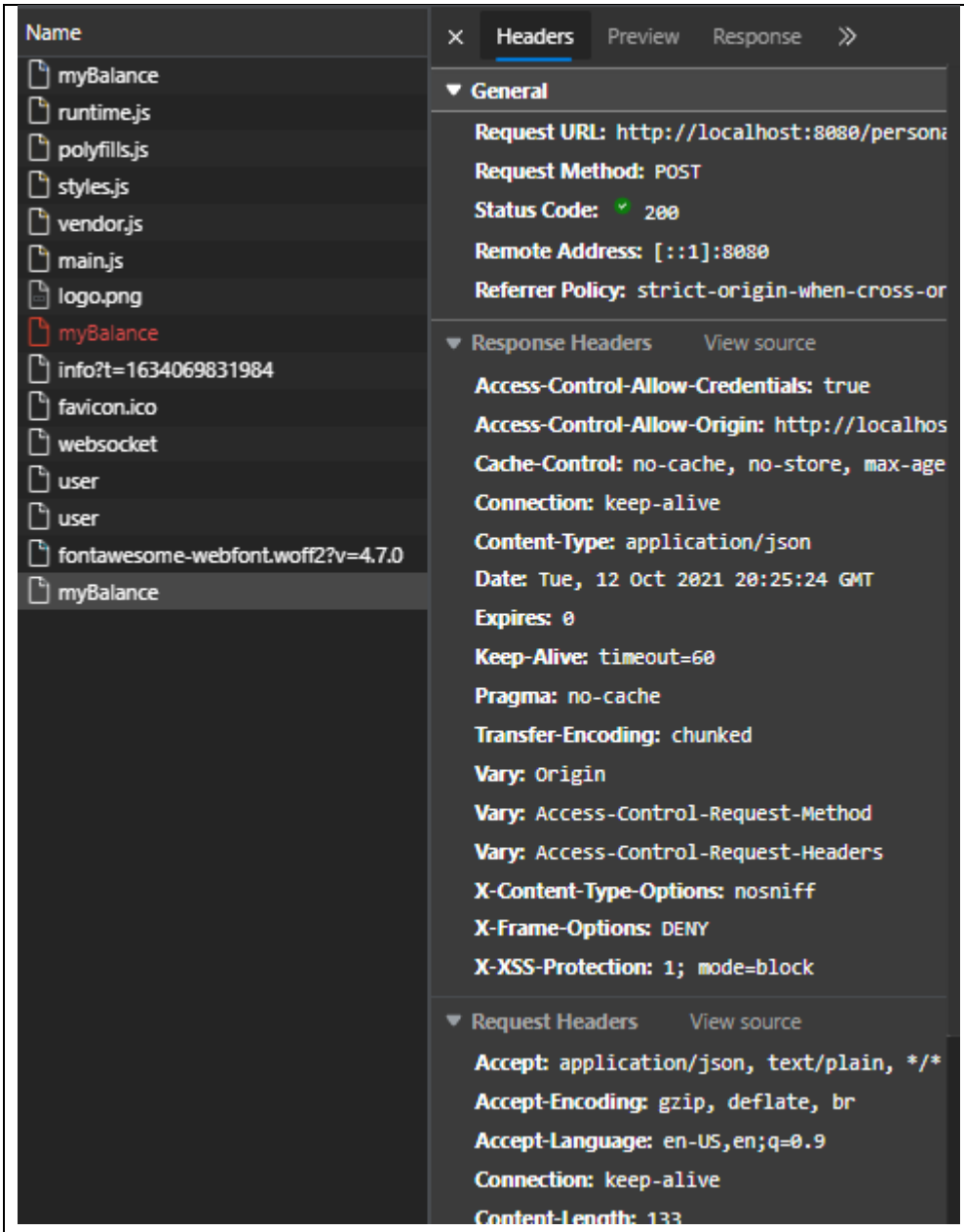
In case of success:

```
select * from authorities a
```

Enter a SQL expression t

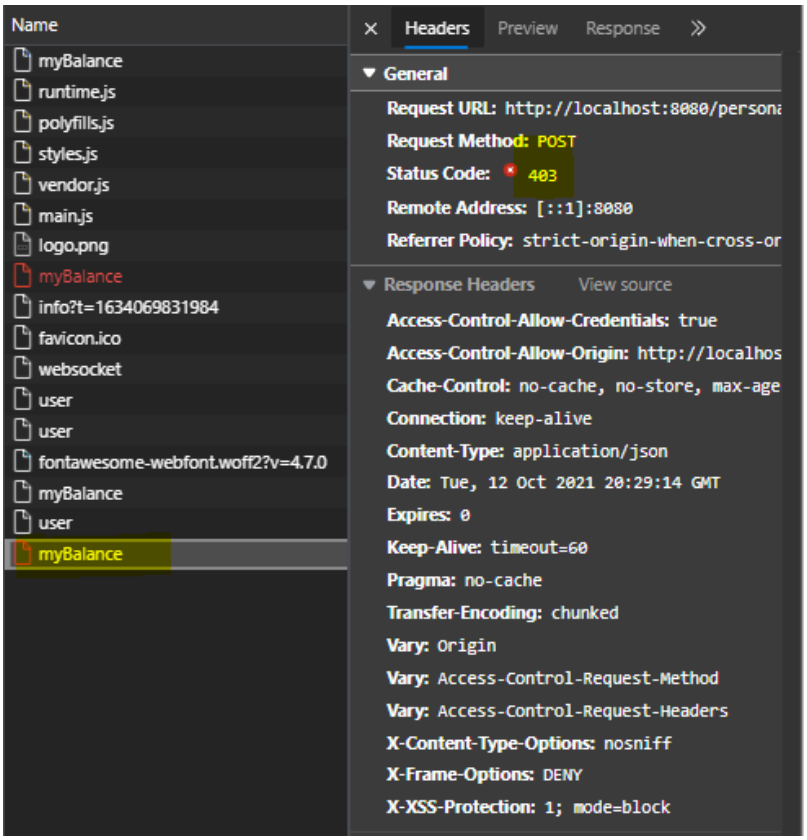
	id	customer_id	name
1	1	2	READ
2	2	2	WRITE
3	3	3	READ
4	4	3	WRITE

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().configurationSource(request -> {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Collections.singletonList("http://localhost:4200/"));
        configuration.setAllowedMethods(Collections.singletonList("*"));
        configuration.setAllowedHeaders(Collections.singletonList("*"));
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L );
        return configuration;
    }) CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .csrf() CsrfConfigurer<HttpSecurity>
            .ignoringAntMatchers("/public/**")
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and() HttpSecurity
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .access( attribute: "hasAuthority('WRITE') && hasAuthority('READ')") ExpressionUrlAuthorizationConfigure
            .antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
            .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .formLogin() FormLoginConfigurer<HttpSecurity>
            .and() HttpSecurity
            .httpBasic();
};
```

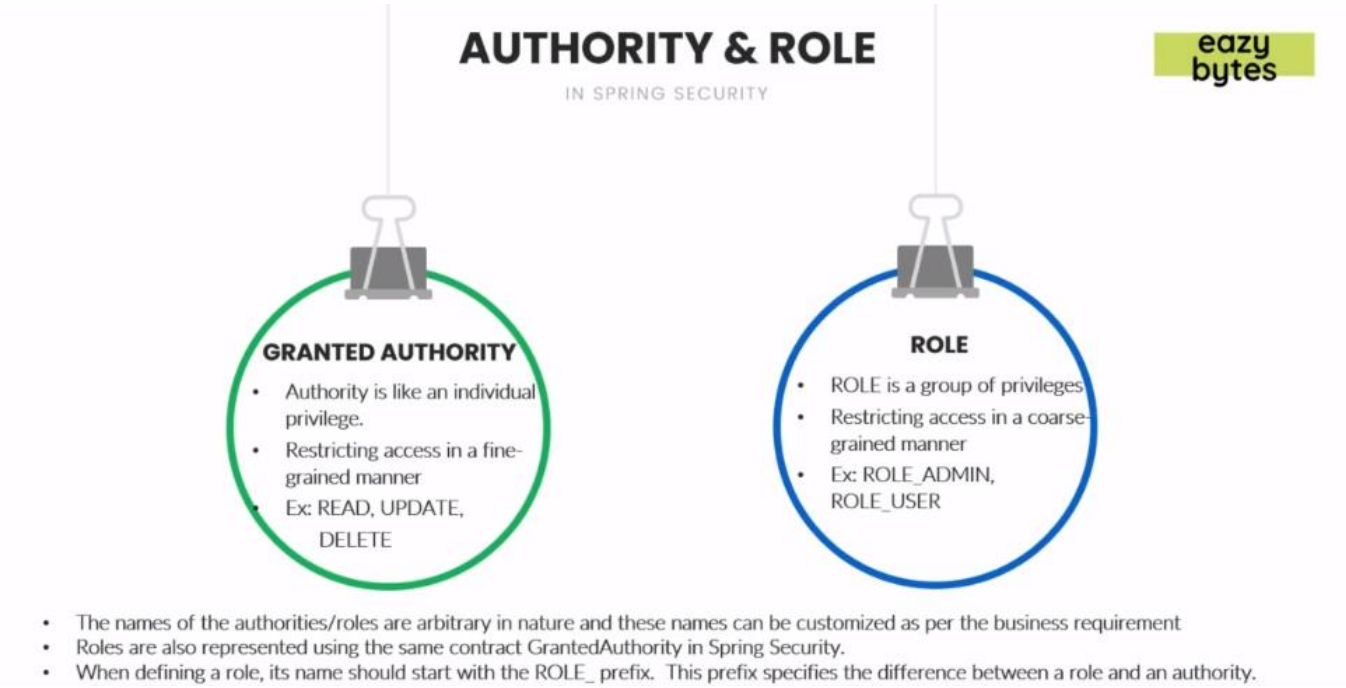

Name
myBalance
runtime.js
polyfills.js
styles.js
vendor.js
main.js
logo.png
myBalance
info?t=1634069831984
favicon.ico
websocket
user
user
fontawesome-webfont.woff2?v=4.7.0
myBalance
Headers
Preview
Response
General
Request URL: http://localhost:8080/persona
Request Method: POST
Status Code: 200
Remote Address: [::1]:8080
Referrer Policy: strict-origin-when-cross-or
Response Headers
View source
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: http://localhos
Cache-Control: no-cache, no-store, max-age
Connection: keep-alive
Content-Type: application/json
Date: Tue, 12 Oct 2021 20:25:24 GMT
Expires: 0
Keep-Alive: timeout=60
Pragma: no-cache
Transfer-Encoding: chunked
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
Request Headers
View source
Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Content-Length: 133

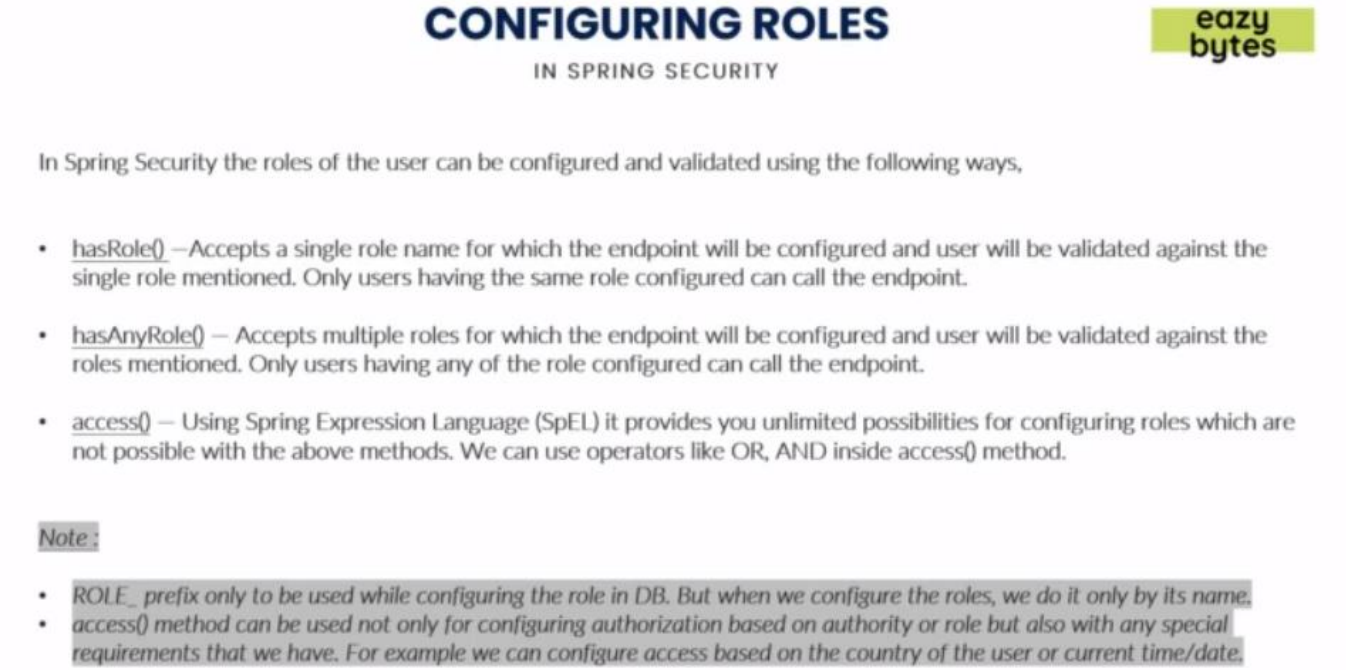
In case of failure (unauthorized)

```
CorsConfigurer<HttpSecurity>
    .and() HttpSecurity
    .csrf() CsrfConfigurer<HttpSecurity>
    .ignoringAntMatchers("/public/**")
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .and() HttpSecurity
    .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegist
    .antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>
    .access( attribute: "hasAuthority('UPDATE')") ExpressionUrlAuthorizationConfigurer<
    .antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.Au
    .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
    .and() HttpSecurity
    .formLogin() FormLoginConfigurer<HttpSecurity>
    .and() HttpSecurity
    .httpBasic();
```



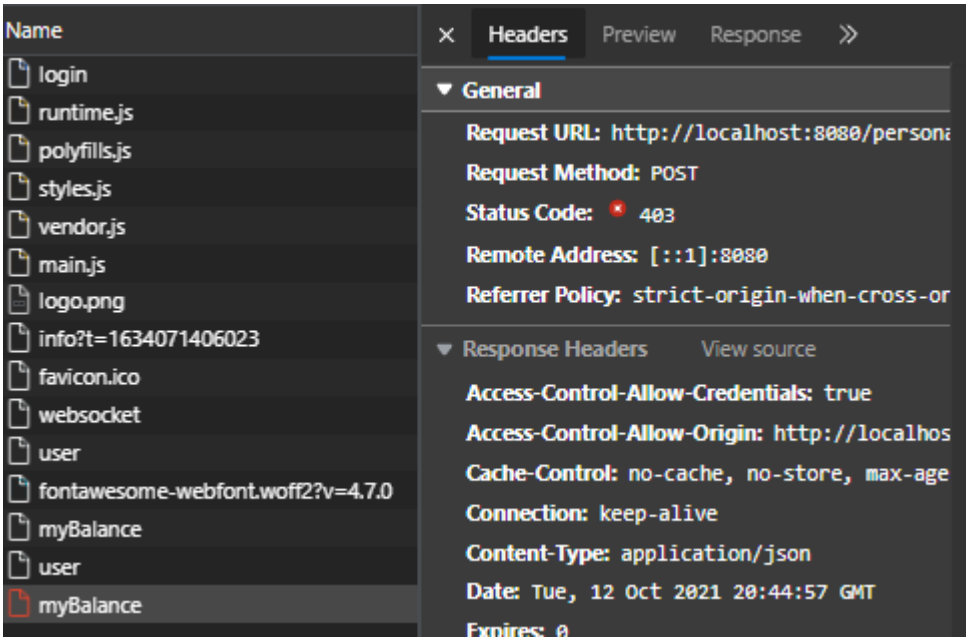
- ***Roles and authorization.***





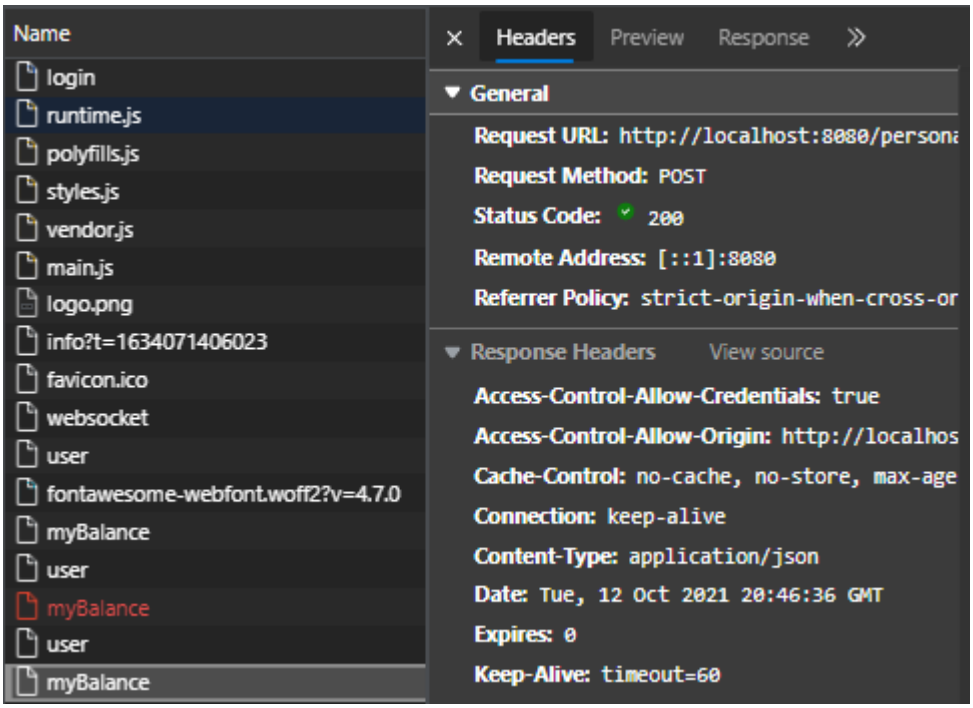
```
.and() HttpSecurity
.csrf() CsrfConfigurer<HttpSecurity>
.ignoringAntMatchers("/public/**")
.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
.and() HttpSecurity
.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistr
.antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>.
.access( attribute: "hasRole('TESTER')") ExpressionUrlAuthorizationConfigurer<H>.Expr
.antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.Aut
.permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
.and() HttpSecurity
.formLogin() FormLoginConfigurer<HttpSecurity>
.and() HttpSecurity
.httpBasic();
```

	id	customer_id	name
1	1	2	ROLE_ADMIN
2	2	2	ROLE_USER
3	3	3	ROLE_ADMIN
4	4	3	ROLE_USER



Allow role:

```
.and() HttpSecurity
.csrf() CsrfConfigurer<HttpSecurity>
.ignoringAntMatchers("/public/**")
.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
.and() HttpSecurity
.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegist
.antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>
.access( attribute: "hasRole('USER')") ExpressionUrlAuthorizationConfigurer<H>.Expres
.antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.Au
.permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
.and() HttpSecurity
.formLogin() FormLoginConfigurer<HttpSecurity>
.and() HttpSecurity
```



► *Ant matcher:*

MATCHERS METHODS

IN SPRING SECURITY

eazy bytes

Spring Security offers three types of matchers methods to configure endpoints security,

1) MVC matchers, 2) Ant matchers, 3) Regex matchers

- MVC matchers

MvcMatcher() uses Spring MVC's HandlerMappingIntrospector to match the path and extract variables.

- *mvcMatchers(HttpMethod method, String... patterns)– We can specify both the HTTP method and path pattern to configure restrictions*

```
http.authorizeRequests().mvcMatchers(HttpMethod.POST, "/example").authenticated()
.mvcMatchers(HttpMethod.GET, "/example").permitAll()
.anyRequest().denyAll();
```

- *mvcMatchers(String... patterns)–We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().mvcMatchers( "/profile/edit/**").authenticated()
.anyRequest().permitAll();
```

Note :

- *** indicates any number of paths. For example, /x/**/z will match both /x/y/z and /x/y/abc/z*
- *Single * indicates single path. For example /x/*/z will match /x/y/z, /x/abc/z but not /x/y/abc/z*

37

MATCHERS METHODS

IN SPRING SECURITY

eazy bytes

- ANT matchers is an implementation for Ant-style path patterns. Part of this mapping code has been kindly borrowed from Apache Ant.

- *antMatchers(HttpMethod method, String... patterns)– We can specify both the HTTP method and path pattern to configure restrictions*

```
http.authorizeRequests().antMatchers(HttpMethod.POST, "/example").authenticated()
.antMatchers(HttpMethod.GET, "/example").permitAll()
.anyRequest().denyAll();
```

- *antMatchers(String... patterns)–We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().antMatchers( "/profile/edit/**").authenticated()
.anyRequest().permitAll();
```

- *antMatchers(HttpMethod method)–We can specify only HTTP method ignoring path pattern to configure restrictions. This is same as antMatchers(httpMethod, "/**")*

```
http.authorizeRequests().antMatchers(HttpMethod.POST).authenticated()
.anyRequest().permitAll();
```

Note : Generally mvcMatcher is more secure than an antMatcher. As an example

- *antMatchers("/secured") matches only the exact /secured URL*
- *mvcMatchers("/secured") matches /secured as well as /secured/, /secured.html, /secured.xyz*

► *Spring filter*

FILTERS IN SPRING SECURITY

eazy bytes

IN AUTHN & AUTHZ FLOW

- Lot of times we will have situations where we need to perform some house keeping activities during the authentication and authorization flow. Few such examples are,
    - Input validation
    - Tracing, Auditing and reporting
    - Logging of input like IP Address etc.
    - Encryption and Decryption
    - Multi factor authentication using OTP
- All such requirements can be handled using HTTP Filters inside Spring Security. Filters are servlet concepts which are leveraged in Spring Security as well.
- We already saw some in built filters of Spring security framework like Authentication filter, Authorization filter, CSRF filter, CORS filter in the previous sections.
- A filter is a component which receives requests, process its logic and handover to the next filter in the chain.
- Spring Security is based on a chain of servlet filters. Each filter has a specific responsibility and depending on the configuration, filters are added or removed. We can add our custom filters as well based on the need.

FILTERS IN SPRING SECURITY

eazy bytes

IN AUTHN & AUTHZ FLOW

- We can always check the registered filters inside Spring Security with the below configurations,
    1. @EnableWebSecurity(debug = true) – We need to enable the debugging of the security details
    2. Enable logging of the details by adding the below property in application.properties
       logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
- Below are the some of the internal filters of Spring Security that gets executed in the authentication flow,

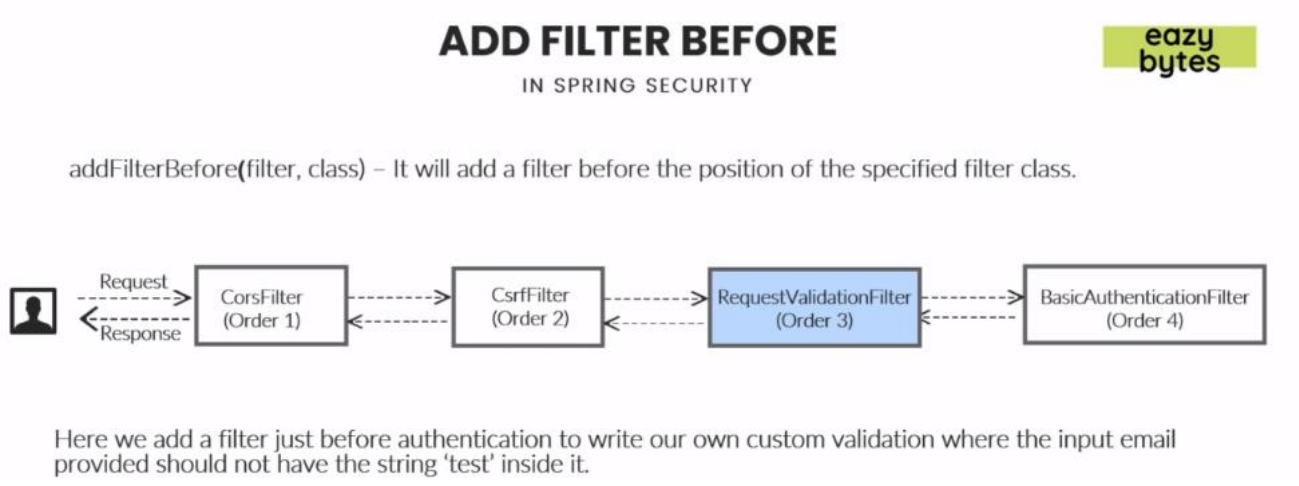
```
Security filter chain: [
  WebAsyncManagerIntegrationFilter
  SecurityContextPersistenceFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  FilterSecurityInterceptor
]
```

Add spring filter chain debug:

```java
@SpringBootApplication
@EnableWebSecurity(debug = true)
public class LocalApplication {
    public static void main(String[] args) { SpringApplication.run(LocalApplication.class, args); }
}
```

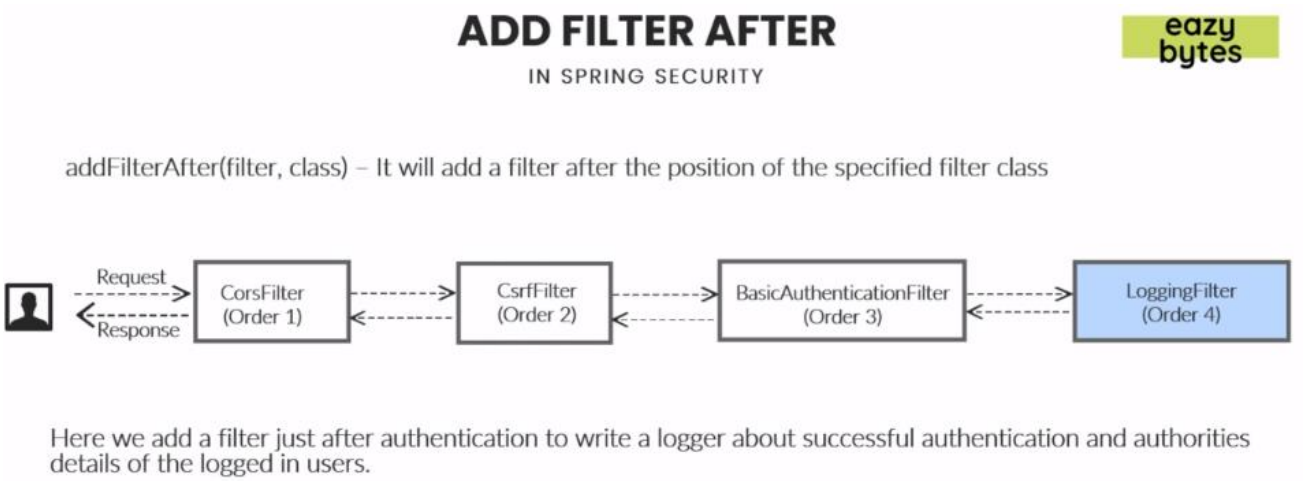
```
logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
```

- ***Customize filter***




```
orsConfigurer<HttpSecurity>
    .and() HttpSecurity
    .csrf() CsrfConfigurer<HttpSecurity>
    .ignoringAntMatchers("/public/**")
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .and() HttpSecurity
    .addFilterBefore(new PreAuthCustomFilter(), BasicAuthenticationFilter.class)
    .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
    .antMatchers( ...antPatterns: "/personal/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
    .access( attribute: "hasRole('USER')") ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
    .antMatchers( ...antPatterns: "/public/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
    .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
    .and() HttpSecurity
    .formLogin() FormLoginConfigurer<HttpSecurity>
    .and() HttpSecurity
    .httpBasic();
```

```java
@Slf4j
public class PreAuthCustomFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest,
                         ServletResponse servletResponse,
                         FilterChain filterChain) throws IOException, ServletException {
        log.debug("Processing PreAuthCustomFilter ...");
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        String header = request.getHeader(AUTHORIZATION);

        if(header != null){
            header = header.trim();
            if (StringUtils.startsWithIgnoreCase(header, "Basic")){
                byte[] base64Token = header.substring(6).getBytes(StandardCharsets.UTF_8);
                log.debug("base64Token {}", base64Token);
                byte[] decoded;

                try {
                    decoded = Base64.getDecoder().decode(base64Token);
                    String token = new String(decoded, StandardCharsets.UTF_8);
                    int delim = token.indexOf(":");
                    if (delim == -1){
                        throw new BadCredentialsException("In valid authentication token");
                    }
                    String email = token.substring(0, delim);
                    log.debug("email {}", email);

                    if (email.toLowerCase().contains("test")){
                        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                        return;
                    }
                } catch (IllegalArgumentException exception){
                    throw new BadCredentialsException("Failed to decode basic authentication token");
                }
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

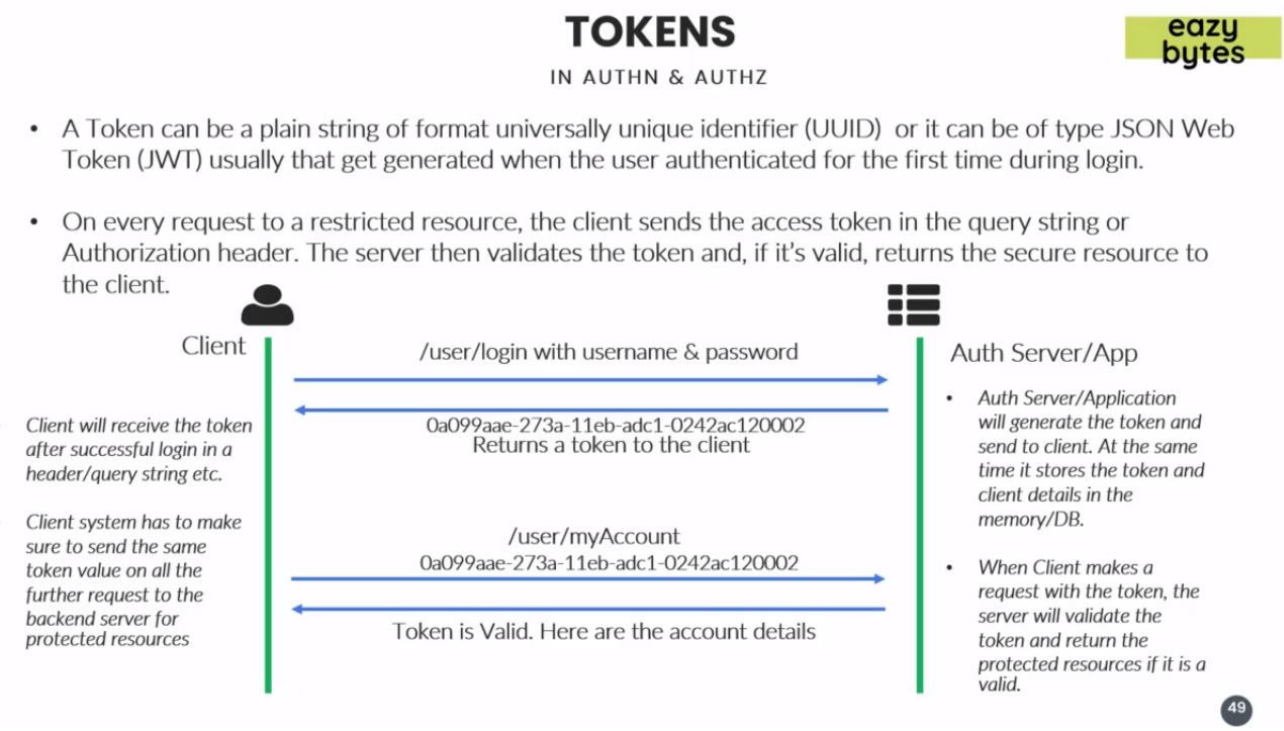



```java
public class PostAuthCustomFilter implements Filter {

    private final Logger LOG = Logger.getLogger(this.getClass().getSimpleName());

    @Override
    public void doFilter(ServletRequest servletRequest,
                         ServletResponse servletResponse,
                         FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
```

```
        Authentication authentication = SecurityContextHolder
                                        .getContext()
                                        .getAuthentication();
        if (authentication != null){
            LOG.info("User "
                    + authentication.getName()
                    + "\n authenticated getPrincipal "
                    + authentication.getPrincipal().toString()
                    + "\n authenticated getAuthorities "
                    + authentication.getAuthorities().toString()
                    + "\n authenticated getCredentials "
                    + authentication.getCredentials()
                    + "\n authenticated getPrincipal "
                    + authentication.getPrincipal());
        }

        filterChain.doFilter(request, response);
    }
}
```

OncePerRequestFilter: filter only execute once each request.

- ***Tokens in authentication flows***



TOKENS

IN AUTHN & AUTHZ

eazy bytes

Advantages of Token based Authentication

- Token helps us not to share the credentials for every request which is a security risk to make credentials send over the network frequently.
- Tokens can be invalidated during any suspicious activities without invalidating the user credentials.
- Tokens can be created with a short life span.
- Tokens can be used to store the user related information like roles/authorities etc.
- Reusability - We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.

TOKENS

IN AUTHN & AUTHZ

eazy bytes

Advantages of Token based Authentication

- Security - Since we are not using cookies, we don't have to protect against cross-site request forgery (CSRF) attacks.
- Stateless, easier to scale. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.
- We already used tokens in the previous sections in the form of CSRF and JSESSIONID tokens.

  1. CSRF Token protected our application from CSRF attacks.
  2. JSESSIONID is the default token generated by the Spring Security which helped us not to share the credentials to the backend every time.

JWT

TOKEN DETAILS

eazy bytes

- JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
- JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.
- JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.
- A JWT token has 3 parts each separated by a dot(.). Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

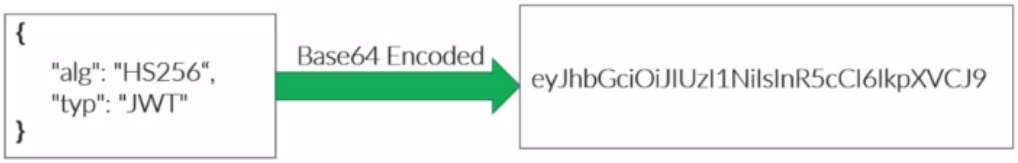
1. Header
2. Payload
3. Signature (Optional)

JWT

TOKEN DETAILS

eazy bytes

- JWTs have three parts: a header, a payload, and a signature.
- In the header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.

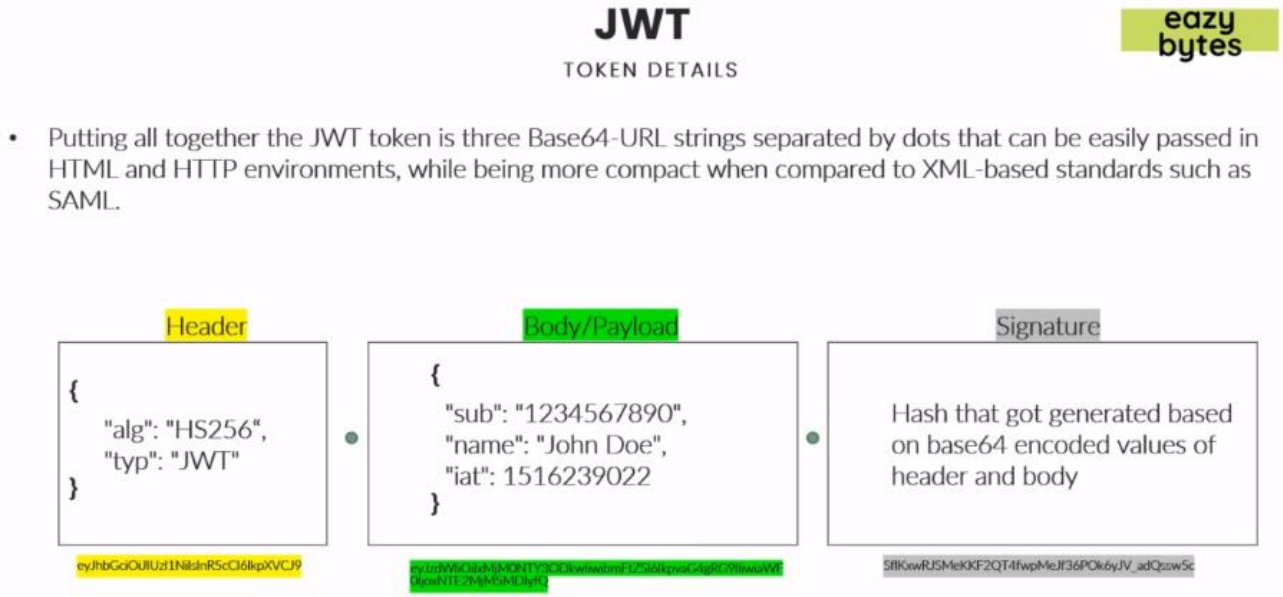


JWT

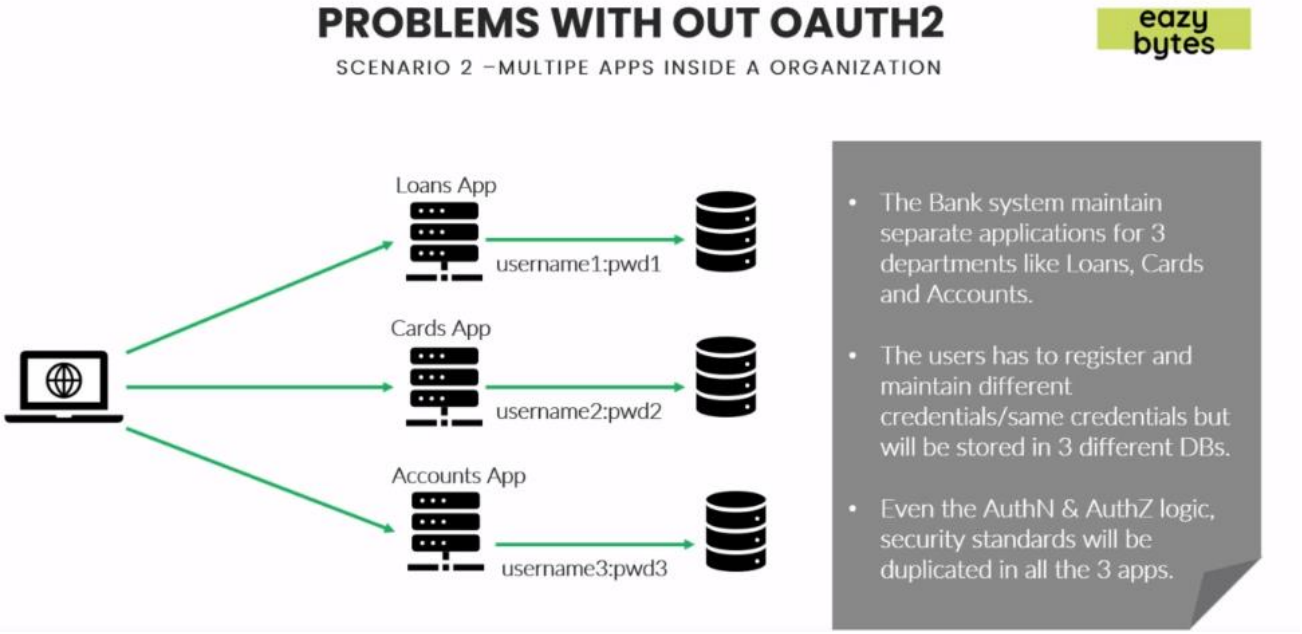
TOKEN DETAILS

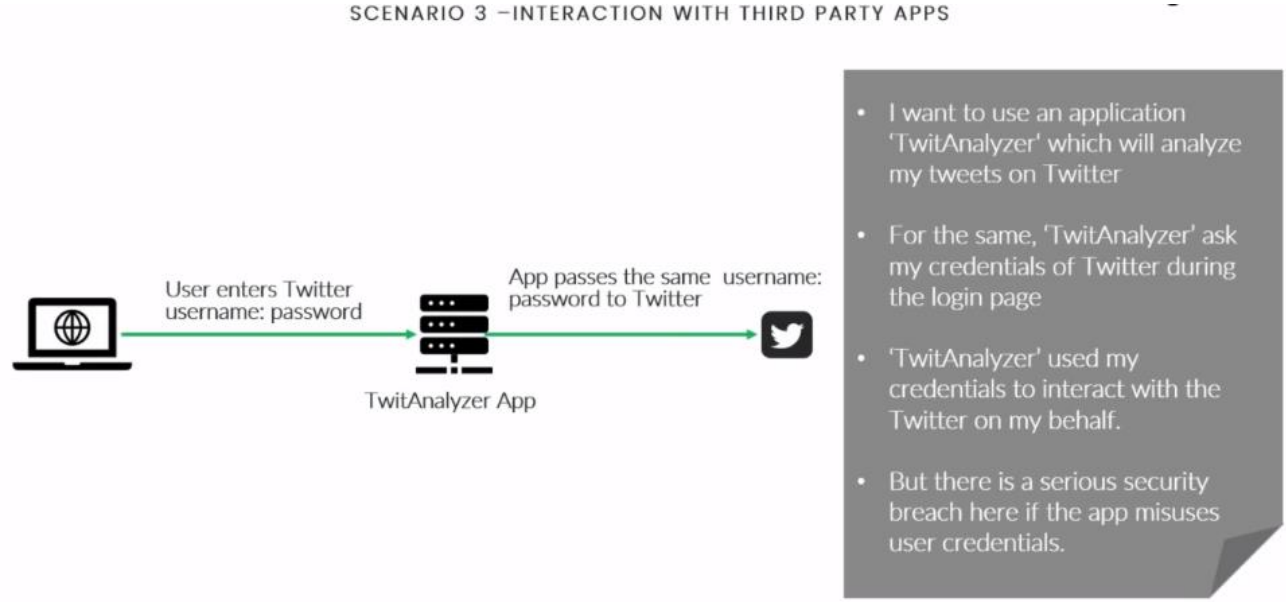
eazy bytes

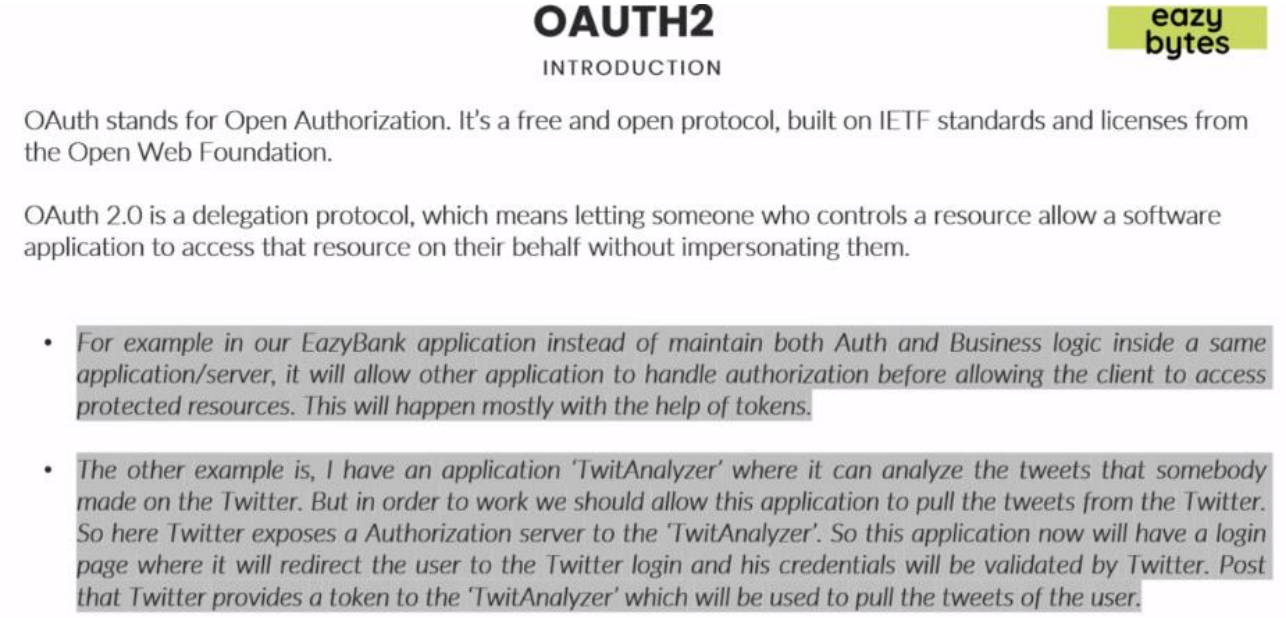
- In the body, we can store details related to user, roles etc. which can be used later for AuthN and AuthZ. Though there is no such limitation what we can send and how we can send in the body, but we should put our best efforts to keep it as light as possible.



- ***Oauth2:***
  - Prevent send credentials all over the networks.
  - No need to store multiple password for multiple apps.
  - Interact with third party apps.

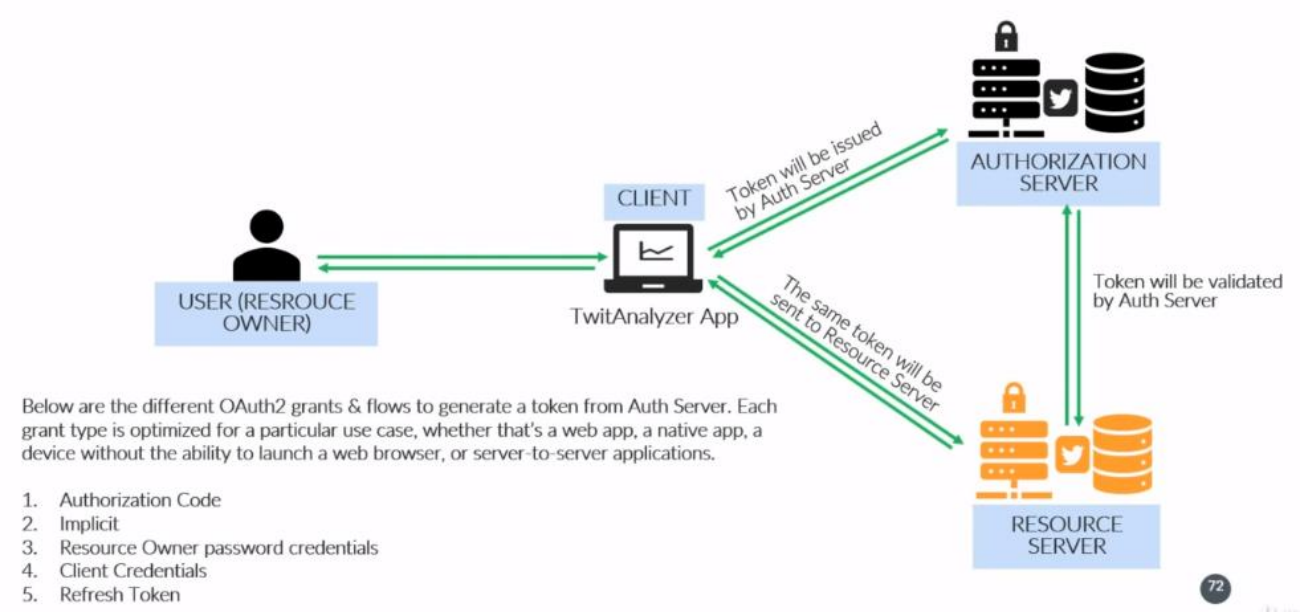






Oauth terms 4 components:

- **Resource server**: content mangager (google drive)
- **Authorization server**: provide **access token** (google authorization server)
- **Resource owner or users**: who have content of google drive.
- **Client**: app.



Grant type:

1. **Authorization code**:
2. **Implicit**
3. **User password credentials**
4. **Client credentials**
5. **Refresh token**

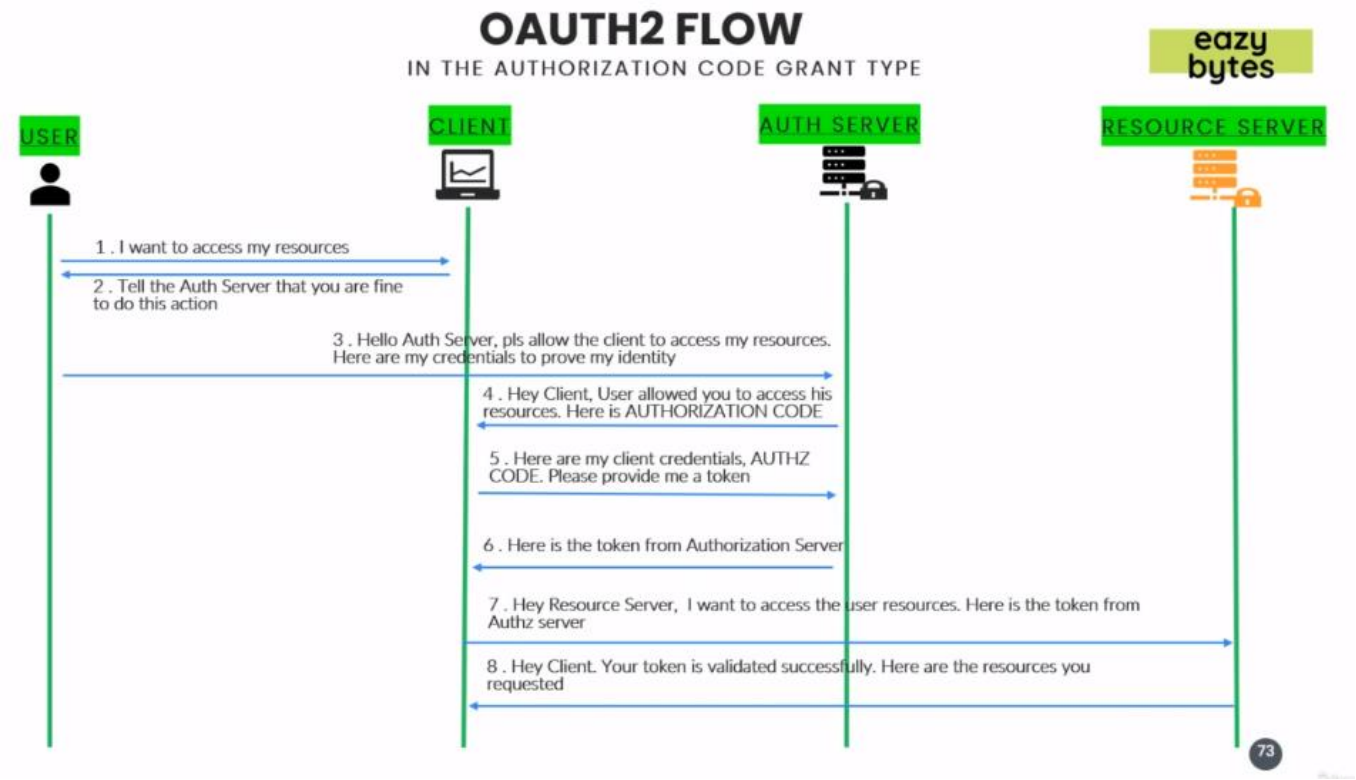
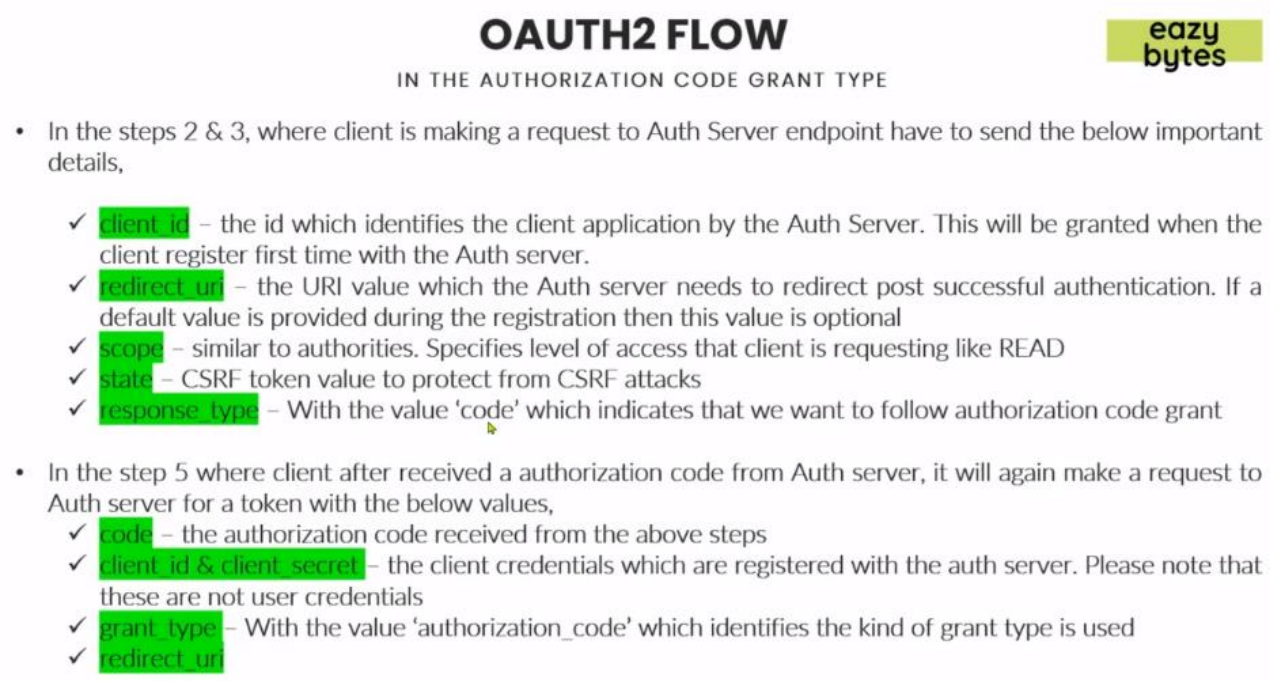


Figure 29 Authorization Code



Why there is two requests to server:

- **First request** is to make sure user consent to authenticate.
- **Second request** to prove client app identity along with authorization code from user consent, to get the access token.

- **Type of grant type** that group 2 request to the same is **implicit grant type**. Implicit grant type is **not secured** since we expose our applicaion **client_id** and **client_secret** in the browsers.

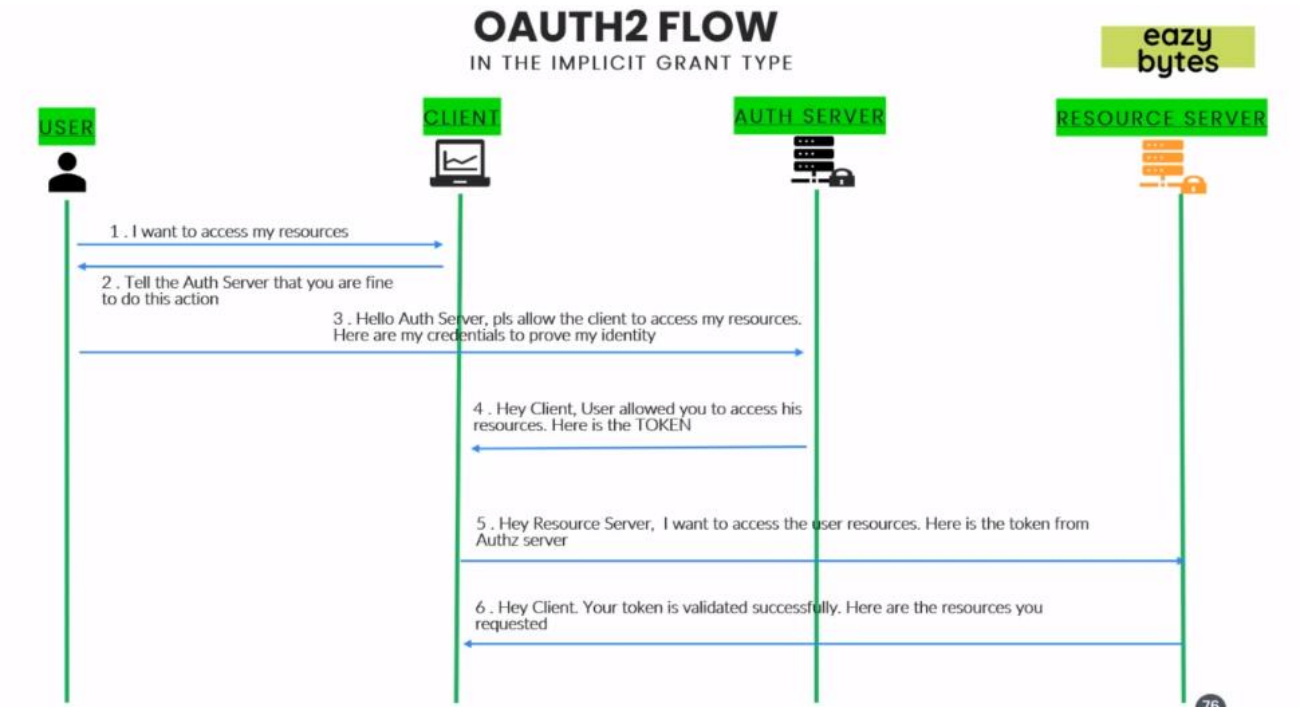


Figure 30 Implicit grant type (not secure)

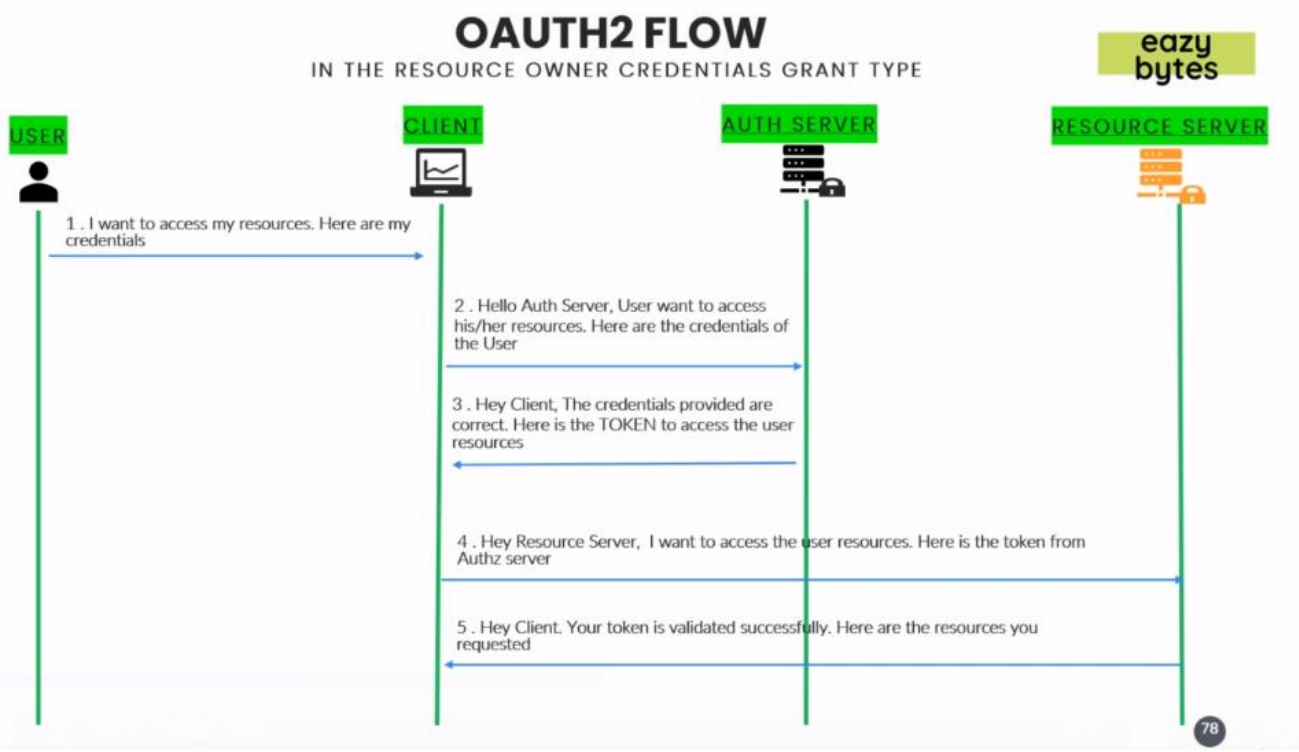
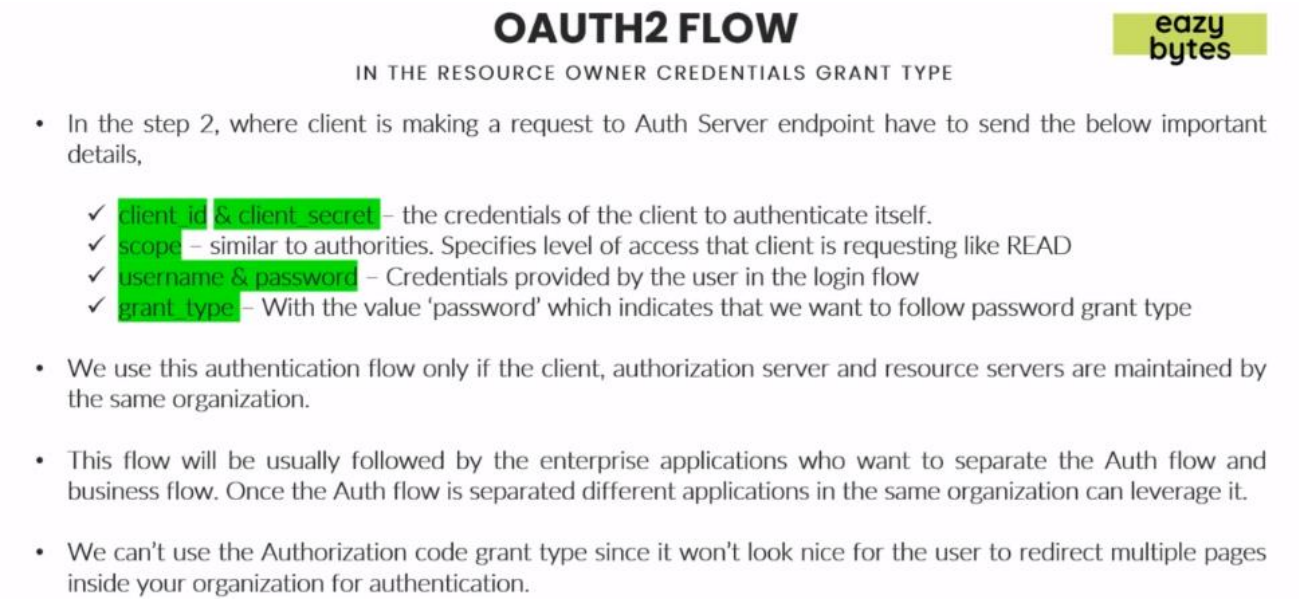


Figure 31 Owner Credentials



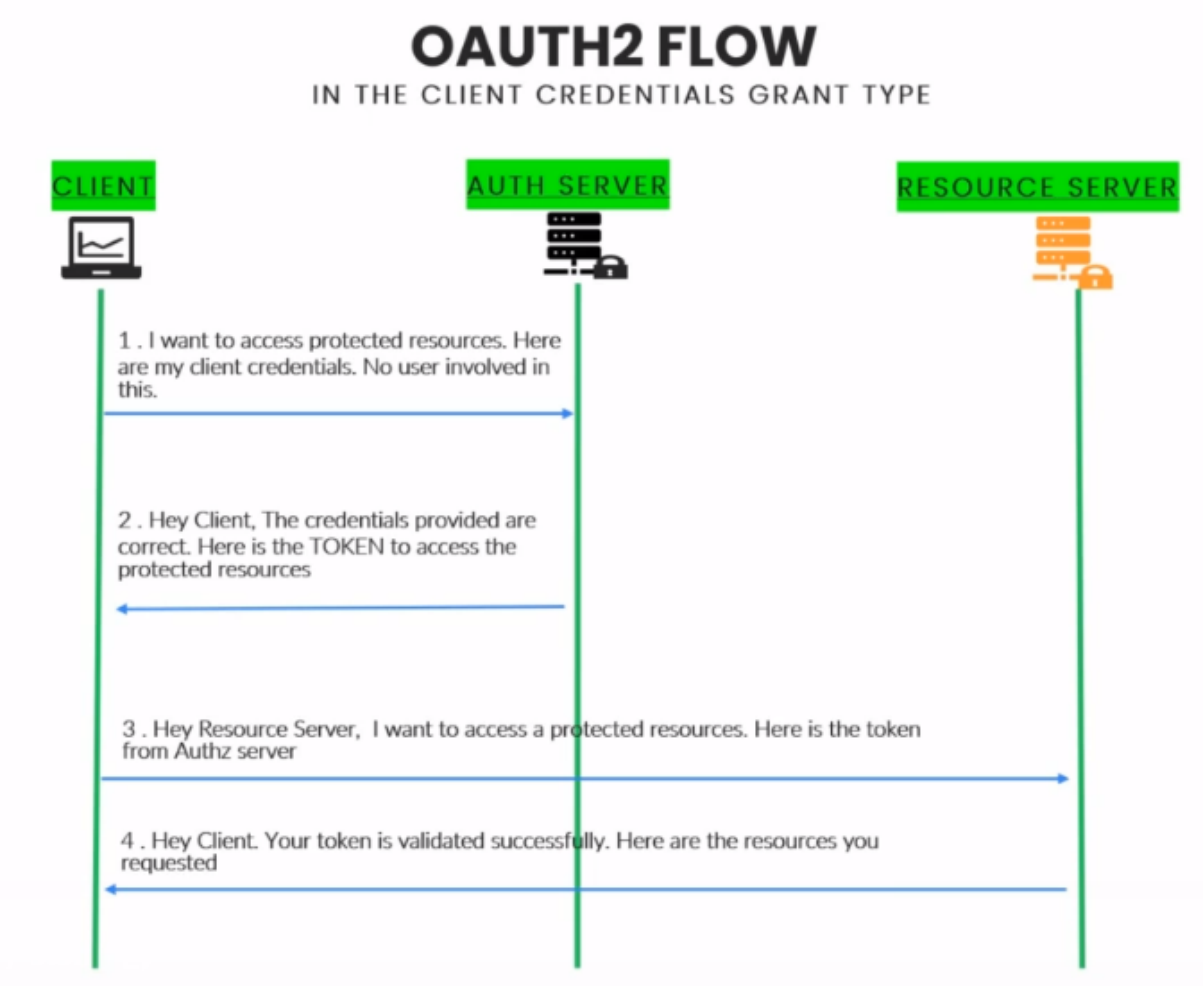


Figure 32 Client Credential grant types

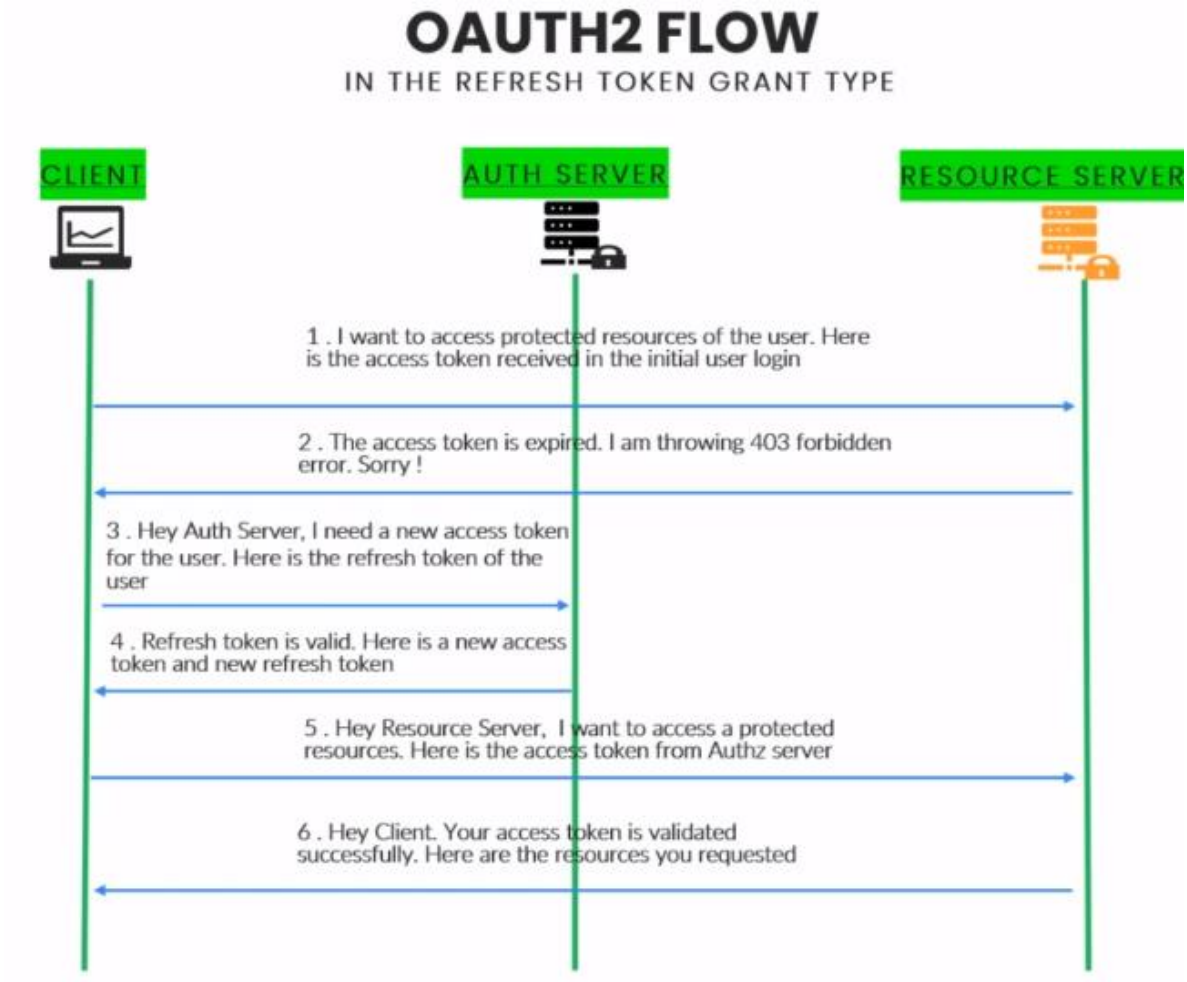
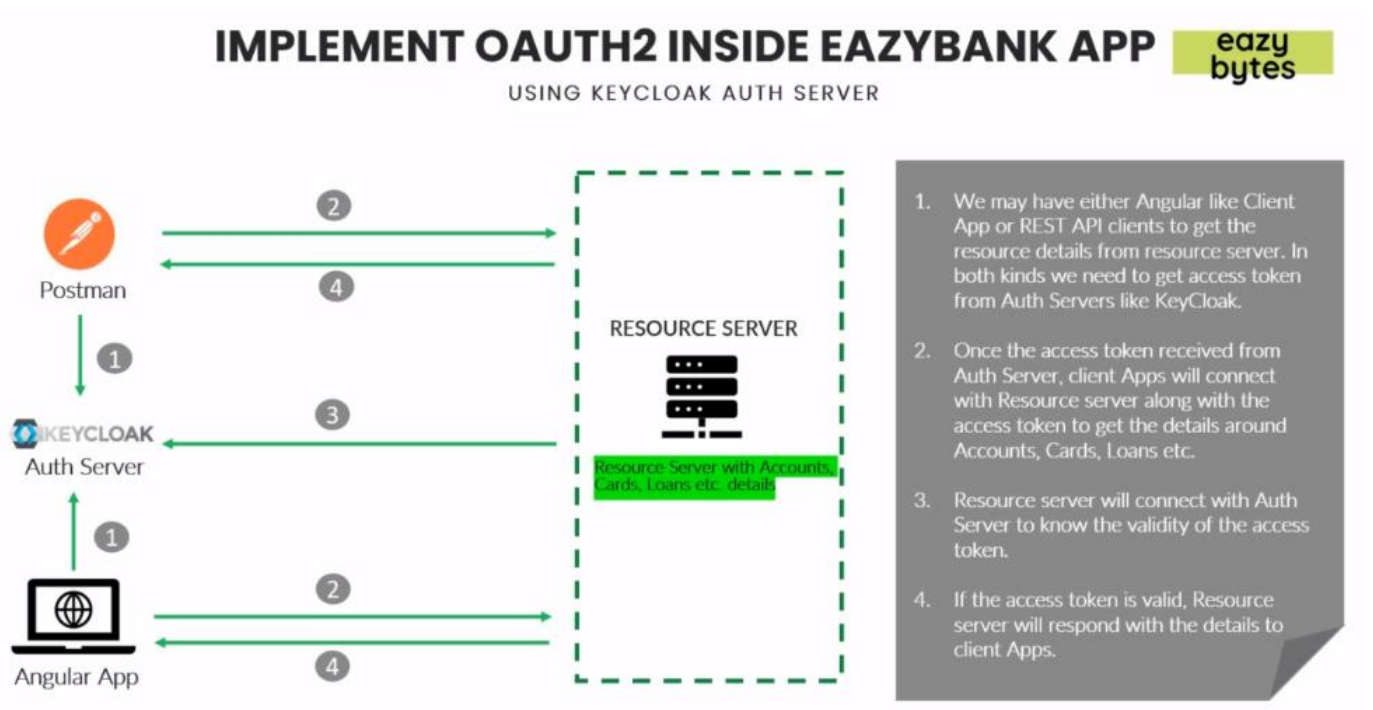


Figure 33 Refresh token grant type



- ***Spring Oauth simple implementation.***

Pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Web config:

```
@Configuration
@EnableWebSecurity
public class WebSecureConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
        .antMatchers( ...antPatterns: "/front-page/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
        .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
        .antMatchers( ...antPatterns: "/**") ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
        .authenticated() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
        .and() HttpSecurity
        .oauth2Login();
    }
}
```

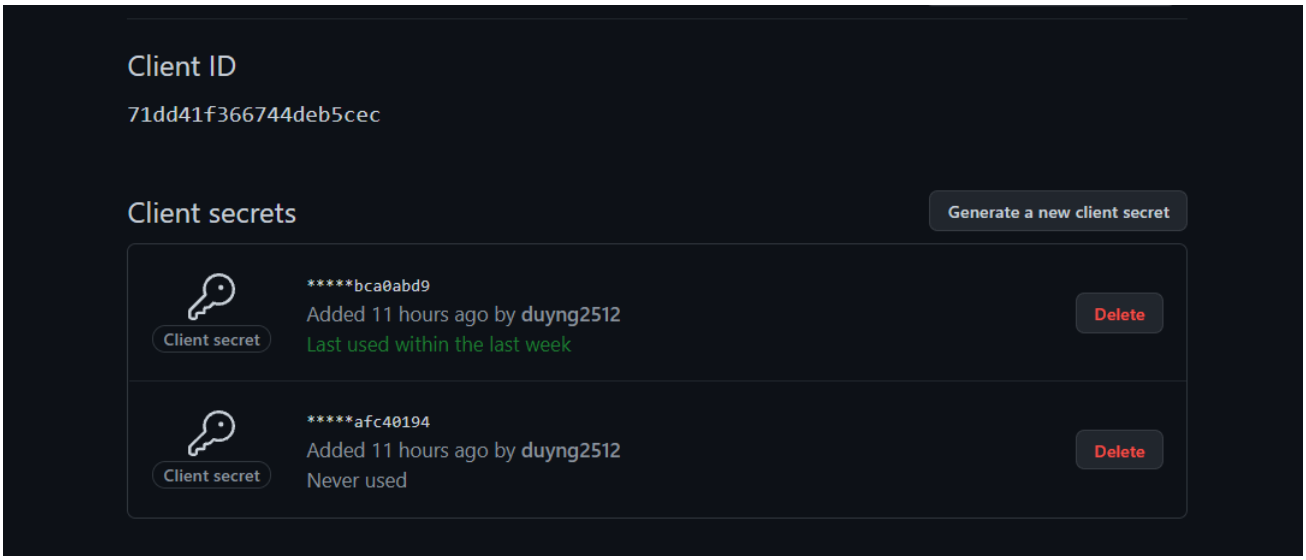
Application:

```
spring.security.oauth2.client.registration.github.client-id=71dd41f366744deb5cec
spring.security.oauth2.client.registration.github.client-secret=422670916fe46fb1a35d48a8aa61f96ebca0abd9
spring.security.oauth2.client.registration.google.client-id=71663517152-8p777ufjk6sg6aetui3hjf1k04bm7o17.apps.googleusercontent.com
spring.security.oauth2.client.registration.google.client-secret=pGifdEkQkf-8P9EDygoS1wTJ
server.port=8080
```

Login URL:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Github Login</title>
</head>
<body>
    <h1>Login Here</h1>
    With GitHub: <a href="/oauth2/authorization/github">click here</a>
    <br>
    With Google: <a href="/oauth2/authorization/google">click here</a>
</body>
</html>
```

Client ID Github + google:



Client ID	71663517152-8p777ufjk6sg6aetui3hjf1k04bm7o17.apps.googleusercontent.com
Client secret	pGifdEkQkf-8P9EDygoS1wTJ
Creation date	October 4, 2021 at 10:51:26 PM GMT+7

- ***Open ID.***

892

OpenID is about authentication (ie. proving who you are), OAuth is about authorisation (ie. to grant access to functionality/data/etc.. without having to deal with the original authentication).

OAuth could be used in external partner sites to allow access to protected data without them having to re-authenticate a user.

What is OpenID Connect?

- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of scopes and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.