Technical Documentation

10 Jan 2022

Audience

Deployment engineers
Future software engineers
Architects

Authors

Arash Nasrat
Jechillo Huang
Hashim Mohammad
Mohamad Hassan
Nazlıcan Eren
Sarah Chrzanowska-Buth

Version

1.0

Table of Contents

1. Introduction

This document is meant for deployment engineers, future software engineers or architects who will be given the task to extend our product with additional functionality or migrate it to another IT technology. Prior experience or knowledge about software engineering is needed for readers to understand this document. Also will be explained how the product is built.

The purpose of this technical documentation is to describe how our project was built. We will do this by clarifying the functional and technical specifications of our project. There is also a software development guide to help the audience understand our product and to make it easy for them to use it for future development.

This document will cover the product vision with its most important features (2); the most important epic stories that have been realised in the product (3); a Navigable Class Diagram of the functional model (4); a Deployment Diagram (5); and our design choices (6).

2. Product vision

This product was made for the Climate Cleanup organisation. This organisation utilises nature-based solutions to remove CO_2 with the help of a specific type of plant located in a greenhouse. This product should help the specialists involved in their project to monitor and regulate the greenhouses via sensors. In the end, they should be able to determine what the perfect conditions for this plant are, to remove the most CO_2 from the atmosphere.

climate cleanup

This product is a web application designed for teams of specialists in the Climate Cleanup project. The most important features of the product are: controlling and monitoring greenhouses using the data from the sensors, graphs with relevant data based on the specialists' needs and the ability to share notes with each other. Other less important features are little things like profile pictures.

Each team is assigned a greenhouse and consists of five different specialties. The specialties are Geology, Agronomy, Hydrology, Botany and Climate Science. This product is capable of creating different teams and adding specialists to those teams. These actions can be performed by either an admin or a super admin.

A specialist can easily access the greenhouse data and change the value of the sensors. The result of this change is stored and the history can be viewed for later evaluation. The specialists can also share notes to keep each other up to date with their observations and the changes they have made. This product is designed from the requirements of the Climate Cleanup organisation, but other companies with similar needs could make use of this software as well.

In contrast to other dashboard applications, our web application has a design that is mobile-friendly. The design is user-friendly and one of the features that contributes to this is the app’s responsiveness. For example, the sensor data can be easily viewed on any device, without elements of the page breaking.

The product is built with the Angular framework and the Spring Boot framework. It is easy for other Software Engineers to extend the product in the future. The product has a lot of potential to grow further. User preferences could be added, such as colourblind mode to help colourblind specialists pick a colour for the greenhouse light. Another way to expand the app would be to add multiple languages. The most interesting future growth of the product is to make a mobile app version for the App Store/Play Store. This way the product could easily be accessed by any user.

3. Epic stories

A summary of the most important epic stories that have been realised in our product:

- Login
- User profile
- Notes page
- Teams page
- Sensors
- Dashboard
- Settings

List of the most important acceptance criteria per story:

Login:

- ➔ Given I'm a specialist, I should be able to log in to my environment with a username and password created by the admin of the website.
- ➔ Given I'm not a registered user of the website, I shouldn't be able to access the pages of the website.

User profile:

- ➔ Given I'm a logged-in user, I should be able to see my profile picture in the navigation bar. If I click on the profile picture I should see what my name, specialty and role in this project are.

Note page:

- ➔ Given I'm a specialist and I visit the notes page, the initial page that I'll see is the notes page of my specialty.
- ➔ Given I'm on the notes page, I should be able to view notes from all the specialists.
- ➔ Given I'm on my own notes page, I should be able to create a note and to edit or remove my previously created note.

Teams page:

- ➔ Given I'm a member of a team and I visit the teams' page, the initial page should show my team.
- ➔ Given I'm an admin of a team, I should be able to add or remove members from my team.
- ➔ Given I'm a super admin, I should be able to add or remove teams, admins, and members.

Sensors:

- ➔ Given I'm a specialist, I should be able to only change the sensors applicable to my specialty.
- ➔ Given I'm a specialist, I should be able to see a list of the 10 most recent changes made by members to the sensors.
- ➔ Given I'm a specialist, I should be able to see the current and desired values of all sensors.

Dashboard:

- ➔ Given I'm a specialist, I should be able to see the history of the CO_2 level against a controlled element relevant to my specialty. The history should be displayed as a chart or table.
- ➔ Given I'm a member, I should be able to navigate the charts or tables

Settings:

- ➔ Given I'm a member, I should be able to change my profile information, such as my full name and display picture.
- ➔ Given I'm a member, I should be able to change my account information, such as my email and password.

4. Navigable Class Diagram

Below you can find a navigable class diagram of the functional model that has been designed for 'Climate Cleanup'. This diagram only includes the main entities, attributes and some operations.

For the sake of readability, we have excluded in this diagram the constructor, getter and setter methods, and standard class methods unless overridden, such as `toString()` or `valueOf(String)`. Public getter and setter methods are used to access private fields. The arrow tips of the associations indicate 'navigable relations'. Some are bi-directional, others are unidirectional. Additional explanation about the navigable class diagram can be found on the next page.

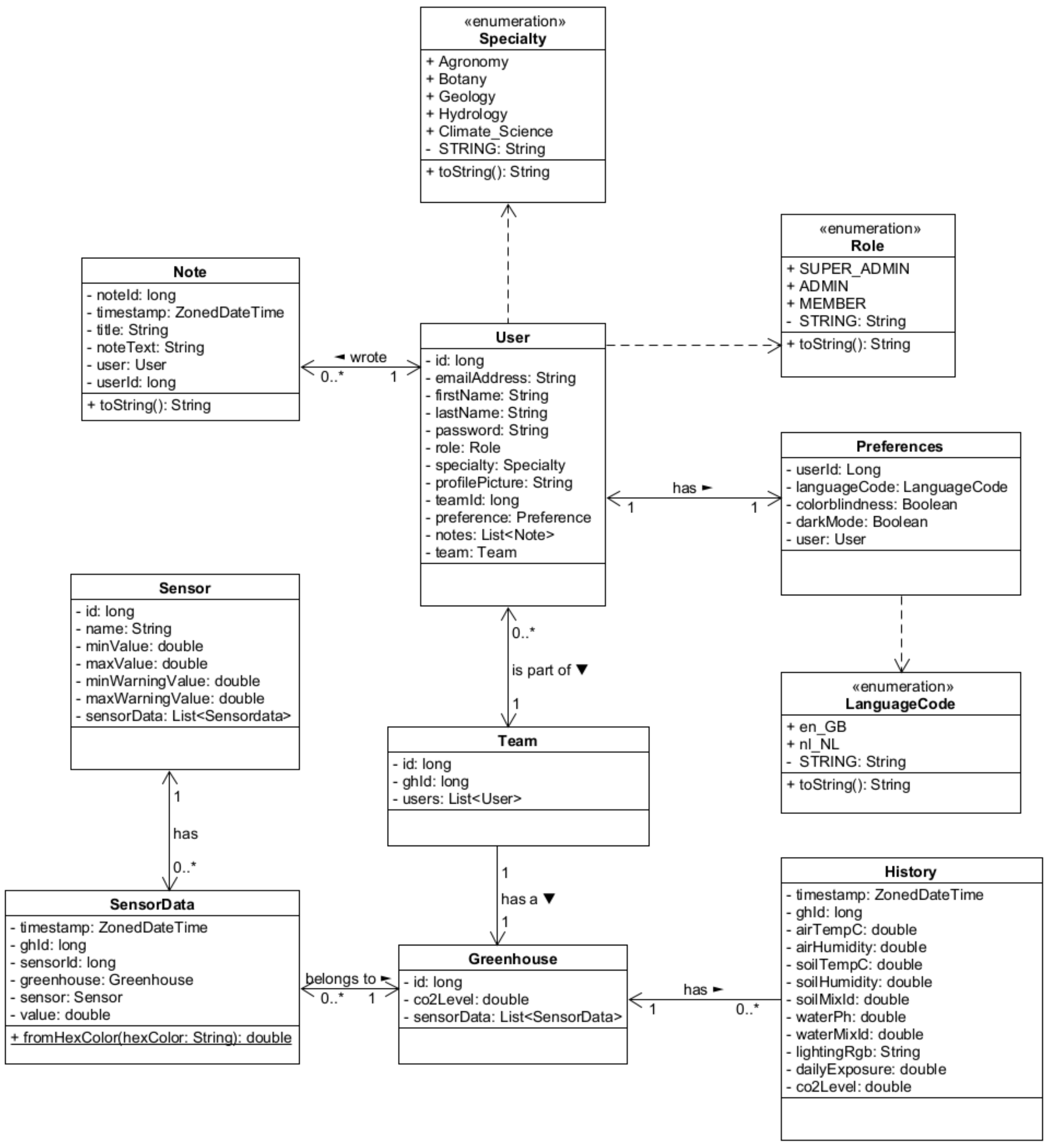

Figure 1. Navigable Class Diagram 'Climate Cleanup'

As can be seen in the diagram a `User` has a few fields. However, when a new `User` is created, only the `id`, `emailAddress`, `firstName`, `lastName`, `password`, `Role` and `Specialty` are recorded.

A `User` has a `Role`, which can be either `SUPER_ADMIN`, `ADMIN`, or `MEMBER`. A `User` with the role `SUPER_ADMIN` can create a `Team` or a `User` and view all teams. A User with the role `ADMIN` can only create a `User` and add it to their own team. And a User with the role `MEMBER` can only view their team. A `SUPER_ADMIN` can do everything that a `MEMBER` also can.

A `User` also has a `Specialty` assigned to them. This can be `Agronomy`, `Botany`, `Geology`, `Hydrology`, or `Climate_Science`.

A `User` has `Preferences` such as language (English or Dutch), colour blindness or dark mode but these functions are in the frontend not fully implemented yet. The `Preferences` is defaulted to `en_GB`, `false`, and `false`.

Notes can be created by users. To create a `Note` you need an `id` of the `Note`, a `timestamp`, a `title`, a `noteText`, and a `userId`. When you create a `Note` and send a `POST` request, you only need the `userId` to store it in the database. But when the `Note` is meant to be viewed in the frontend, an instance of `User` is needed to see the user's information on the `Note`.

As mentioned before, a `User` belongs to a `Team`. A `Team` has an `id`, a `greenhouseId` and can contain zero, one or multiple users. When you create a new `Team` only the `id` and `greenhouseId` are recorded.

A `Greenhouse` has an `id`, `co2Level` and a list of `sensorData`. When you create a new `Greenhouse` only the `id` and `co2Level` are recorded.

Data from the greenhouses is periodically saved as `History`. It saves a `timestamp`, all sensors, the `co2Level`, and the `greenhouseId`.

`SensorData` has a similar function: it saves a `timestamp`, `greenhouseId`, `sensorId`, and `value`. The difference is that `History` saves all sensors in a row while `SensorData` saves the changes per sensor. `SensorData` is used for storing the desired values of a `User`.

Then we also have a `Sensor`. It has an `id`, a `name`, a `minValue`, a `maxValue`, a `minWarningValue`, and a `maxWarningValue`. It is used for generating sensors and as validation for the `SensorData`. When a `User` is changing the sensor values, their desired values may not exceed the min or max values of a `Sensor`.

4.1. Class diagram of the structure and dependencies in the note repository

This section explains the structure and dependencies of one of the repositories in the JPA persistence layer. Below is a diagram that shows the notes repository and its dependencies.

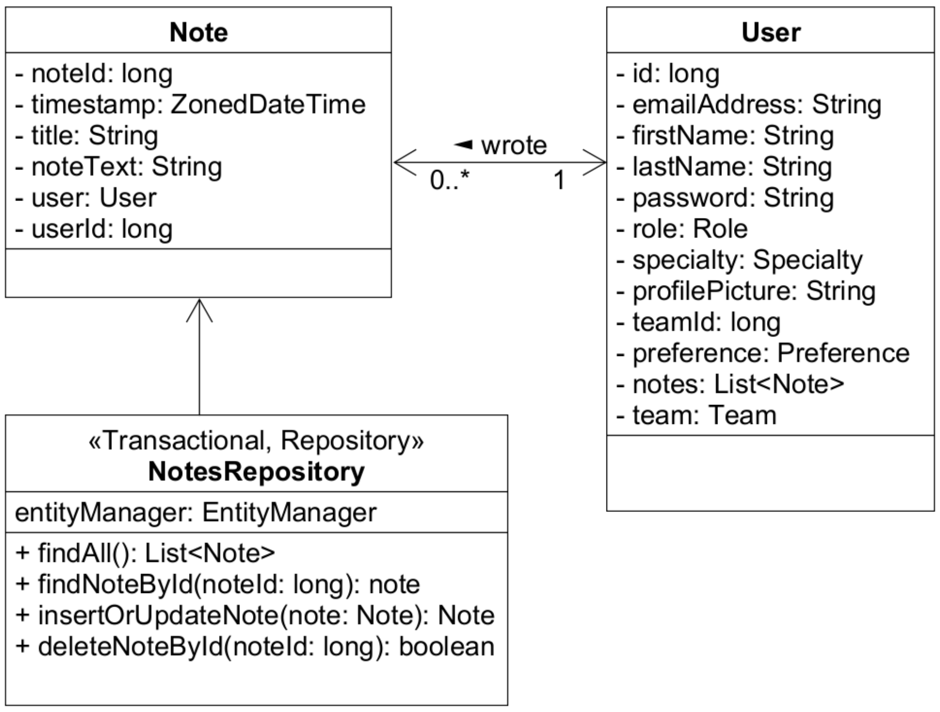

Figure 2. Class diagram of the notes repository

This is the class diagram of the note repository in our JPA persistence layer. The JPA persistence layer is about the storage logic.

`NotesRepository` is annotated by `@Transactional` and `@Repository`. The reason why we annotate with `@Transactional` is that JPA on itself doesn't provide any type of declarative transaction management. By using `@Transactional`, many important aspects such as transaction propagation are handled automatically (Ferreira, 2014), so that you don't have to worry about it.

The reason why we also annotate with `@Repository` is to indicate that the class provides the mechanism for storage, retrieval, search, update, and delete operations on objects. (Kumar, n.d.)

The `EntityManager`, seen in `NotesRepository`, is a part of the Java Persistence API and is annotated with the `@PersistenceContext`. `EntityManager` is an API that manages the lifecycle of entity instances. EntityManager has some very useful methods that are used often such as `remove`, `find`, `merge`, etc. This way you don't have to write native queries to send to the database.

The `NotesRepository` has 4 methods that do the following: `findAll()` gets all the notes out of the database, `findNoteById(noteId)` gets the note by id , `insertOrUpdateNote(note)` adds or update a note in the database and `deleteNoteById(noteId)` removes a note from the database. It does all these actions by using entityManager.

4.2. Class diagram of an external interface

This section explains the usage of an external interface via a class diagram. Below is a diagram that shows how the backend communicates with the email service to send emails.

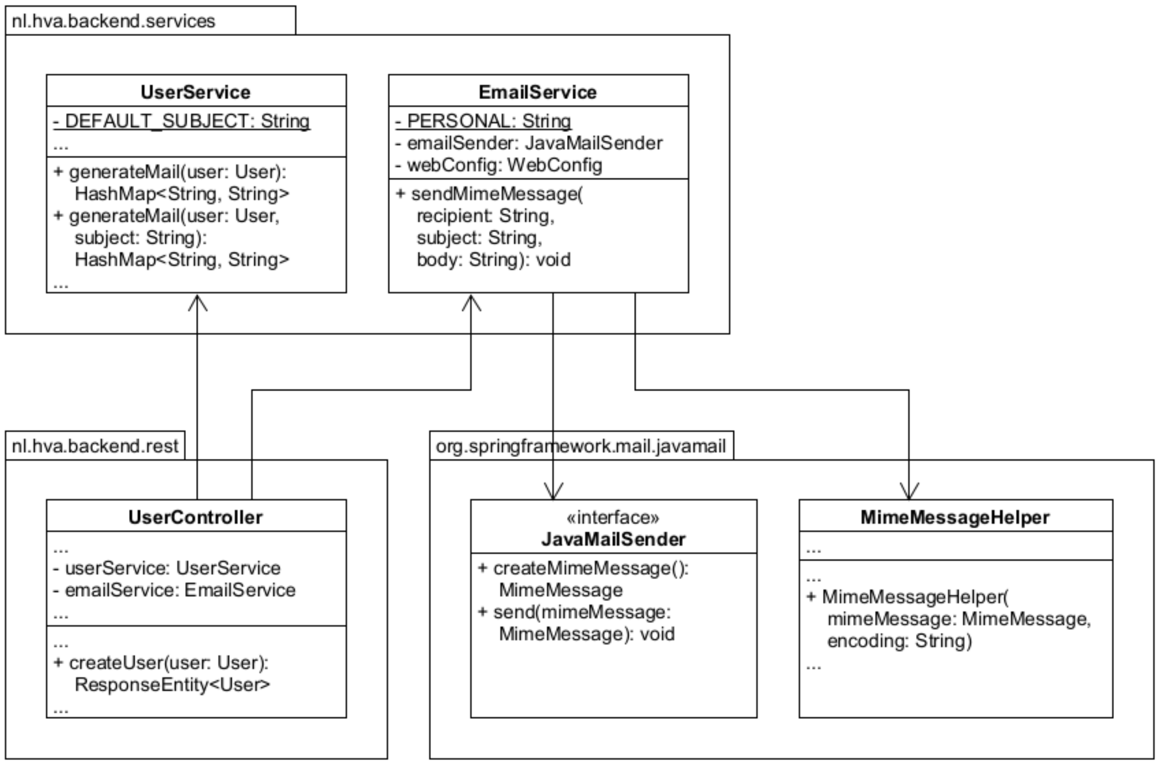
nl.hva.backend.services
UserService
- DEFAULT_SUBJECT: String
...
+ generateMail(user: User): HashMap<String, String>
+ generateMail(user: User, subject: String): HashMap<String, String>
...
EmailService
- PERSONAL: String
- emailSender: JavaMailSender
- webConfig: WebConfig
+ sendMimeMessage(recipient: String, subject: String, body: String): void
nl.hva.backend.rest
UserController
...
- userService: UserService
- emailService: EmailService
...
...
+ createUser(user: User): ResponseEntity<User>
...
org.springframework.mail.javamail
«interface»
JavaMailSender
+ createMimeMessage(): MimeMessage
+ send(mimeMessage: MimeMessage): void
MimeMessageHelper
...
...
+ MimeMessageHelper(mimeMessage: MimeMessage, encoding: String)
...

Figure 3. A diagram that illustrates communication with an external interface

Both the `UserService` and `EmailService` get injected into `UserController` using JPA's `@Autowired` annotation. The `createUser` method will call the `UserService`'s `generateMail` method to create a personalised email. This method returns a `HashMap` containing the recipient's email address, the subject of the mail, and the mail's body. If all went well, the `createUser` method will call the `EmailService`'s `sendMimeMessage` method. This method will be supplied with the `HashMap`'s entries and then send the email. An email is sent using the `JavaMailSender` interface, which gets autowired into the `EmailService`. This interface needs the hostname of the email service provider, the port number, a username and a password. Additional options, i.e. `SMTP` authentication and `STARTTLS` protocol need to be enabled. These values and options can be supplied in either the `pom.xml` file or a config file. Once supplied, the `EmailService` can use the `send` method of `JavaMailSender` to send the mail. The `MimeMessageHelper` class is used to add `PERSONAL` as the display name of the mail.

5. Deployment Diagram

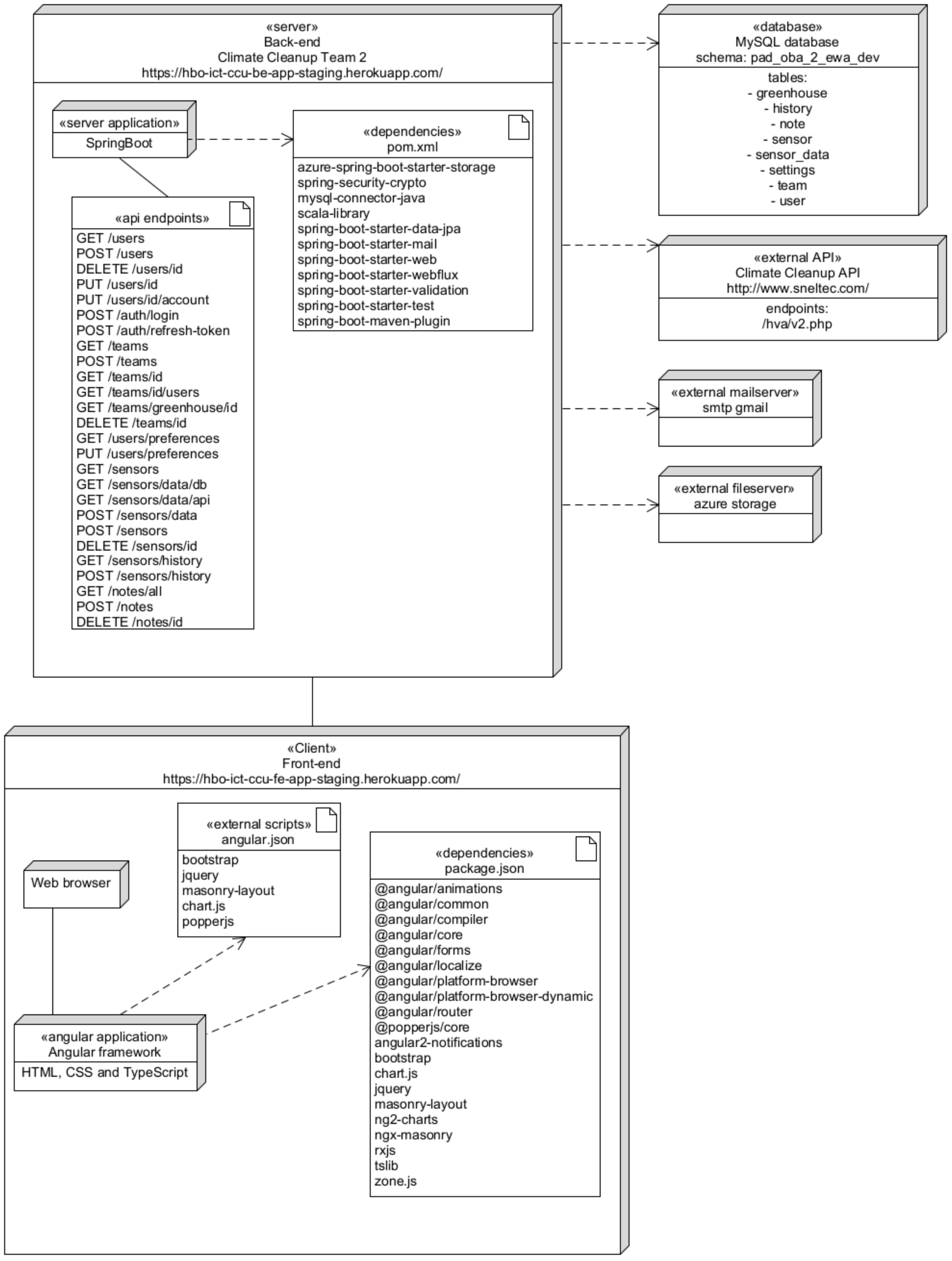

Figure 4. Deployment diagram

On the previous page is the deployment diagram of our application. The backend relies on the Spring Boot framework and therefore it’s an important part of the backend diagram. The backend exposes resources via various API endpoints so that data can be fetched, updated or deleted. This information is eventually sent to the database. For example, if someone wants to create a new note in our application, the note data can be stored in the database by sending a `POST` request to the `/notes` endpoint.

This project makes use of an external API, provided by our client Climate Cleanup. With that endpoint data can be retrieved from the sensors in the greenhouse. The values of the sensors can also be changed and this change can be sent as a `POST` request to that external API.

In the frontend the application runs in the web browser and we used the Angular framework to accomplish that. The frontend also uses some external scripts. With the masonry-layout[1] we could align the notes on our webpage in an optimal position based on available vertical space. We also used Chart.js so that we could easily make graphs that were the best suitable for us.

[1] A JavaScript grid layout library

6. Design

This section elaborates on the challenges and alternative solution options of our design theme. It will discuss themes related to security, performance, storage, the REST API and external interfaces.

As for security measures, we've encrypted the users' passwords using the BCrypt hashing function and chosen to authenticate sessions using JWT. We've chosen to use JWT authentication because we were most familiar with this form of authentication. On top of that, we found it more secure to authenticate users in the backend rather than in the frontend. Every request sent to the backend should have a valid token in the header's `Authorization` field in order to get an authorised response. This token can be acquired when logging in via the `/login/token` endpoint. The generation of a token is based on the user's email and role. It is signed with a secret key using the HS512 algorithm. As for password encryption, we've used the Spring Security Crypto library which provided us with the `BCryptPasswordEncoder`. This encoder enabled us to either encode a password or decode and match passwords.

For our DAO design pattern, we have made use of the Spring Data JPA library. This enabled us to store and retrieve data from the database. We could think of three different methods to implement the DAOs. We had opted to use two of them. The first method of implementing a DAO was by utilising Spring Data's DAO generation. This was achieved by implementing the `JpaRepository` interface into the `UserRepository` DAO. With this approach, we could declare our own relevant query methods or use the standard CRUD methods available from the implemented interface. For the other approach, we implemented DAOs that utilise JPA's `entityManager`. This method allows us to easily define complex or custom queries. For either method, each model is treated as an entity and can access its corresponding table in the database. JPA also provides the `@Transactional` annotation to ensure that queries run properly. If something goes wrong while executing a query, JPA will rollback any changes made in the persistence context.

To communicate with external interfaces, such as file storage, an email service, or an external API, we could either include the logic into the DAO or create a separate utility class. For the latter, we created `Service` classes that contain the business logic for communication with these interfaces. Each class is annotated with JPA's `@Service` annotation.

One such example is `EmailService`. This service allowed us to send emails using the `JavaMailSender` interface. For this, we needed an email service provider. With the limited time in the last sprint, we had limited options available. We thought of starting our own email server but decided to use Gmail instead.

As for storing files, such as display pictures, we've decided to use Azure Storage. We've looked at different ways of storing files such as AWS, Heroku or somewhere in the backend server. We found that Azure was easier for us to implement and within our budget as students compared to AWS. Heroku wasn't ideal either as it would clean its filesystem on each cycle. Also, we found storing files in the backend to be insecure.

Instead of using an Angular dashboard template, we chose to build the frontend from scratch. We did this to avoid redundant files and almost none of the templates we found matched our mobile design. We've used Bootstrap as our CSS framework and included the dependencies ng2-charts and ngx-masonry.

The dependency ng2-charts allowed us to draw various graphs for the sensors. We chose this chart engine because we were most familiar with it. A good alternative would've been ngx-charts as it works well with Angular.

The notes page needed a way to layout the notes on a grid to match our prototype design. We tried to implement our own solution with CSS but that proved unsuccessful. To remedy this, we included the ngx-masonry dependency. This automatically places the notes in the right position on the masonry grid.

6.1. Analytical reflection

The implementation of JWT was rushed as it was implemented in the last week of the 4th sprint. Therefore there is room for improvement in the frontend. For example, before the JWT was added to the authentication service, updating the logged-in user happened mainly in the user service. This should be moved to the authentication service since it's the job of the authentication to update a logged-in user. There is also a small area of improvement in the backend in regard to JWT. Currently, the `/auth/refresh-token` endpoint refreshes a token, without setting a limit on the number of refreshes. There is also the possibility to use JWT in conjunction with Spring Security but that's not necessary.

BCrypt is slow which makes it a good password hashing function. However, it is a dated hashing function. So a better and newer alternative would be Argon2. With the use of the Spring Security Crypto library, this would require an extra dependency, namely the Bouncy Castle package. Also, it should be noted that BCrypt has a maximum password length of 72 bytes. (CheatSheets Series, n.d.)

All of the repositories in the backend adhere to the DAO design pattern, but not all of them are implemented the same way. For example, some use Spring Data's generation of DAO by implementing the `JpaRepository` interface while most are DAOs that utilise the `entityManager` explicitly. This could be enhanced by converting all the repositories to implement the `JpaRepository` interface.

However, another way of improving would be the aforementioned third approach. With this approach, there would be an `EntityRepository` interface and an `AbstractEntityRepository` abstract class. This would decrease the number of artefacts needed to maintain.

Most of the services in the backend do not have an interface or abstract class to implement or extend. This makes it difficult for future engineers to expand or use a different implementation of for example a `FileService`. Currently, the application is only capable of handling requests with Azure's Blob Storage. Services need a corresponding service interface and config file so that the service can be expanded with different implementations.

The project is structured with regard to the MVC design pattern. However, not all files are placed in the right location. Reorganisation of the (sub-)directories and files will improve the quality of the project structure.

Glossary

This section contains an alphabetical list of words or phrases that could be difficult to understand and their definitions.

Term	**Definition**
Angular	A typescript-based open-source web application framework led by the Angular Team at Google. This framework works from the frontend.
Annotation	A special type of interface. It is a form of metadata used on fields, methods, and classes in Java.
API	Application Programming Interface: a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other services.
Autowire	Injecting a Bean into a class’s field, constructor, property, or setter.
Bean	An object instantiated, assembled, and managed by Spring’s dependency injection.
Bi-directional navigability	A bidirectional relationship has both an owning side and an inverse side. So that means that each entity has a relationship field/property that refers to the other entity.
Deployment diagram	A diagram that shows the configuration of run-time processing nodes and the components that live on them. A deployment diagram is a kind of structure diagram used in modelling the physical aspects of an object-oriented system. They are often used to model the static deployment view of a system.
Endpoint	An endpoint is one end of a communication channel. When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or service.

External interface	A shared boundary across which two or more separate external components exchange information.
JPA persistence layer	A software layer that makes it easier for a program to persist its state. The JPA persistence layer consists of a persistence layer with persistent data stored in a relational database.
Scrum	A set of practices used in agile project management that emphasize daily communication and the flexible reassessment of plans that are carried out in short, iterative phases of work
Spring Boot	An open-source framework that is maintained by a company called Pivotal. It provides Java developers with a platform to get started with an auto configurable production-grade Spring application. This framework works from the backend.
Sprint	A repeatable fixed time-box during which a "Done" product of the highest possible value is created.
`@Transactional`	A JPA annotation that’s used on a method or class when you want query methods to be executed in a transaction.
Uni-directional navigability	A unidirectional relationship has only an owning side. So that means that only one entity has a relationship field/property that refers to the other.

References

CheatSheets Series. (n.d.). *Password Storage*. OWASP Cheat Sheet Series. Retrieved January 9, 2022, from https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Ferreira, V. (2014, June 5). *How Does Spring @Transactional Really Work?* DZone. Retrieved January 9, 2022, from https://dzone.com/articles/how-does-spring-transactional

Kumar, P. (n.d.). *Spring @Repository Annotation*. JournalDev. Retrieved January 9, 2022, from https://www.journaldev.com/21460/spring-repository-annotation

Oracle. (n.d.). *Direction in Entity Relationships (The Java EE 6 Tutorial)*. Oracle. Retrieved January 7, 2022, from https://docs.oracle.com/cd/E19798-01/821-1841/bnbqi/index.html

Spring. (n.d.). *16. Transaction Management*. Spring Framework Reference Documentation. Retrieved January 9, 2022, from https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html