Task manager application

Software Design

1st assignment

Soos David

3rd group

2024

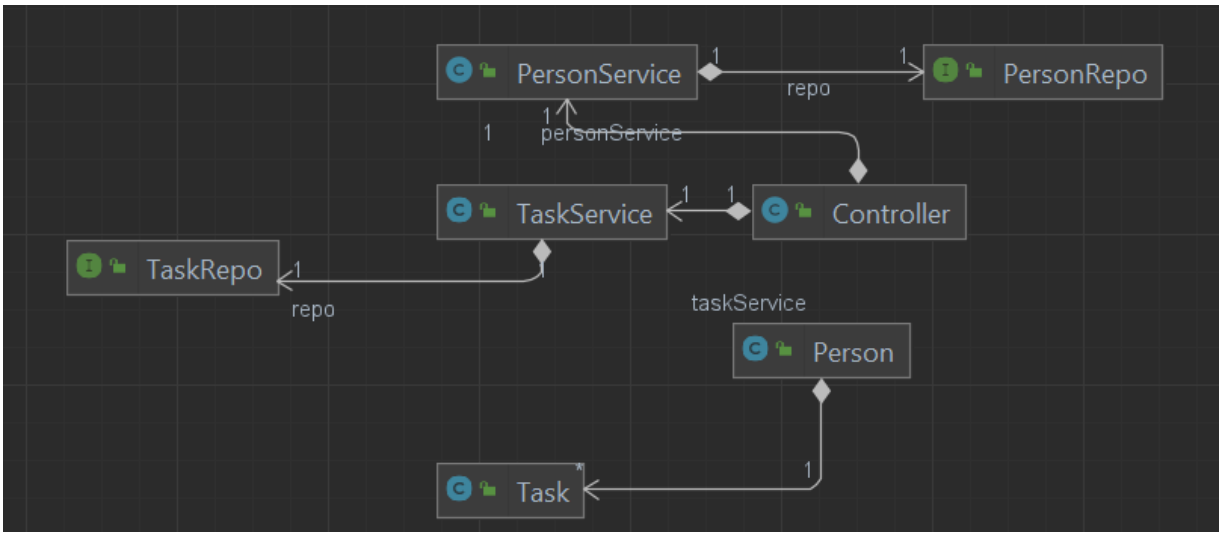
Table of Contents

1. Project Proposal

My aim with this project is to create an application that successfully manages tasks that are assigned to people working in an office setting. At the first stage, I would like to deliver a working backend that is set up from scratch.

The technology that will be used for the implementation is Java Spring , a Postgres database will be employed and the testing will be done with the help of Postman.

2. Analysis and Design



1. Figure, UML class diagram

The internal architecture should be composed of three levels: the Model level, the Repository level, and the Service level. For each of these there will be two separate entities as there are two tables to be modeled. With the help of the Repositories, we can establish a connection between the database and our code, enabling us to perform CRUD operations. The Services store functions that handle the mix of data and the interaction between the Controller, Models and Repositories. The Models embody the object representation of our database entities and are the basic building blocks of our application.

The Controller is the brain behind the operation. With the help of annotations, it is able to establish the direction of the API call and should orient the towards the proper flow of operations.

The Person model should have a name, email, password and a tasks element, while the Task object contains only a brief description of its purpose.

3. Implementation

The first problem I tackled was the incrementality of the id fields. I used a Long variable and the annotations: *@Id @GeneratedValue(strategy = GenerationType.IDENTITY)*. Secondly I made the fields name and email unique with an annotation with the same name. The tasks of a person are stored in a list and are annotated with: *@OneToMany(cascade = CascadeType.ALL) @JoinColumn(name = "fk_person_id", referencedColumnName = "id")*. This enables us to create the one to many relationship with the help of a primary and a foreign key.

The biggest problem, that maybe wasn`t so straightforward initially, was the creation of the update operation. By only using a put operation, we lose the fields that are not mentioned, but with the help of reflection and the patch operation, we are able to truly implement an update.

```
public Person updatePersonBetter(Long id, Map<String, Object> fields) {
    Optional<Person> currentPerson = repo.findById(id);

    if (currentPerson.isPresent()) {
        fields.forEach((key, value) -> {
            Field field = ReflectionUtils.findField(Person.class, key);
            field.setAccessible(true);
            ReflectionUtils.setField(field, currentPerson.get(), value);
        });

        return repo.save(currentPerson.get());
    }
    return null;
}
```

2. Figure, Reflection used in PATCH

In the Controller class the data flow and the upcoming steps are established. To be able to distinguish between operations, annotations are employed. These make sure the type of operation, the proper variable and the task at hand are given and this way all the information can be handled.

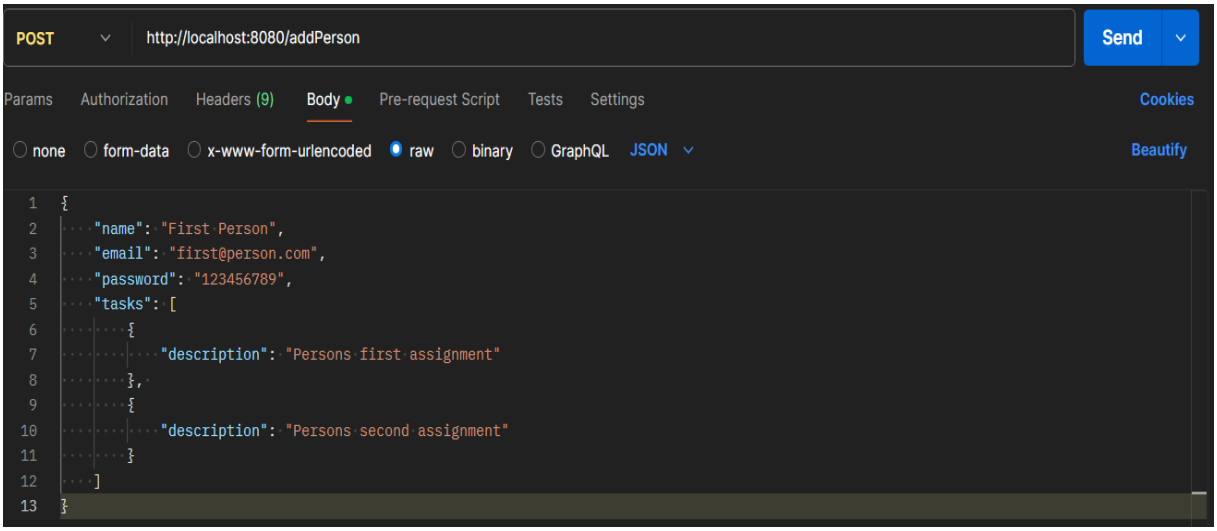
```
@PutMapping("/updatePerson/{id}")
public Person updatePerson(@PathVariable Long id, @RequestBody Person person) {
    return personService.updatePerson(id, person);
}
```

3. Figure, Controller class

4. Testing

The testing was done through Postman. When the application is run, if there are no tables, then brand-new ones are created.

First, we will add a couple of people with a POST call.



4. Figure, POST

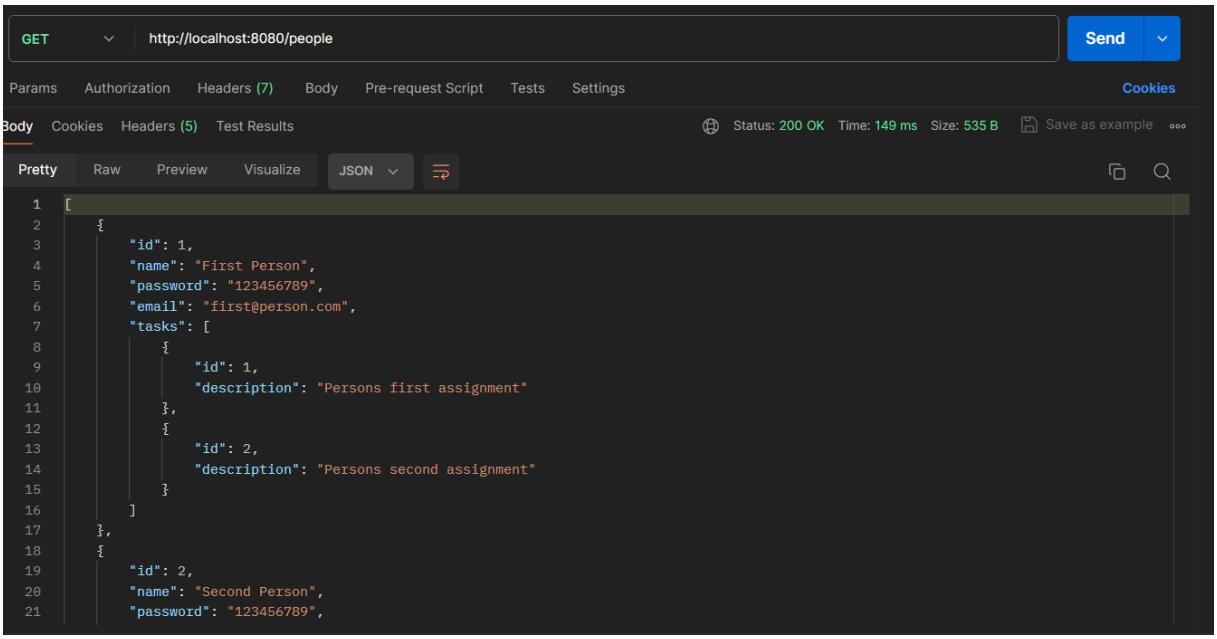
At this phase our database looks like this:

	id [PK] bigint	email character varying (255)	name character varying (255)	password character varying (255)
1	1	first@person.com	First Person	123456789
2	2	second@person.com	Second Person	123456789

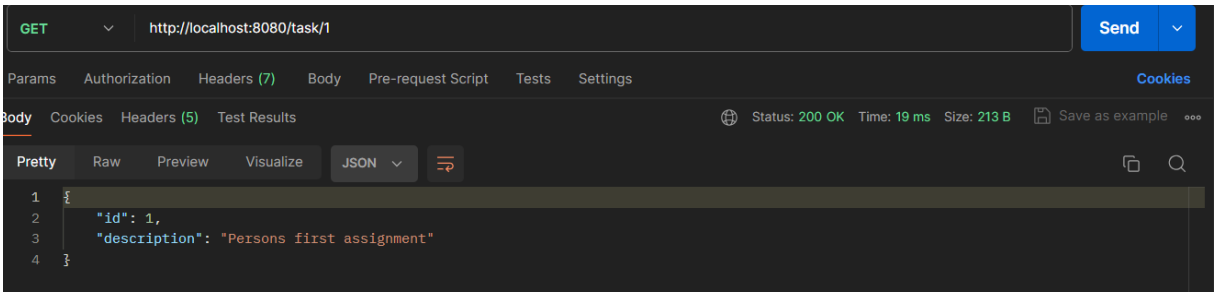
	id [PK] bigint	description character varying (255)	fk_person_id bigint
1	1	Persons first assignment	1
2	2	Persons second assignment	1
3	3	THird assignment	2
4	4	Fourth assignment	2

5. Figure, Initial database

We can view the all, or filter by with the help of GET calls:

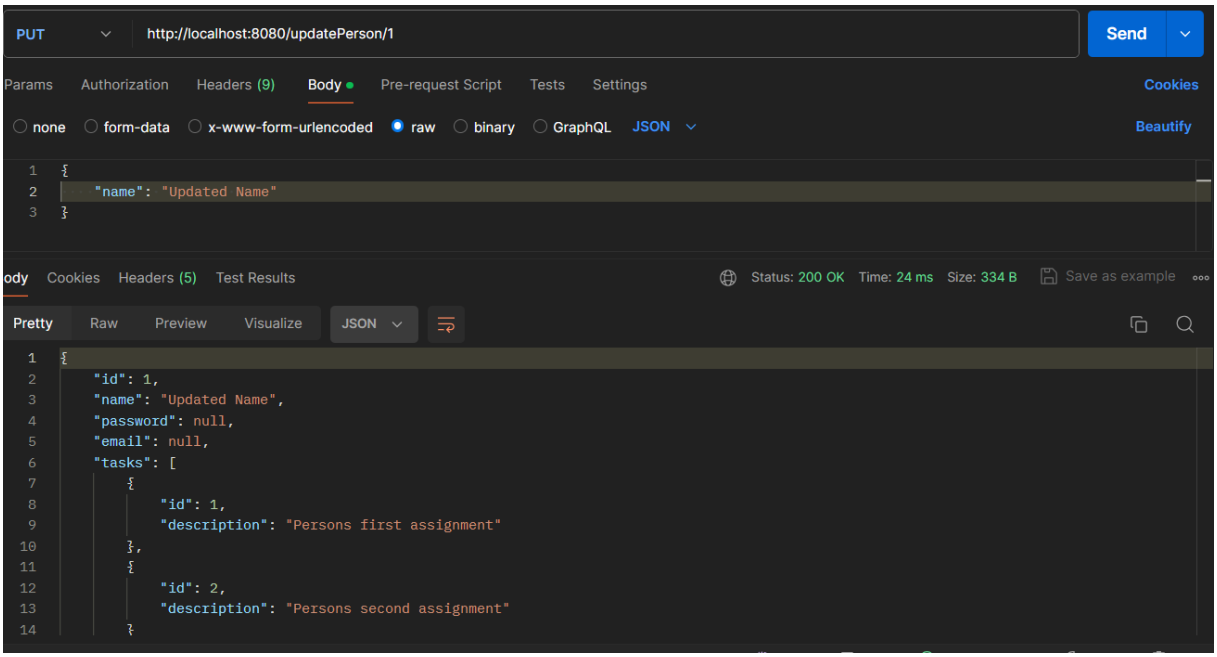


6. Figure, GET all people

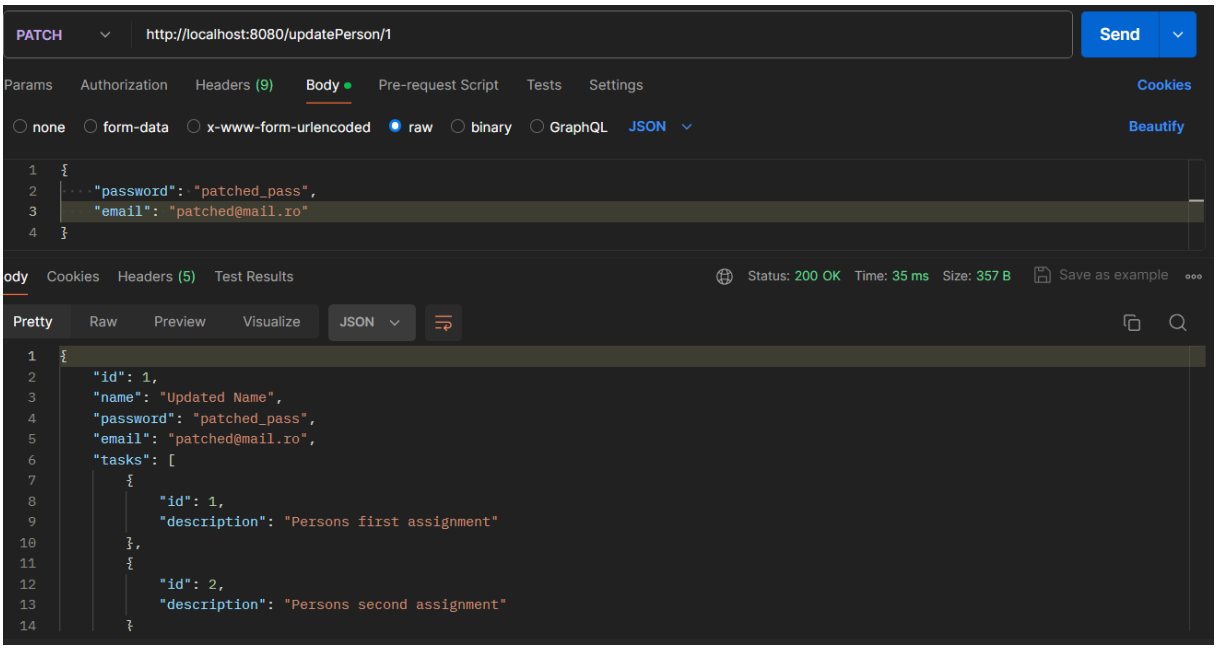


7. Figure, GET the first task

For the update we can use either PUT or PATCH, depending on the use case.

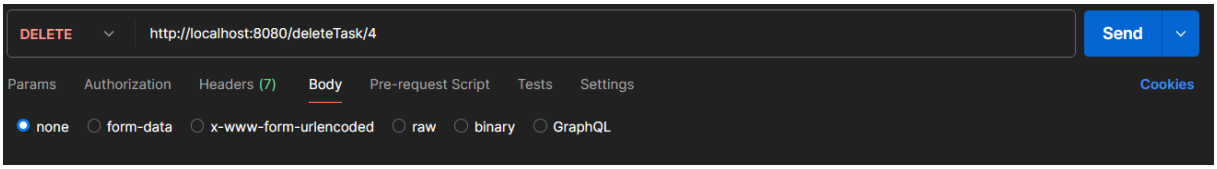


8. Figure, PUT operation on Person



9. Figure, PATCH operation on Person

Lastly, we can also DELETE.



10. Figure, DELETE operation on Task

This way our database gains this form:

	id [PK] bigint	email character varying (255)	name character varying (255)	password character varying (255)
1	2	second@person.com	Second Person	123456789
2	1	patched@mail.ro	Updated Name	patched_pass

	id [PK] bigint	description character varying (255)	fk_person_id bigint
1	1	Persons first assignment	1
2	2	Persons second assignment	1
3	3	THird assignment	2

5. Conclusion

I found it fun to explore this side od Java, as I have never used it for a project up until now. The hardest part was gaining enough information, but when all was clear, the implementation became straightforward. All in all I enjoyed this assignment.