1) Answering Questions

Q: What is ORM and how is JPA used?
A: ORM stands for Object-Relational Mapping, which facilitates the conversion of data between object-oriented programming languages and relational databases. JPA, or Java Persistence API, is a specification in Java for ORM. It provides interfaces and annotations for developers to map Java objects to database tables and vice versa. JPA implementations like Hibernate and EclipseLink provide the actual implementation of these specifications.

Q: What is the application.properties used for and where must it be stored?
A: The `application.properties` file is a configuration file commonly used in Spring Boot applications. It stores configuration properties such as database connection details, server port, and logging settings. This file must be stored in the `src/main/resources` directory of a Spring Boot project. It's utilized for configuring the application at runtime.

Q: Which annotations are frequently used for entity types? Which key points must be observed?
A: Frequently used annotations for entity types in JPA include `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`, and `@ManyToOne`/`@OneToMany` for defining relationships between entities. Key points to observe include ensuring that each entity has a primary key (`@Id`), specifying the table name (`@Table`) if it differs from the class name, and correctly mapping relationships between entities using appropriate annotations.

Q: What methods do you need for CRUD operations?
A: For CRUD operations (Create, Read, Update, Delete) in JPA, you typically need the following methods:

1. **Create (Insert)**: `EntityManager.persist()` or `CrudRepository.save()` method.
2. **Read (Select)**: `EntityManager.find()` or `CrudRepository.findById()` method for single record retrieval, and `EntityManager.createQuery()` or `CrudRepository.findAll()` method for multiple record retrieval.
3. **Update (Modify)**: `EntityManager.merge()` or `CrudRepository.save()` method.
4. **Delete (Remove)**: `EntityManager.remove()` or `CrudRepository.delete()` method.

2) GKÜ documentation

1. installing MySQL via Docker-compose:

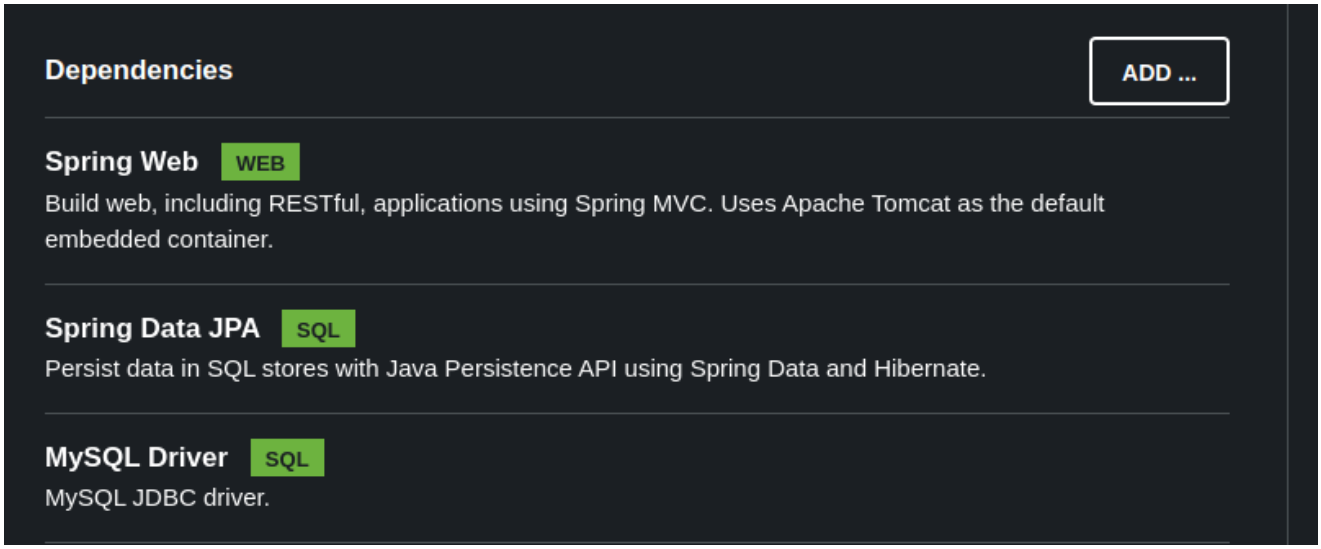
```
# Use root/example as user/password credentials
version: '3.1'

services:
```

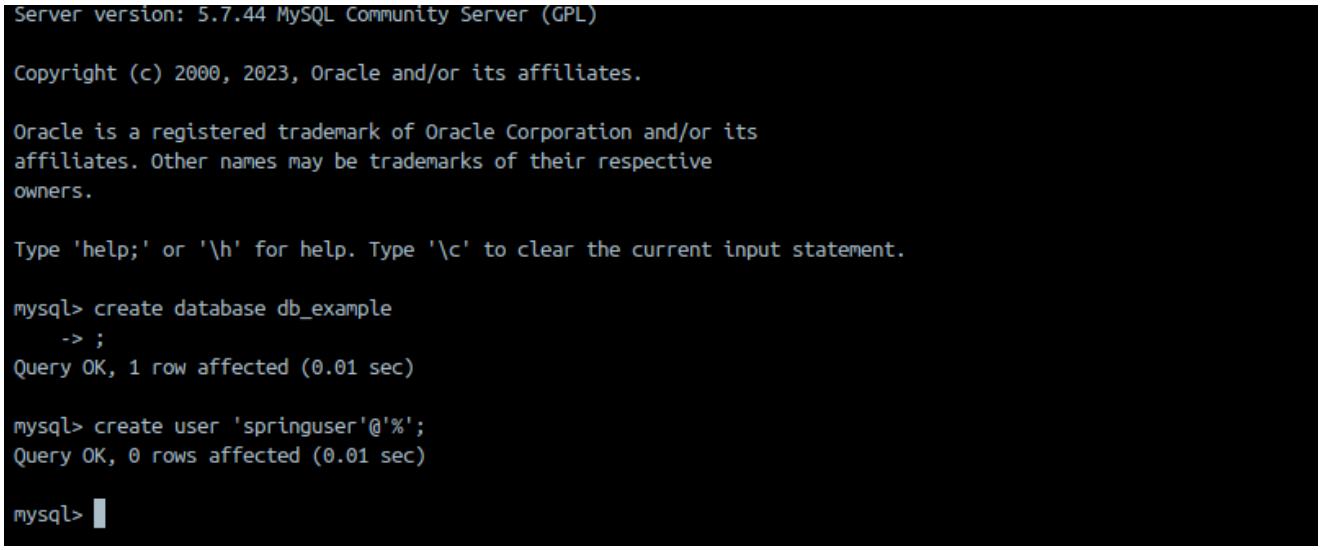
```
  db:
    image: mysql
    # NOTE: use of "mysql_native_password" is not recommended:
https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-
series.html#upgrade-caching-sha2-password
    # (this is just an example, not intended to be a production
configuration)
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example

  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080
```

2. Using the spring initialiazr with these dependencies to initialize the project:



3. create a test table and user that can do queries in the mysql DB:



4. application properties:

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true

5. adding User Entity, the Repository and a controller to handle Requests
6. Testing the program:

   By curling $ curl http://localhost:8080/demo/add -d name=First -d email=someemail@someemailprovider.com

   the output is now "Saved"

   and a simple $ curl http://localhost:8080/demo/all

   outputs all the entries in this table

2) GKV

For This I had to simply change the Entities (in our case the warehouse and products) and their respective Repositories.

```
@Entity
public class Warehouse {

    @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    @JsonIgnore
    private List<Product> products;

    @Column
    @Id    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
```

Here you can see the broad variety of Annotations used by JPA to signal different things such as an Entity, a "OneToMany" Association and a Column in the DB.

I also had to add a new Controller to handle incoming and outgoing Data, which looked like the following:

```
@RestController
@RequestMapping("/api/warehouses")
```

```
public class StorageManagementController {

    private Warehouse generateWarehouseWithProducts() {
        Warehouse warehouse = new Warehouse();
        warehouse.setName("New Warehouse " + new Random().nextInt(1000));
        warehouse.setAddress("New Address " + new Random().nextInt(100));
        warehouse.setParkinslots(new Random().nextInt(50));
        warehouse.setStorage(new Random().nextInt(10000));

        List<Product> products = createProducts(warehouse);
        warehouse.setProducts(products);

        return warehouse;
    }
```

Here you can see how the general route for this application is /api/warehouses and a simple warehouse generator to create a random Warehouse which is then used in the /add route for example when writing a new warehouse into the DB