Software Architecture and Methodologies

Corso di Laurea Magistrale in Ingegneria Informatica

Università degli Studi di Firenze

Online Videogame

Victor Soto Berenguer (7078230)

2022

Index

Introduction

In this project I am extending the application developed for the course of *Ingegneria del Software* of the *Laurea Triennale* as the back end of a Restful architecture.

The project is connected to a **database** in localhost called “assignment-restful-architecture” with **user** “java-client” and **password** “password”.

Tools

The application was developed in Java with the IDE Eclipse (for Enterprise Java and Web developers), as a Maven project.

JPA was used to define how the entities of the Domain Model should be mapped in the database.

CDI was used to manage the lifecycle of different components.

JAX-RS was used to expose endpoint services.

Finally, JUnit was used to perform unit tests, along with several tools like Hamcrest, Rest Assured, or Mockito.

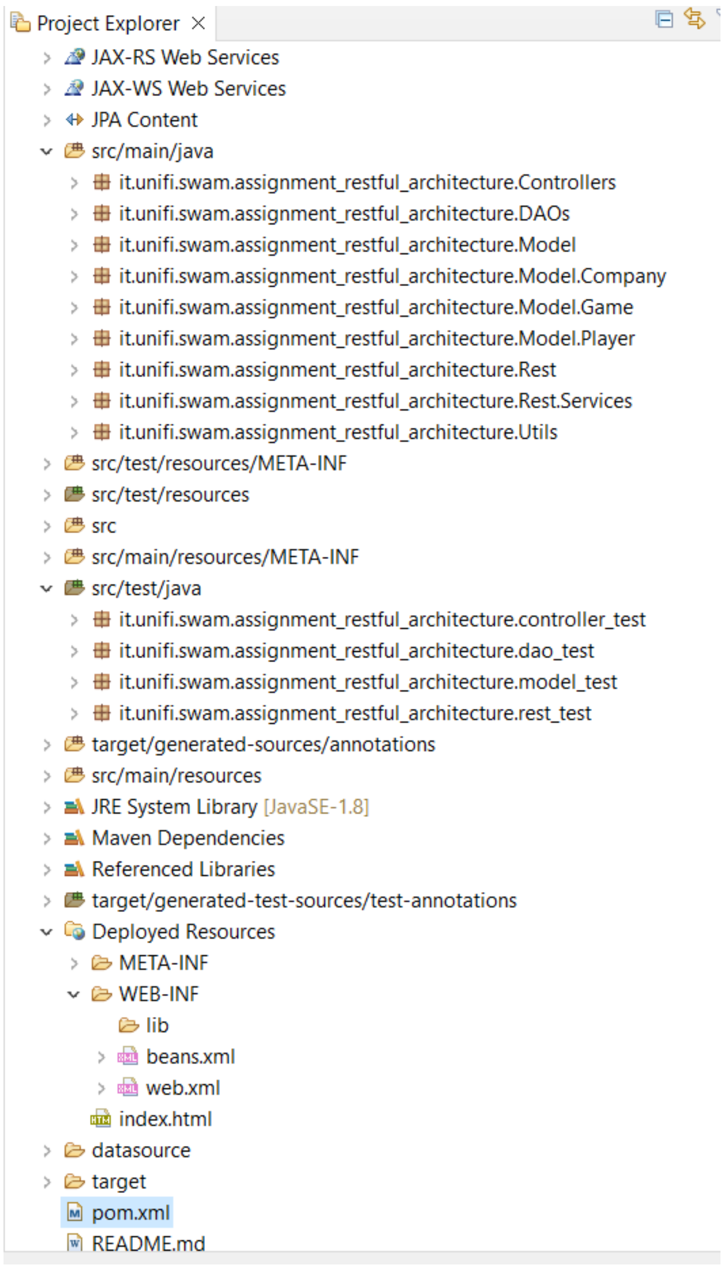
Elements added to the classpath in Java Build Path: java-jwt-3.19.2.jar, json-20220320.jar, mysql-connector-java-8.0.19.jar and spring-security-crypto-5.7.1.jar

All dependencies included in the pom.xml file are what follows:

```
<dependencies>
		<dependency>
				<groupId>junit</groupId>
				<artifactId>junit</artifactId>
				<version>4.12</version>
				<scope>test</scope>
		</dependency>
		<dependency>
				<groupId>javax</groupId>
				<artifactId>javaee-api</artifactId>
				<version>8.0.1</version>
				<scope>provided</scope>
		</dependency>
		<dependency>
				<groupId>org.hibernate</groupId>
				<artifactId>hibernate-core</artifactId>
				<version>5.4.13.Final</version>
				<scope>provided</scope>
		</dependency>
		<dependency>
				<groupId>com.h2database</groupId>
				<artifactId>h2</artifactId>
				<version>1.4.195</version>
				<scope>test</scope>
		</dependency>
		<dependency>
				<groupId>org.apache.commons</groupId>
				<artifactId>commons-lang3</artifactId>
				<version>3.12.0</version>
		</dependency>
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-databind</artifactId>
			<version>2.9.6</version>
		</dependency>
		<dependency>
			<groupId>com.jayway.restassured</groupId>
			<artifactId>rest-assured</artifactId>
			<version>2.9.0</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>org.hamcrest</groupId>
			<artifactId>hamcrest-all</artifactId>
			<version>1.3</version>
		</dependency>
		<dependency>
			<groupId>org.mockito</groupId>
			<artifactId>mockito-core</artifactId>
			<version>3.9.0</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>com.auth0</groupId>
			<artifactId>java-jwt</artifactId>
			<version>3.19.2</version>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt</artifactId>
			<version>0.7.0</version>
		</dependency>
		<dependency>
			<groupId>com.lambdaworks</groupId>
			<artifactId>scrypt</artifactId>
			<version>1.4.0</version>
		</dependency>
		<dependency>
			<groupId>org.springframework.security</groupId>
			<artifactId>spring-security-crypto</artifactId>
			<version>5.7.1</version>
		</dependency>
		<dependency>
			<groupId>org.json</groupId>
			<artifactId>json</artifactId>
			<version>20220320</version>
		</dependency>
</dependencies>
```

pom.xl dependencies

Project Organization

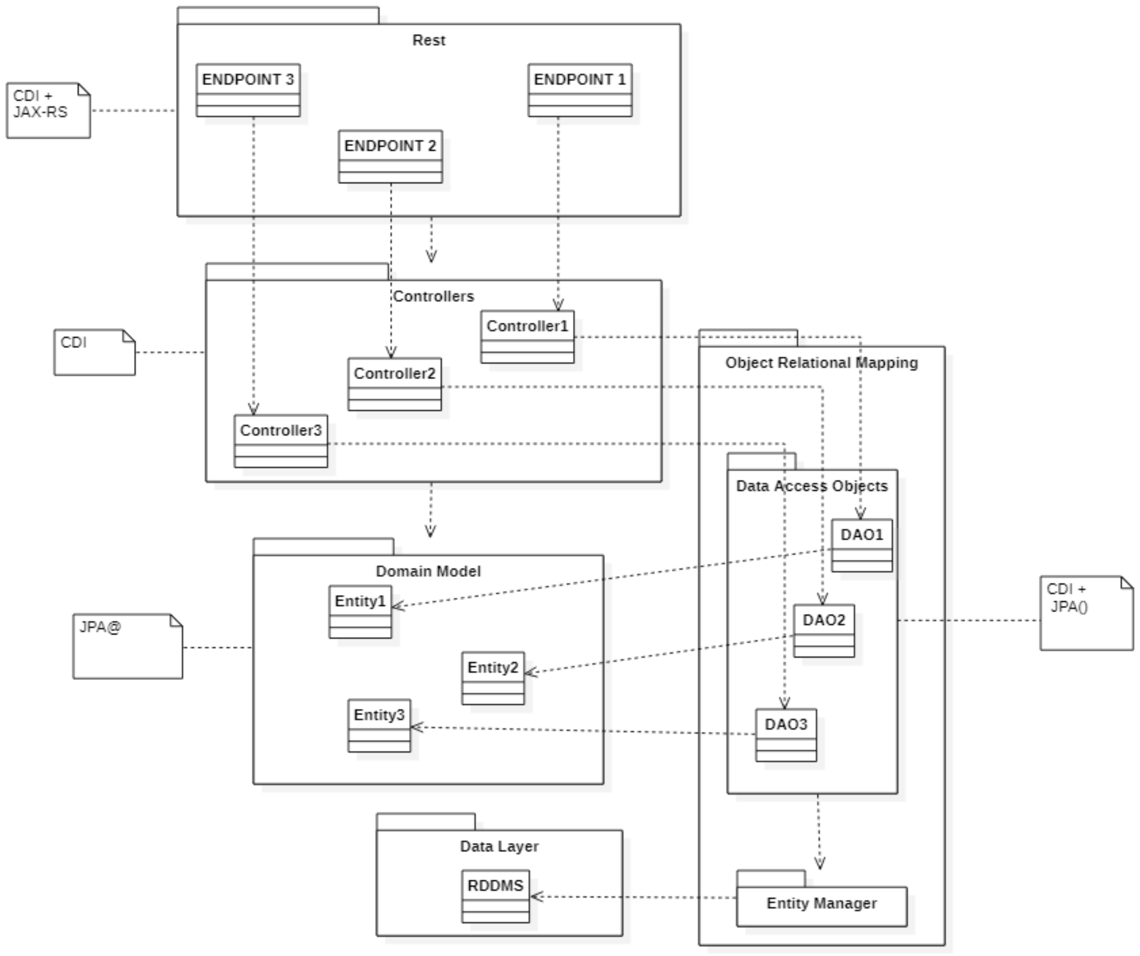


```
Project Explorer
> JAX-RS Web Services
> JAX-WS Web Services
> JPA Content
v src/main/java
    > it.unifi.swam.assignment_restful_architecture.Controllers
    > it.unifi.swam.assignment_restful_architecture.DAOs
    > it.unifi.swam.assignment_restful_architecture.Model
    > it.unifi.swam.assignment_restful_architecture.Model.Company
    > it.unifi.swam.assignment_restful_architecture.Model.Game
    > it.unifi.swam.assignment_restful_architecture.Model.Player
    > it.unifi.swam.assignment_restful_architecture.Rest
    > it.unifi.swam.assignment_restful_architecture.Rest.Services
    > it.unifi.swam.assignment_restful_architecture.Utils
> src/test/resources/META-INF
> src/test/resources
> src
> src/main/resources/META-INF
v src/test/java
    > it.unifi.swam.assignment_restful_architecture.controller_test
    > it.unifi.swam.assignment_restful_architecture.dao_test
    > it.unifi.swam.assignment_restful_architecture.model_test
    > it.unifi.swam.assignment_restful_architecture.rest_test
> target/generated-sources/annotations
> src/main/resources
> JRE System Library [JavaSE-1.8]
> Maven Dependencies
> Referenced Libraries
> target/generated-test-sources/test-annotations
v Deployed Resources
    > META-INF
    v WEB-INF
        lib
        > beans.xml
        > web.xml
      index.html
> datasource
> target
  pom.xml
  README.md
```

Project organization in Eclipse

- src/main/java: contains entities of Domain Model, DAOs for each entity, Controllers, Rest Services, as well as Utility Classes.
- src/main/resources/META-INF: contains the persistence.xml file to the MySql database.
- src/test/java: contains the unit tests for the entities, DAOs, Controllers and Endpoints.
- src/test/resources/META-INF: contains the persistence.xml file a simple database in memory to execute some of the tests.
- WEB-INF: contains the beans.xml and web.xml files. The web.xml defines the Roles existing in the application, which are then used to defined who is authorized to do certain operations.

From stateless to Restful architecture



General structure

This is the general structure of the solution to be implemented.

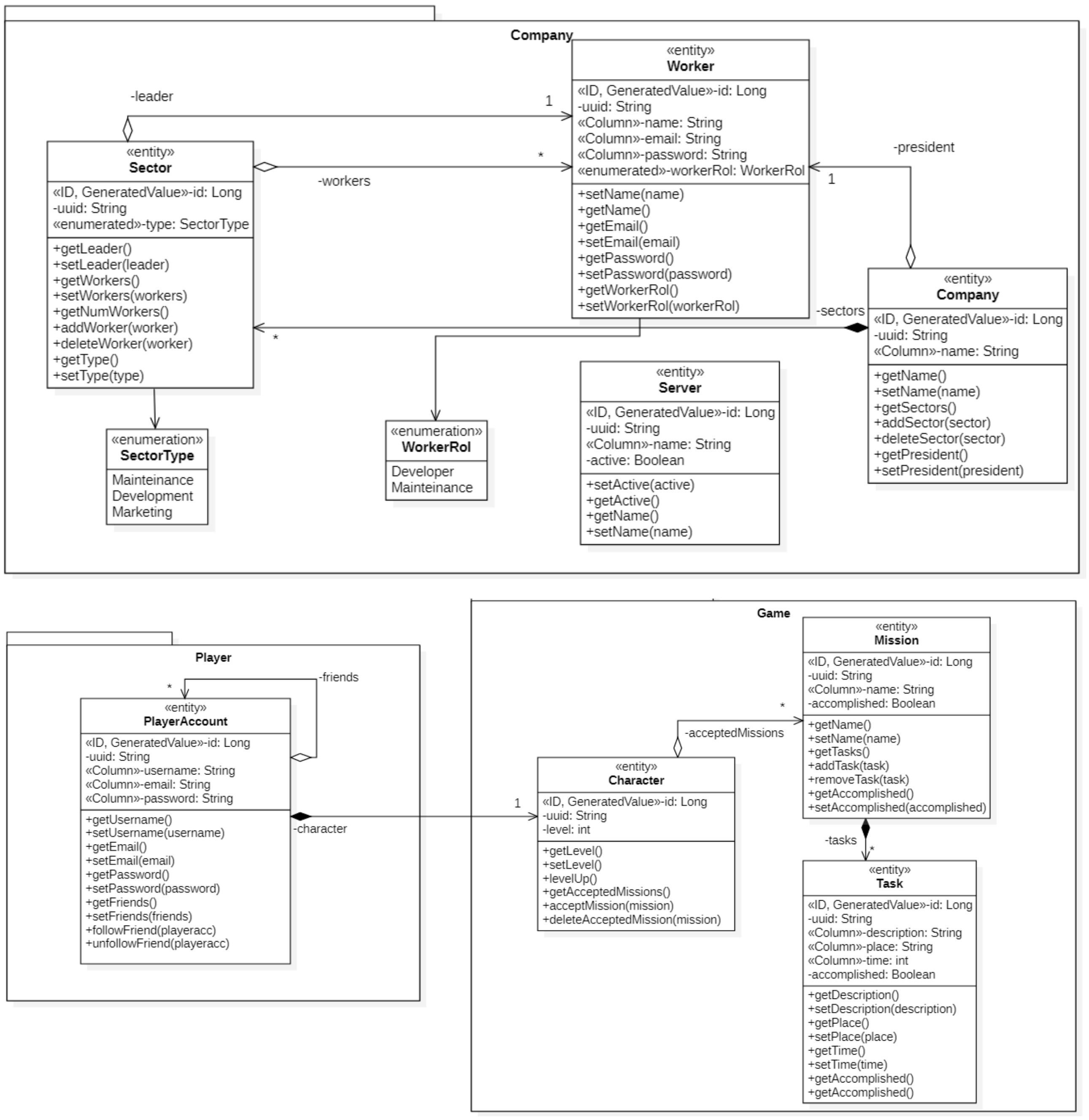
Starting from the Domain Model designed and implemented for the SWE exam, several components have been added to upgrade the solution as the back end of a Restful architecture.

From the old Domain Model everything has been taken as it was, that is the classes forming it and the different packages in which they were contained (Company, Game and Player). Some details have been added to some of the classes in order to enrich the idea a little bit. For example, in the class Worker a Rol has been added as a type Enum. This Rol wasn't part of the original idea, instead the class Worker could have other classes inheriting from it. This change was implemented to make easier the way in which the Rest endpoints check if the user trying to do an action is authorized to do it or not. This idea will be shown later (in the endpoints part). Also in the Worker class, new attributes have been added such as the email and password. These new attributes also help to have a way in which workers can authorize themselves (and again, this will be used with the endpoints later). The players also have these attributes as a way to identify themselves.

In some classes, for instance in the PlayerAccount class, some methods that were part of the original idea have been changed or removed because, in this implementation, these methods are developed as part of other components like the controllers.

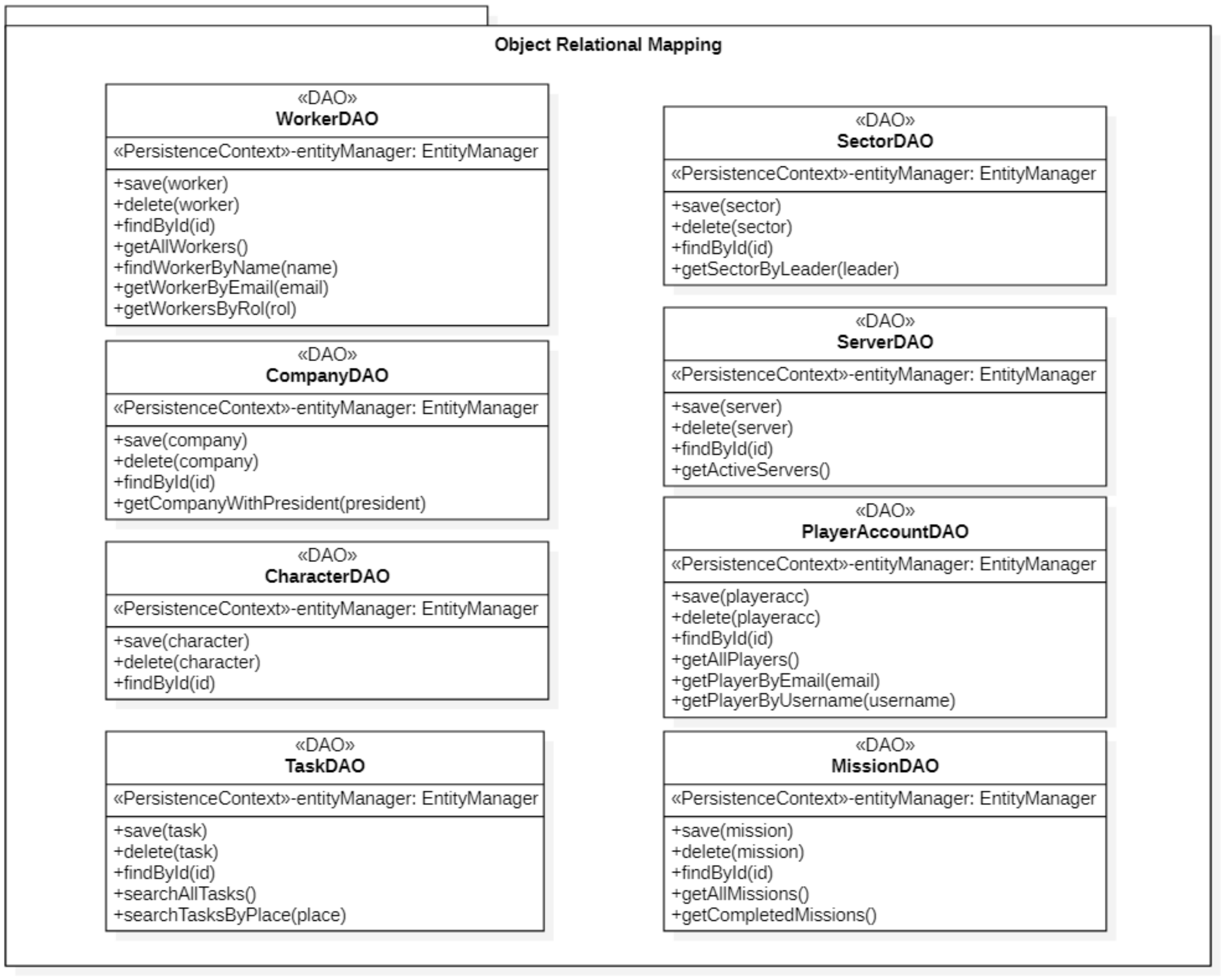
Apart from these slight changes, two new classes have been created, the BaseEntity Class and the ModelFactory Class. More details will be given next.

After this, JPA annotations were added to these Domain Model classes. These annotations define how the classes are modeled into the database, which ones are independent entities with their own identity, which ones are dependent on others, which attribute serves as the Primary Key of the entity, how are associations mapped, etc.



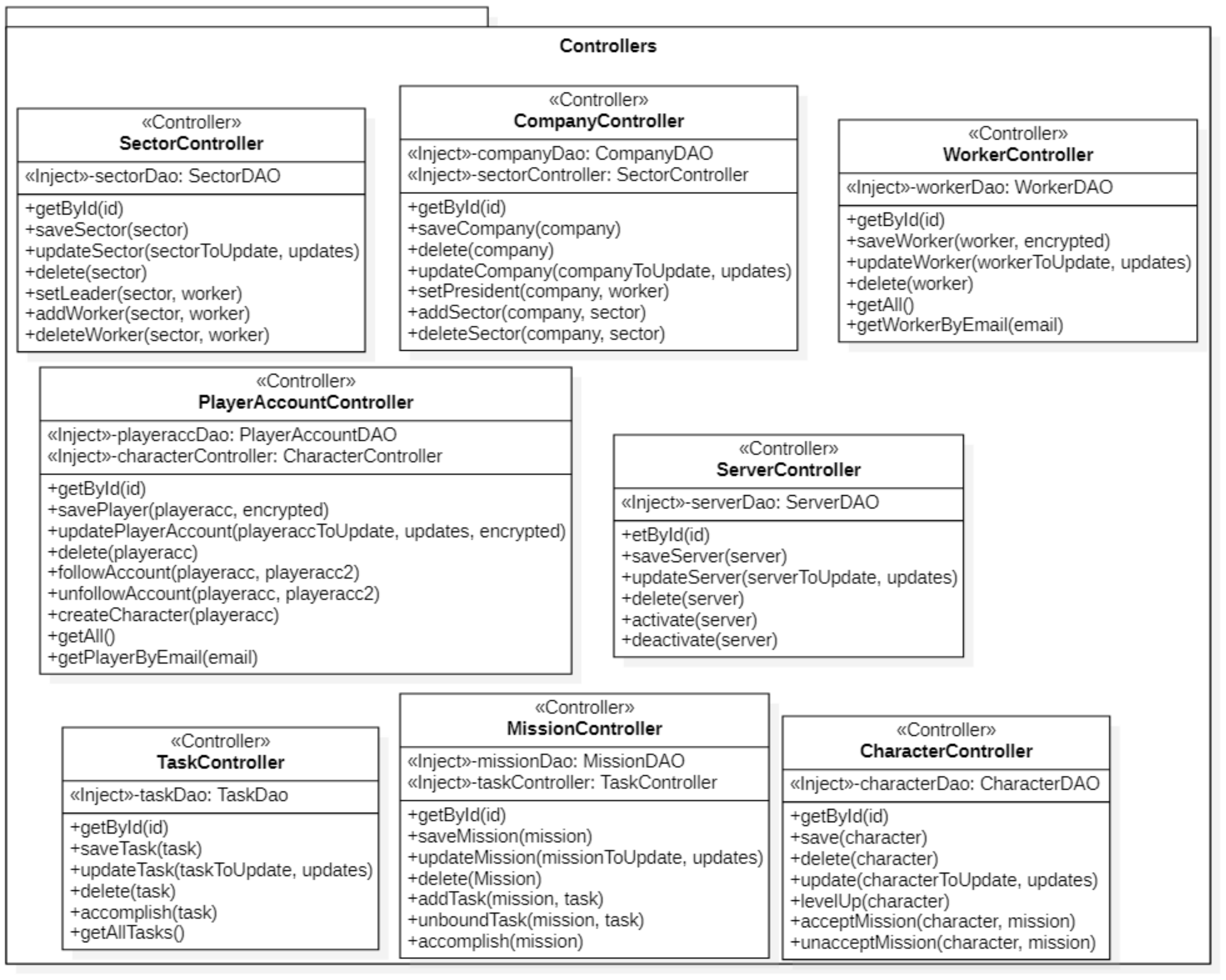
Domain Model

Once the classes in the Domain Model have been implemented and tested (with the testing tools that will be described later), we can proceed with the DAOs. The DAOs make use of an injected EntityManager (implemented with the @PersistenceContext annotation) to perform different operations. These operations are persisting a new object in the database, updating it, removing it from the database, or find it in the database, whether using the Primary Key or a query that can be defined using JPQL (Java Persistence Query Language). The DAOs also have CDI annotations declaring their scope (for example @SessionScoped), because the Controllers will make use of them by injection with CDI.



DAOs

After being tested, the Controllers were developed. The controllers have CDI annotations (mostly the @Inject annotation) to make use of the different DAOs previously implemented. These controllers create new instances of the classes in the Domain Model and then perform different operations with them. For example, once a new instance has been created, the controller uses the corresponding DAO to save it in the database. Some of these controllers have also special utilities. For instance, if the method unacceptMission is called in the CharacterController, it checks first if the mission is part of the accepted mission of the character we are working with. Similar to this is the case of the PlayerAccountController, that checks if the Player is following someone before unfollowing them.

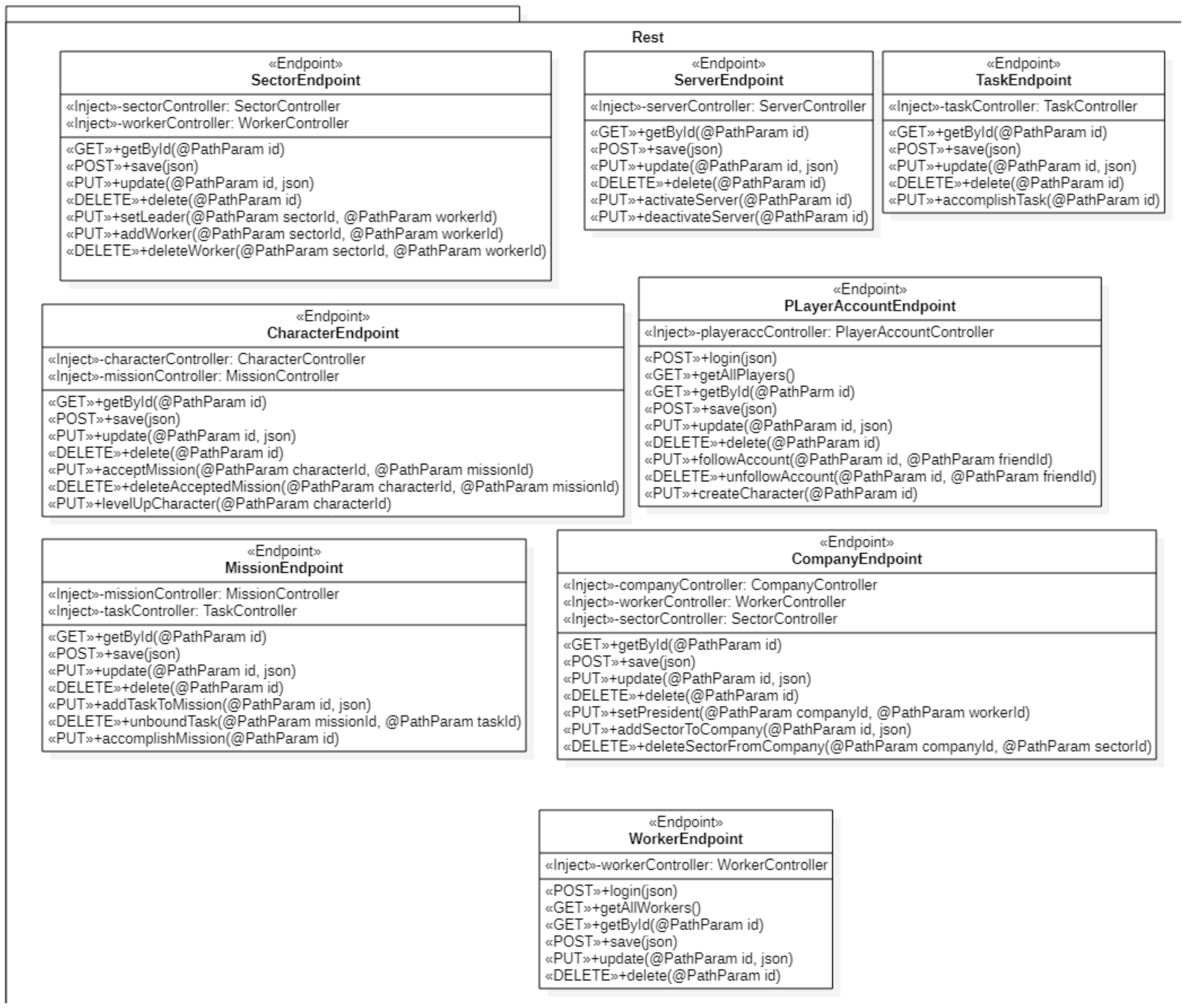

Controllers
«Controller» SectorController
«Inject»-sectorDao: SectorDAO
+getById(id)
+saveSector(sector)
+updateSector(sectorToUpdate, updates)
+delete(sector)
+setLeader(sector, worker)
+addWorker(sector, worker)
+deleteWorker(sector, worker)
«Controller» CompanyController
«Inject»-companyDao: CompanyDAO
«Inject»-sectorController: SectorController
+getById(id)
+saveCompany(company)
+delete(company)
+updateCompany(companyToUpdate, updates)
+setPresident(company, worker)
+addSector(company, sector)
+deleteSector(company, sector)
«Controller» WorkerController
«Inject»-workerDao: WorkerDAO
+getById(id)
+saveWorker(worker, encrypted)
+updateWorker(workerToUpdate, updates)
+delete(worker)
+getAll()
+getWorkerByEmail(email)
«Controller» PlayerAccountController
«Inject»-playeraccDao: PlayerAccountDAO
«Inject»-characterController: CharacterController
+getById(id)
+savePlayer(playeracc, encrypted)
+updatePlayerAccount(playeraccToUpdate, updates, encrypted)
+delete(playeracc)
+followAccount(playeracc, playeracc2)
+unfollowAccount(playeracc, playeracc2)
+createCharacter(playeracc)
+getAll()
+getPlayerByEmail(email)
«Controller» ServerController
«Inject»-serverDao: ServerDAO
+etById(id)
+saveServer(server)
+updateServer(serverToUpdate, updates)
+delete(server)
+activate(server)
+deactivate(server)
«Controller» TaskController
«Inject»-taskDao: TaskDao
+getById(id)
+saveTask(task)
+updateTask(taskToUpdate, updates)
+delete(task)
+accomplish(task)
+getAllTasks()
«Controller» MissionController
«Inject»-missionDao: MissionDAO
«Inject»-taskController: TaskController
+getById(id)
+saveMission(mission)
+updateMission(missionToUpdate, updates)
+delete(Mission)
+addTask(mission, task)
+unboundTask(mission, task)
+accomplish(mission)
«Controller» CharacterController
«Inject»-characterDao: CharacterDAO
+getById(id)
+save(character)
+delete(character)
+update(characterToUpdate, updates)
+levelUp(character)
+acceptMission(character, mission)
+unacceptMission(character, mission)

Controllers

Finally, we have the Rest services. In this case we have some utility classes that are used by the endpoints. These classes are in the package 'Rest'. For example, the class JWTService defines some methods of creation and check of JWTs (Java Web Tokens). This will be used to check if the user is authenticated and authorized to do something or not. The idea is that every request requires being authenticated, and for this, the Worker and PlayerAccount endpoints contains a login method which returns a token. After the login, each request should contain the token as an Authorization header of type Bearer, and the program will check if this token is valid, and the user is authorized. This depends on the Roles allowed for each method, which are define as an annotation in the endpoints (@RolesAllowed). To do this we have the classes SecurityRequestFilter (checks the header of the request for the Authorization token) and SecurityContextPlayer/SecurityContextWorker. These last classes check the role of the user (player or worker) and see if it is correct in order to proceed with the action.

The class MyRestApplication defines the path of the application with the annotation @ApplicationPath.

Lastly, the endpoints contain the JAX-RS annotations. The most important ones are @Path, which defines the path to access the specific endpoint, the annotations @GET, @POST, @PUT, @DELETE that determine the type of functionality, as well as the @RolesAllowed annotation previously mentioned.


Rest
«Endpoint»
SectorEndpoint
«Inject»-sectorController: SectorController
«Inject»-workerController: WorkerController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+setLeader(@PathParam sectorId, @PathParam workerId)
«PUT»+addWorker(@PathParam sectorId, @PathParam workerId)
«DELETE»+deleteWorker(@PathParam sectorId, @PathParam workerId)
«Endpoint»
ServerEndpoint
«Inject»-serverController: ServerController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+activateServer(@PathParam id)
«PUT»+deactivateServer(@PathParam id)
«Endpoint»
TaskEndpoint
«Inject»-taskController: TaskController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+accomplishTask(@PathParam id)
«Endpoint»
CharacterEndpoint
«Inject»-characterController: CharacterController
«Inject»-missionController: MissionController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+acceptMission(@PathParam characterId, @PathParam missionId)
«DELETE»+deleteAcceptedMission(@PathParam characterId, @PathParam missionId)
«PUT»+levelUpCharacter(@PathParam characterId)
«Endpoint»
PLayerAccountEndpoint
«Inject»-playeraccController: PlayerAccountController
«POST»+login(json)
«GET»+getAllPlayers()
«GET»+getById(@PathParm id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+followAccount(@PathParam id, @PathParam friendId)
«DELETE»+unfollowAccount(@PathParam id, @PathParam friendId)
«PUT»+createCharacter(@PathParam id)
«Endpoint»
MissionEndpoint
«Inject»-missionController: MissionController
«Inject»-taskController: TaskController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+addTaskToMission(@PathParam id, json)
«DELETE»+unboundTask(@PathParam missionId, @PathParam taskId)
«PUT»+accomplishMission(@PathParam id)
«Endpoint»
CompanyEndpoint
«Inject»-companyController: CompanyController
«Inject»-workerController: WorkerController
«Inject»-sectorController: SectorController
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)
«PUT»+setPresident(@PathParam companyId, @PathParam workerId)
«PUT»+addSectorToCompany(@PathParam id, json)
«DELETE»+deleteSectorFromCompany(@PathParam companyId, @PathParam sectorId)
«Endpoint»
WorkerEndpoint
«Inject»-workerController: WorkerController
«POST»+login(json)
«GET»+getAllWorkers()
«GET»+getById(@PathParam id)
«POST»+save(json)
«PUT»+update(@PathParam id, json)
«DELETE»+delete(@PathParam id)

Endpoints

We can proceed to see everything in more detail:

In the package Model we can find the classes ModelFactory and BaseEntity.

Base Entity is an abstract class which only defines the ID and the UUID. All entities will extend this class. The ID will be the Primary Key in the tables (to check identity in the database context), whereas the UUID will be the identifier to check if two objects are the same (in the java context).

ModelFactory creates new instances of the entities in the DomainModel and generates random UUIDs for them.

```
@MappedSuperclass
public abstract class BaseEntity {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String uuid;
```

```
	protected BaseEntity() {}
	public BaseEntity(String uuid) {
		if(uuid == null) {
			throw new IllegalArgumentException("uuid cannot be null");
		}
		this.uuid = uuid;
	}

	@Override
	public boolean equals(Object obj) {
		if(this==obj) {
			return true;
		}
		if(obj==null) {
			return false;
		}
		if(!(obj instanceof BaseEntity)) {
			return false;
		}
		return uuid.equals(((BaseEntity) obj).getUuid());
	}

	public String getUuid() {
		return this.uuid;
	}

	public Long getId() {
		return this.id;
	}
```

BaseEntity abstract class

```
public class ModelFactory {
	private ModelFactory() {}

	public static Company company() {
		return new Company(UUID.randomUUID().toString(), null, null);
	}

	public static Worker worker() {
		return new Worker(UUID.randomUUID().toString(), "", "", "");
	}

	public static Worker worker(Worker worker) {
		return new Worker(worker);
	}

	public static Sector sector() {
		return new Sector(UUID.randomUUID().toString(), null);
	}
```

ModelFactory class

In the rest of the packages (Model.*) we can find the entities with the JPA annotations:

Examples:

```
@Entity
public class Worker extends BaseEntity {
		@Column(nullable = false)
		private String name;

		@Column(nullable = false, unique = true)
		private String email;

		@Column(nullable = false)
		private String password;

		@Enumerated(EnumType.STRING)
		private WorkerRol workerRol;


		Worker() {}
		public Worker(String uuid, String name, String email, String password) {
				super(uuid);
				this.name = name;
				this.email = email;
				this.password = password;
				this.setWorkerRol(null);
		}
```

Example Worker class

```
@Entity
public class Mission extends BaseEntity{
		@Column(nullable = false, unique = true)
		private String name;

		@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
		@JoinTable(name="mission_tasks",joinColumns=@JoinColumn(name="missionId"),inverse
JoinColumns=@JoinColumn(name = "taskId", foreignKey = @ForeignKey(
				name = "FK_TASKID",
				foreignKeyDefinition = "FOREIGN KEY (taskId) REFERENCES task(id) ON DELETE
CASCADE" )))
		@OrderBy
		private List<Task> tasks;

		private Boolean accomplished;

		Mission() {}

		public Mission(String uuid, String name, List<Task> tasks) {
				super(uuid);
				this.name = name;
				this.tasks = tasks;
				this.setAccomplished(false);
		}
```

Example Mission Class

Different types of annotations have been used such as: @Column to define characteristics of that column for the table (like not nullable or unique) or annotations for associations such as @OneToMany, @ManyToMany, etc.

Entities Worker and PlayerAccount make use of the library BScrypt to encrypts the passwords and/or verify them.

DAOs

It is defined a DAO for each entity in the Domain Model.
These DAOs have an EntityManager injected through the annotation @PersistenceContext which they use to perform several operations like persist, find, delete, or even different queries.

Example:

```
@SessionScoped
@Default
public class WorkerDAO implements Serializable {
	static final long serialVersionUID = 1L;

	@PersistenceContext
	private EntityManager entityManager;

	public void save(Worker worker) {
		if(worker.getId() != null) {
			entityManager.merge(worker);
		}else {
			entityManager.persist(worker);
		}
	}

	public void delete(Worker worker) {
		this.entityManager.remove(
				this.entityManager.contains(worker) ? worker :
				this.entityManager.merge(worker));
	}

	public Worker findById(Long id) {
		return entityManager.find(Worker.class, id);
	}

	public List<Worker> getAllWorkers() {
		return this.entityManager.createQuery(
				"FROM Worker", Worker.class)
				.getResultList();
	}

	public List<Worker> findWorkerByName(String name) {
		List<Worker> result = this.entityManager.createQuery(
				"FROM Worker WHERE name = :name", Worker.class)
		.setParameter("name", name)
		.getResultList();
```

```
				if(result.isEmpty()) {
					return null;
				}else {
					return result;
				}
		}
```

Example WorkerDAO Class

Controllers

Controllers can create new instances of entities of the DomainModel and also make use of the previously defined DAOs to perform operations with these instances.

DAOs are injected into the Controllers with the CDI annotation @Inject.

Example:

```
@SessionScoped
@Named
public class WorkerController implements Serializable{
		private static final long serialVersionUID = 1L;

		@Inject
		WorkerDAO workerDao;

		public Worker getById(Long id) {
				if(id==null) {
					throw new IllegalArgumentException("Id cannot be null");
				}else {
					return workerDao.findById(id);
				}
		}

		public Worker saveWorker(Worker worker, Boolean encrypted) {
				Worker workerToPersist = ModelFactory.worker();

				workerToPersist.setName(worker.getName());
				workerToPersist.setEmail(worker.getEmail());
				workerToPersist.setPassword(worker.getPassword(), encrypted);

				workerToPersist.setWorkerRol(worker.getWorkerRol());

				workerDao.save(workerToPersist);
				return workerToPersist;
		}

		public void updateWorker(Worker workerToUpdate, Worker updates, Boolean encrypted) {
				if(updates.getName()!=workerToUpdate.getName() &&
updates.getName()!=null) {
					workerToUpdate.setName(updates.getName());
				}
				if(updates.getEmail()!=workerToUpdate.getEmail() && updates.getEmail()!=null){
					workerToUpdate.setEmail(updates.getEmail());
				}
```

```
			if(updates.getPassword()!=workerToUpdate.getPassword() &&
updates.getPassword()!=null) {
				workerToUpdate.setPassword(updates.getPassword(), encrypted);
			}
			if(updates.getWorkerRol()!=workerToUpdate.getWorkerRol() &&
updates.getWorkerRol()!=null) {
				workerToUpdate.setWorkerRol(updates.getWorkerRol());
			}

			workerDao.save(workerToUpdate);
	}
```

Example WorkerController Class

Endpoints

In the package Rest we can find some useful classes that will be used later for the Endpoints:

- MyRestApplication: defines the path of the endpoints.
- ResponseFilter: allows CORS among other uses.
- JWTService: defines method to create JWT Tokens or verify a token that was passed as a parameter***.***
- SecurityRequestFilter: makes sure that Authorization is performed for every request. When performing login or singup of a worker or player we will receive a JWT Token which we will have to pass as Authorization Header for every request. In this case the Authorization is of type Bearer.
- SecurityContextPlayer/SecurityContextWorker: after the Token is passed and verified in the SecurityRequestFilter, a Player or Worker (depending on what the user of the Token is) is set. With this class and the annotation @RolesAllowed in the Endpoints, the methods will only be performed if the role of the user identified in the Token is allowed for that method.
  Players have role "User" whereas Workers can have different roles like "Development", "Maintenance" or "All" if no role has been set when creating that worker.
  The method "isUserInRole" is the one used automatically along with the @RolesAllowed annotation.

Finally, in the package Rest.Services we can find the different endpoints for each entity. These endpoints use annotations like @Path or @PathParam to specify the path to access that endpoint and/or take the param in the path to use it in the method. The basic methods are @GET, @POST, @PUT, @DELETE.

Also, annotations @Consumes and @Produces defines what content is expected at the request or at the response.

The endpoints for Worker and PlayerAccount have login and signup methods, which returns the Token that will be used in subsequent requests.

Login methods used BScrypt library to check if the encrypted password coincides with the one introduced by the user.

Examples:

```
@Path("/workerendpoint")
public class WorkerEndpoint {

		@Inject
		WorkerController workerController;

		@GET
		@Path("/ping")
		public Response ping() {
				return Response.ok().entity("WorkerEndpoint is ready").build();
		}

		@GET
		@Produces({MediaType.APPLICATION_JSON})
		@RolesAllowed({"All","Developer","Mainteinance"})
		public Response getAllWorkers() {
				ObjectMapper mapper = new ObjectMapper();
				try {
						List<Worker> workers = workerController.getAll();
						String json = mapper.writeValueAsString(workers);
						return Response.ok(json, MediaType.APPLICATION_JSON).build();

				} catch (JsonProcessingException e) {
						e.printStackTrace();
						return Response.serverError().entity("An unexpected error
ocurred").build();
				}
		}
```

Example WorkerEndpoint

Jackson Framework has been used to parse a Json as an instance of entities or to parse instances/text as Json.

Testing

To perform the unit tests, some test suites containing test cases have been defined with the use of JUnit annotations like @Test. Other useful annotations worth noticing are @BeforeClass which executes a method before all the test cases in the test suite, @Before which is similar to BeforeClass but instead of executing the method before all the tests only once, it executes the method before each test case, and the equivalents @AfterClass and @After. These methods are the setup and teardown methods performed before and after the tests.

Model Test

The objective of these tests is to check that initialization and identity/equality comparisons between Objects work correctly.

To do this, first we define a simple class called “FakeBaseEntity” that extends BaseEntity. This is useful to avoid the problem of BaseEntity being an abstract class.

```
package it.unifi.swam.assignment_restful_architecture.model_test;

import it.unifi.swam.assignment_restful_architecture.Model.BaseEntity;

//solves the problem of having BaseEntity abstract
public class FakeBaseEntity extends BaseEntity {
		public FakeBaseEntity (String uuid) {
				super(uuid);
		}
}
```

FakeBaseEntity

Now we can define the test suite BaseEntityTest.

```
//We test correct initialization as well as identity and equality between Objects
public class BaseEntityTest {
		private FakeBaseEntity entity1;
		private FakeBaseEntity entity2;
		private FakeBaseEntity entity3;


		//Before = BeforeEach of junit5 (in junit4)
		@Before
		public void setup() {
				System.out.println("Perform the setup...");
				String uuid1 = UUID.randomUUID().toString();
				String uuid2 = UUID.randomUUID().toString();

				entity1 = new FakeBaseEntity(uuid1);
				entity2 = new FakeBaseEntity(uuid2);
				entity3 = new FakeBaseEntity(uuid1);
		}

		//Expected exception when trying to create an object without uuid
		//In juni5 exists Assertions.assertThrows(...)
		@Test(expected = IllegalArgumentException.class)
		public void testNullUUID() {
				System.out.println("Perform testNullUUID");
				new FakeBaseEntity(null);
		}

		@Test
		public void testEquals() {
				System.out.println("Perform testEquals");
				assertEquals(entity1, entity1); //Check Identity
				assertEquals(entity1, entity3); //Check Equality
				assertNotEquals(entity1, entity2); //Check Not Equality
		}
```

BaseEntityTest

DAO Test

These tests make use of a simple database in memory defined in the persistence.xml of test/resources/META-INF.

It consists of an abstract class called JPATest that contains the methods of setup and teardown. These methods create an EntityManager manually through an EntityManagerFactory, which will be used later for the tests to perform operations with this database. Furthermore, it contains the method init(), which will be defined specifically for each concrete DAOTest.

```
//includes the four annotations of junit (@Before, @BeforeClass, @After, @AfterClass)
//initializes the EntityManager and factory
public abstract class JPATest {
		private static EntityManagerFactory entityFactory;
		protected EntityManager entityManager;

		//creates EntityManagerFactory
		//once for every Test Suite (costful operation)
		@BeforeClass
		public static void setupEM() {
				System.out.println("Creates EntityManagerFactory");
				//not real DB system, "in memory" to give the DAO something to work with
				entityFactory = Persistence.createEntityManagerFactory("test");
		}

		//initializes EntityManager and calls init() method
		//Performed before each single TestCase
		@Before
		public void setup() throws IllegalAccessException {
				System.out.println("Creates EntityManager");
				entityManager = entityFactory.createEntityManager();
				entityManager.getTransaction().begin(); //starts cleaning transaction

				//cleans DB keeping the tables
				String sql = "SET FOREIGN_KEY_CHECKS = 0";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE server";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE sector;";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE worker;";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE playeraccount;";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE playercharacter;";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE company;";
				entityManager.createNativeQuery(sql).executeUpdate();

				sql = "TRUNCATE TABLE mission;";
```

```
                    entityManager.createNativeQuery(sql).executeUpdate();

                    sql = "TRUNCATE TABLE task;";
                    entityManager.createNativeQuery(sql).executeUpdate();

                    sql = "SET FOREIGN_KEY_CHECKS = 1";
                    entityManager.createNativeQuery(sql).executeUpdate();


                    entityManager.getTransaction().commit(); //closes cleaning transaction
                    entityManager.getTransaction().begin(); //starts transaction for custom init
                    System.out.println("Calls method init");
                    //this method is abstract and it is specialized in each concrete class
                    init();
                    entityManager.getTransaction().commit();
                    entityManager.clear();
                    entityManager.getTransaction().begin();
                    System.out.println("Setup completed");
          }
          //Closes the transaction of the EntityManager
          //Performed after each single TestCase
          @After
          public void close() {
                    if(entityManager.getTransaction().isActive()) {
                              entityManager.getTransaction().rollback();
                              //If there is active transaction, perform a rollback
                    }
                    System.out.println("Closes EntityManager");
                    entityManager.close();
          }

          //Closes the EntityManagerFactory
          //Performed after every TestSuite
          @AfterClass
          public static void tearDownDB() {
                    System.out.println("Closes EntityManagerFactory");
                    entityFactory.close();
          }

          //Abstract method that will be defined for each ConcreteClass extending JPATest
          protected abstract void init()
                    throws IllegalAccessException;

}
```

JPATest

Once we have defined this abstract class, we can make each concrete class to test the different DAOs. For example:

```
public class WorkerDAOTest extends JPATest {
          private Worker worker;
          private WorkerDAO workerDao;

          //concrete init() method
          @Override
          protected void init() throws IllegalAccessException {
                    System.out.println("Start init custom for WorkerDAOTest");
```

```
		worker = ModelFactory.worker();
		worker.setName("testWorker1");
		worker.setEmail("testWorker1@gmail.com");
		worker.setPassword("testWorker1",false);
		entityManager.persist(worker); //Persisted manually, without using the DAO
		//this is done to test the retrieve afterwards
		workerDao = new WorkerDAO();
		FieldUtils.writeField(workerDao,"entityManager",entityManager,true);
	}

	//retieve test
	//check that the entity retrieved is the same as the one defined in the init()
	@Test
	public void testFindById() {
		System.out.println("Perform testFindById in WorkerDAOTest");
		Worker result = workerDao.findById(worker.getId()); //It has an ID since it has 
been persisted in the init()
		assertEquals(worker.getId(), result.getId());
		assertEquals(worker.getName(), result.getName());
	}

	//In this case we check persisting the data through the DAO
	@Test
	public void testSave() {
		System.out.println("Perform testSave in WorkerDAOTest");
		Worker workerToPersist = ModelFactory.worker();
		workerToPersist.setName("testWorker2");
		workerToPersist.setEmail("testWorker2@gmail.com");
		workerToPersist.setPassword("testWorker2",false);
		workerDao.save(workerToPersist);
		Worker manuallyRetrievedWorker =  entityManager.
				createQuery("FROM Worker WHERE uuid = :uuid", 
Worker.class)
				.setParameter("uuid", workerToPersist.getUuid())
				.getSingleResult();
		assertEquals(workerToPersist, manuallyRetrievedWorker);
	}

	@Test
	public void testDelete() {
		System.out.println("Perform testDelete in WorkerDAOTest");
		Worker workerToDelete = ModelFactory.worker();
		workerToDelete.setName("testWorker3");
		workerToDelete.setEmail("testWorker3@gmail.com");
		workerToDelete.setPassword("testWorker3",false);
		entityManager.persist(workerToDelete);
		workerDao.delete(workerToDelete);
		List<Worker> manuallyRetrievedWorker =  entityManager.
				createQuery("FROM Worker WHERE uuid = :uuid", 
Worker.class)
				.setParameter("uuid", workerToDelete.getUuid())
				.getResultList();
		assertTrue(manuallyRetrievedWorker.isEmpty());
	}
	@Test
	public void testFindWorkerByName() {
		System.out.println("Perform testFindWorkerByName in WorkerDAOTest");
		Worker worker1 = ModelFactory.worker();
		worker1.setName("Victor");
		worker1.setEmail("testemail@gmail.com");
		worker1.setPassword("testworker1",false);
		Worker worker2 = ModelFactory.worker();
		worker2.setName("Victor");
		worker2.setEmail("testemail2@gmail.com");
		worker2.setPassword("testworker2",false);
		entityManager.persist(worker1);
		entityManager.persist(worker2);
		List<Worker> result = workerDao.findWorkerByName("Victor");
		assertEquals(result.size(), 2);
	}
```

WorkerDAOTest

These classes test the methods defined in each of the different DAOs, using the assert methods that JUnit provides. Test Cases are independent between each other (they don't depend on the result of a previous Test Case).

Controller Test

These classes test the different Controllers. To do this, apart from JUnit methods and annotations, it is also used the Mockito Framework to "mock" the behavior of the DAOs inside the Controllers.

For Example:

```
public class PlayerAccountControllerTest {
	private PlayerAccountController playerController;
	private PlayerAccountDAO playerDao;
	private PlayerAccount fakePlayer;

	@Before
	public void setup() throws IllegalAccessException {
		playerController = new PlayerAccountController();

		playerDao = mock(PlayerAccountDAO.class); //Mock instance (defines behaviour manually)

		fakePlayer =  ModelFactory.playeracc();
		fakePlayer.setUsername("Fake Player Username");
		fakePlayer.setEmail("fakeplayeremail@gmail.com");
		fakePlayer.setPassword("Fake Player Password", false);

		FieldUtils.writeField(playerController, "playeraccDao", playerDao, true); //Mock instance injected in the controller manually
	}
	@Test
	public void testGetPlayer() {
		when(playerDao.findById(1L)).thenReturn(fakePlayer);

		PlayerAccount retrievedPlayer = playerController.getById(1L);
		assertEquals(retrievedPlayer.getUsername(), fakePlayer.getUsername());
		assertEquals(retrievedPlayer.getEmail(), fakePlayer.getEmail());
		assertEquals(retrievedPlayer.getPassword(), fakePlayer.getPassword());
		assertEquals(retrievedPlayer, fakePlayer);
	}

	@Test
	public void testUnfollowFriend() {
		PlayerAccount fakePlayer2 =  ModelFactory.playeracc();
		fakePlayer2.setUsername("Fake Player Username2");
		fakePlayer2.setEmail("fakeplayeremail2@gmail.com");
		fakePlayer2.setPassword("Fake Player Password2", false);

		playerController.followAccount(fakePlayer, fakePlayer2);
		assertEquals(fakePlayer.getFriends().size(), 1);
		assertEquals(fakePlayer.getFriends().get(0), fakePlayer2);

		playerController.unfollowAccount(fakePlayer, fakePlayer2);
		assertEquals(fakePlayer.getFriends().size(), 0);
```

PlayerAccountControllerTest

Endpoint Test

These classes test the Rest Endpoints. It uses JUnit methods as well as the REST Assured library.

Example:

```
public class ServerEndpointTest {
	private final static String baseURL = "assignment-restful-architecture/rest/";
	private static String token;
	private static Worker worker;

	@BeforeClass
	public static void setup() throws IllegalAccessException {
		RestAssured.baseURI = "http://localhost/";
		RestAssured.port = 8080;
		Populate.truncate();
		Populate.populate();

		worker = ModelFactory.worker();
		worker.setEmail("testdeveloper@gmail.com");
		worker.setPassword("testdeveloper", true);
		Response response = given()
					.contentType("application/json")
					.body(worker)
					.when().post(baseURL + "workerendpoint/login");
		token = response.getBody().asString(); //token of an existing worker in the
database with role Developer
	}
```

```
		@AfterClass
		public static void teardown() {
				Populate.truncate();
		}

		@Test
		public void getServerByIdTest() {
				Response response  = given().header("Authorization", "Bearer "+
token).pathParam("id","1").get(baseURL + "serverendpoint/" + "{id}");
				response.then().statusCode(200)
				.body("name",org.hamcrest.Matchers.equalTo("test server"));
		}

		@Test
		public void addServerTest() {
				Server server =  ModelFactory.server();
				server.setName("test server 2");

				Response response = given()
				.header("Authorization", "Bearer "+ token)
				.contentType("application/json")
				.body(server)
				.when().post(baseURL + "serverendpoint/");

				response.then().statusCode(201);
		}
```

ServerEndpointTest

The setup method uses truncate() and populate() from the Populate class in Utils to introduce some initial data in the database. Then a login is performed with the worker defined in the populate method. This is done because Endpoints are protected, so we need to perform the login to obtain the JWT Token to be able to make use of the endpoint operations.

In the test cases REST Assured is used to perform the operations defined in the endpoints, passing in the header the token received after logging.

Furthermore, the endpoints have not been tested only in this way, by also making using of a REST client (in this case Advanced REST Client).

References

[1] Java Authentication with JSON Web Tokens (JWTs) - https://www.baeldung.com/java-json-web-tokens-jjwt

[2] Jason Web Tokens (JWT) - https://jwt.io/

[3] Java: Create a Secure Password Hash - https://howtodoinjava.com/java/java-security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/

[4] BCrypt Docs - https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCrypt.html