withdrive

withdrive

Applied research & software design document

withdrive

Table of Contents

Contents

withdrive

1. Introduction

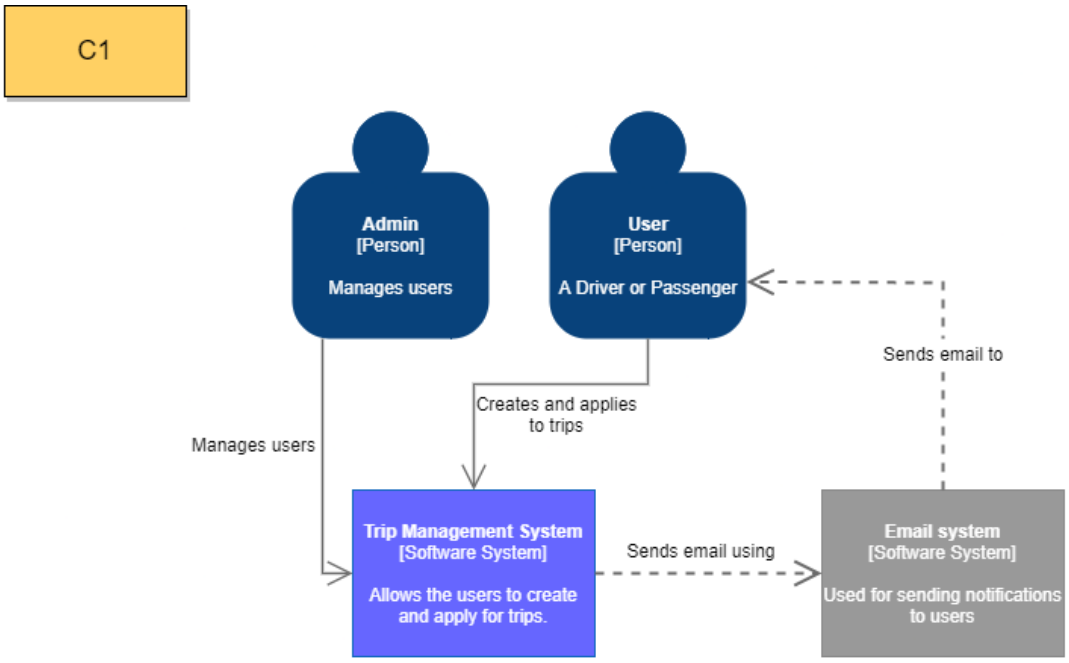
1.1 Document purpose

The purpose of this document is to provide some context and explanations behind certain design and software related choices for this project. I will discuss the reasonings for my stack choice. As well as explain and discuss my architecture and design choices for my application. This will include some researched information that is used to reinforce the reasoning for my choices.

2. System architecture and design

2.1 C4 model

C1:



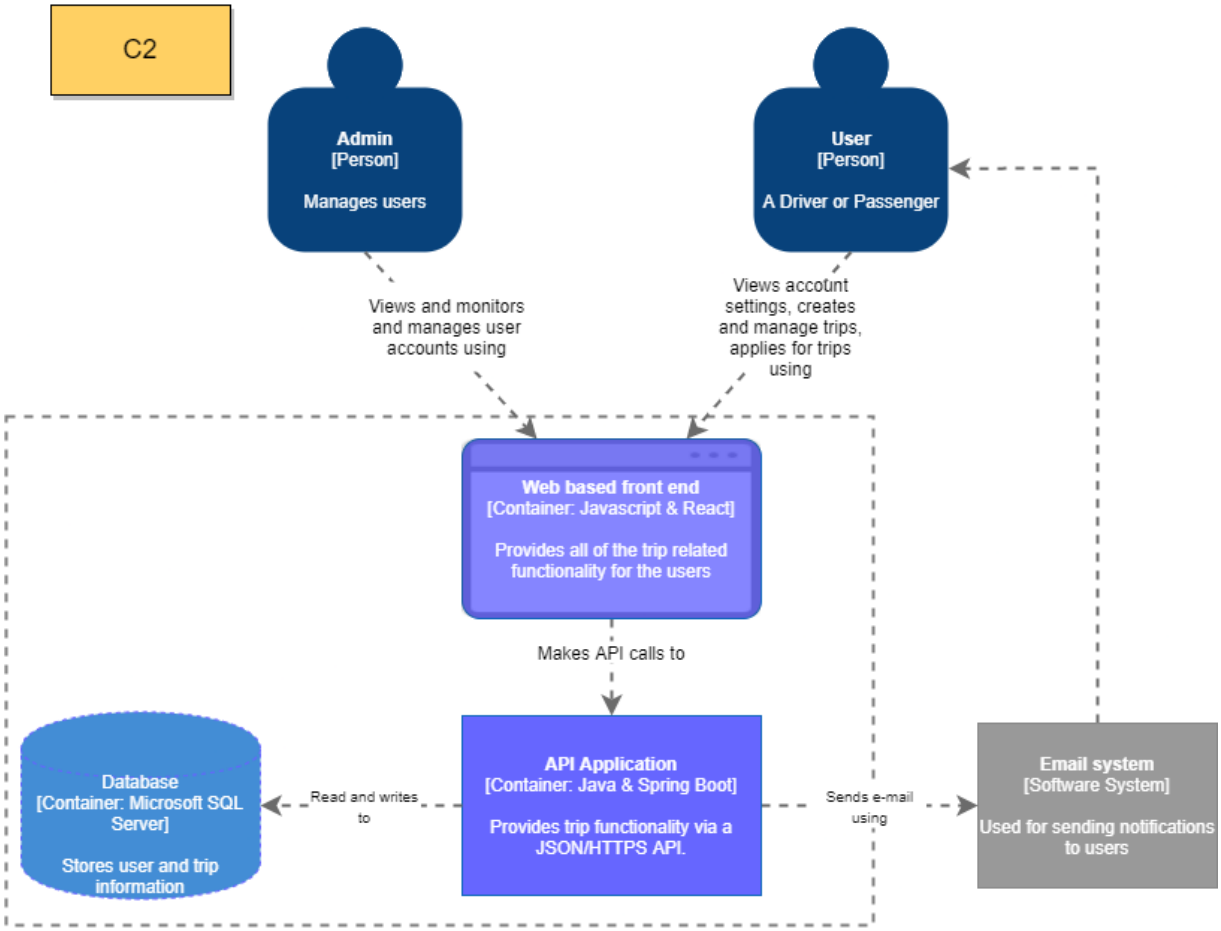
The withdrive application has to do with two main types of users. One being the general user of the application and the second being the admin of the platform. They have completely different roles on the platform, but is website should be used by both:

- The main function of the administrator is to manage users and act on support tickets.
- The user can be both a driver and a passenger. They can create a ride, in addition to applying for trips listed by other users.

The actions of both user types are undertaken on the same Trip Management System. Whilst their views are different the access roadmap is the same.

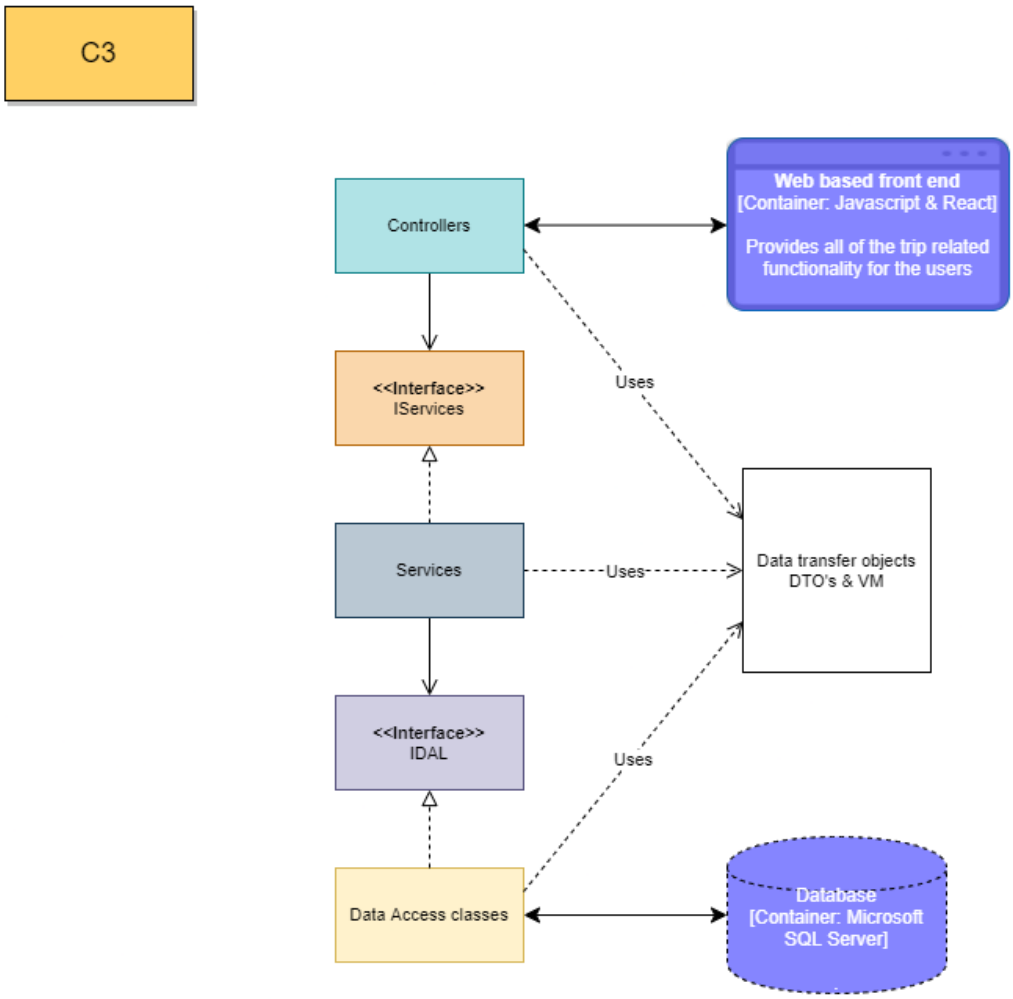
The Trip management system also uses an Email system to notify users/clients about events that have to do with their accounts, as well as remind them of actions that they should take.

C2:



Looking deeper into the Trip Management System we expand onto it into three main components. That being the Web based front end, the back-end API as well as the database. The front end allows the interaction of the two users with the application, this is done using web interfaces that allow for the users to edit personal data, CRUD trips, apply and react to applications. This frontend makes calls to the backend, the API of the application. This part manages all the requests and reads and writes to the database in order to make the data persistent. The API also uses the Email system to send emails to the users.

C3:



Looking deeper into the API Application, we arrive at a high level, three tier design. The system architecture is designed to adhere to the SOLID principles.

The application is split into various layers/modules that each interact with one another, and each has its own specific function. We make this separation to improve scalability, readability as well as to adhere to the Single responsibility principle. Each component therefore consists of a Controller, Service, and DAL (Data Access Layer). *(Organize your application code in three-tier architecture, 2021)*

Every time one of these layers interact with the underlying layer, they need to send/receive data, which are DTOs, aka Data Transfer Objects. Only DTOs are used between layers. Similarly, the Controller may rearrange the data to prepare it for presentation, so the data sent to the web browser is different from the data received from the Service Layer.

The interfaces in between the layers are used to take advantage of dependency injection. As well as allowing the application to adhere to ID of the SOLID principles. This process is made

easier, using Spring Auto-wired. As the Spring framework automates the process of injecting the dependency and completes many responsibilities under the hood.
The web based front end exchanges HTTPS requests with the controllers and data is exchanged via JSON files. The Data Access classes interact with the database and read and write data to and from it.

C4:

Work in progress.

2.2 Important design decisions

Selected technologies:

The technologies selected for this project were deliberately ones that have many online resources and are used in the industry. In the case of the withdrive project, the back-end rest service is developed using **Java & Spring** and dependencies are managed using Gradle. As mentioned before there are many resources online about how to work with this combination and therefore it makes development easier as there are sources to consult and cross reference to find optimal solutions. Spring Boot allows us start coding instantly without making the developer waste time on preparing and configuring the environment. Plug and play of sorts. The framework is also a perfect fit for the assignment as it has made making the API for web applications easy.

For the same reasons the front-end Javascript framework called **React** was selected. As it too has a vast collection of extensive online resources and documentation. React also has many packages that work well with it. A great example of one that is being used within the project is react-bootstrap. It is a library that contains many react components that are prebuilt using bootstrap. This allows for an elegant user-interface to be created in a shorter time span than it would have taken to create said interfaces using pure HTML & CSS.

2.2 Applied research

The universally unique identifier (UUID) is a 128-bit ‘label’ that is used for information tagging in computer systems. A UUID can be used to identify something with near certainty that the identifier will not be duplicated if created to identify something else.

Information labelled with UUIDs by independent parties can therefore be later combined into a single database or transmitted on the same channel, with a negligible probability of duplication. *(Commons Id Team, 2021)* As UUID's are unique and due to their length and randomness are ideally harder to guess.

When working on the backend rest API of the withdrive project. An issue of security and access came up. When a record is generated, say a trip or the user. This is usually done by returning an id to differentiate an instance of a record. In a restful service, interaction is done using HTTPS requests directed at an URL. *(Almighty Java, 2021)* As an example if records do not use UUID's, and instead are identified using int's or long's then this exposes a glaring security issue. Say if the URL for a necessary get request for an user with ID 0 is 'http://localhost:8080/user/0'.

A user with malicious intent could potentially access another user's data by changing the 0 in the URL to a 1 and so on. And as a result, they could make HTTPS calls and requests on other users' profiles. Therefore, an option to counteract this is by using UUID's for interaction between the front-end and the rest points to the back end. When UUID's are utilised in this way, then a malicious user would not be able to easily discern what the next users UUID is as no obvious pattern is present in the generation. As is shown in this example: 'http://localhost:8080/user/aaf06f07-8e1b-46c5-9d3a-5610f9eb30ea'. This essentially refers to the user of id 0. But that is not obvious from the returned path. This is further reinforced when getting a GET request of ID 1: 'http://localhost:8080/user/25bce054-29d1-40d9-beb1-1d4788364a2b'.

The use of UUID's for this purpose, however, is not fault proof. A new way of brute forcing UUID's is now being applied using a method called a 'Sandwich attack'. Essentially once an attacker knows what kind of version of UUID was used for generation, the hacker can apply the sandwich attack and attempt to brute force a result. This however requires deep background understanding. *(Breaking Down UUIDs, 2019)* Therefore, other security solutions should be used in addition to this, but those are to be expanded on in later iterations of this document.

3. Test strategy

1.1 Why do we test?

The goal of software testing is to find errors, gaps, or missing requirements in comparison to the actual requirements. We do this so that we can find and squash bugs before delivering a software product to the client. Testing is helpful when we incorporate new features, as we may run older tests that worked before, and ensure that the implementation of new features did not disrupt the application.

1.2 What should we test?

It is important to test the C4 part of the implementation. At the C4 level, the whole program is separated into units. These units (pieces of code) may be tested, and this action is called unit-testing. The point of unit testing is to ensure that each unit of the program is working correctly as per defined assertions.

After you make sufficient unit test and ensure that the code coverage of said tests encompasses all units of the application, then you must check if all the components within the scope of the application are working correctly and as intended. This is called integration testing, and as the name suggests we test whether all the components when put together are working correctly and as described.

Next, we must test how the entire system functions. Again, we zoom out and now we are testing the behaviour of all the elements working together. This is called system testing. We do this to evaluate the system's compliance with requirements. System testing takes, as its input, all the integrated components that gone through and passed the integration testing phase.

Finally, to ensure that the end-product meets all the requirements set out by our clients, we do acceptance testing. These tests involve testing the software based on the needs set out by the client/s. The goal is to test and successfully pass the requirements and criteria that was set out by the client.

1.3 How do we test?

This section will look at how the aforementioned testing phases are applied in the real world with a real project. This will include an outline on the tools that will be used for each of the test stages.

Unit testing – For unit testing of the withdrive project, I will be using Junit5 for the testing, as it is a popular framework for Java testing. To mock the database, I am using Mockito, this allows me to test the service layer efficiently.

Integration testing – Once all unit tests are made and pass, I then can start integration testing all the components of the backend. There are frameworks available to do help do this, my choice being Selenium, it is one of the most popular integration testing frameworks for Java.

System testing – Once all unit tests and integration tests are made and pass, I then can start system testing the entire application. This is done by testing the test cases that can be found test-plan document. For this part I will be testing the application and checking whether the desired outcome is met or not.

Acceptance testing – Once all of the aforementioned tests are made and pass, then I will be able to try the acceptance tests. Here we test the requirements set out by the client/s and see whether the application does everything that there was set out for it to do. This is done to ensure the end client is satisfied with the experience.

Once all of these tests pass, then you may consider your application thoroughly tested and ready for deployment to the client and subsequent end users.

3. Bibliography

1. YouTube. (2021, July 1). *68 - Spring Boot : How to use UUID instead of long? | UUID as primary key*. YouTube. Retrieved October 6, 2021, from https://www.youtube.com/watch?v=KkvAFRKgJ8E&t=0s.

2. Nick Steele Senior R&D Engineer @codekaiju, Steele, N., & Engineer, S. R. D. (n.d.). *Breaking down uuids*. Duo Security. Retrieved October 6, 2021, from https://duo.com/labs/tech-notes/breaking-down-uuids#:~:text=UUIDs%20are%20generally%20used%20for,physical%20hardware%20within%20an%20organization.

3. Team, C. I. (n.d.). *UUID COMMONS*. Commons ID - UUID documentation. Retrieved October 6, 2021, from *https://commons.apache.org/sandbox/commons-id/uuid.html*.

4. *Organize your application code in three-tier architecture*. OpenClassrooms. (2020, July 3). Retrieved October 6, 2021, from https://openclassrooms.com/en/courses/5684146-create-web-applications-efficiently-with-the-spring-boot-mvc-framework/6156961-organize-your-application-code-in-three-tier-architecture.

5. *Security considerations when building an application.* VerSprite Cybersecurity Consulting Services. (n.d.). Retrieved October 6, 2021, from https://versprite.com/blog/universally-unique-identifiers/.