DISTRIBUTED SYSTEMS

Lab 3

João Leitão, Sérgio Duarte, Pedro Camponês

GOALS

In the end of this lab you should be able to:

- Understand what are transient communication errors in the context of REST clients.
- Know how to deal with errors on REST clients
- Know how to persist "plain old" Java objects (POJO) using Hibernate and a backing relational database
- Know how to query Hibernate to recover your “plain old” Java Objects.

GOALS

In the end of this lab you should be able to:

- **Understand what are transient communication errors in the context of REST clients.**
- **Know how to deal with errors on REST clients**
- Know how to persist "plain old" Java objects (POJO) using Hibernate and a backing relational database
- Know how to query Hibernate to recover your “plain old” Java Objects.

Transient Errors on REST requests

The execution of a REST request might fail due to multiple reasons:

- The server might not be running when the request is performed (a TCP connection cannot be established)
- The server is too slow in processing the request and the client timeouts.
- A TCP connection is dropped.
- The network interface fails.
- A temporary network anomaly (e.g., routing).

IDENTIFYING AN ERROR IN EXECUTING A REST REQUEST

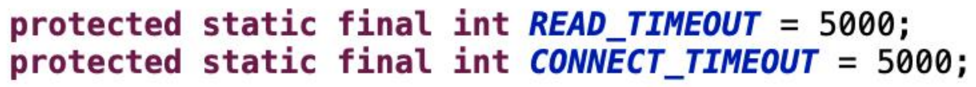
In Jersey (JAX-RS) clients are notified of errors during the execution of a request through a Java exception:

Javax.ws.rs.ProcessingException

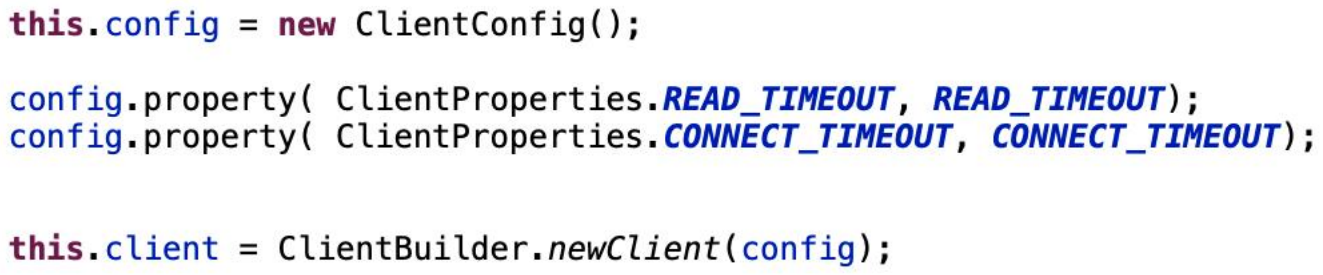
Therefore, REST requests should be enclosed within a try{}catch{} block so that this Exception can be captured and lead to retry the request automatically after a small amount of time and for a limited number of reattempts.

Timeouts can lead to the ProcessingException being generated.

Therefore, these timeouts should be configured at the level of the ClientBuilder.



```
protected static final int READ_TIMEOUT = 5000;
protected static final int CONNECT_TIMEOUT = 5000;
```

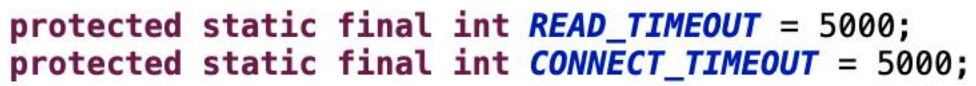


```
this.config = new ClientConfig();

config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);

this.client = ClientBuilder.newClient(config);
```

Example is present in the support code on class **RestUsersClient**

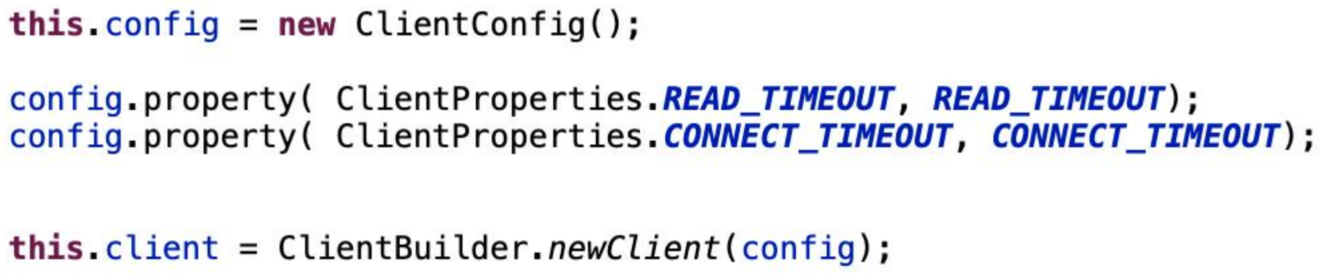
Timeouts can lead to the ProcessingException being generated.

Therefore, these timeouts should be configured at the level of the ClientBuilder.



```
protected static final int READ_TIMEOUT = 5000;
protected static final int CONNECT_TIMEOUT = 5000;
```

These translate respectively to:
Wait at most 5 seconds for a TCP connection to be established for the server
Wait at most 5 seconds to receive the reply from the server after sending the request.



```
this.config = new ClientConfig();

config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);


this.client = ClientBuilder.newClient(config);
```

Example is present in the support code on class **RestUsersClient**

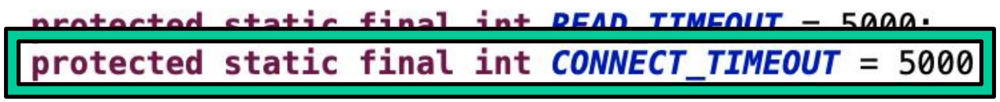
TIMEOUTS CAN LEAD TO THE PROCESSINGEXCEPTION BEING GENERATED.

Therefore, these timeouts should be configured at the level of the ClientBuilder.

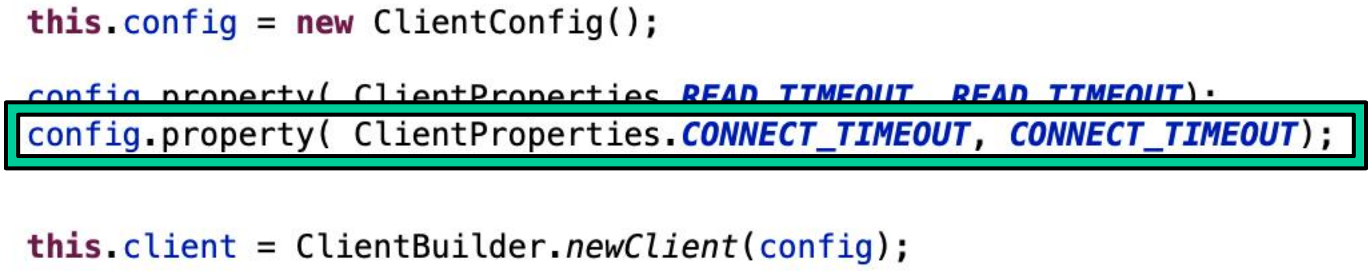
These translate respectively to:

Wait at most 5 seconds for a TCP connection to be established for the server

Wait at most 5 seconds to receive the reply from the server after sending the request.



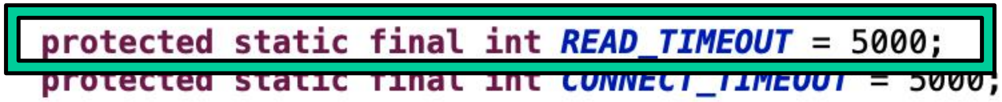
```
protected static final int READ_TIMEOUT = 5000;
protected static final int CONNECT_TIMEOUT = 5000
```



```
this.config = new ClientConfig();

config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);

this.client = ClientBuilder.newClient(config);
```

TIMEOUTS CAN LEAD TO THE PROCESSINGEXCEPTION BEING GENERATED.

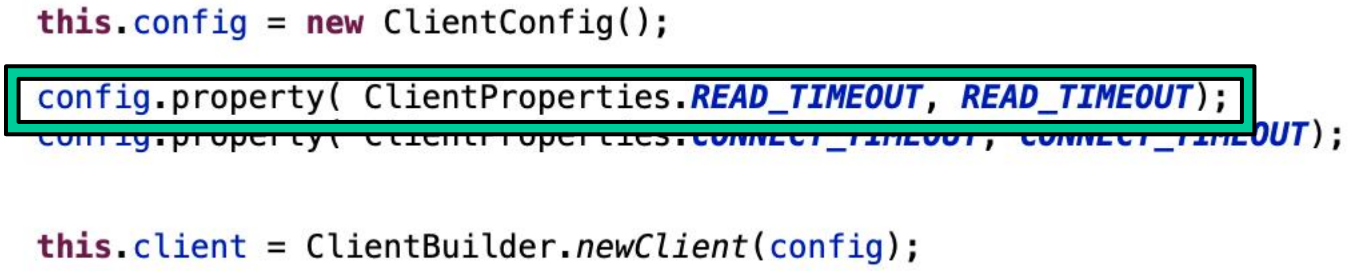
Therefore, these timeouts should be configured at the level of the ClientBuilder.



```
protected static final int READ_TIMEOUT = 5000;
protected static final int CONNECT_TIMEOUT = 5000;
```

These translate respectively to:
Wait at most 5 seconds for a TCP connection to be established for the server

Wait at most 5 seconds to receive the reply from the server after sending the request.

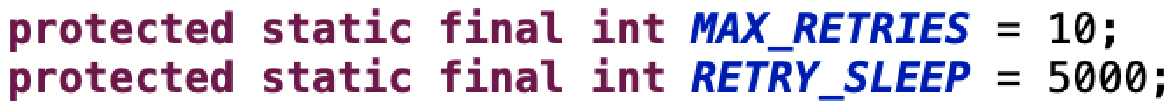


```
this.config = new ClientConfig();

config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);

this.client = ClientBuilder.newClient(config);
```

Re-execution of a REST request

It should be noted that immediate re-execution is not a great idea, since the transient failure might need some time to recover.

Also, a maximum number of attempts should be made to avoid blocking if the failure is permanent.



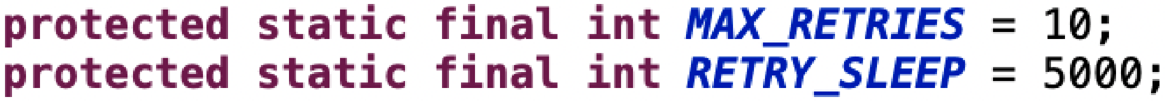
```
protected static final int MAX_RETRIES = 10;
protected static final int RETRY_SLEEP = 5000;
```

Example is present in the support code on class **RestUsersClient**

Re-execution of a REST request

It should be noted that immediate re-execution is not a great idea, since the transient failure might need some time to recover.

Also, a maximum number of attempts should be made to avoid blocking if the failure is permanent.

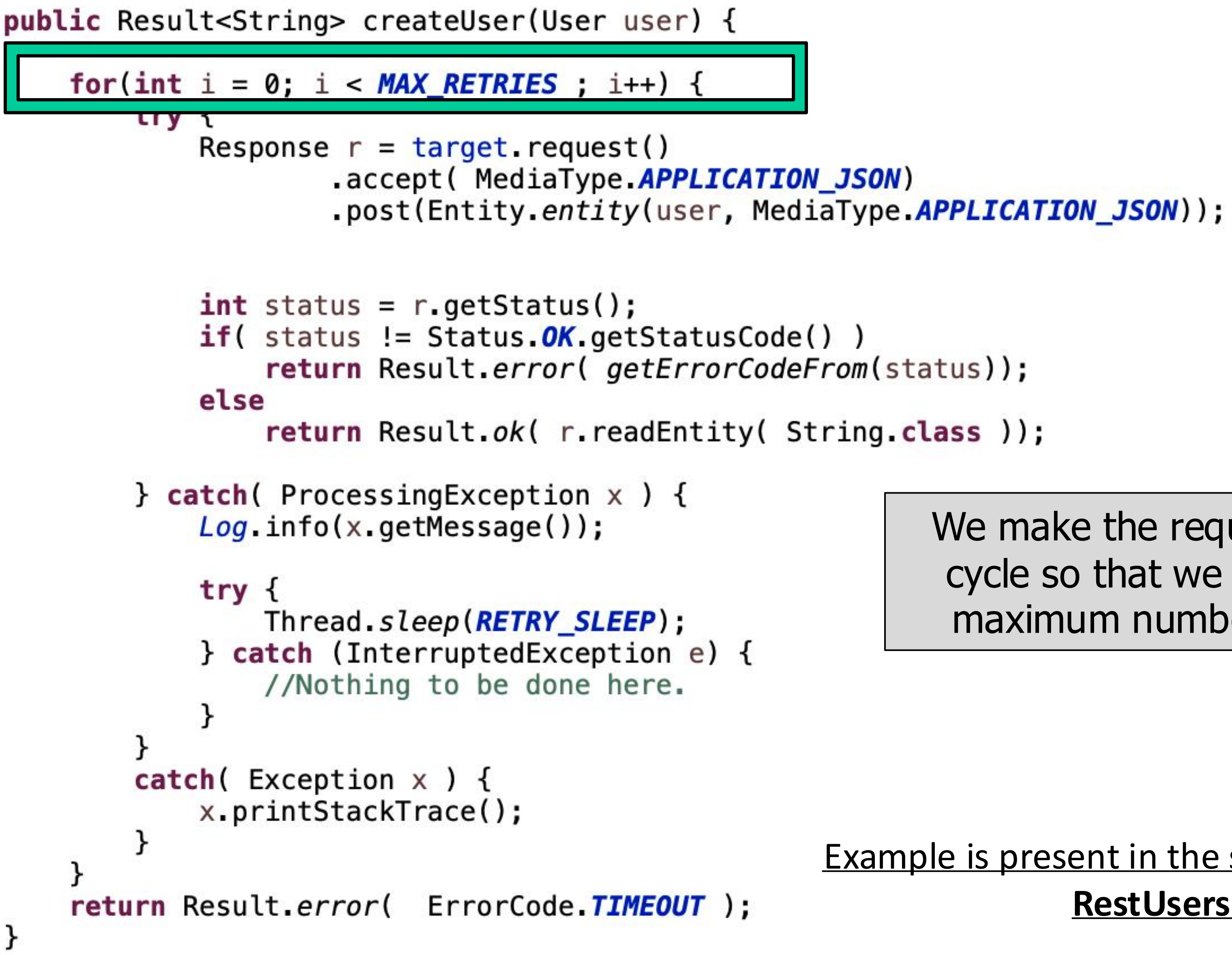


```
protected static final int MAX_RETRIES = 10;
protected static final int RETRY_SLEEP = 5000;
```

In the lab example we are limiting this to 10 retries, with a wait time of 5 seconds between each retry (worst case the client will be blocked 50 seconds).

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST

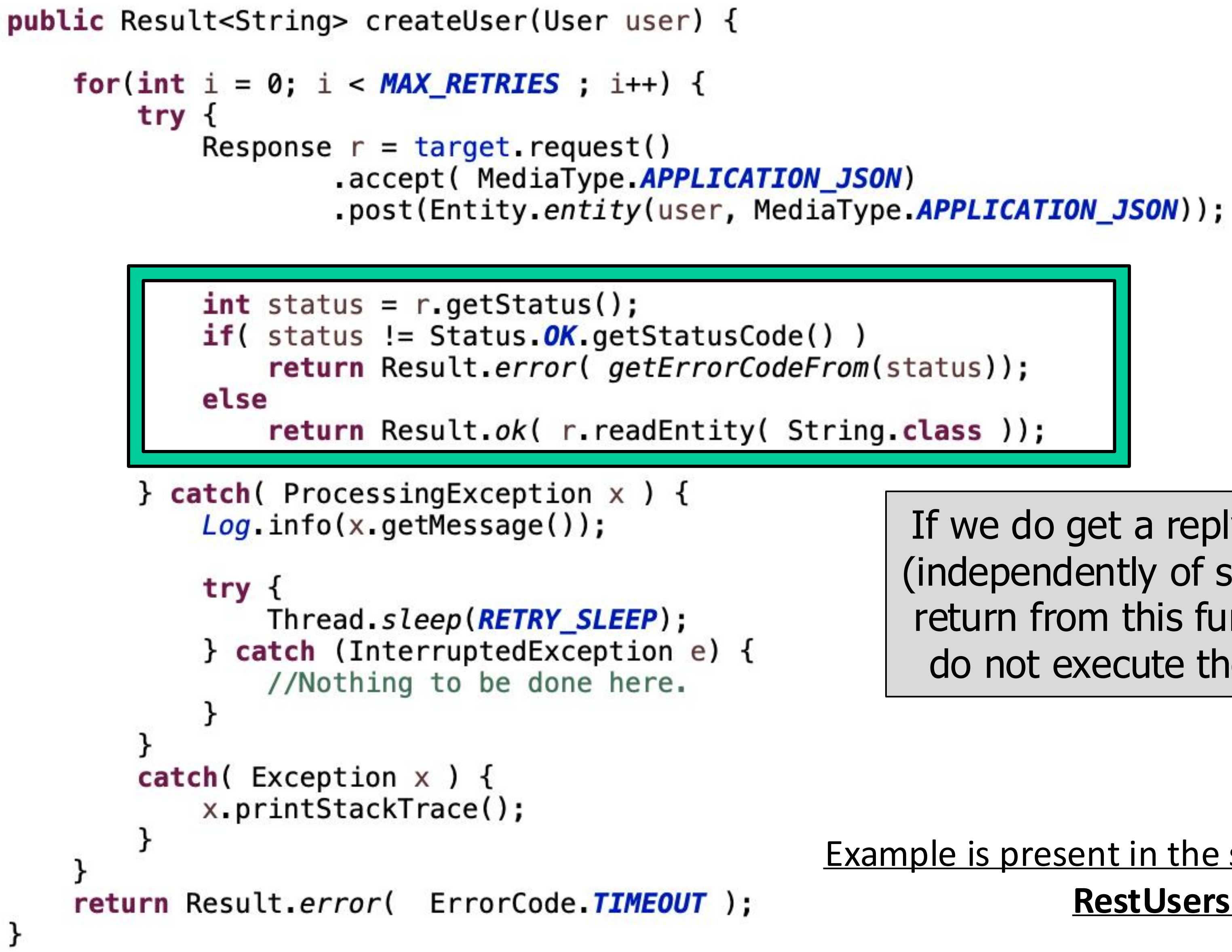


```java
public Result<String> createUser(User user) {
    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

We make the request within a for cycle so that we can control the maximum number of attempts.

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST

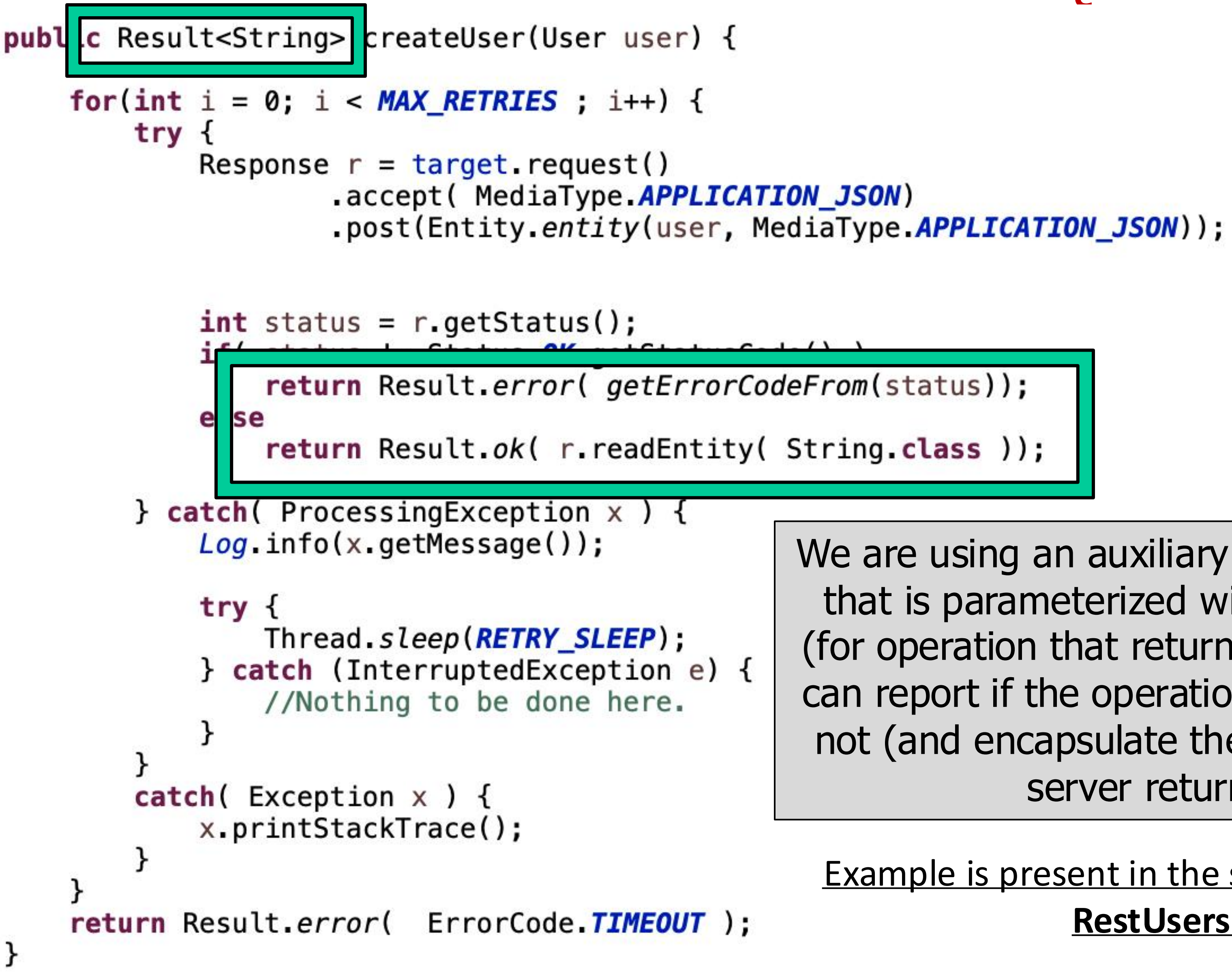


```java
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));

            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));
        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

If we do get a reply from the server (independently of success or not) we return from this function so that we do not execute the request again.

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST

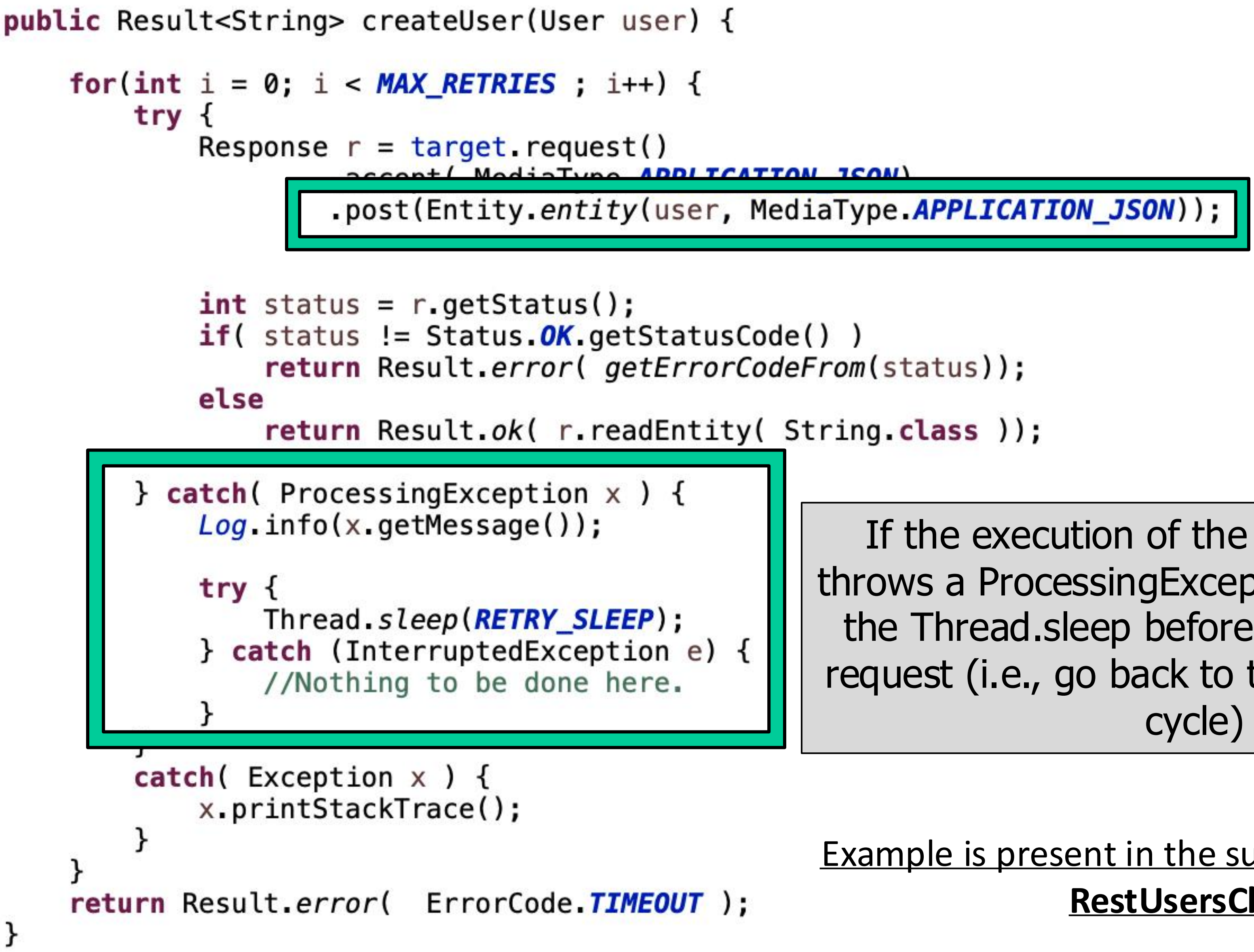


```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

We are using an auxiliary class named Result that is parameterized with the return type (for operation that return a Java object) and can report if the operation was successful or not (and encapsulate the Java object if the server return one)

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST

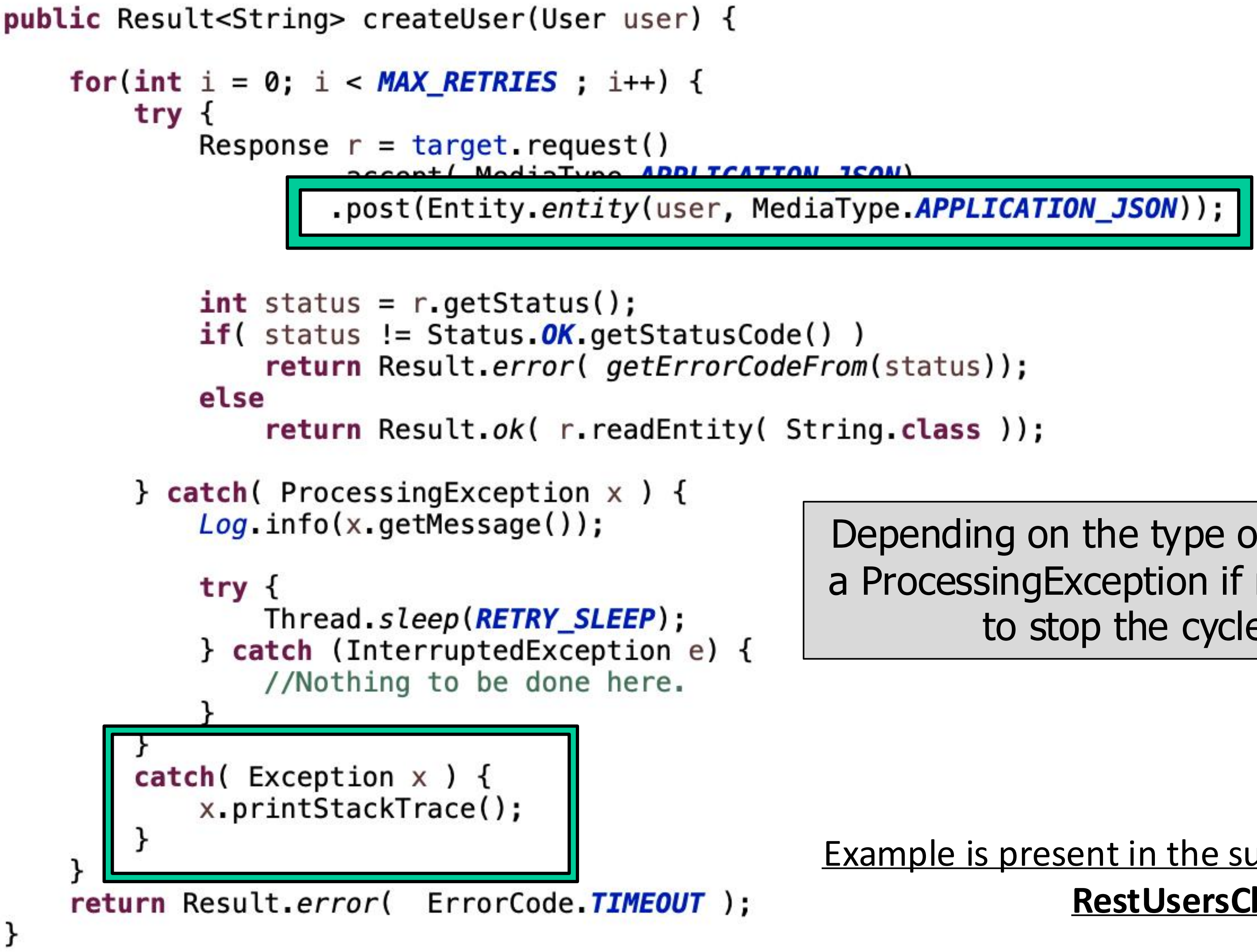


```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

If the execution of the REST operation throws a ProcessingException we wait using the Thread.sleep before re-executing the request (i.e., go back to the start of the for cycle)

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST

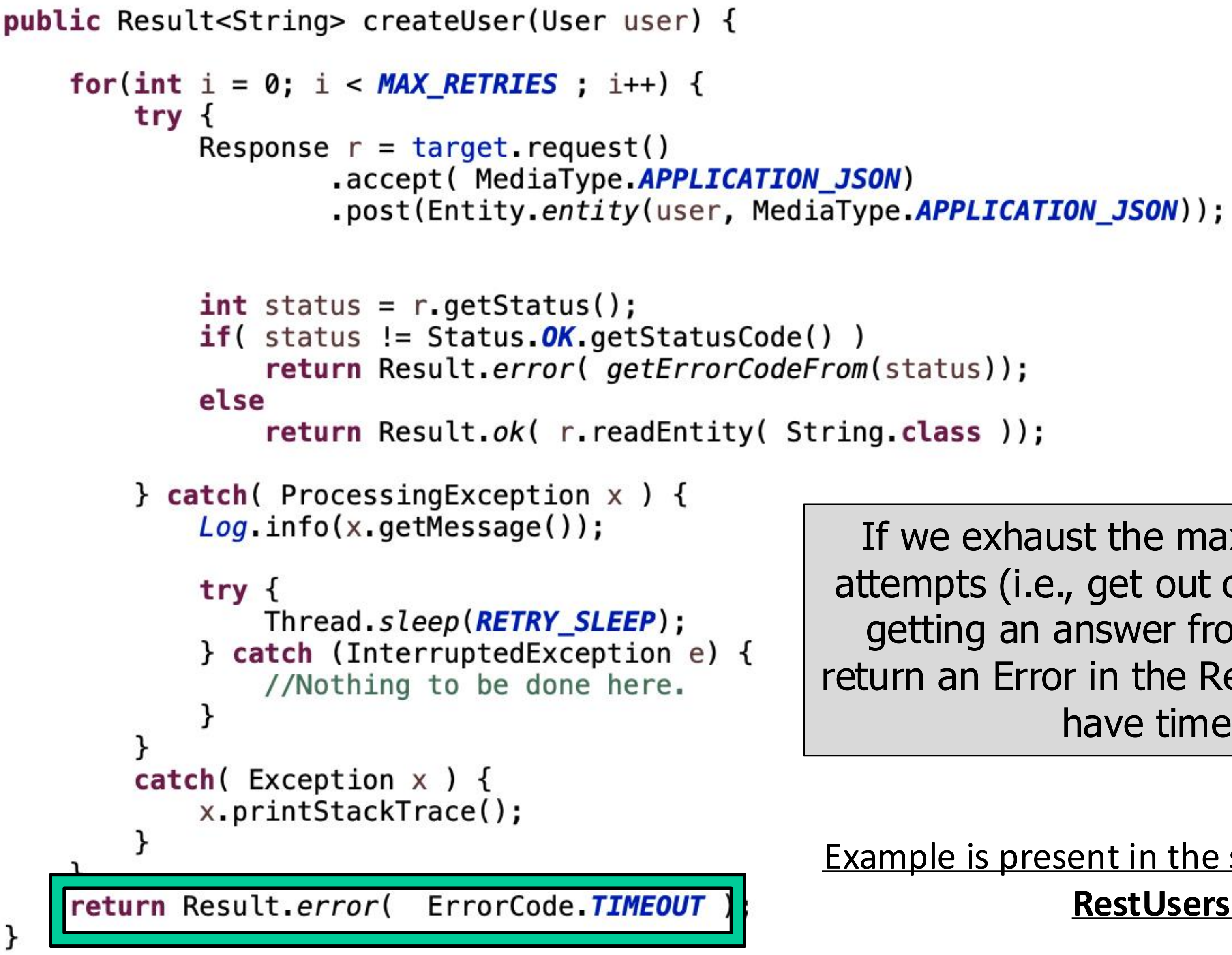


```
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

Depending on the type of Exception (if not a ProcessingException if might make sense to stop the cycle earlier).

Example is present in the support code on class **RestUsersClient**

RE-EXECUTION OF A REST REQUEST



```java
public Result<String> createUser(User user) {

    for(int i = 0; i < MAX_RETRIES ; i++) {
        try {
            Response r = target.request()
                    .accept( MediaType.APPLICATION_JSON)
                    .post(Entity.entity(user, MediaType.APPLICATION_JSON));


            int status = r.getStatus();
            if( status != Status.OK.getStatusCode() )
                return Result.error( getErrorCodeFrom(status));
            else
                return Result.ok( r.readEntity( String.class ));

        } catch( ProcessingException x ) {
            Log.info(x.getMessage());

            try {
                Thread.sleep(RETRY_SLEEP);
            } catch (InterruptedException e) {
                //Nothing to be done here.
            }
        }
        catch( Exception x ) {
            x.printStackTrace();
        }
    }
    return Result.error(  ErrorCode.TIMEOUT );
}
```

If we exhaust the maximum number of attempts (i.e., get out of the cycle without getting an answer from the server) we return an Error in the Result stating that we have timed-out.

Example is present in the support code on class **RestUsersClient**

AVOIDING CODE REPETITION

The code that is provided to support this lab uses a generic RestUsersClient that provides methods to exercise all endpoints of Users (without the endpoints to manipulate the avatar of the user).

- The code only has the mechanisms to retry operations on the create user operation.
- Repeat this code in all operations can lead to errors and in general is a bad practice.
- In the future we will discuss how to factorize this portion of the code.

GOALS

In the end of this lab you should be able to:

- Understand what are transient communication errors in the context of REST clients.
- Know how to deal with errors on REST clients
- **Know how to persist "plain old" Java objects (POJO) using Hibernate and a backing relational database**
- **Know how to query Hibernate to recover your “plain old” Java Objects.**

PERSISTENCE USING HIBERNATE

In this course we are going to use Hibernate to create a persistence layer associated with WebServices.

Hibernate is as open-source Object Relational Mapping (ORM) tool

It fundamentally provides mechanisms to map object-oriented domain (i.e., data) models to a relational database.

Therefore, Hibernate can be used to implement persistency of state and storage for web services.

https://hibernate.org/

PERSISTENCE ENGINE

Hibernate allows to persist objects using different engines, while providing a unified API for the programmer.

HSQLDB (Hyper SQL Database) is a relational database management system written in Java. It has a JDBC driver and supports a large subset of SQL-92, SQL:2008, SQL:2011, and SQL:2016 standards.

Maven dependencies

```
<dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.7.2</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.4.Final</version>
</dependen
```

CONFIGURATION FILE

```
<!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>
                <!-- JDBC Database connection settings -->
                <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
                <property name="connection.url">jdbc:hsqldb:file:/tmp/db</property>
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool settings ... using built-in test pool -->
                <property name="connection.pool_size">1</property>
                <!-- Echo the SQL to stdout -->
                <property name="show_sql">true</property>
                <!-- Set the current session context -->
                <property name="current_session_context_class">thread</property>
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create-drop</property>
                <!-- dbcp connection pool configuration -->
                <property name="hibernate.dbcp.initialSize">5</property>
                <property name="hibernate.dbcp.maxTotal">20</property>
                <property name="hibernate.dbcp.maxIdle">10</property>
                <property name="hibernate.dbcp.minIdle">5</property>
                <property name="hibernate.dbcp.maxWaitMillis">-1</property>
                <mapping class="lab3.api.User" />
        </session-factory>
</hibernate-configuration>
```

CONFIGURATION FILE

```
<!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>
                <!-- JDBC Database connection settings -->
                <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
                <property name="connection.url">jdbc:hsqldb:file:/tmp/db</property>
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool settings ... using built-in test pool -->
                <property name="connection.pool_size">1</property>
                <!-- Echo the SQL to stdout -->
                <property name="show_sql">true</property>
                <!-- Set the current session context -->
                <property name="current_session_context_class">thread</property>
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create-drop</property>
                <!-- dbcp connection pool configuration -->
                <property name="hibernate.dbcp.initialSize">5</property>
                <property name="hibernate.dbcp.maxTotal">20</property>
                <property name="hibernate.dbcp.maxIdle">10</property>
                <property name="hibernate.dbcp.mi
                <property name="hibernate.dbcp.ma
                <mapping class="lab3.api.User" />
        </session-factory>
</hibernate-configuration>
```

This file defines several properties, including the connection to the backing database (hsqldb in this case).

CONFIGURATION FILE

```
<!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>
                <!-- JDBC Database connection settings -->
                <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
                <property name="connection.url">jdbc:hsqldb:file:/tmp/db</property>
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool settings ... using built-in test pool -->
                <property name="connection.pool_size">1</property>
                <!-- Echo the SQL to stdout -->
                <property name="show_sql">true</pr
                <!-- Set the current session context -->
                <property name="current_session_con
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create-drop</property>
                <!-- dbcp connection pool configuration -->
                <property name="hibernate.dbcp.initialSize">5</property>
                <property name="hibernate.dbcp.maxTotal">20</property>
                <property name="hibernate.dbcp.maxIdle">10</property>
                <property name="hibernate.dbcp.minIdle">5</property>
                <property name="hibernate.dbcp.maxWaitMillis">-1</property>
                <mapping class="lab3.api.User" />
        </session-factory>
</hibernate-configuration>
```

The configuration file must also contain the list of entities that will be mapped to the relational model.

CONFIGURATION FILE

```
<!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>
                <!-- JDBC Database connection settings -->
                <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
                <property name="connection.url">jdbc:hsqldb:file:/tmp/db</property>
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool settings ... using built-in test pool -->
                <property name="connection.pool_size">1</property>
                <!-- Echo the SQL to stdout -->
                <property name="show_sql">true</pr
                <!-- Set the current session context -->
                <property name="current_session_con
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create-drop</property>
                <!-- dbcp connection pool configuration -->
                <property name="hibernate.dbcp.initialSize">5</property>
                <property name="hibernate.dbcp.maxTotal">20</property>
                <property name="hibernate.dbcp.maxIdle">10</property>
                <property name="hibernate.dbcp.minIdle">5</property>
                <property name="hibernate.dbcp.maxWaitMillis">-1</property>
                <mapping class="lab3.api.User" />
        </session-factory>
</hibernate-configuration>
```

Additional entries of this type can be added to support storing additional types of Objects.

CONFIGURATION FILE

This configuration restart the database whenever Hibernate is initialized (which is adequate for testing purposes)

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hiber
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd
<hibernate-configuration>
        <session-factory>
                <!-- JDBC Database connection settings -->
                <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
                <property name="connection.url">jdbc:hsqldb:file:/tmp/db</property>
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool settings ... using built-in test pool -->
                <property name="connection.pool_size">1</property>
                <!-- Echo the SQL to stdout -->
                <property name="show_sql">true</property>
                <!-- Set the current session context -->
                <property name="current_session_context_class">thread</property>
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create-drop</property>
                <!-- dbcp connection pool configuration -->
                <property name="hibernate.dbcp.initialSize">5</property>
                <property name="hibernate.dbcp.maxTotal">20</property>
                <property name="hibernate.dbcp.maxIdle">10</property>
                <property name="hibernate.dbcp.minIdle">5</property>
                <property name="hibernate.dbcp.maxWaitMillis">-1</property>
                <mapping class="lab3.api.User" />
        </session-factory>
</hibernate-configuration>
```

USING HIBERNATE

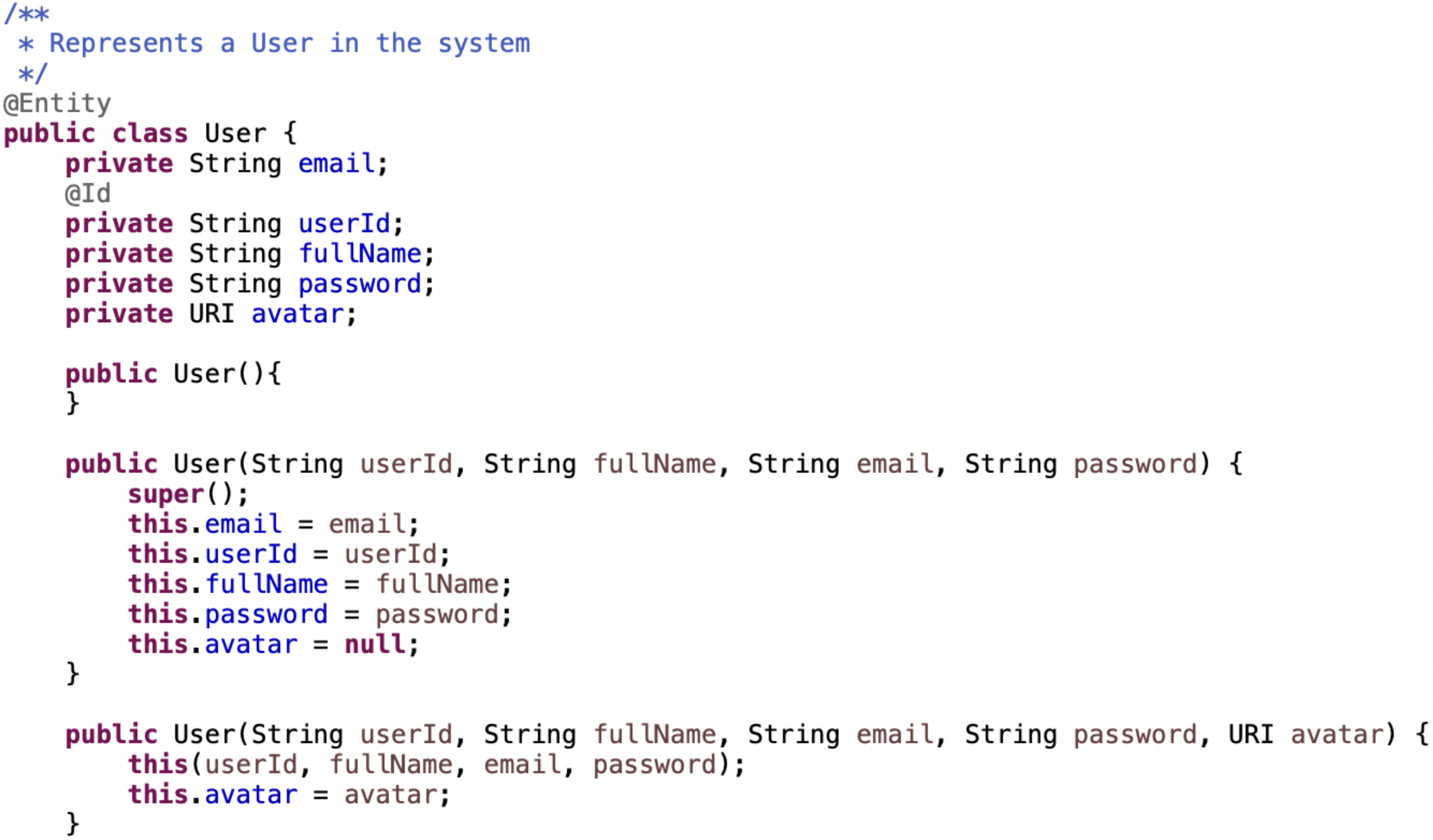
To persist a Java object, it needs to modelled as an Entity

The key requirements are:

- Public default constructor (i.e., no arguments);
- Setters and getters;
- Non-final fields;
- Non-final class;

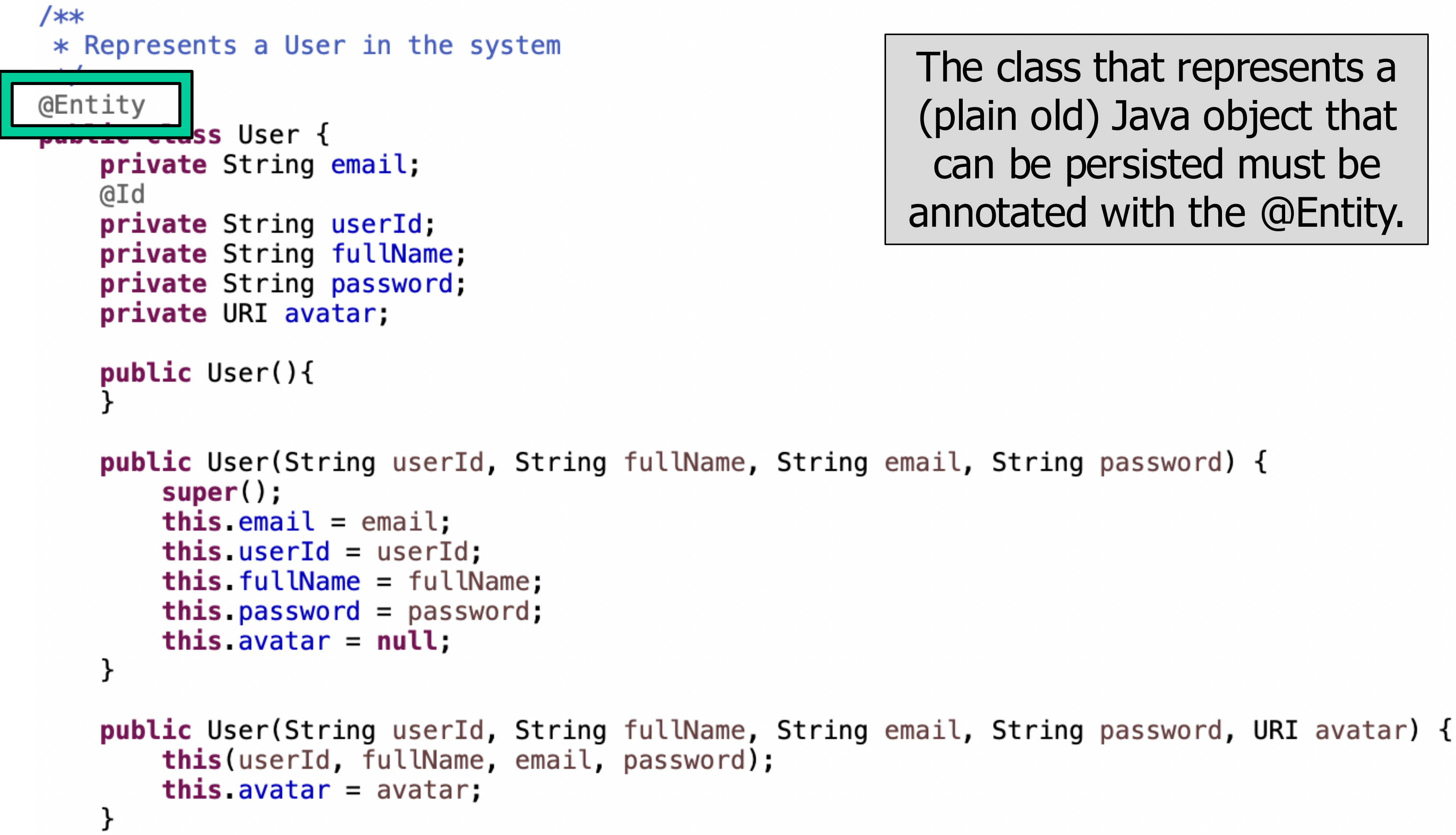
Due to these requirements Java records cannot be used as Hibernate entities (e.g., fields in records are final).

ANNOTATION ON POJOS



```java
/**
 * Represents a User in the system
 */
@Entity
public class User {
    private String email;
    @Id
    private String userId;
    private String fullName;
    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String fullName, String email, String password) {
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

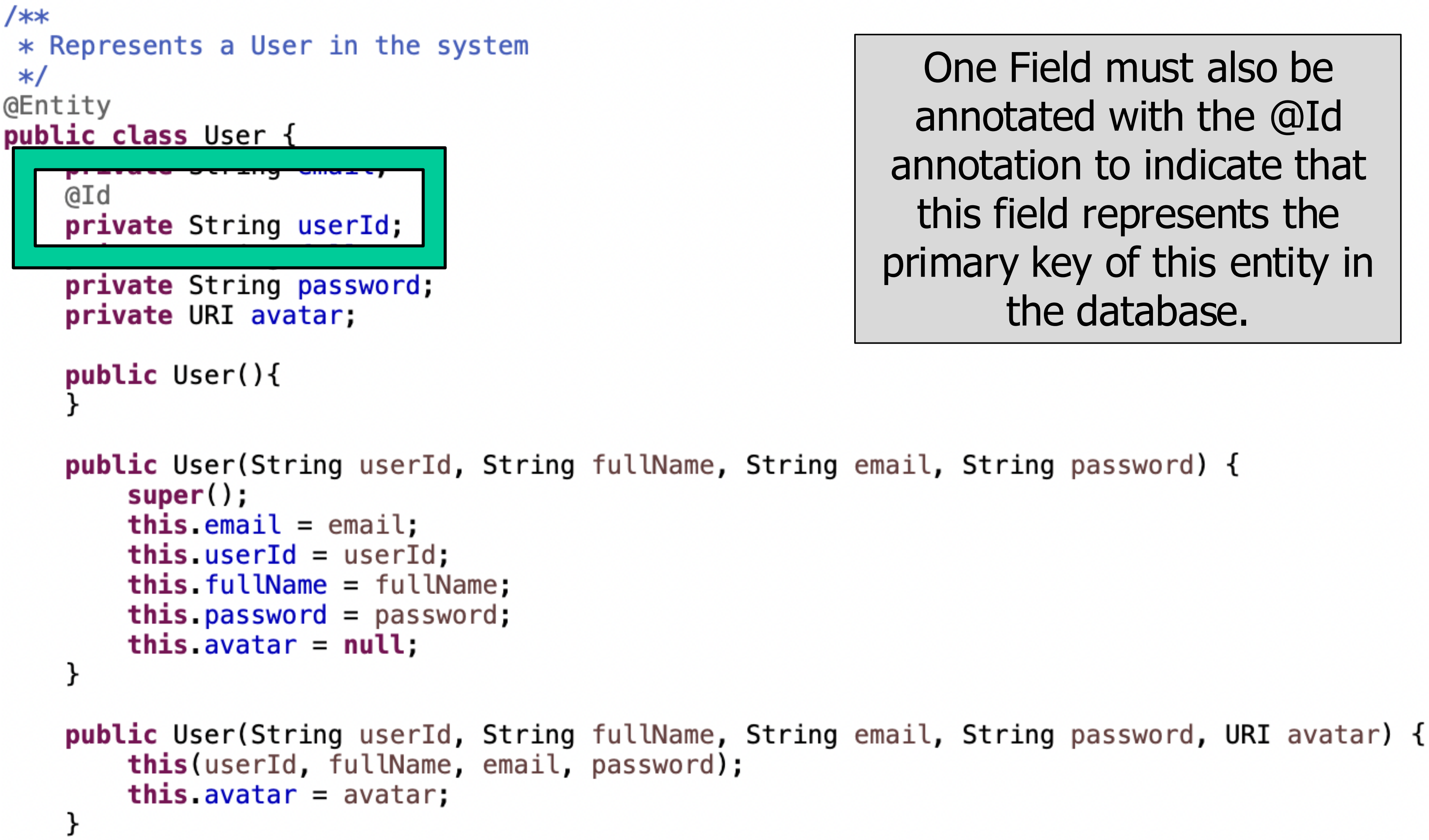
ANNOTATION ON POJOS



```
/**
 * Represents a User in the system
 */
@Entity
public class User {
    private String email;
    @Id
    private String userId;
    private String fullName;
    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String fullName, String email, String password) {
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

The class that represents a (plain old) Java object that can be persisted must be annotated with the @Entity.

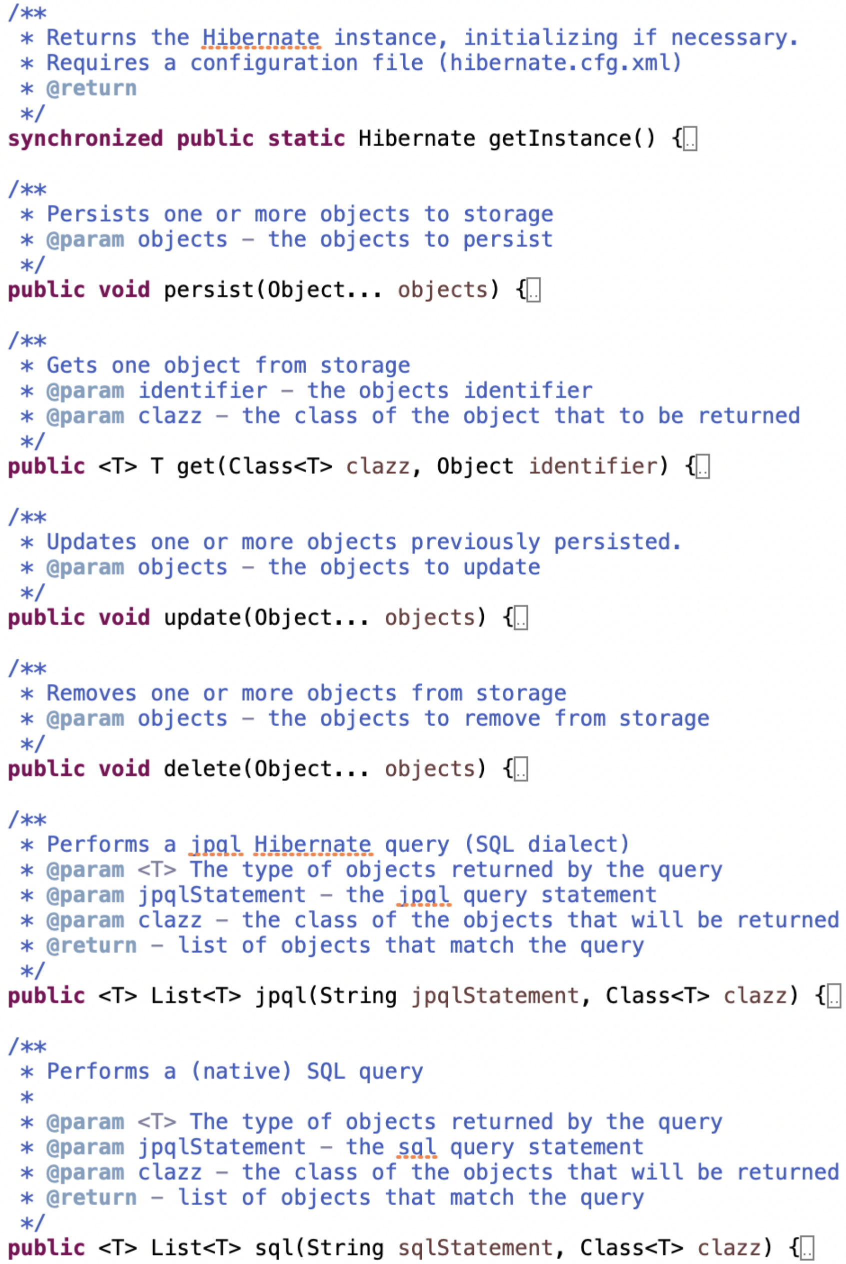
ANNOTATION ON POJOS



```
/**
 * Represents a User in the system
 */
@Entity
public class User {
    private String email;
    @Id
    private String userId;
    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String fullName, String email, String password) {
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

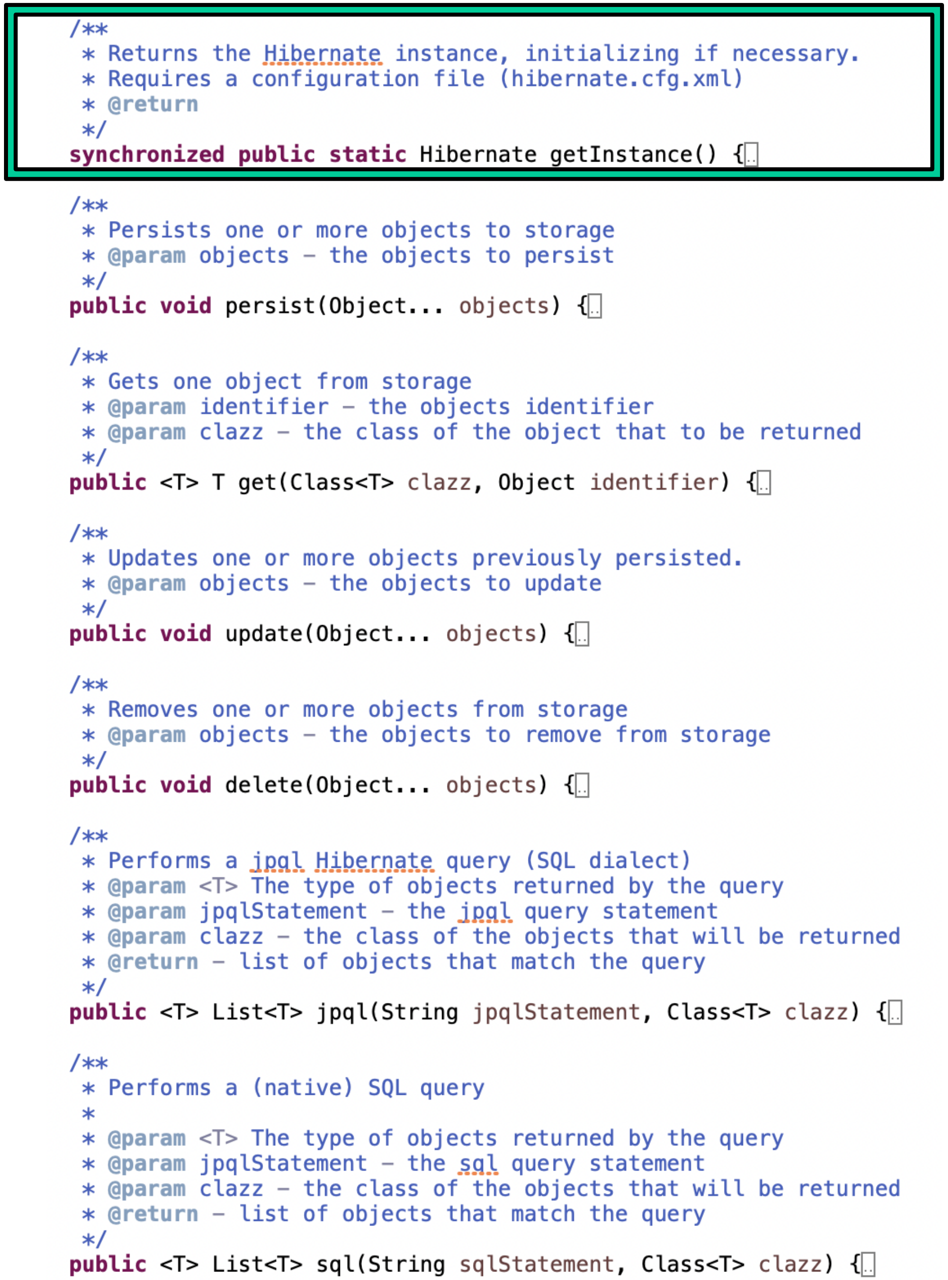
One Field must also be annotated with the @Id annotation to indicate that this field represents the primary key of this entity in the database.

AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {…

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {…

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {…

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {…

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {…

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {…

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {…
```

Auxiliary class to interact with Hibernate

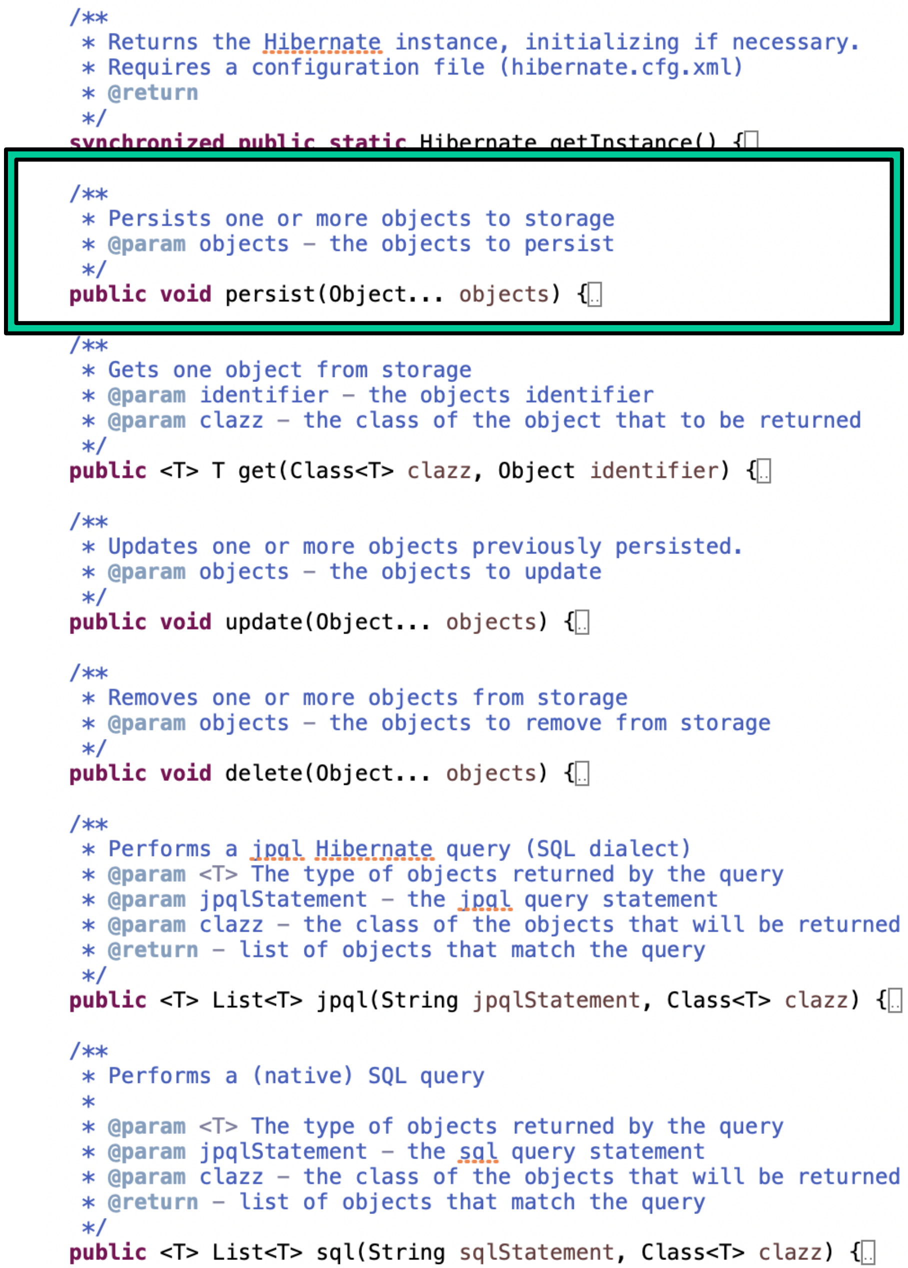


```java
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {□

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {□

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {□

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {□

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {□

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {□

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {□
```

We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

It uses the singleton pattern, hence an instance of it can be obtained using the getInstance static method.

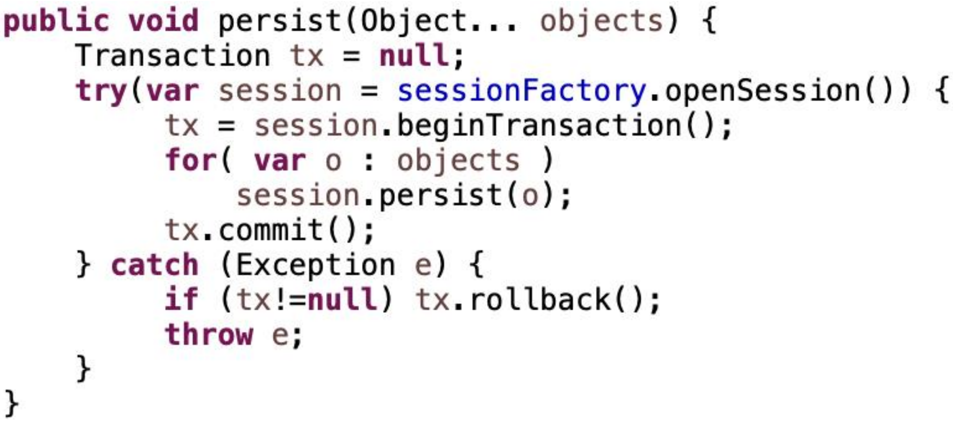
AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```java
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {□

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {□
/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {□

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {□

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {□

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {□

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {□
```

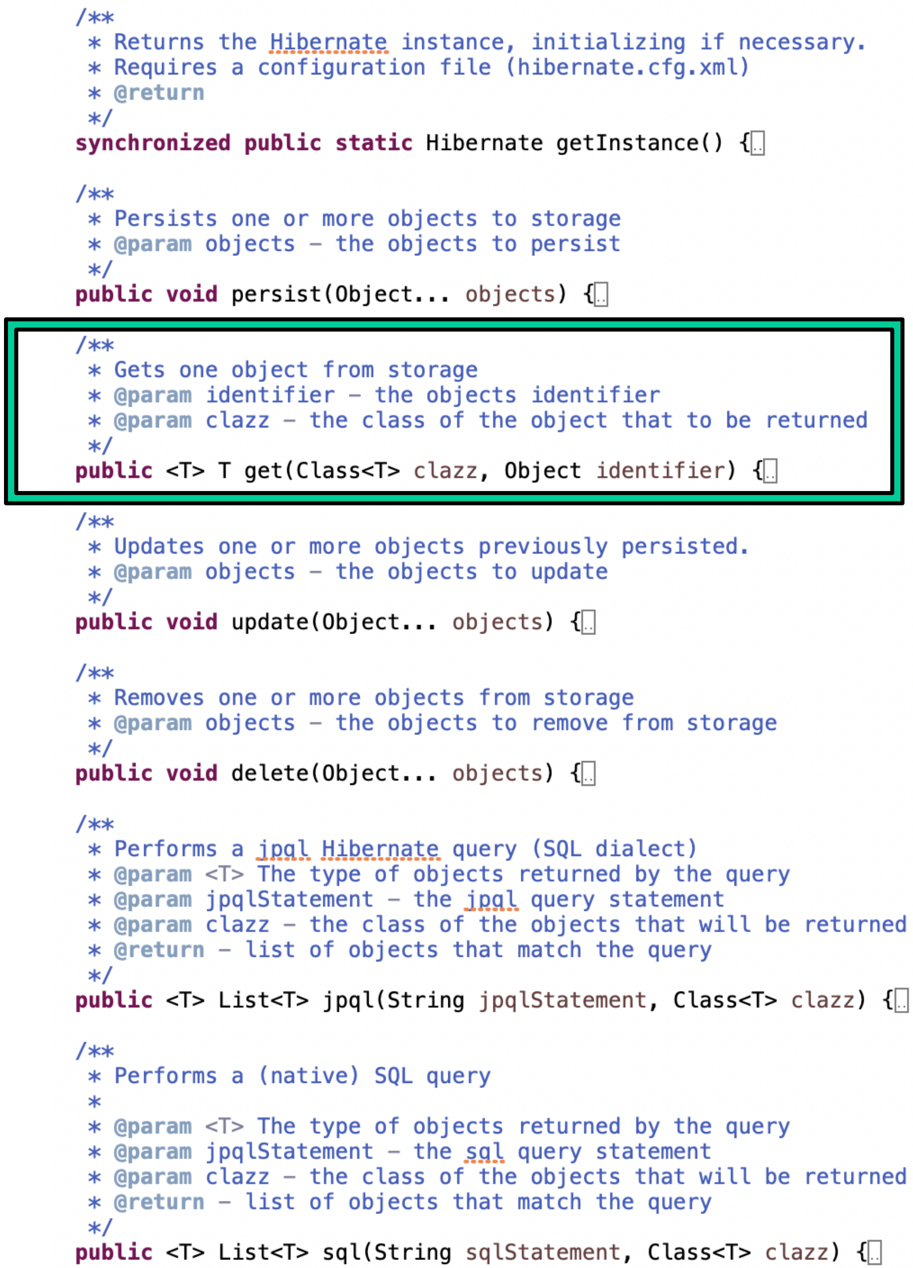
We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can use the method persist (passing as argument a list of Objects annotated as described previously) to be persisted to the database.



```java
public void persist(Object... objects) {
    Transaction tx = null;
    try(var session = sessionFactory.openSession()) {
        tx = session.beginTransaction();
        for( var o : objects )
            session.persist(o);
        tx.commit();
    } catch (Exception e) {
        if (tx!=null) tx.rollback();
        throw e;
    }
}
```

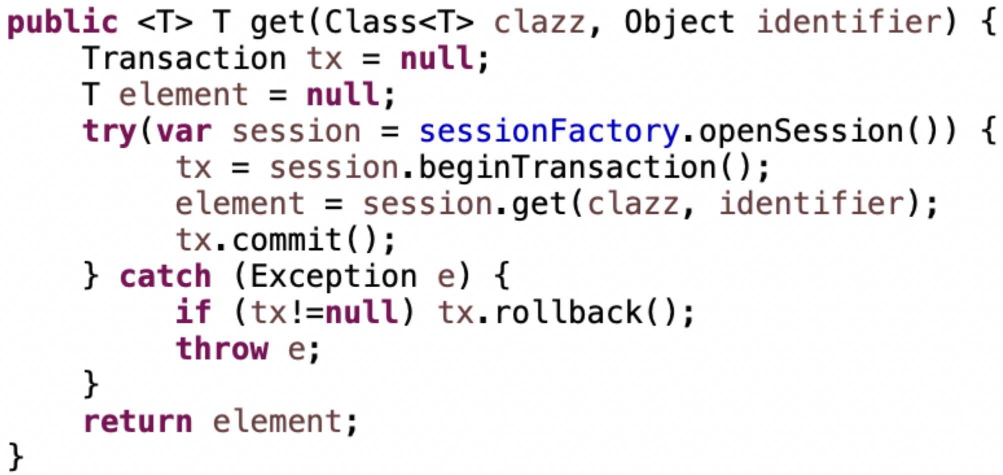
AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {□

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {□

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {□

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {□

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {□

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {□

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {□
```

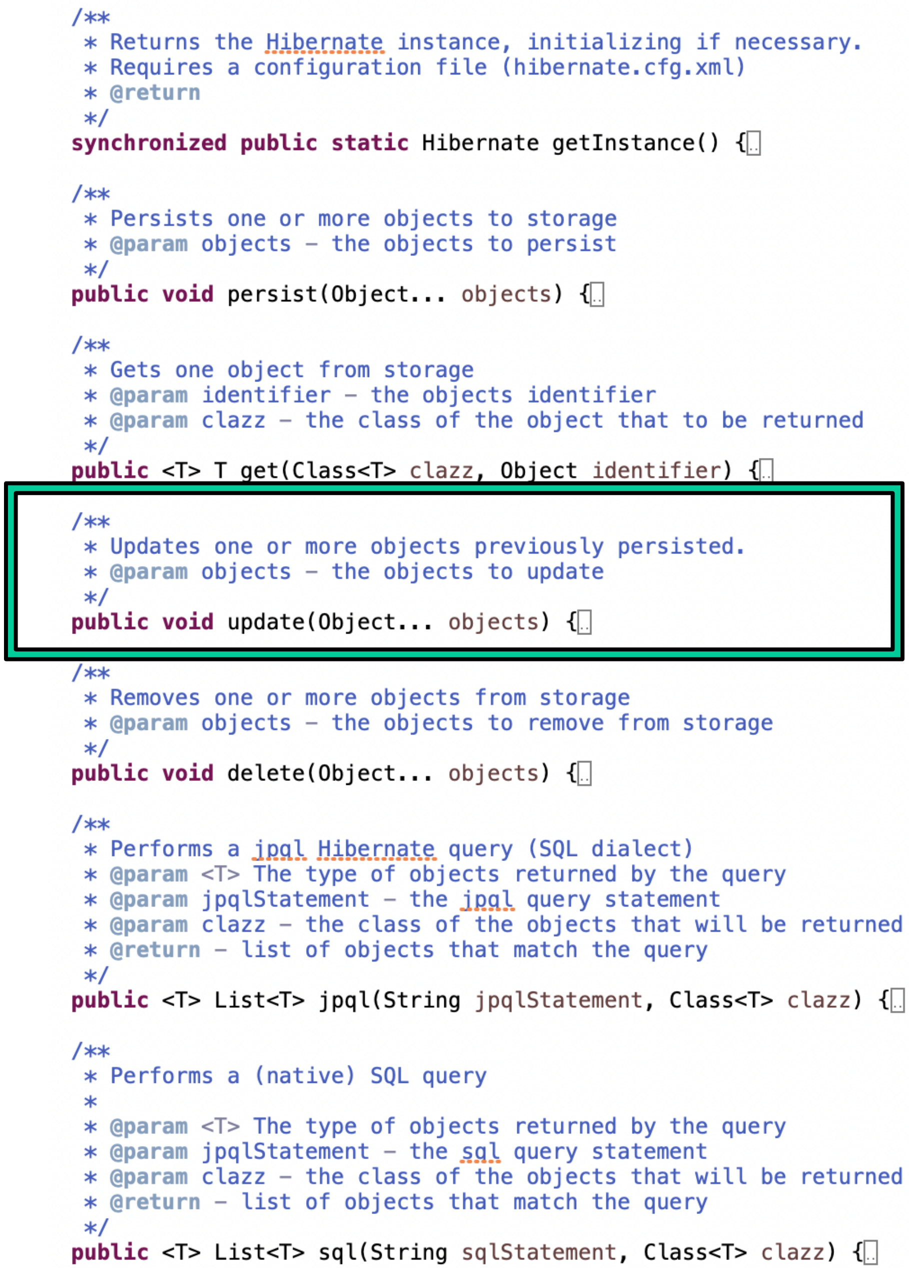
We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can use the method get to obtain an object previously persisted by indicating the class of the object and the value that serves as primary key of that object.



```
public <T> T get(Class<T> clazz, Object identifier) {
    Transaction tx = null;
    T element = null;
    try(var session = sessionFactory.openSession()) {
        tx = session.beginTransaction();
        element = session.get(clazz, identifier);
        tx.commit();
    } catch (Exception e) {
        if (tx!=null) tx.rollback();
        throw e;
    }
    return element;
}
```

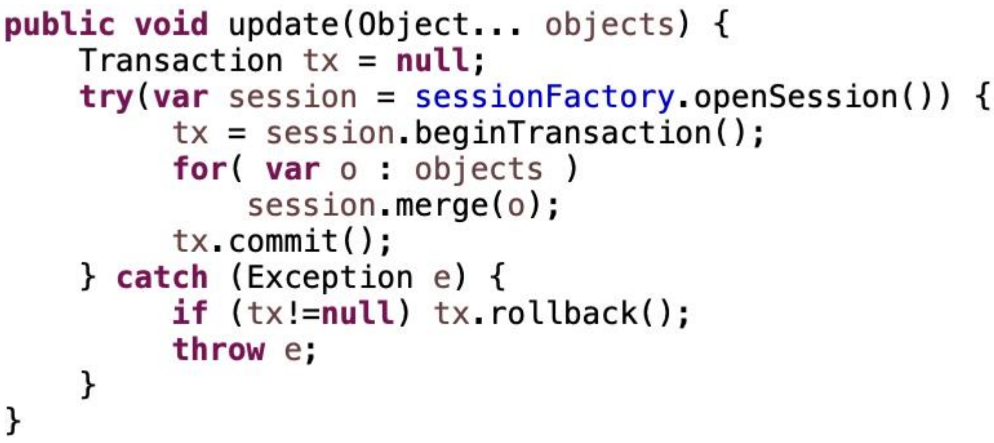
AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {□

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {□

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {□

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {□

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {□

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {□

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {□
```

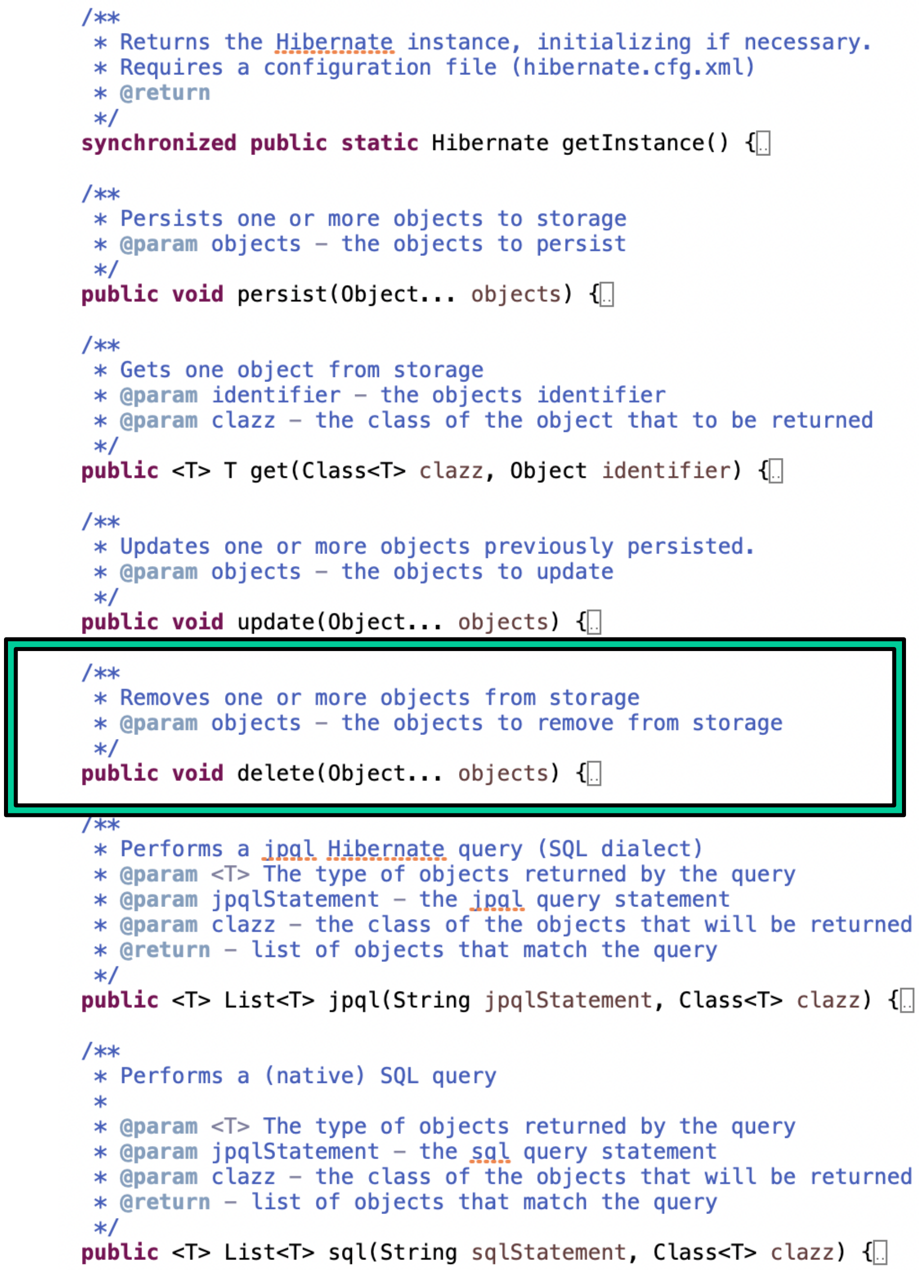
We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can use the update method to update multiple objects at the same time (variable number of arguments) within the context of a single transaction.



```
public void update(Object... objects) {
     Transaction tx = null;
     try(var session = sessionFactory.openSession()) {
          tx = session.beginTransaction();
          for( var o : objects )
               session.merge(o);
          tx.commit();
     } catch (Exception e) {
          if (tx!=null) tx.rollback();
          throw e;
     }
}
```

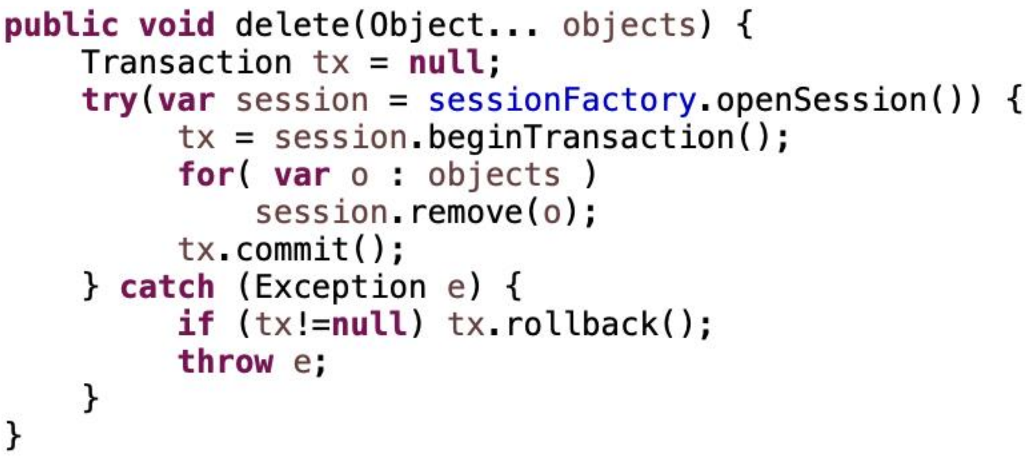
Auxiliary class to interact with Hibernate



```
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {□

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {□

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {□

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {□

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {□

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {□

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {□
```

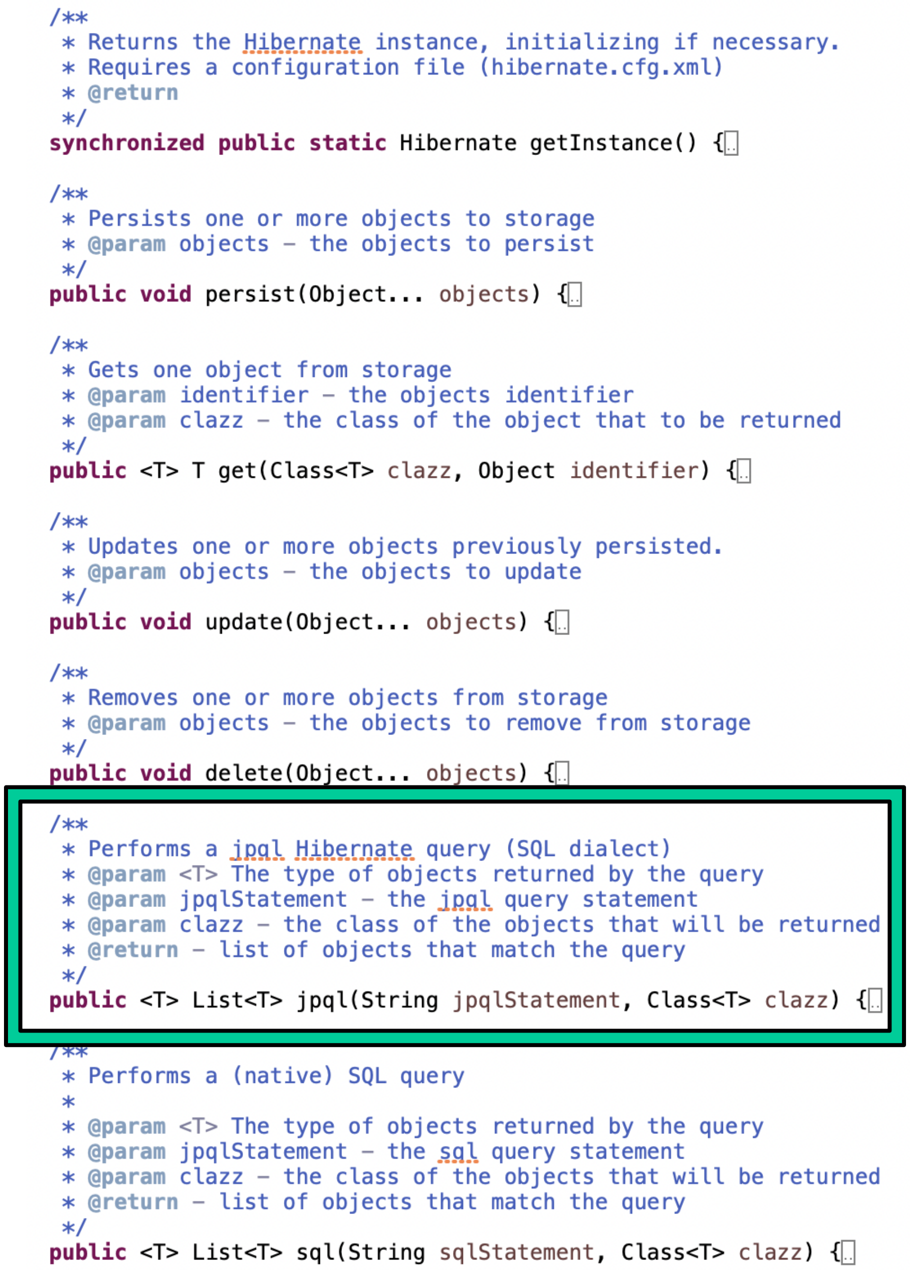
We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can use the delete method to delete multiple objects at the same time (variable number of arguments) within the context of a single transaction.



```
public void delete(Object... objects) {
    Transaction tx = null;
    try(var session = sessionFactory.openSession()) {
        tx = session.beginTransaction();
        for( var o : objects )
            session.remove(o);
        tx.commit();
    } catch (Exception e) {
        if (tx!=null) tx.rollback();
        throw e;
    }
}
```

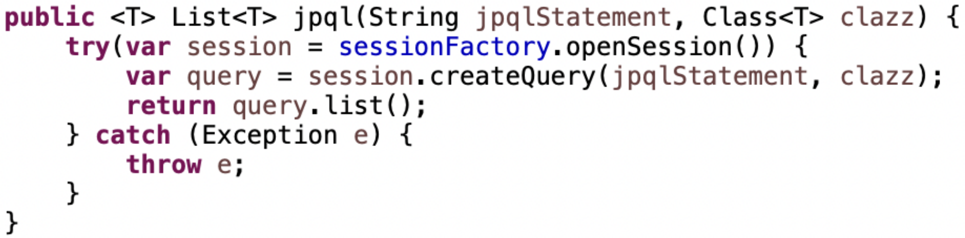
AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {...}

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {...}

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {...}

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {...}

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {...}

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {...}

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {...}
```

We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can execute a jpql (Jakarta Persistence Query Language) statement to execute complex queries over the objects within the database.

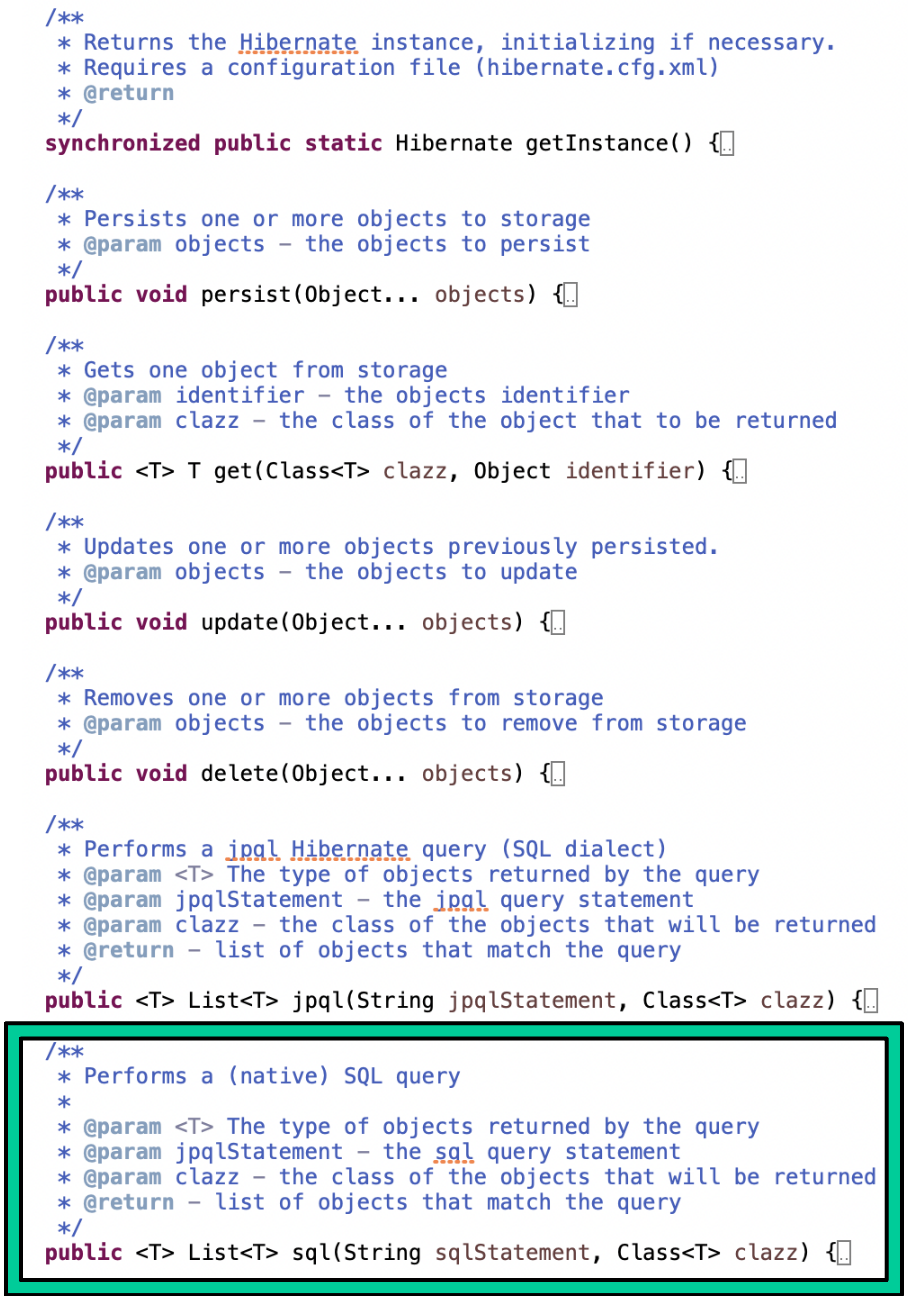


```
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {
    try(var session = sessionFactory.openSession()) {
        var query = session.createQuery(jpqlStatement, clazz);
        return query.list();
    } catch (Exception e) {
        throw e;
    }
}
```

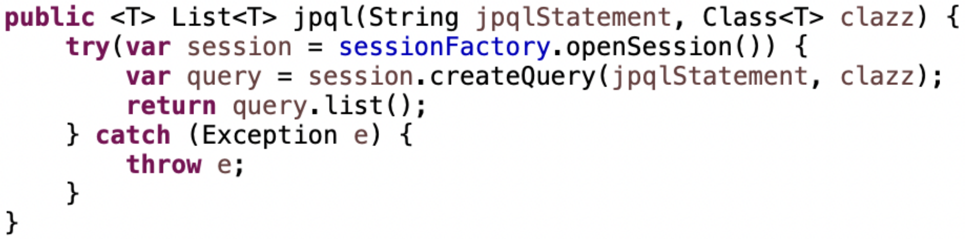
AUXILIARY CLASS TO INTERACT WITH HIBERNATE

We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can execute a native sql (Simple Query Language) statement to execute complex queries over the objects within the database.

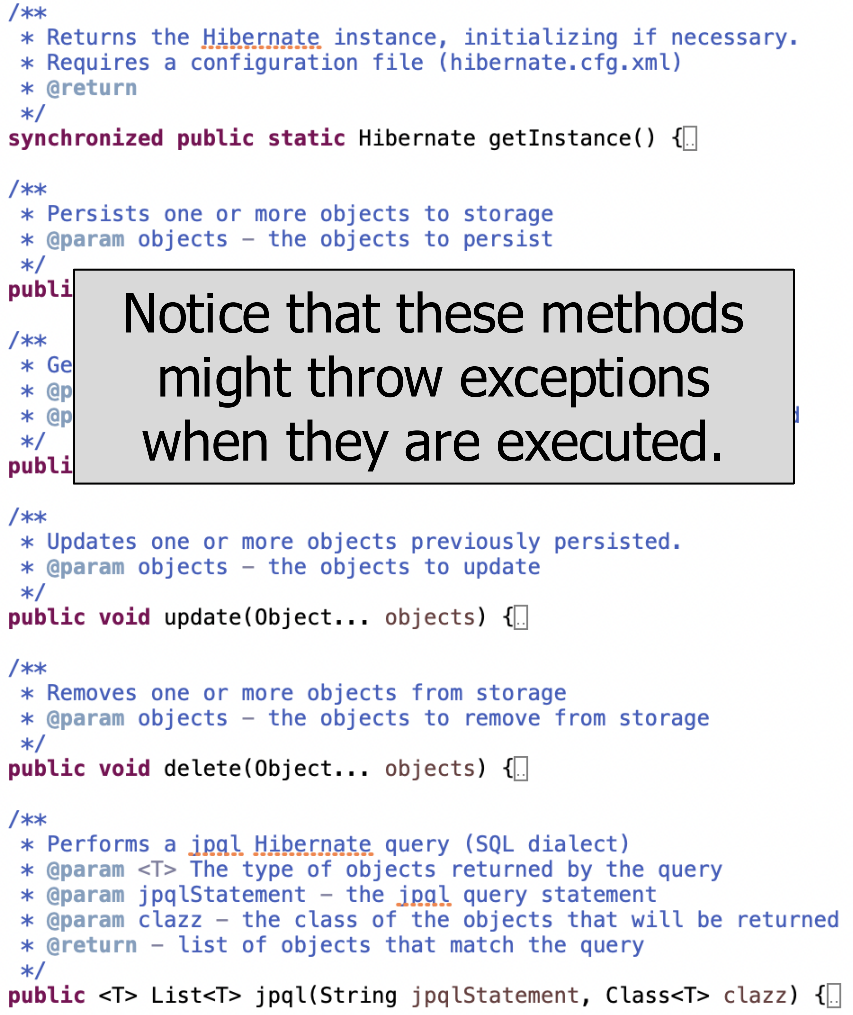


```java
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {…}

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
public void persist(Object... objects) {…}

/**
 * Gets one object from storage
 * @param identifier - the objects identifier
 * @param clazz - the class of the object that to be returned
 */
public <T> T get(Class<T> clazz, Object identifier) {…}

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {…}

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {…}

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {…}

/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {…}
```



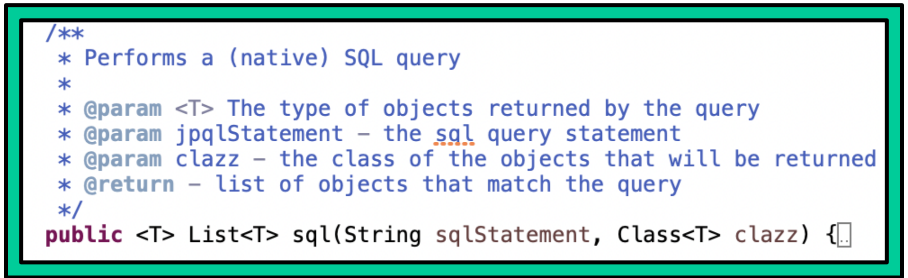
```java
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {
    try(var session = sessionFactory.openSession()) {
        var query = session.createQuery(jpqlStatement, clazz);
        return query.list();
    } catch (Exception e) {
        throw e;
    }
}
```

AUXILIARY CLASS TO INTERACT WITH HIBERNATE



```java
/**
 * Returns the Hibernate instance, initializing if necessary.
 * Requires a configuration file (hibernate.cfg.xml)
 * @return
 */
synchronized public static Hibernate getInstance() {…

/**
 * Persists one or more objects to storage
 * @param objects - the objects to persist
 */
publi

/**
 * Ge
 * @p
 * @p
 */
publi

/**
 * Updates one or more objects previously persisted.
 * @param objects - the objects to update
 */
public void update(Object... objects) {…

/**
 * Removes one or more objects from storage
 * @param objects - the objects to remove from storage
 */
public void delete(Object... objects) {…

/**
 * Performs a jpql Hibernate query (SQL dialect)
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the jpql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {…
```

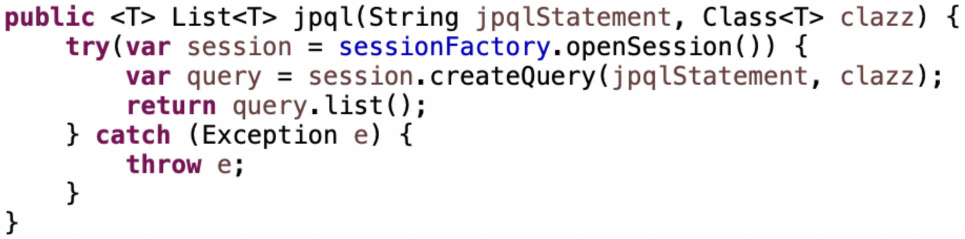
Notice that these methods might throw exceptions when they are executed.



```java
/**
 * Performs a (native) SQL query
 *
 * @param <T> The type of objects returned by the query
 * @param jpqlStatement - the sql query statement
 * @param clazz - the class of the objects that will be returned
 * @return - list of objects that match the query
 */
public <T> List<T> sql(String sqlStatement, Class<T> clazz) {…
```

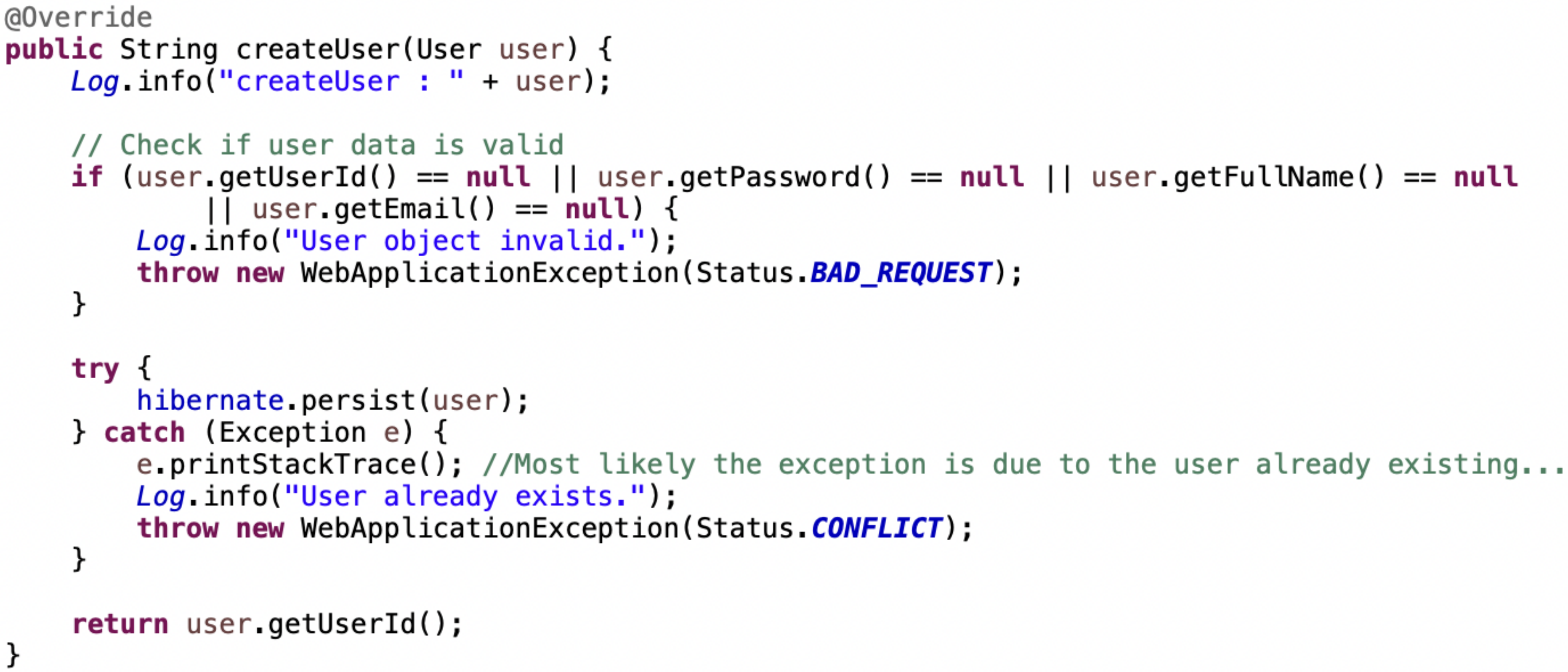
We have provided the class lab3.server.persistence.Hibernate to simplify the interaction with it.

You can execute a native sql (Simple Query Language) statement to execute complex queries over the objects within the database.



```java
public <T> List<T> jpql(String jpqlStatement, Class<T> clazz) {
    try(var session = sessionFactory.openSession()) {
        var query = session.createQuery(jpqlStatement, clazz);
        return query.list();
    } catch (Exception e) {
        throw e;
    }
}
```

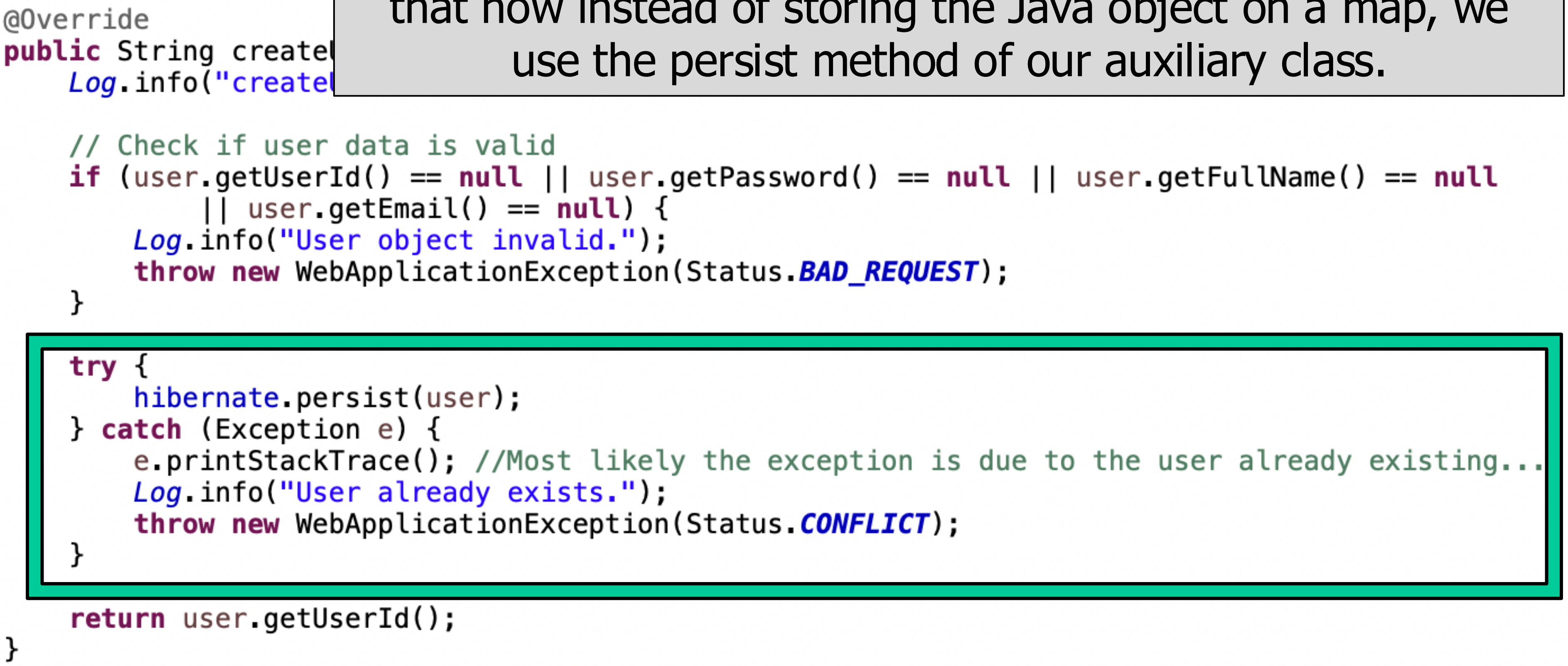
Hibernate simple operations: Create User



```java
@Override
public String createUser(User user) {
    Log.info("createUser : " + user);

    // Check if user data is valid
    if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
            || user.getEmail() == null) {
        Log.info("User object invalid.");
        throw new WebApplicationException(Status.BAD_REQUEST);
    }

    try {
        hibernate.persist(user);
    } catch (Exception e) {
        e.printStackTrace(); //Most likely the exception is due to the user already existing...
        Log.info("User already exists.");
        throw new WebApplicationException(Status.CONFLICT);
    }

    return user.getUserId();
}
```

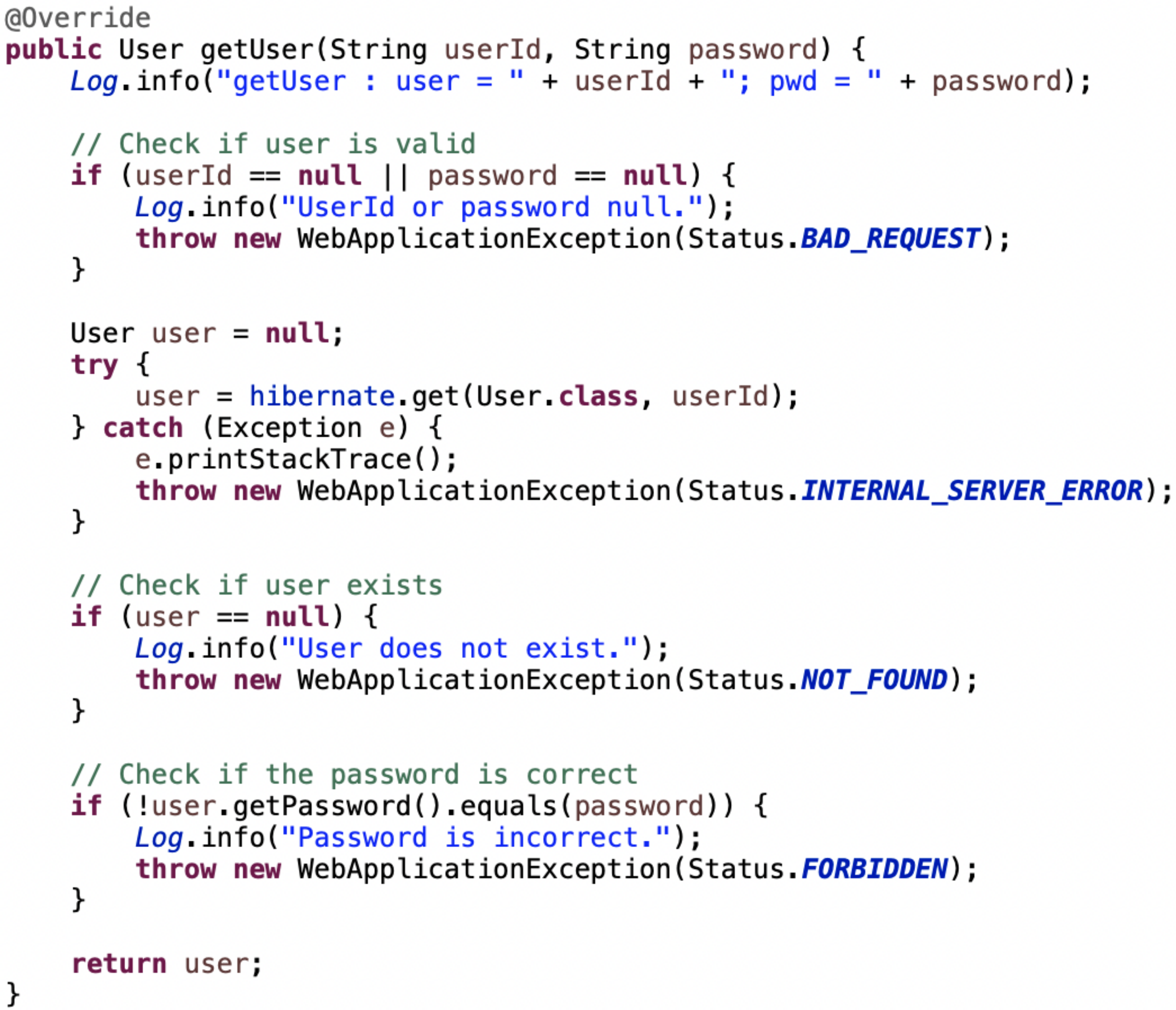
HIBERNATE SIMPLE OPERATIONS: CREATE USER

The code is similar to what you have seen before (error testing and returning the value back to the client), except that now instead of storing the Java object on a map, we use the persist method of our auxiliary class.



```java
@Override
public String create
    Log.info("create

    // Check if user data is valid
    if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
            || user.getEmail() == null) {
        Log.info("User object invalid.");
        throw new WebApplicationException(Status.BAD_REQUEST);
    }

    try {
        hibernate.persist(user);
    } catch (Exception e) {
        e.printStackTrace(); //Most likely the exception is due to the user already existing...
        Log.info("User already exists.");
        throw new WebApplicationException(Status.CONFLICT);
    }

    return user.getUserId();
}
```

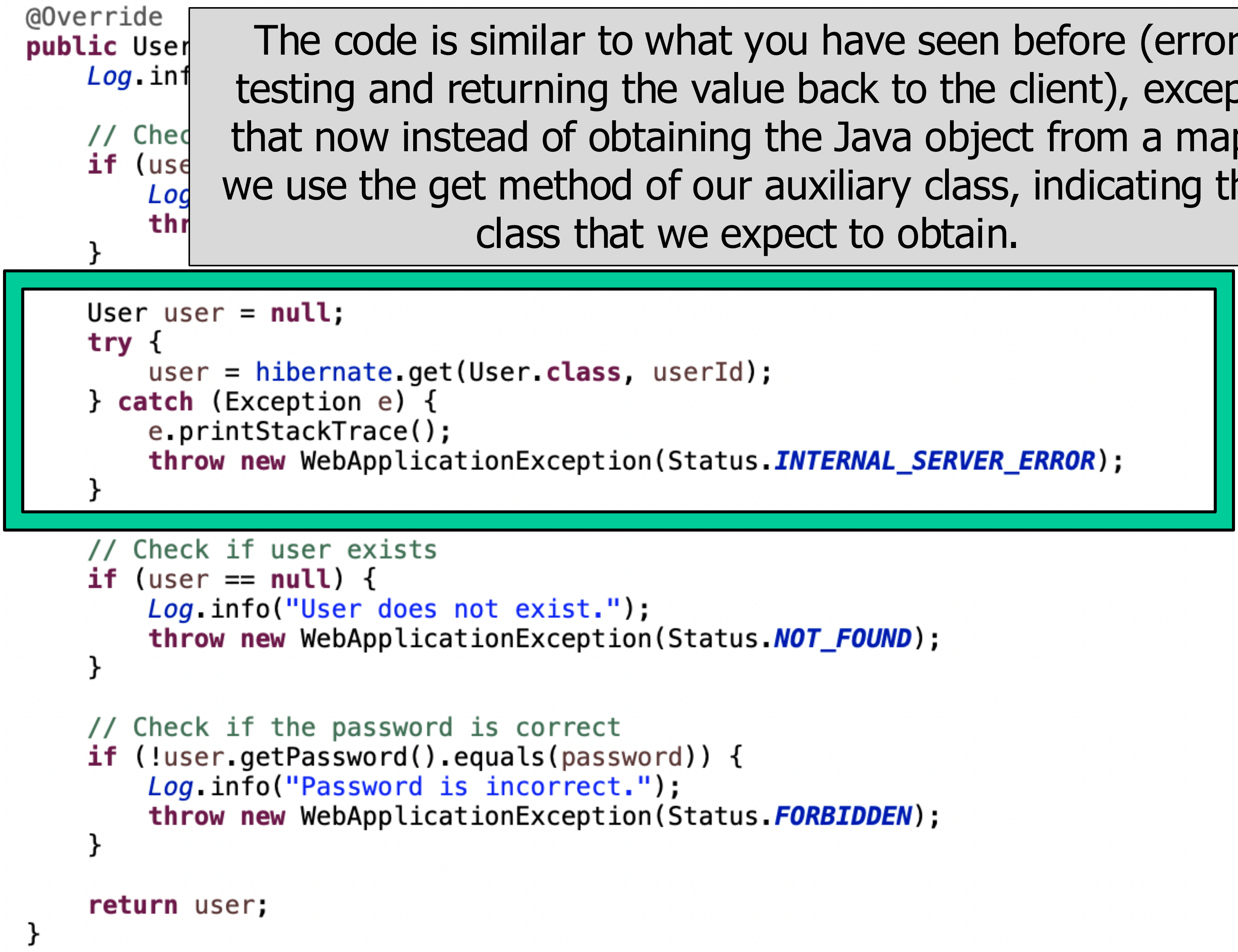
HIBERNATE SIMPLE OPERATIONS: GET USER



```java
@Override
public User getUser(String userId, String password) {
    Log.info("getUser : user = " + userId + "; pwd = " + password);

    // Check if user is valid
    if (userId == null || password == null) {
        Log.info("UserId or password null.");
        throw new WebApplicationException(Status.BAD_REQUEST);
    }

    User user = null;
    try {
        user = hibernate.get(User.class, userId);
    } catch (Exception e) {
        e.printStackTrace();
        throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }

    // Check if user exists
    if (user == null) {
        Log.info("User does not exist.");
        throw new WebApplicationException(Status.NOT_FOUND);
    }

    // Check if the password is correct
    if (!user.getPassword().equals(password)) {
        Log.info("Password is incorrect.");
        throw new WebApplicationException(Status.FORBIDDEN);
    }

    return user;
}
```

HIBERNATE SIMPLE OPERATIONS: GET USER

The code is similar to what you have seen before (error testing and returning the value back to the client), except that now instead of obtaining the Java object from a map, we use the get method of our auxiliary class, indicating the class that we expect to obtain.



```
@Override
public User
    Log.inf

    // Chec
    if (use
        Log
        thr
    }

    User user = null;
    try {
        user = hibernate.get(User.class, userId);
    } catch (Exception e) {
        e.printStackTrace();
        throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }

    // Check if user exists
    if (user == null) {
        Log.info("User does not exist.");
        throw new WebApplicationException(Status.NOT_FOUND);
    }

    // Check if the password is correct
    if (!user.getPassword().equals(password)) {
        Log.info("Password is incorrect.");
        throw new WebApplicationException(Status.FORBIDDEN);
    }

    return user;
}
```

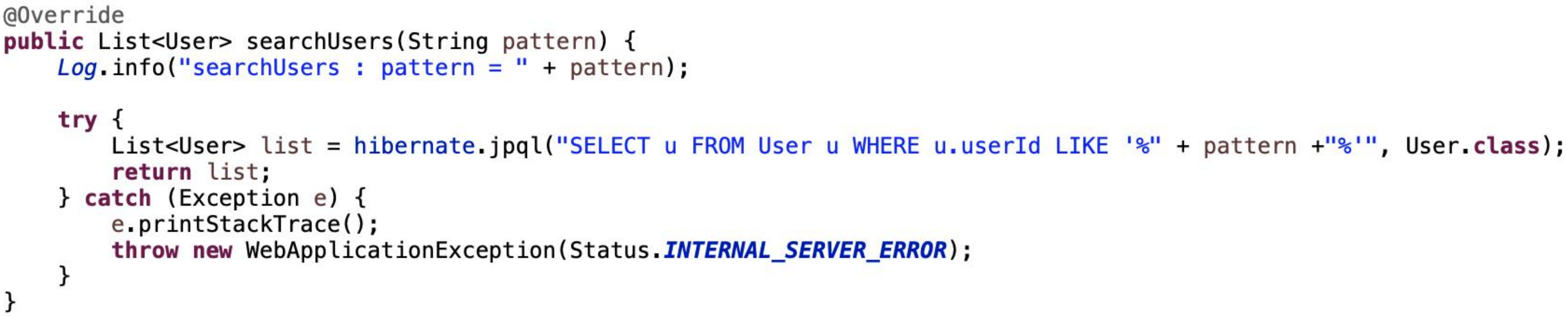
HIBERNATE QUERIES

Has shown before we have, in our auxiliary class, provided methods to support two types of query languages:

- **JPQL** to define searches against persistent entities independent of the mechanism used to store those entities (i.e., the backend storage engine).

- **Native SQL** to define queries using the SQL dialect of the backend storage engine (i.e., the underlying database)

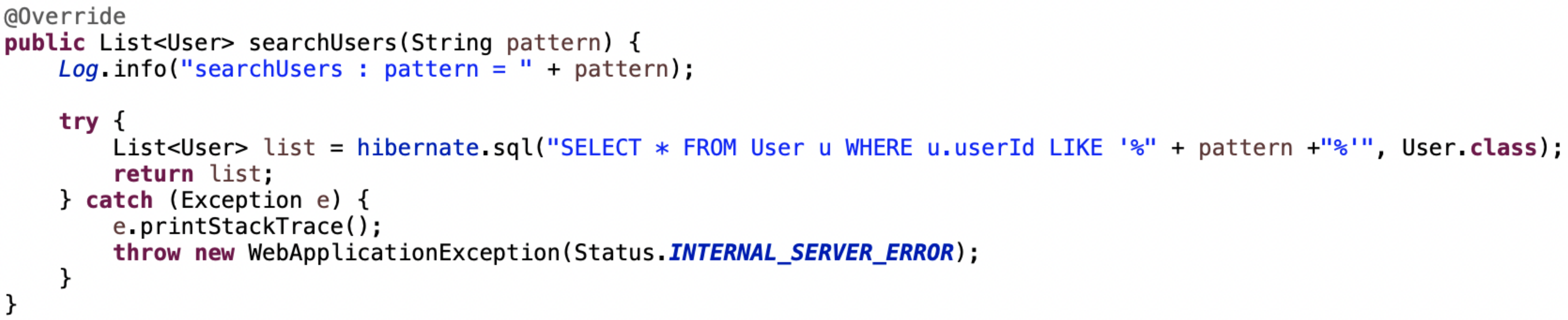
Hibernate Queries: JPQL



```
@Override
public List<User> searchUsers(String pattern) {
    Log.info("searchUsers : pattern = " + pattern);

    try {
        List<User> list = hibernate.jpql("SELECT u FROM User u WHERE u.userId LIKE '%" + pattern +"%'", User.class);
        return list;
    } catch (Exception e) {
        e.printStackTrace();
        throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }
}
```

SELECT u FROM User u WHERE u.userId LIKE '%jl%'

HIBERNATE QUERIES: NATIVE SQL



```
@Override
public List<User> searchUsers(String pattern) {
    Log.info("searchUsers : pattern = " + pattern);

    try {
        List<User> list = hibernate.sql("SELECT * FROM User u WHERE u.userId LIKE '%" + pattern +"%'", User.class);
        return list;
    } catch (Exception e) {
        e.printStackTrace();
        throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
    }
}
```

SELECT * FROM User u WHERE u.userId LIKE '%jl%'

FURTHER CONTROL THE MAPPING TO TABLES

Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User (avatar varbinary(255), email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

FURTHER CONTROL THE MAPPING TO TABLES

Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User (avatar varbinary(255), email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

Each type of POJO that we can persist is stored in its own table (by default the name of the table will be the name of the POJO).

FURTHER CONTROL THE MAPPING TO TABLES

Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User (avatar varbinary(255), email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

Elements that are part of the primary key (i.e., have the @Id annotation) will have a not null constraint.

FURTHER CONTROL THE MAPPING TO TABLES

Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User avatar varbinary(255) email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

The class attribute avatar that has the type URI is converted to varbinary(255): Binary information with at most 255 bytes.

FURTHER CONTROL THE MAPPING TO TABLES

Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User (avatar varbinary(255), email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

Every String is converted into type varchar(255), textual format with at most 255 characters.

FURTHER CONTROL THE MAPPING TO TABLES

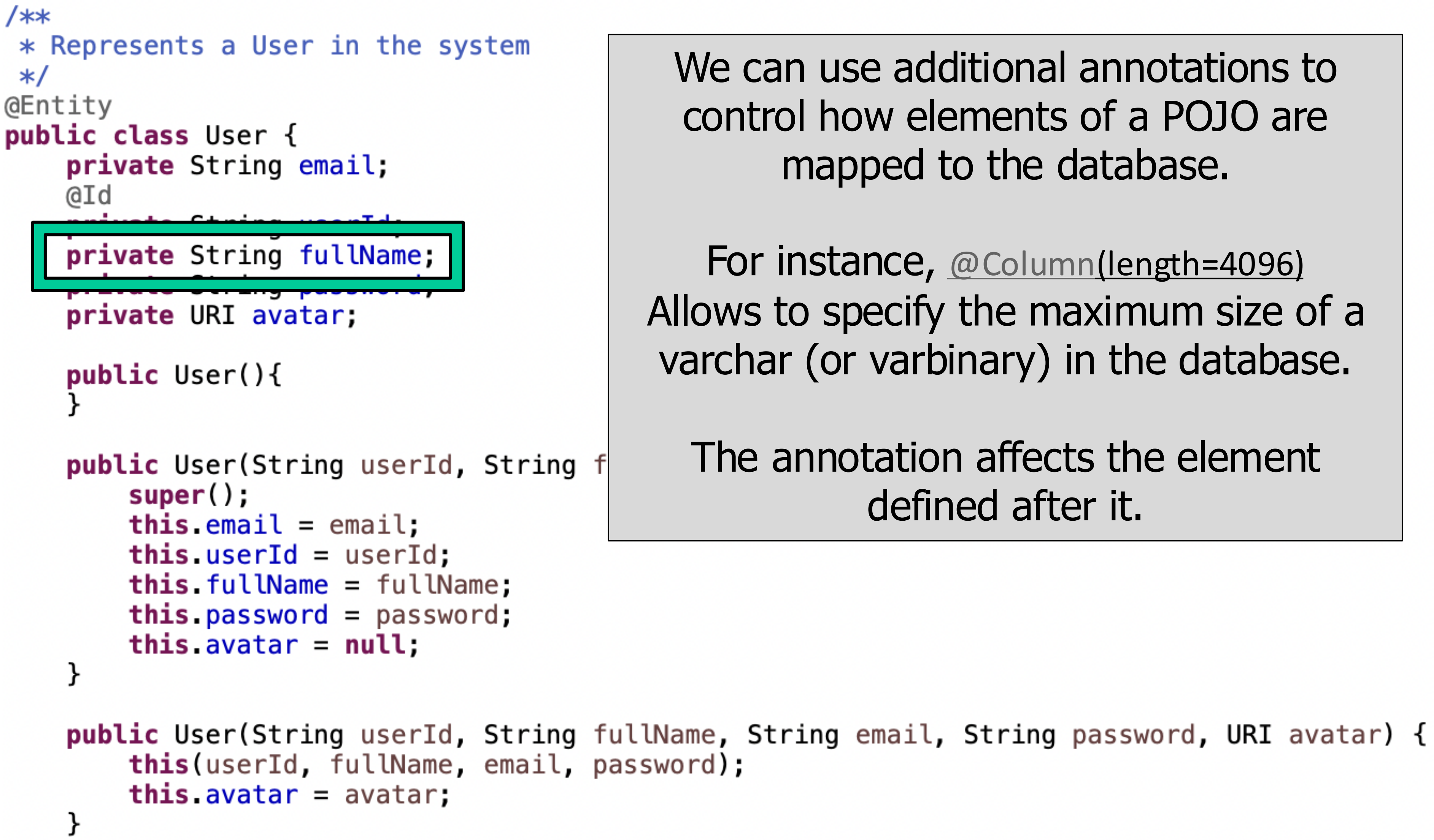
Since we have setup the option to show the queries executed over the database, we can see the effects of operations that we execute over our server in the server terminal.

For instance, we we persist the first User object, you can see the creation of the table.

```
Hibernate: create table User (avatar varbinary(255), email
varchar(255), fullName varchar(255), password varchar(255),
userId varchar(255) not null, primary key (userId))
```

What if one of these variables are required to store more than 255 characters?

Additional Annotation on POJOs



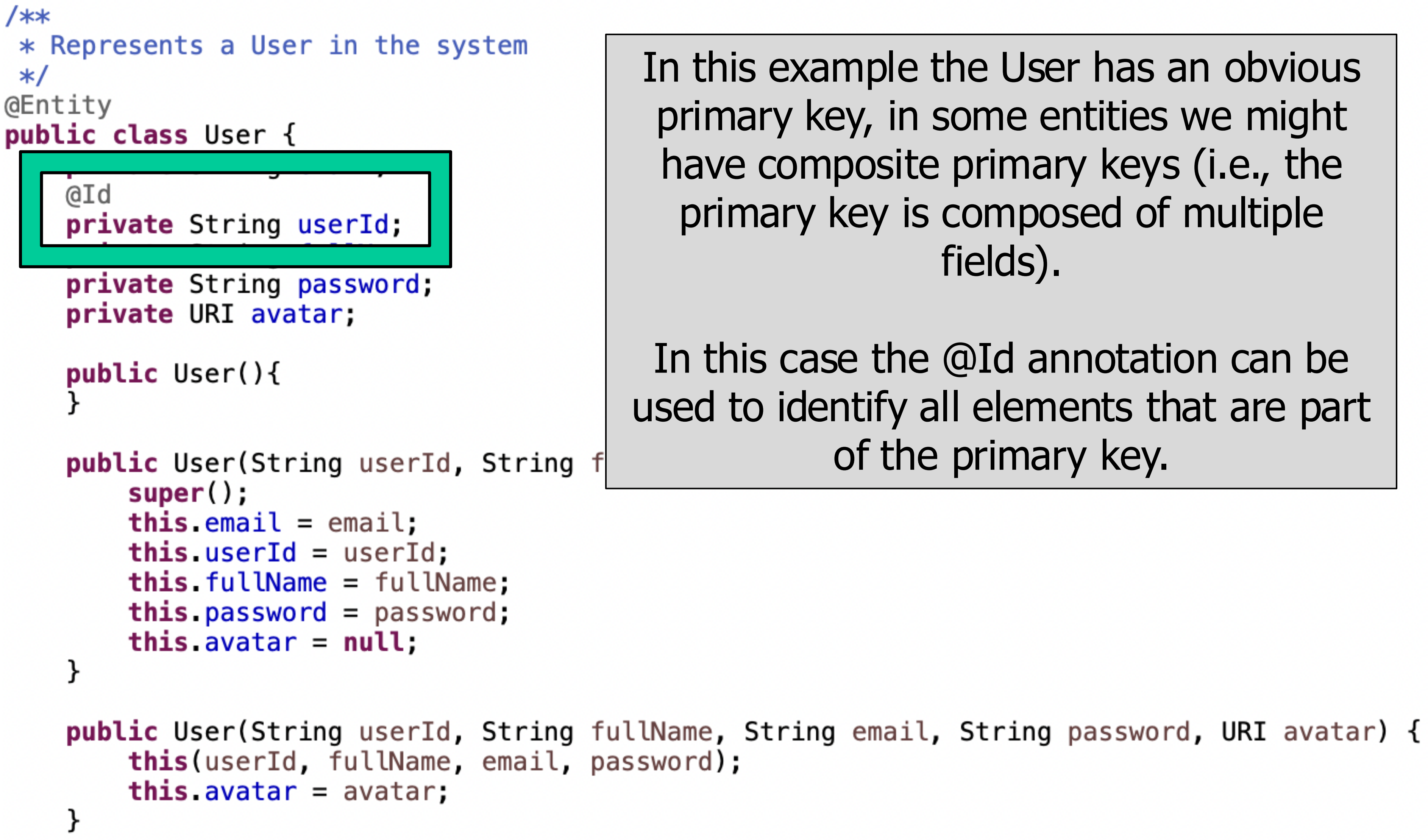
```
/**
 * Represents a User in the system
 */
@Entity
public class User {
    private String email;
    @Id
    private String userId;
    private String fullName;
    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String f
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

We can use additional annotations to control how elements of a POJO are mapped to the database.

For instance, @Column(length=4096)
Allows to specify the maximum size of a varchar (or varbinary) in the database.

The annotation affects the element defined after it.

ADDITIONAL ANNOTATION ON POJOS

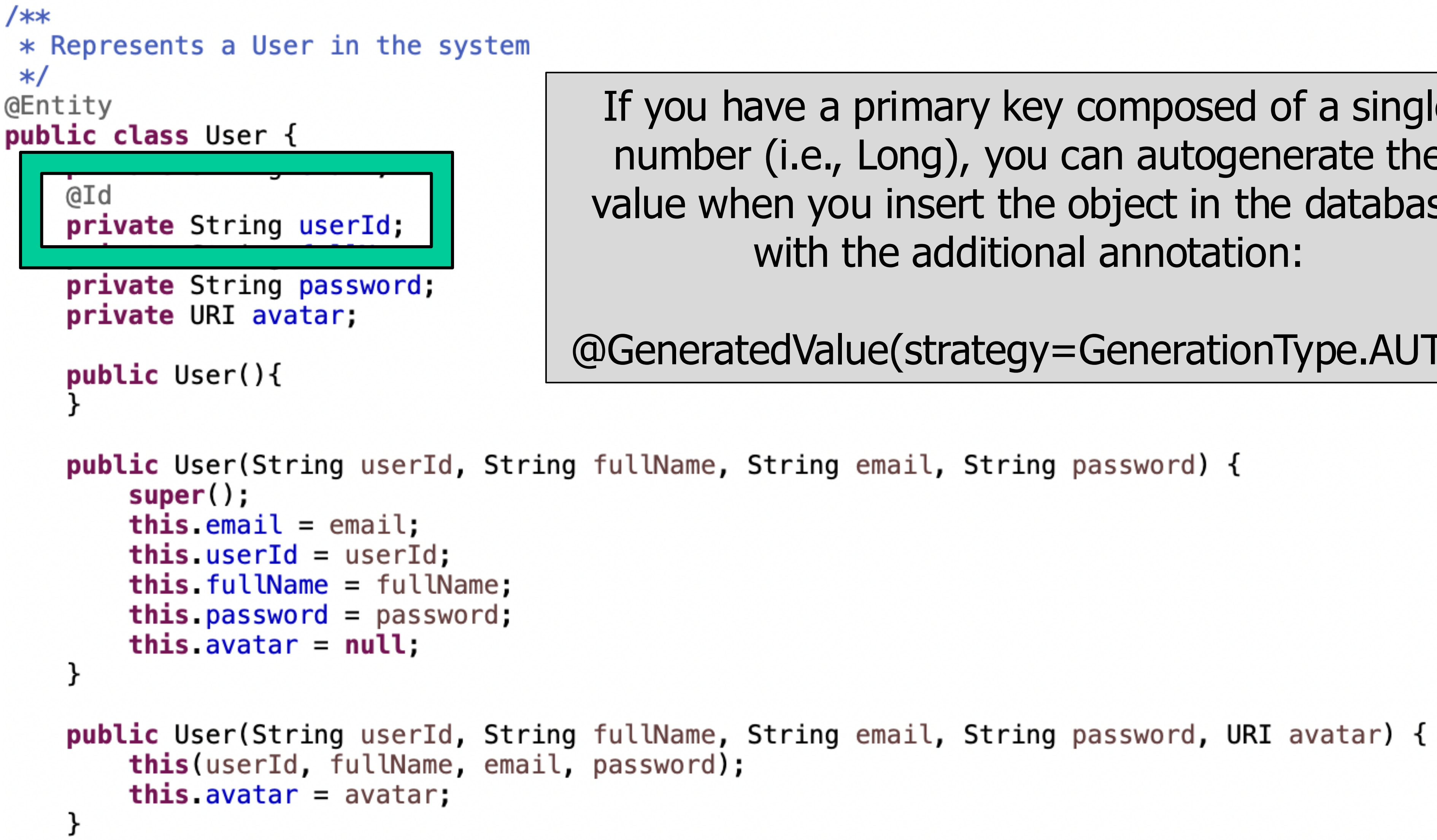


```
/**
 * Represents a User in the system
 */
@Entity
public class User {

    @Id
    private String userId;

    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String f
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

In this example the User has an obvious primary key, in some entities we might have composite primary keys (i.e., the primary key is composed of multiple fields).

In this case the @Id annotation can be used to identify all elements that are part of the primary key.

ADDITIONAL ANNOTATION ON POJOS



```
/**
 * Represents a User in the system
 */
@Entity
public class User {
    @Id
    private String userId;
    private String password;
    private URI avatar;

    public User(){
    }

    public User(String userId, String fullName, String email, String password) {
        super();
        this.email = email;
        this.userId = userId;
        this.fullName = fullName;
        this.password = password;
        this.avatar = null;
    }

    public User(String userId, String fullName, String email, String password, URI avatar) {
        this(userId, fullName, email, password);
        this.avatar = avatar;
    }
```

If you have a primary key composed of a single number (i.e., Long), you can autogenerate the value when you insert the object in the database with the additional annotation:

@GeneratedValue(strategy=GenerationType.AUTO)

EXERCISE

1. Test the client provided that already has the logic for doing retries (CreateUser operation), complete the remaining operations and based on the clients provided last week create those clients.

2. Complete the logic of the operation of the UsersResource (you can use what you have done last week) by storing data managed by the service to Hibernate (classes to support this are provided in the lab materials).

3. We are now desegregating the logic to manage images (avatars) to a new service, for which an interface (more general) is already provided. Create this new server and modify the logic of the UsersResource to now store a full URI of the user avatar (if one exists).