withdrive

withdrive

Software design document

withdrive

Table of Contents

Contents

withdrive

1. Introduction

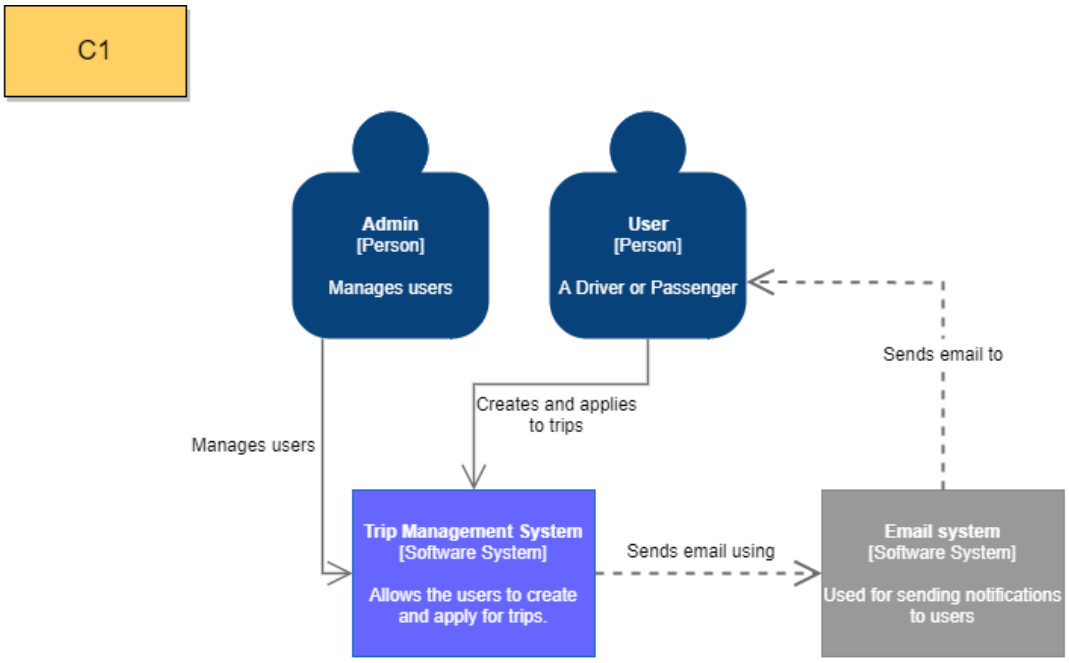
1.1 Document purpose

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:



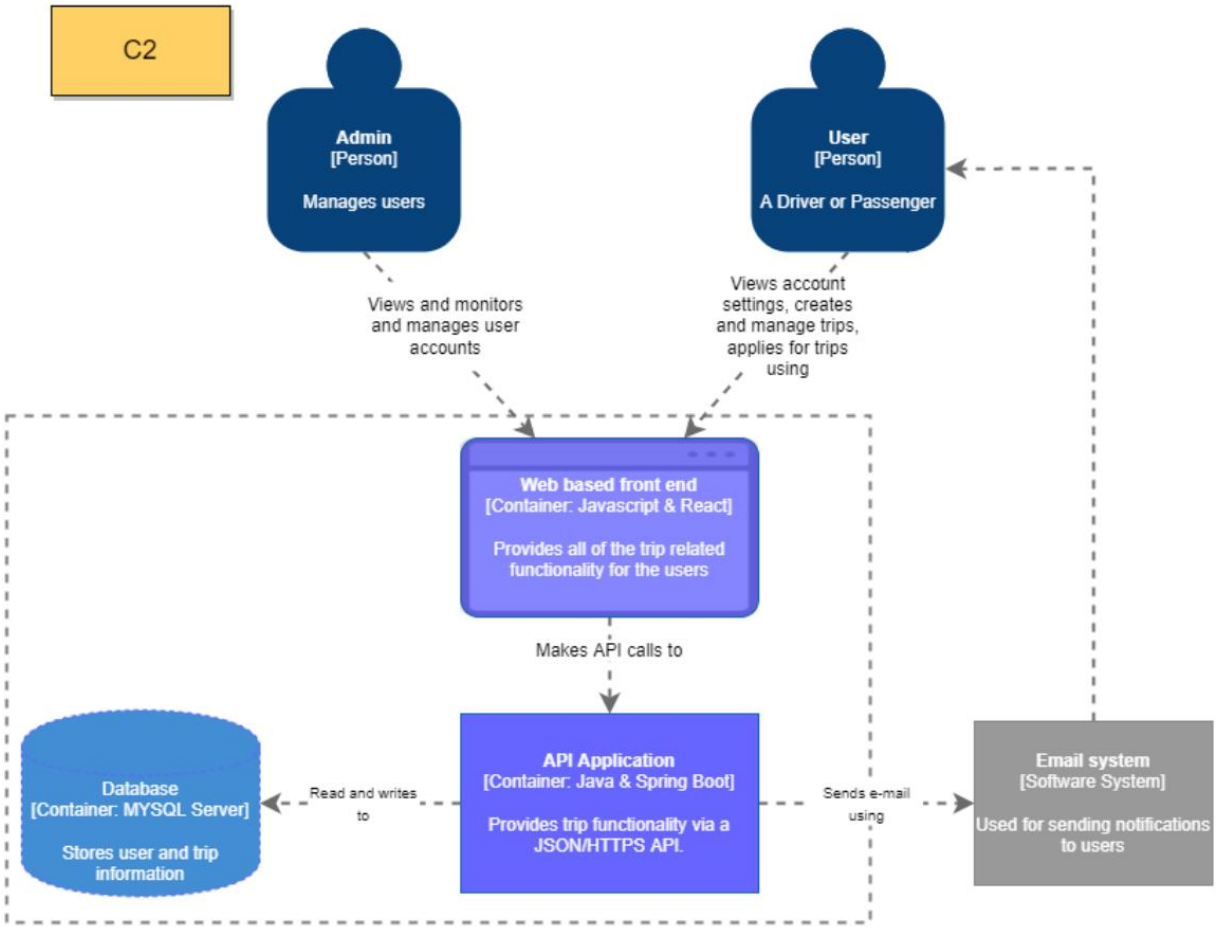
The withdrive application has to do with two main types of users. One being the general user of the application and the second being the admin of the platform. They have completely different roles on the platform, but is website should be used by both:

- 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.

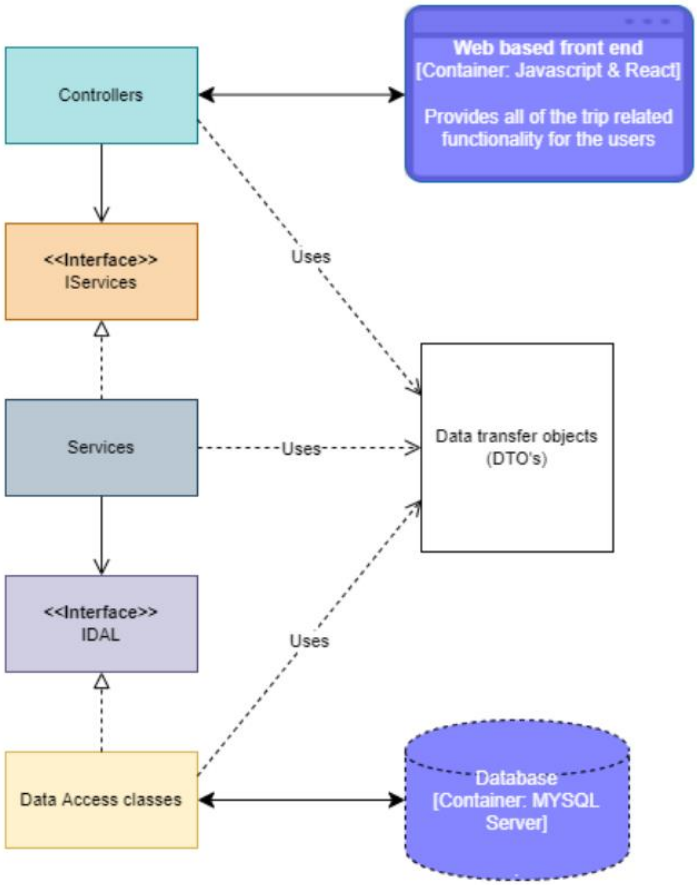
The Trip management system also uses an Email system to notify users/clients about events that have to do with their accounts, as well as remind them of actions that they should take.

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 DTO used depends on the request, say if a user is creating a trip, the DTO the controller expects would just hold the UUID of the driver, as that all it gets. Then deeper in the

application (in the service) using the converter it extracts a different DTO that has the actual user driver object. This is also present when a trip may hold a different DTO version of the user object that has less fields, so that 'full' user objects are ever sent to the frontend as that is a security risk. This is also elaborated on later in this document. Continuing, 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, write and update data to and from the database.

2.2 Important design decisions

Selected technologies:

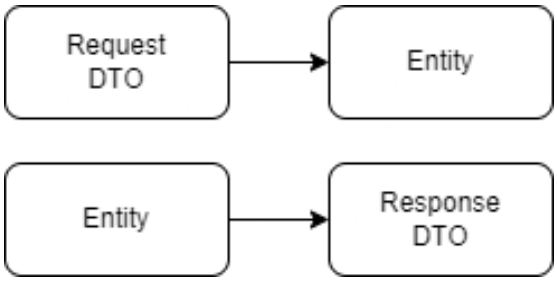
The technologies selected for this project were deliberately ones that have many online resources and are used in the industry. In the case of the withdrive project, the back-end rest service is developed using **Java & Spring** and dependencies are managed using Gradle. As mentioned before there are 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. I also use other libraries, such as MUI, react-router-dom and others. We also use Cypress for testing the frontend.

A general reason I chose these technologies for their wide use around the industry, as well as the aforementioned details of how many resources there are available online. This is a point discussed in the Best good and bad practices in the **DOT framework**. Which mentions that incorporating what has proven to work somewhere else forms the basis of any high-quality project. (*Best good and bad practices - ICT research methods*, 2018)

Miscellaneous decisions & notes:

- Use of converters



The premise of using DTO/ENTITY converters the backend is to use them to convert between types of DTO's and entities that are used throughout many parts of the program. This requires for say coming in Request DTO's that hold UUID for say fields like driver or passengers, to be converted from UUID's to the actual objects which are stored in entities so that you may write to the database using JPA easily. This is also true vice versa, say an entity has too much information/data, you must transform it into a DTO that has only the necessary details for a set operation so that you are not sending sensitive data to the frontend which as we know is not secure and the data can be pried into.

- Backend Pagination

Backend pagination is an important consideration and is integral to creating an application that will scale. Essentially data/resources are only loaded when Pagination is done by specifying to the user how many results to display per page, and which page of results they are currently viewing. *(Everything You Need to Know About API Pagination | Nordic APIs |, 2019)* Within the withdrive application, this is only present in one page. Namely, on the main trip page where all upcoming trips are shown. Due to time constraints of the project, this was the only place where this was done as it required quite extensive refactoring to existing code. That page specifically has this feature since, that page can be accessed by anyone, regardless of if they are logged in or not. Meaning that if there are many visitors this would strain the API with potentially thousands of calls based on the

active trip count. This implementation negates this problem as data is only fetches based on the tables page.

- Backend input validation

Input in the backend is an important because hackers may circumvent any potential frontend validation of fields and may make requests with data (JSON ect.) that may break the application. To prevent this, all inputs should be checked in the service classes. Again, due to time constraints of this project, these kinds of checks were only included in one part of the application, where inputs are checked with regular expressions to sanitize and validate inputs so that they don't violate defined/allowed formats.

- Backend filtering

Filtering in the backend is also something that is included in this project, namely in all upcoming trips page. Where the user is given the ability to select to show all trips originating from their city. This string is then sent to the backend where a custom query is written which finds upcoming trips that match the given origin city string.

- Security

The security aspect in the withdrive project is done using the JWT (JSON Web Token). This is an essential part to the application say when a user would like to apply for a trip, they may only do so if they can be identified, and this exactly is done using a JWT for authentication. This is also used to disallow unauthenticated users to make certain calls to the API as endpoints require auth through headers sent with requests. This is also done to stop unauthorized users to access pages not for them, say the admin page. As they don't have the necessary roles that are set out in the JWT and ensured by the signature. Currently, the token is stored in local storage, this is however unsafe. In the future the token could be stored in a HTTPS only cookie as that is less vulnerable to attacks. Another security addition would be to store the deception secret externally and have it changed once every 24 hours. The refresh token can also be implemented to refresh the access token in the future, to improve usability and security. *(auth0.com, 2013)*

- Reusable Components

Within the frontend of the application, components are reused. Whilst this is an anti-pattern and is not recommended by some in the React community *(Gard, 2018)*. This is done to shorted development time, as when components are created, they may be also adapted and reused, and this has helped me develop the frontend in a cleaner manner.

- WebSocket

WebSocket's are used in the project to allow the admin user to send notifications to all the users that are present on the website. This is done using the Stomp client and SockJS. A popup notification is shown on all user's screens. It also has a timer after which is disappears.

- Use of scheduled tasks in SB

As per one of the user stories, a trip who's start date has passed must be set to closed and started by the system rather than by the driver. To make this happen the Spring boots annotation called @Scheduled is used which allowed for a method to be set to execute every 30 minutes. This means that every time interval the method is executed, and trips are checked whether they have started or not. If they have then they are locked in application and are marked as begun. From that point on passengers of said trip may leave reviews for the driver.

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

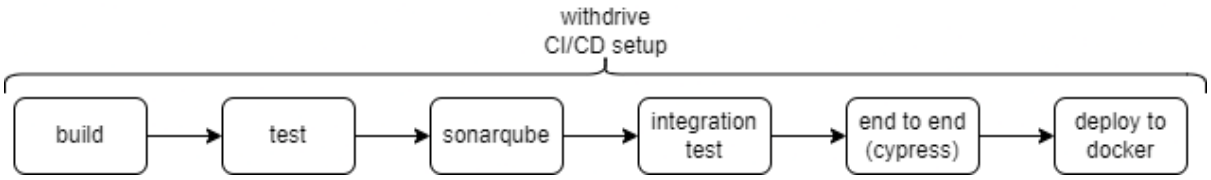
interaction between the front-end and the rest points to the back end. When UUID's are utilised in this way, then a malicious user would not be able to easily discern what the next users UUID is as no obvious pattern is present in the generation. As is shown in this example: 'http://localhost:8080/user/aaf06f07-8e1b-46c5-9d3a-5610f9eb30ea'. 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.

This was gathered with the **Literature study** DOT framework research method in mind, as I looked at many sites to extract knowledge and information from to come to a conclusion from many sources, which led me to seeing that such a addition would assist my application.

2.2 CI/CD setup

The problem of continuous integration is one that has been tackled many times, and with the knowledge acquired by people who have worked in the industry for many years we may use try tested tools. As community research into the topic has yielded extensive results. (*Community research - ICT research methods, 2018)* In my case a Gitlab tool is used.



The above diagram is a representation of how the CI/CD pipeline works, so initially as a commit is pushed to the main (master) branch on the project's git. Then the CI/CD pipeline kicks in and goes through these three steps to ensure that the new code that has been added in said commit did not have an effect on the application and did not break tests and checks that were tested beforehand. This is also done automatically without the need for the user to initialise the check and this makes it very convenient fore use. Also, when branches are merged then they only do so if the branch that you are merging from also passes all tests. This ensures integrity. *(GitLab - CI/CD, 2021)*

It is important to explain the stages of the pipeline as each plays an important part of creating and maintaining an application and ensuring that what is given to the client is in working order. The first stage checks whether Gradle can build your solution, the second stage tests defined unit tests, the third is SonarQube which checks code coverage of tests as well as any code smells that have been added. The next two stages coincide with one another where the integration test checks that the API works correctly, and then the cypress end to end test the frontend is tested in unison with essentially the rest of the application (hence the name end to end). Where the user stories are simulated and tested. Then the final stage is the building of images and deploying of all the elements into docker for production use to the client.

It is also important to mention that if any of the stages before the docker stage fail then the pipeline will stop, meaning only working solutions that past all tests will be put to production. This is an important part of the CI/CD pipeline as the integrity of the continuous deployment is upkept.

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/.

6. *Methods - ICT research methods*. (2021). Ictresearchmethods.nl. https://ictresearchmethods.nl/Methods

7. *Everything You Need to Know About API Pagination | Nordic APIs* |. (2019, October 17). Nordic APIs. https://nordicapis.com/everything-you-need-to-know-about-api-pagination/

8. Gard, N. (2018, June 2). *Not-So-Reusable React Components - Nick Gard - Medium*. Medium; Medium. https://ntgard.medium.com/not-so-reusable-react-components-d09ded69afe9

9. *GitLab - CI/CD*. (2021). Tutorialspoint.com. https://www.tutorialspoint.com/gitlab/gitlab_ci_cd.htm

10 auth0.com. (2013). *JSON Web Tokens - jwt.io*. Jwt.io; Auth0. https://jwt.io/introduction