



Rapid Prototyping in TypeScript

Semester Thesis - HS 2024

Department: Computer Science
Field of Study: Software Engineering

Authors

Isaia Brassel
isaia.brassel@ost.ch
Matriculation: 22-173-264

Silvan Kisseleff
silvan.kisseleff@ost.ch
Matriculation: 22-176-135

Involved People

Olaf Zimmermann
olaf.zimmermann@ost.ch
Advisor

Abstract

Building a fullstack web application with NodeJS in today's landscape of countless frameworks and libraries can be daunting. Which ones should be chosen? How to configure the chosen libraries so that they work together seamlessly rather than against each other? Many technologies and libraries feature a simple "hello world" tutorial but combining the technologies together to craft a powerful fullstack web application is not straightforward. The lack of a good introduction to a new technology slows down the development process and distracts developers from focusing on what truly matters: building features. For this purpose we developed an approach with rapid prototyping. Our proof of work focuses on fast development without the need of always adapting the API when changes are made. Our approach to rapid prototyping is based on a GraphQL API that supports code generation with the help of TypeGraphQL-Prisma, which creates the resolvers for create read update and delete operations for each data model out of the box. To bring rapid prototyping to the developers, we created an easy to understand tutorial with Docusaurus, a static website generator that makes it easy to build documentation-focused web applications that are accessible and engaging, that we refined based on feedback we gathered from user testing. The tutorial demonstrates our technologies in a realistic and easy to understand example application where you can make ticket reservations for a cinema chain.

Management Summary

Context

Research into rapid prototyping tools highlighted the potential to significantly simplify application development by focusing on adaptability and efficiency. To achieve this, we selected tools and techniques to build an application with a GraphQL API that dynamically adjusts through generated code. This setup eliminates the need for manual updates whenever changes are introduced, making the process faster and more reliable. To help users understand the dynamic approach, we created a demo application and a tutorial. The demo application is an online ticket reservation for a cinema. We chose this domain because it is easy to understand. The tutorial guides users through every aspect of our rapid prototyping method in an interactive way. As users work through the tutorial, they gain practical experience with the tools and techniques, enabling them to apply this approach to their own projects. The GraphQL API is implemented with Apollo Server, combined with automatically generated resolvers based on TypeGraphQL Prisma. TypeGraphQL Prisma provides out-of-the-box support for create, read, update, and delete (CRUD) operations for every data model, reducing the effort required to build and maintain the API. For custom operations we leverage TypeGraphQL so that complex processes can still be realized. The frontend is built with React. This setup ensures a dynamic, responsive user experience. For the tutorial, we used Docusaurus, a static website generator that makes it easy to build documentation-focused web applications that are accessible and engaging for users to learn and explore our template. This combination of technologies streamlines the development process, providing both a practical introduction to rapid prototyping and a robust foundation for scalable application development.

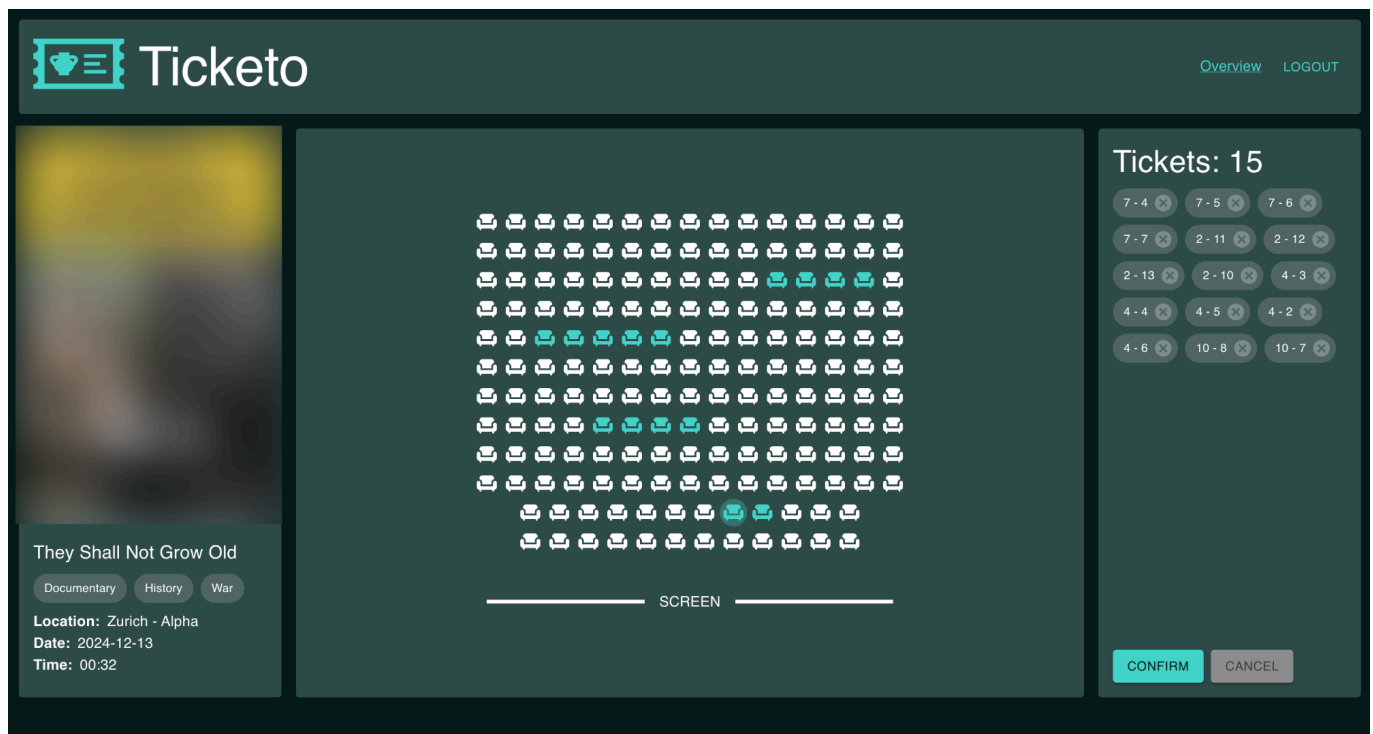
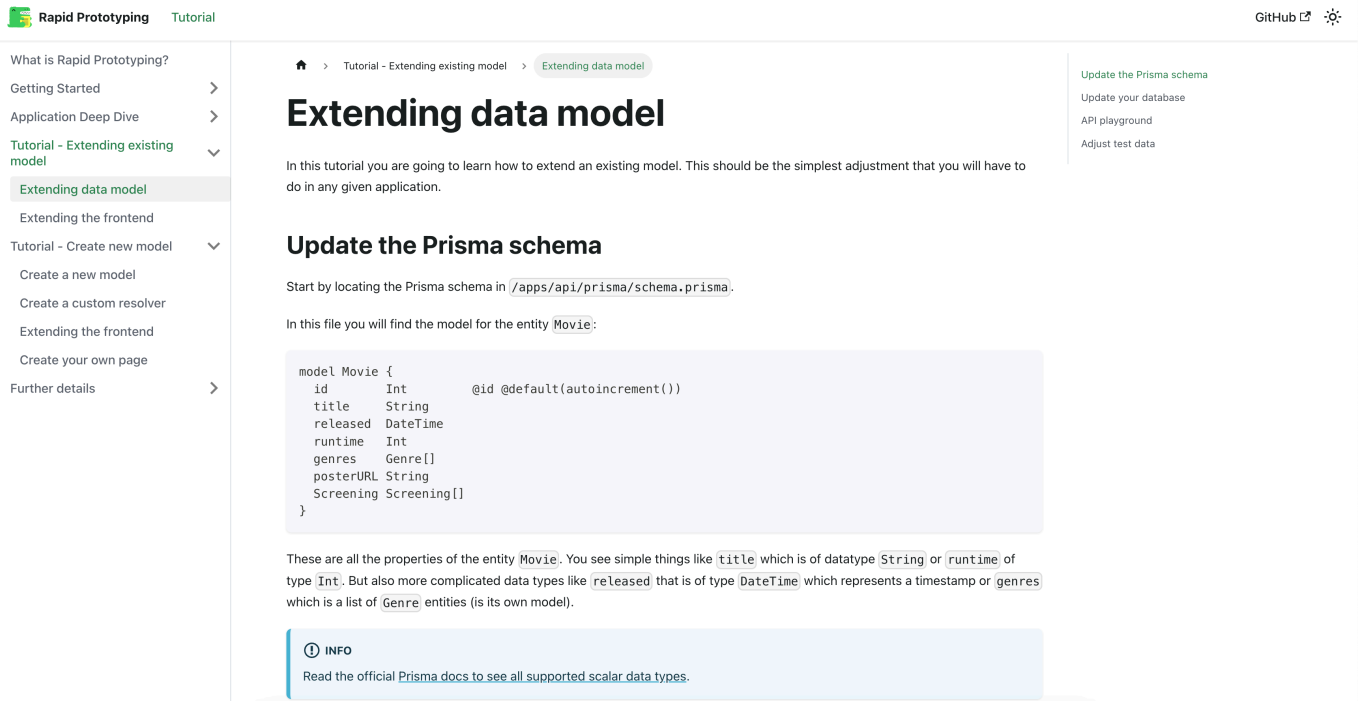



Figure 1: Seating plan screen of the demo application

Solution

Our solution is a simple project template that demonstrates how to use our chosen technologies in a rapid prototype manner. We use a GraphQL API with Apollo Server, combined with resolvers automatically generated from our object relational mapper Prisma that supports create, read, update and delete out of the box for each data model. The user interface is built with React and uses the GraphQL client Apollo Client to talk with our GraphQL API. By writing easy to understand tutorials and creating a demo application in a well-known domain, we created a product

that brings rapid prototyping to the user. The tutorial shows how it is possible to adapt the already existing features with rapid prototyping. It is also possible to create new features from scratch. To assure, that the developer does not get stuck on some steps, we also provided a solution for every single step, which the developer can look at if he needs to. We also ensured that the developer does not have to waste time on a complicated setup by implementing the entire application in Docker containers, simplifying the process. Different user tests showed that the tutorials are easy to understand and intuitive and that our template increases productivity.



Rapid Prototyping Tutorial GitHub 

What is Rapid Prototyping? >
Getting Started >
Application Deep Dive >
Tutorial - Extending existing model >
 Extending data model
 Extending the frontend
Tutorial - Create new model >
 Create a new model
 Create a custom resolver
 Extending the frontend
 Create your own page
Further details >

Home > Tutorial - Extending existing model > Extending data model

Extending data model

In this tutorial you are going to learn how to extend an existing model. This should be the simplest adjustment that you will have to do in any given application.

Update the Prisma schema

Start by locating the Prisma schema in `/apps/api/prisma/schema.prisma`.

In this file you will find the model for the entity `Movie`:

```
model Movie {
  id      Int      @id @default(autoincrement())
  title   String
  released DateTime
  runtime Int
  genres  Genre[]
  posterURL String
  Screening Screening[]
}
```

These are all the properties of the entity `Movie`. You see simple things like `title` which is of datatype `String` or `runtime` of type `Int`. But also more complicated data types like `released` that is of type `DateTime` which represents a timestamp or `genres` which is a list of `Genre` entities (is its own model).

INFO
Read the official [Prisma docs](#) to see all supported scalar data types.

Update the Prisma schema
Update your database
API playground
Adjust test data

Figure 2: Screen of the first tutorial

Future work

This project also serves as a basis for a consecutive project where one could build a web tool that allows the creation of CRUD API's with an interactive user interface to define and build the data model. We defined a list of possible features and deliverables in the [appendix](#).

Table of Contents

1 Introduction	1
1.1 Why should one read this documentation?	1
1.2 Target audience	1
1.3 How should one read this documentation?	1
1.4 Rapid prototyping	1
2 Glossary	3
Product description	4
3 Domain analysis	5
4 Requirements	6
4.1 Actors	6
4.2 Use case diagram	6
4.3 Functional requirements	7
4.3.1 FR1: View next movies	7
4.3.2 FR2: View seating plan for movie	7
4.3.3 FR3: Reserve movie ticket online	7
4.3.4 FR4: Confirm movie ticket reservations offline	7
4.4 Non-functional requirements	8
4.4.1 NFR1: Browser independent functionality	8
4.4.2 NFR2: Operating system independent functionality	8
4.4.3 NFR3: Simple setup with minor effort.	8
4.4.4 NFR4: The demo app is simple and compact.	9
4.4.5 NFR5: Responsive user interface	9
5 Tutorials	10
5.1 Structure	11
5.1.1 Getting started	11
5.1.2 Application deep dive	12
5.1.3 Extending a data model	12
5.1.4 Creating a data model	12
5.1.5 Further details	12
Product realization	13
6 Architecture	14
6.1 Evaluation of technologies	14
6.1.1 Web framework	14
6.1.2 Component library	14
6.1.3 GraphQL code generation	14
6.1.3.0.1 Postgraphile	15
6.1.3.0.2 GraphQL-Code-Generator	15
6.1.3.0.3 TypeGraphQL-Prisma:	15
6.1.4 Build tool	15
6.1.5 Movie data	16
6.1.6 API patterns	16
6.2 C4 model	17

6.2.1 System context model (C1) and Container diagram (C2)	17
6.2.2 Component diagram (C3)	18
6.3 Sequence diagram	18
6.4 User interface sketches	19
6.5 Folder structure	22
6.6 Project setup	22
6.6.1 Linting and formatting	22
6.6.2 Containerization	23
6.6.3 Docker orchestration	23
6.6.4 Building	23
6.6.5 CI/CD	24
6.7 Authentication / Authorization	25
6.7.1 Important note	25
6.7.2 Authentication	25
6.7.3 Authorization	28
6.8 Error handling	28
6.8.1 Invalid JWT	29
6.8.2 RLS violations	29
6.8.3 Client side error handling	30
7 Quality measures	31
7.1 Test concept	31
7.1.1 Quadrant one	31
7.1.2 Quadrant two	32
7.1.3 Quadrant three	32
7.1.4 Quadrant four	32
7.1.5 Level of testing	32
7.1.6 Time of testing	32
7.1.7 Summary	32
7.2 Testing protocols	32
7.2.1 Manual tests	32
7.2.1.1 Browser independent functionality (NFR1)	32
7.2.1.2 Operating system independent functionality (NFR2)	33
7.2.1.3 Simple setup with minor effort (NFR3)	33
7.2.1.4 The demo app is simple and compact (NFR4)	33
7.2.1.5 Responsive user interface (NFR5)	33
8 API generation tool	35
8.1 Requirements	35
8.1.1 R1 Create and edit a data schema	35
8.1.2 R2 Verify data schema	35
8.1.3 R3 Custom resolvers	35
8.1.4 R4 Save the data schema	35
8.1.5 R5 Error handling	36
8.1.6 R6 Allow access constraints	36
8.2 What should get generated	36
Summary	37

9 Summary	38
9.1 Overall thoughts	38
9.2 Alternative ways	38
Appendix	39
10. Test protocols	40
10.1. Integration test template	40
10.2. End to End testing (E2E)	41
10.2.1. Login roundtrip	41
10.2.2. Genre filter on overview page	41
10.2.3. Detail movie view	42
10.3. User tests	43
Figures	45
Bibliography	46

1 Introduction

We, Silvan Kisseleff and Isaia Brassel, are software engineering students at OST Rapperswil. We both have a strong interest in web development and focus our studies around it. Since APIs are always a big topic when it comes to web development, we decided that our semester thesis will revolve around it. Sadly working with APIs isn't always as easy as you would like it to be. And because of this we decided to work on a proof of work for rapid prototyping.

1.1 Why should one read this documentation?

Great product ideas arise every single day, but implementing these ideas quickly in a prototype to test it, is not always as easy and cheap as we would want it to be. One can create a static wireframe to visualize a product idea, but there is much more value when there is a clickable and working prototype that enables the implementation of processes and that have some sort of persistence to discover issues with the project idea and the required tweaks.

When implementing such a prototype as a developer, one would want to spend most of the time implementing features and quickly adjust existing data structures rather than spending time writing boiler plate code, repeating the same model bindings for the tenth time, and adjusting big models to fit for the newly adapted view of the problem. There are many existing libraries and frameworks in the NodeJS environment that do most of the heavy lifting, but it is not very easy to combine these parts into a powerful tool but rather a messy and confusing process until everything works together as intended. There are plenty "Hello World" tutorials for each of these libraries but the combination of these tools is not trivial and straightforward. This is why we see a gap where we can provide a clear structured solution.

This is where this proof of work comes to work. We demonstrate in a easy to understand and practical example how rapid prototyping can be simpler, faster and safer than traditional approaches. This allow a developer to spend the valuable time developing new features rather then fighting the technologies and frameworks while still have the extendability and maintainability that one would desire. Our practical example is a simple cinema ticket reservation application where a user can view the next movies and make a reservation for a certain screening.

Trough the tutorials it is possible to learn how to extend this application in a practical manner with guided tutorials that let the developer discover the effectiveness of our chosen frameworks and libraries. It also serves as a base for future tools that might get developed.

1.2 Target audience

This documentation is intended for tech-leads or software architects or software engineers that want to discover a simple approach to rapid prototyping. The tutorial and documentation are somewhat understandable for non-professionals, but certain details are not fully explained and require additional research by the reader. We are writing this project as a semester thesis so this documentation also serves as a result based documentation for our advisor or for any competitor interested in our project.

1.3 How should one read this documentation?

This documentation provides more of a deep dive into the project and shines a light to our decisions and thoughts when designing the application and rapid prototyping approach. For a more hands-on approach we have created a step by step tutorial that allows a developer to discover the functionality of this tech stack and understand how to extend this prototype. We did not copy all parts from either side to the other so it will not suffice to just read either one, they are indented to be read together. We recommend completing the tutorial first. For those interested in additional background information or further details about our decisions, this documentation can be consulted.

1.4 Rapid prototyping

Before we dive into our project we want to clarify the term "Rapid prototyping" in a more detailed manner. Rapid prototyping is the process of building a MVP (minimal viable product) in a quick fashion. A MVP is working application that represents the complexity and scale of the actual product that requires to be build. Traditional

approaches to rapid prototyping sometimes lack the ability for customization and fine grain control which do not make them viable production software. Our approach allows the developer to still have these controls and with a few tweaks to the setup one could even deploy it as a production grade software.

2 Glossary

See [agile glossary for common abbreviations used in this documentation](#). See [Domain analysis](#) for further explanation for domain specific terms.

Apollo Client A GraphQL client that can create and send GraphQL queries.

Apollo Server A GraphQL server that can resolve GraphQL requests.

CRUD Create, read, update and delete.

Docusaurus Static site generator for creating documentation websites using Markdown and React.

Earthfile Configuration file for Earthly.

Earthly Build tool to build docker images with local artifacts.

Express Minimal framework to create web applications and APIs, used by Apollo Server.

GraphQL Query language for APIs and a runtime to fulfill these queries.

Javascript A programming language commonly used for web development.

JWT JSON Web Token, a secure compact way to transmit credentials.

Material UI Component library for React that follows the Material Design guideline.

Mutation A GraphQL query that modifies data (create, update and delete).

Nginx High-Performance web server and reverse proxy.

Node A JavaScript runtime that can run on different platforms (cross-platform).

PostgreSQL Open source relational database.

Prisma ORM for TypeScript, also used for database access.

Prisma datasource Defines the connection to the persistence layer for Prisma.

Prisma generator A generator that determines what artifacts are created from the Prisma schema.

Prisma schema The source of truth, that defines the data models.

Resolver A function that is responsible to populate a field in a GraphQL query.

RLS Row level security, per table row access restriction.

TypeGraphQL Library to create resolvers for a GraphQL server.

TypeGraphQL-Prisma Prisma generator to create TypeGraphQL resolvers.

Part I
Product description

3 Domain analysis

We demonstrate the rapid prototyping with the help of a sample application: a cinema ticket reservation system. We decided for this simple and known domain so that it is easy to understand the technical part of our template rather than being confused with the chosen domain. In this chapter we describe the domain and the definitions of different jargon words.

Moviegoer The customer that wants to watch a movie.

Cashier The employee that confirms a ticket reservation offline at the cinema locally.

Movie A movie can be showed in multiple hall at multiple dates.

Screening A single instance where the movie is being showed.

Hall The room where the movie is being played at. It contains seats.

Seat A single seat inside a hall that can be reserved for a single screening. A seat knows its position inside a hall and if he is reserved or not.

Reservation A reservation for one or multiple seats in a screening for a specific movie in a specific hall. The seats are not confirmed yet, but no one else can reserve the seat.

In Figure 3 we visualize the relations of the different domain classes with a UML class diagram. Note that this diagram shows the domain state at the end of the tutorials and not the current implementation state of the template.

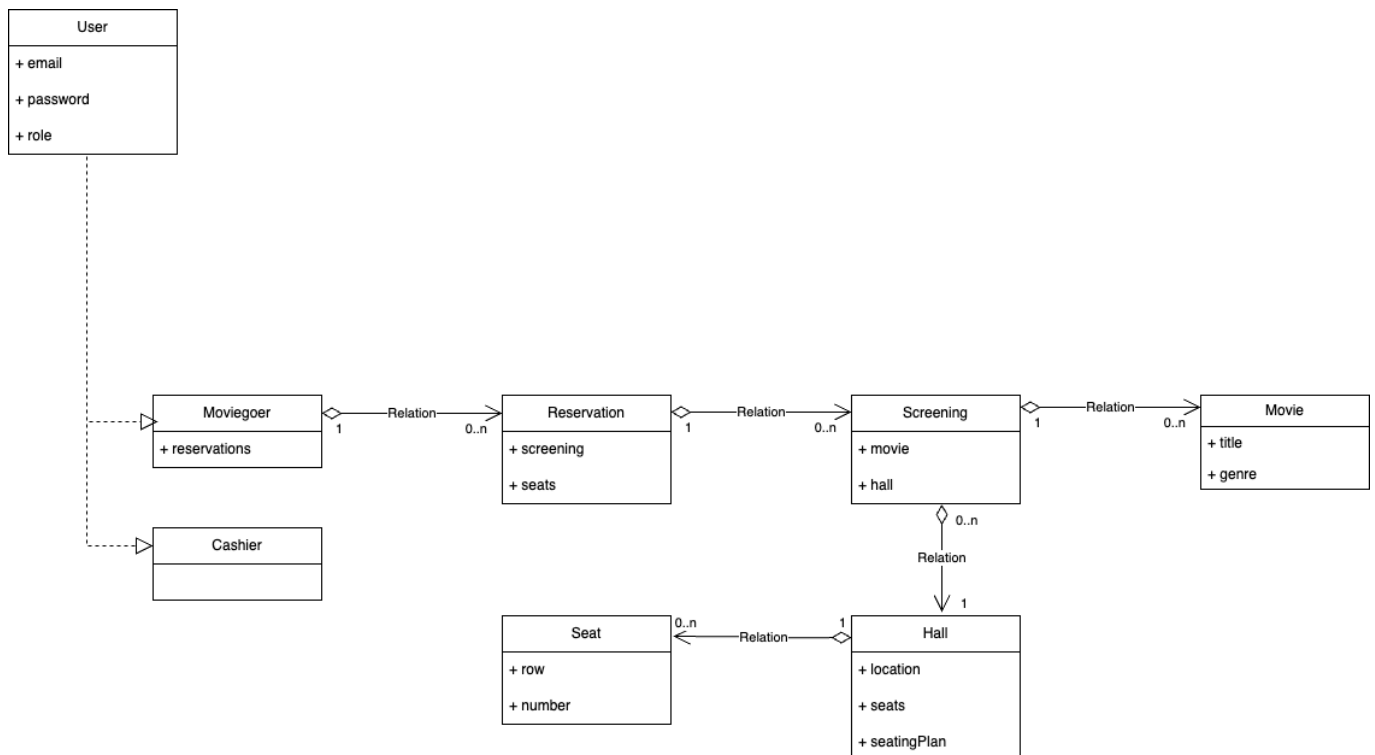


Figure 3: Analysis level domain model (UML class diagram)

4 Requirements

Our product and domain story is based upon the example product from the Domain Story Telling company [1]. We modified and enhanced the example domain story to develop a user-friendly product that effectively facilitates teaching various aspects of the technologies.

4.1 Actors

Here are the actors for our example application and their goals listed:

- Moviegoer
 - Confirm the reserved tickets at the Cashier in person.
 - View upcoming screenings of movies.
 - View seating plan for a movie.
 - Make an online reservation for a screening of a movie.
- Cashier
 - Can confirm reserved tickets in person to a Moviegoer.
 - View upcoming screenings of movies.
 - View seating plan for a screening of a movie.
- Cinema System
 - Processes the confirmations of the cashier.
 - Updates seating plan with the reservations.

4.2 Use case diagram

In the Figure 4 we visualize the different actors and their use cases in a use case diagram. It shows for example the primary actor “Moviegoer” with the use case “View next screenings of movies” with the secondary actor “Cinema System”. The remaining use cases are to be read the same way.

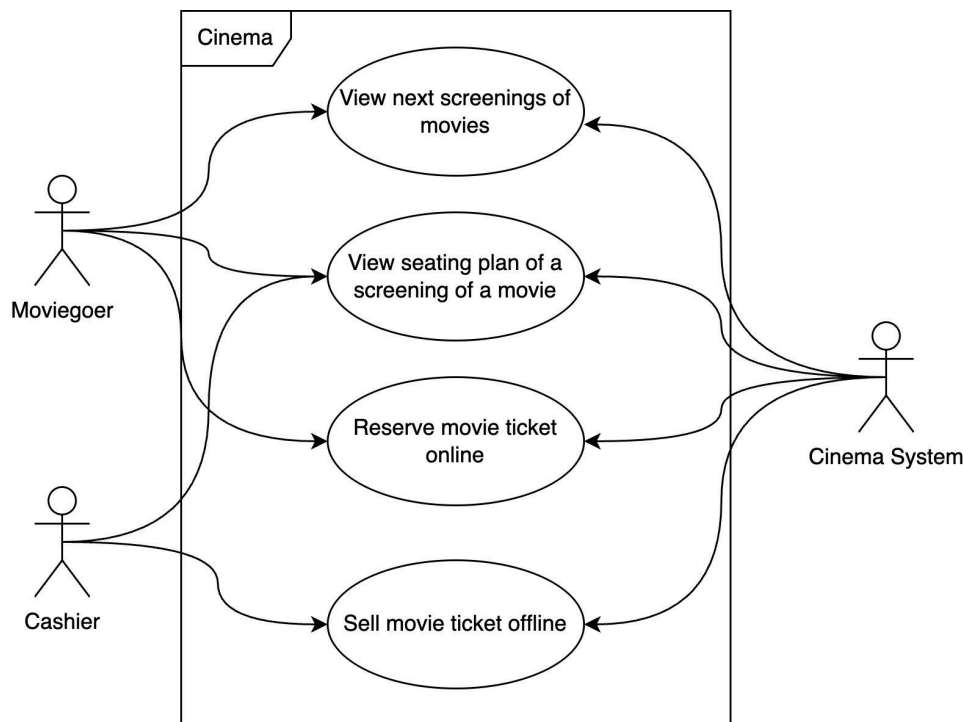


Figure 4: Use case diagram

4.3 Functional requirements

In this chapter we describe the functional requirements for the example application. Certain requirements are kept brief while the most important requirements are described with a fully dressed requirement template. The template for the use cases is based on Craig Larman's template for brief and fully dressed use cases [2].

4.3.1 FR1: View next movies

Main success scenario

- A Moviegoer and a Cashier can see a list of the next available movies and general information about the movie like duration, genre, time and cinema hall.

Alternate scenario

- If no movies are available, a text informing that no upcoming movies exist should be displayed.

4.3.2 FR2: View seating plan for movie

Main success scenario

- A Moviegoer and a Cashier can see the current seating plan for a specific movie and see which seats are still free.

Alternate scenario

- If no seats are available anymore, it should not be possible to do any reservations. Also a fully booked seating plan should be displayed.

4.3.3 FR3: Reserve movie ticket online

Scope Frontend

Level User-goal

Primary actor Moviegoer

Stakeholders and interests

- **Moviegoer:** The Moviegoer wants a simple way to make a reservation for a certain screening of a movie.
- **Backend:** The Backend wants to ensure that no double booking of an available seat can be done.

Preconditions

- Movie exists
- Screenings for movie exists
- Seating for screening exists

Main success scenario

1. Moviegoer selects movie
2. Moviegoer selects screening
3. Moviegoer selects seat(s)
4. Moviegoer reserves the seat(s)
5. Moviegoer sees reserved seat(s)

4.3.4 FR4: Confirm movie ticket reservations offline

Scope Frontend

Level User-goal

Primary actor Cashier

Stakeholders and Interests

- **Cashier:** The Cashier wants a simple way to confirm a ticket reservation for a certain screening of a movie.
- **Backend:** The backend wants to ensure that no double booking of an available seat can be done.

Preconditions

- Movie exists
- Screenings for movie exists
- Seating for screening exists

Main success scenario

1. Cashier selects a movie.
2. Cashier selects a screening.
3. Cashier suggests available seat(s).
4. Moviegoer chooses seat(s).
5. Cashier selects seat(s).
6. Cashier confirms the selected seat(s) for the Moviegoer.

4.4 Non-functional requirements

The following non-functional requirements are listed and categorized according to ISO/IEC 25010 [3].

4.4.1 NFR1: Browser independent functionality

Category Compatibility -> Interoperability

Description The demo application, that we use for our proof of work, has to run in the most used browsers on their latest versions. Such as: Google Chrome and FireFox.

Acceptance criteria A user should be able to run the app locally on his device and use any of the above mentioned browsers on their latest LTS-version. Without any additional setup the app should work.

Verification process

- Clone the git repository.
- Setup according to the README.
- Start app with docker-compose.
- Open the app in one of the browsers.
- Check if the app can complete all its processes.
- Repeat with another browser until you checked both browsers.

Verification period After every sprint, the app will be checked for compatibility on the different browsers.

4.4.2 NFR2: Operating system independent functionality

Category Portability -> Installability

Description The demo application, that we use for our proof of work, has to run on the most common operating systems such as: Mac OS, Windows and Ubuntu Linux.

Acceptance criteria A user should be able to run the app locally on his device that runs any of the above mentioned operating systems on their latest LTS-version.

Verification process

- Clone and start the app according to the README on Windows.
- Clone and start the app according to the README on Mac OS.
- Clone and start the app according to the README on Ubuntu Linux.

Verification period After the first construction sprint and in the transition sprint, the app will be checked for compatibility on the different operating systems.

4.4.3 NFR3: Simple setup with minor effort.

Category Flexibility -> Installability

Description The application, that we use for our proof of work, must be easy to setup. The user should be able to run it without a lot of effort on MacOS, Linux or Windows. Note: We do not support native setup. The main reason for this is that we have multiple applications that require different multiple steps to setup which would have to be done manually if you would want to run the project locally in a native way. To avoid impactful drift between development setup and production setup that would run on your servers we also streamlined the setup to minimize the differences. Furthermore we also have different applications that talk to each other and to avoid violating Cross-Origin Resource Sharing (CORS) policy we use a reverse-proxy which would have to be configured and running locally aswell. Due to all these reasons we opted to not support a native setup and focused on our streamlined setup with Docker and Earthly.

Acceptance criteria A user should be able to clone the git repository and start the application with a few commands.

Verification process

- Clone the app from the gitlab repo.
- Start the app according to the README.
- Check if the app is running without any errors.

Verification period After the alpha release of our project

4.4.4 NFR4: The demo app is simple and compact.

Category Product Quality -> Quality use

Description The app should have a domain that everyone is familiar with. It shouldn't be for software engineers exclusive. Furthermore should the app be held simple.

Acceptance Criteria A person that isn't specifically versified in software engineering should be able to use the application. The application also shouldn't have more than 5 screens and every screen should be reachable within 2 clicks.

Verification process

- Have someone who is not particularly familiar with web development test the app.
- The person should be able to use the web the intended way.
- The person has to reach every screen within 2 clicks from the landingpage.

Verification period After the alpha release of our project

4.4.5 NFR5: Responsive user interface

Category Interaction Capability -> Operability

Description The application should be responsive on multiple screen sizes. Reaching from mobile to big screens.

Acceptance criteria The app should be responsible on a screen-width reaching from 640px to 2560px.

Verification process

- Open a dev tool and test a screen with a width of 640px.
- Open a dev tool and test a screen with a width of 780px.
- Open a dev tool and test a screen with a width of 1024px.
- Open a dev tool and test a screen with a width of 1280px.
- Open a dev tool and test a screen with a width of 2560px.

Verification period After the alpha release of our project

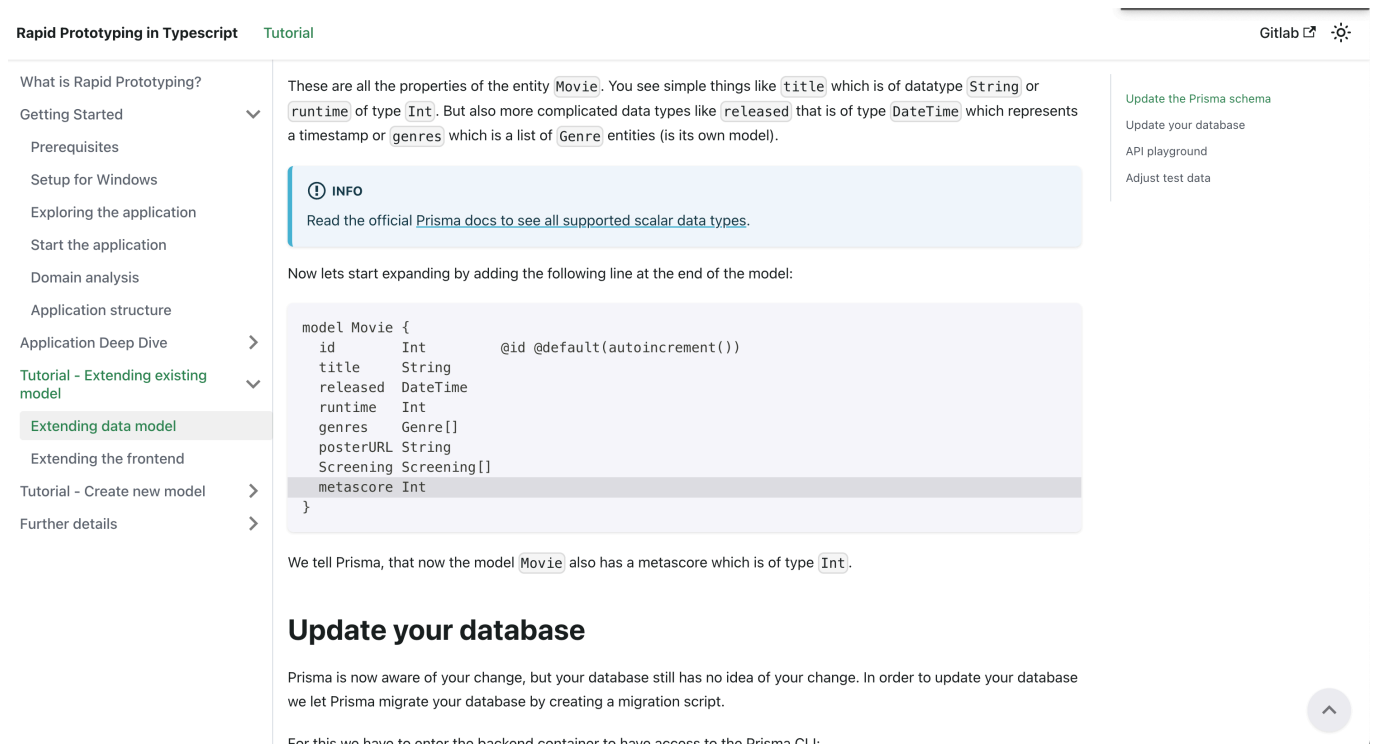
In the next chapter we will describe the tutorials created to teach the use of our template.

5 Tutorials


We want to provide the developer with tutorials that allow him to understand how the technologies and frameworks work together and guide him on how he can extend the current state of the application ranging from simpler to more complex adjustments that could also arise if he would decide to build a new application with these technologies.

For the step by step tutorials we provide the exact code snippet required, the file where it should be added or replaced and give instruction on how to confirm if the changes were successful, we did not include exact screenshots of the UI because we already provide the exact snippet and thus the result is quite obvious if it worked or not.

To create engaging and user friendly tutorials we use [Docusaurus](#) as seen in Figure 5 that allows us to create rendered React pages with code snippets, info boxes and much more from simple Markdown files.



The screenshot shows a web application interface for a tutorial. The title is "Rapid Prototyping in Typescript" and it's a "Tutorial". The page has a sidebar on the left with a navigation menu. The main content area is titled "Tutorial - Extending existing model" and "Extending data model". It contains text explaining the properties of the `Movie` entity, a code snippet for the Prisma model, and instructions on how to update the database. There is also an "INFO" box and a "Update your database" section.

Rapid Prototyping in Typescript Tutorial Gitlab 

What is Rapid Prototyping?
Getting Started
Prerequisites
Setup for Windows
Exploring the application
Start the application
Domain analysis
Application structure
Application Deep Dive
Tutorial - Extending existing model
Extending data model
Extending the frontend
Tutorial - Create new model
Further details

These are all the properties of the entity `Movie`. You see simple things like `title` which is of datatype `String` or `runtime` of type `Int`. But also more complicated data types like `released` that is of type `DateTime` which represents a timestamp or `genres` which is a list of `Genre` entities (is its own model).

INFO
Read the official [Prisma docs](#) to see all supported scalar data types.

Now lets start expanding by adding the following line at the end of the model:

```
model Movie {
  id      Int      @id @default(autoincrement())
  title   String
  released DateTime
  runtime Int
  genres  Genre[]
  posterURL String
  Screening Screening[]
  metascore Int
}
```

We tell Prisma, that now the model `Movie` also has a `metascore` which is of type `Int`.

Update your database

Prisma is now aware of your change, but your database still has no idea of your change. In order to update your database we let Prisma migrate your database by creating a migration script.

For this we have to enter the backend container to have access to the Prisma CLI:

[Update the Prisma schema](#)
[Update your database](#)
[API playground](#)
[Adjust test data](#)

Figure 5: Rendered Markdown pages

5.1 Structure

To keep the tutorial interesting and engaging we split up the tutorials into smaller tasks that are linked together to achieve a certain goal. In Figure 6 you can see a screenshot of the tutorial website showcasing the different chapters of the tutorials and how they are linked together. In the bottom half of the figure you can see that you can directly jump to the previous or next step of the tutorials allowing a guided reading in the intended order.

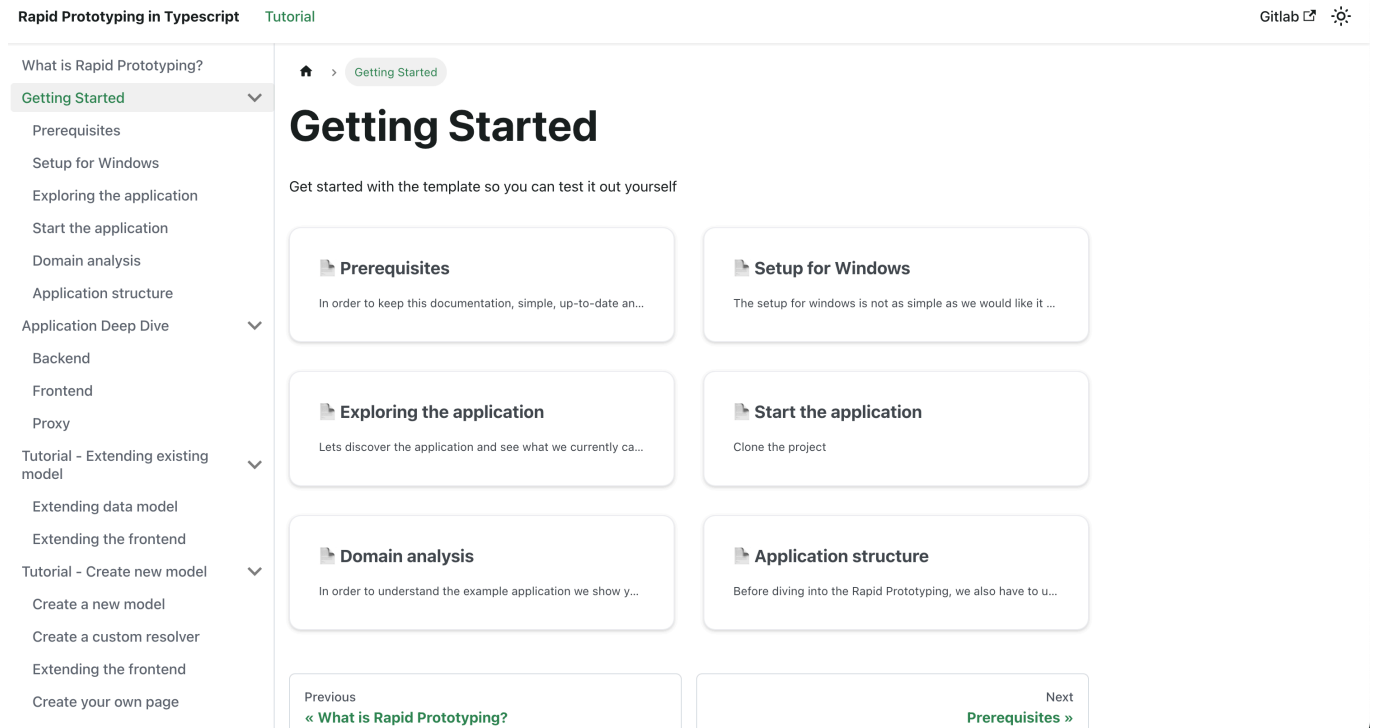


Figure 6: Getting started tutorials

5.1.1 Getting started

The first guide explains how he, the developer, can get started with this tutorial. This includes prerequisites on both the software side, meaning software that the developer requires to be installed on his machine, and on the knowledge side, meaning what the developer should already be familiar with in order to understand this tutorial. We do not list every prerequisites like “should be familiar with the terminal” our target audience will already fulfill these and would only clutter our already sizeable prerequisites scaring potential developers away.

To allow the developer to confirm his knowledge is suffice for the tutorial, we list important topics that we do not explain inside our tutorial but use for our solutions.

It is important to explain how the developer can start the application locally in order to follow the tutorials. Our setup sadly is not perfect. The watchers that try to recompile the project whenever you perform changes in your code do not always register all the changes. This will lead to errors that can be fixed by just triggering the watchers again or restarting them. We made sure to also explain these errors right at the beginning when trying to start the application.

Before we explain the application we give a brief overview of the domain that the application resides in. In this case it's the cinema ticket reservation domain. Similar to the domain analysis in this documentation we state the different domain models and their relationships. We also list the use cases that the application wants to cover. Note that not all the use cases stated there are already implemented.

Finally we give an overview of the architecture of the application so that the developer can understand what components there are and with what they interact with. To further boost the understandability of our technology

stack we added a sequence diagram to show exactly what technology or library is talking to each other and in what chronological order.

5.1.2 Application deep dive

We want to provide further details about each of the major components of our architecture, the backend, the frontend and the proxy. This will help the developer to understand the greater picture before trying to adjust it. In this section we also explain framework or library specific parts, and what adjustments can be done for a production variant.

5.1.3 Extending a data model

For our first tutorial we want a simple start into the whole tech stack. As a first task we want the developer to just add a property to an existing model. He will learn how he has to perform migrations, encounter a common error with migrations that happens when you add a column to existing data but do not provide a clear instruction how the existing data should be handled. Next he will also get his first experience in using the playground to test the API in a simple and direct matter. Lastly he will also learn how to extend the frontend by adjusting the query and the components responsible for the respective information.

5.1.4 Creating a data model

A more advanced tutorial is required to handle the whole workflow that the developer would encounter on a regular basis when he would intend to use this prototype for a project of his own. The developer will learn how to add a new model with a various relationships including many-to-many and one-to-many. Not all features can be covered with CRUD operations so we also let the developer create a custom resolver for a custom operation. He will learn how to protect and expose resources and create different advanced access policies. He will make more complicated adjustments in the frontend by writing his own mutation and lastly even creating a whole page from scratch that is protected against unauthorized access.

5.1.5 Further details

The last part of our documentation contains an FAQ (frequently asked questions) where we document common errors that you could encounter with the technologies and its limitations.

Part II
Product realization

6 Architecture

In this chapter we summarize decisions, patterns and explain the structure of the software components and how they interact with each other.

6.1 Evaluation of technologies

In this section we give an overview of the major decisions of the evaluations we did for the possible technologies and compared different competitors. The decisions are documented using the “Markdown Architectural Decision Records” (MADR) template [4].

6.1.1 Web framework

Context and problem statement

For an interactive user experience we want to use a web framework in order to build a representative example application that a developer also would use for their respective project.

Considered options

- [Angular](#)
- [Vue](#)
- [React](#)

Decision outcome

We decided to use React for our project because:

- It is a widely popular framework.
- It is well known even with newcomers in the developer community.
- Project setup is quite easy to understand and slim in comparison with e.g. Angular.
- Apollo Client (the GraphQL query client) natively supports React.
- Silvan already has extensive knowledge with the framework.

6.1.2 Component library

Context and problem statement

To save time and cut down on complexity we want to use a component library. It will help us to build a web application that is appealing without having to construct complex components.

Considered options

- [Material UI](#)
- [Ant Design](#)
- [React Bootstrap](#)

Decision outcome

We decided to use Material UI for our project because:

- It is the most popular component library in React.
- It provides sophisticated components with configurable features.
- It is based on the Google Material Design which is created by experts in the area of UI/UX design patterns and frontend development.
- It is simple to use and the documentation is easy to understand.

6.1.3 GraphQL code generation

Context and problem statement

The major focus of this proof of work is to enable rapid prototyping for a developer. In order to focus on prototyping rather than writing boilerplate code for CRUD operations we want a way to generate these GraphQL resolvers automatically from the database schema of our ORM Prisma.

Considered options

- [Postgraphile](#)
- [GraphQL-Code-Generator](#)
- [TypeGraphQL-Prisma](#)

6.1.3.0.1 Postgraphile

Postgraphile is a library that automatically generates a GraphQL server from a running database allowing CRUD operations on all tables without coding a single line.

Pros:

- Low effort is required to get a running GraphQL server.
- Postgraphile is optimized for performance.

Cons:

- Extending to manual queries and mutations requires the manual typing of the type definitions.
- Can't keep type definitions and functions in synchronization automatically, would have to be done manually.

6.1.3.0.2 GraphQL-Code-Generator

This library can generate empty resolvers out of a predefined GraphQL schema. The resolvers do not have any business logic inside them but only the typings in an empty skeleton.

Pros:

- Can generate resolvers and types out of a graphql schema.
- Has great Typescript integration.

Cons:

- No automated integration to the database.
- Not practical for CRUD operations since you would have to write a lot of types in the schema and connect them tediously with the database.

6.1.3.0.3 TypeGraphQL-Prisma:

Generates CRUD resolver from the Prisma schema in TypeGraphQL. The resolvers have full database access and can expose them to the GraphQL API.

Pros:

- Resolvers are always in sync with the database schema.
- Allows custom resolvers with the help of TypeGraphQL.

Cons:

- Heavy abstraction because the library does all the heavy lifting.
- Not a batteries included variant, requires bundling and tooling configurations

Decision outcome

We decided to use TypeGraphQL-Prisma for our project because:

- It automatically generates a complete CRUD resolver which allows complete usage of the Prisma Client via GraphQL.
- Allows custom resolvers to be created for custom processes with TypeGraphQL.
- Does not require any manual work to keep GraphQL schema and database schema in sync.

6.1.4 Build tool

Context and problem statement

We require generation from Prisma (client and resolver), compilation from TypeScript to Javascript, and bundling for Docker.

Considered options

- Dockerfile
- Native installing and generation
- [Earthly](#)

Decision outcome

We decided to use Earthly for this project because:

- Earthly is a isolated build tool running inside docker leading to reproducible builds without misconfiguration errors.
- In contrast to pure Dockerfile it does allow to save artifacts like node_modules or the generated code to the local machine to allow IDE support. This is a huge plus considering that we want to create a project that is easy and smooth to setup. It also helps us to minimize the risk of the “it runs on my machine” problem.
- It is still a simple way of writing the build process in an understandable way.

6.1.5 Movie data

Context and problem statement

In order to display movies in the application we need an example data set of possible movies including certain information about the movie like title, genre and movie poster.

Considered options

- Create a data set from scratch
- Use an external API like [OMDB](#)
- Use an existing data set from a public Github repo like [Movie-Demo](#)

Decision outcome

We decided to use an existing JSON with the data because:

- It is simple to use.
- We do not need to worry about an account for an api or any other credentials.
- It saves us a lot of time

6.1.6 API patterns

The TypeGraphQL-Prisma library generates resolvers for all CRUD operations, these are unprotected by default. This is a problem since we don't want to expose all the data to every user, we will use the [information holder resource pattern \(IHR\)](#). The IHR is designed to expose domain data entities in APIs while maintaining implementation encapsulation and supporting concurrent access. It allows structured, shareable data to be accessible across distributed systems, enabling the create, read, update, delete (CRUD), and search operations. By abstracting data manipulation into dedicated endpoints, IHR promotes consistency and integrity in systems with complex entity relationships. This approach aligns well with microservices principles, like loose coupling and independent deployability. Real-world examples include APIs for customer information, document databases, and metadata in analytics platforms. Our resolvers, that will be generated, are implementing this pattern. The reason for this is, that we have an independent resolver for every operation.

6.2 C4 model

We based our domain analysis on the C4 model that we learned in a bachelor module at OST called “Software Engineering Practices 2” [5].

6.2.1 System context model (C1) and Container diagram (C2)

As seen in Figure 7 we combined the system context diagram (C1) and the container diagram (C2) into a single one, since we have no external system interactions and this way you get a good understanding of the system.

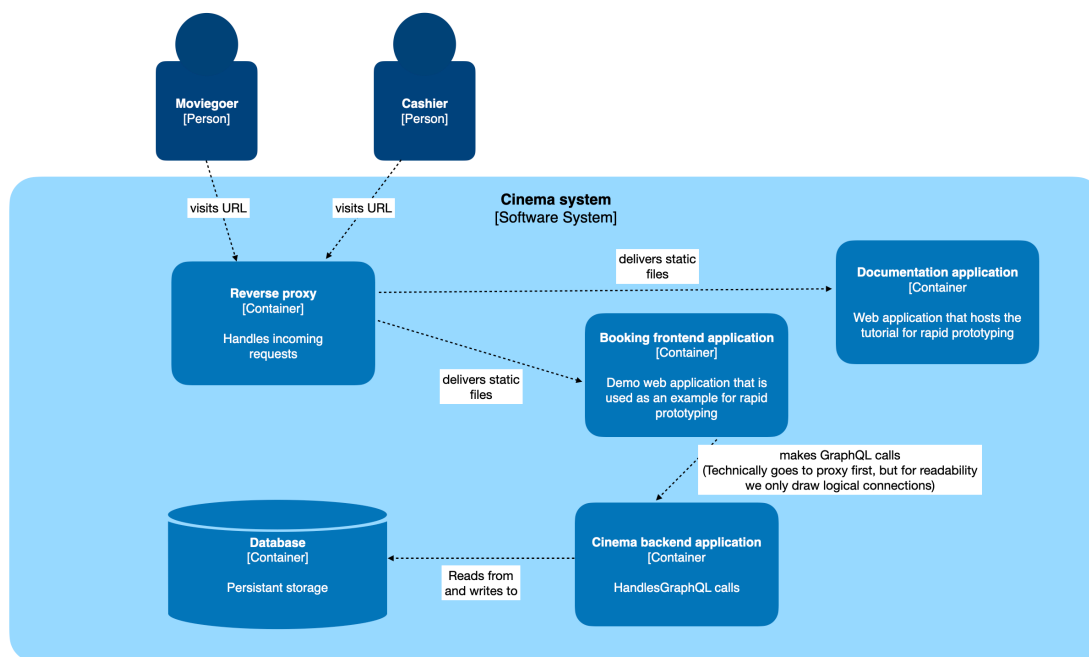


Figure 7: System context model (C1) and Container diagram (C2)

6.2.2 Component diagram (C3)

On level three (C3) as seen in Figure 8 we added a component diagram to get an overview of the API container. Since a C4 code diagram would be too detailed without any benefits we skipped it.

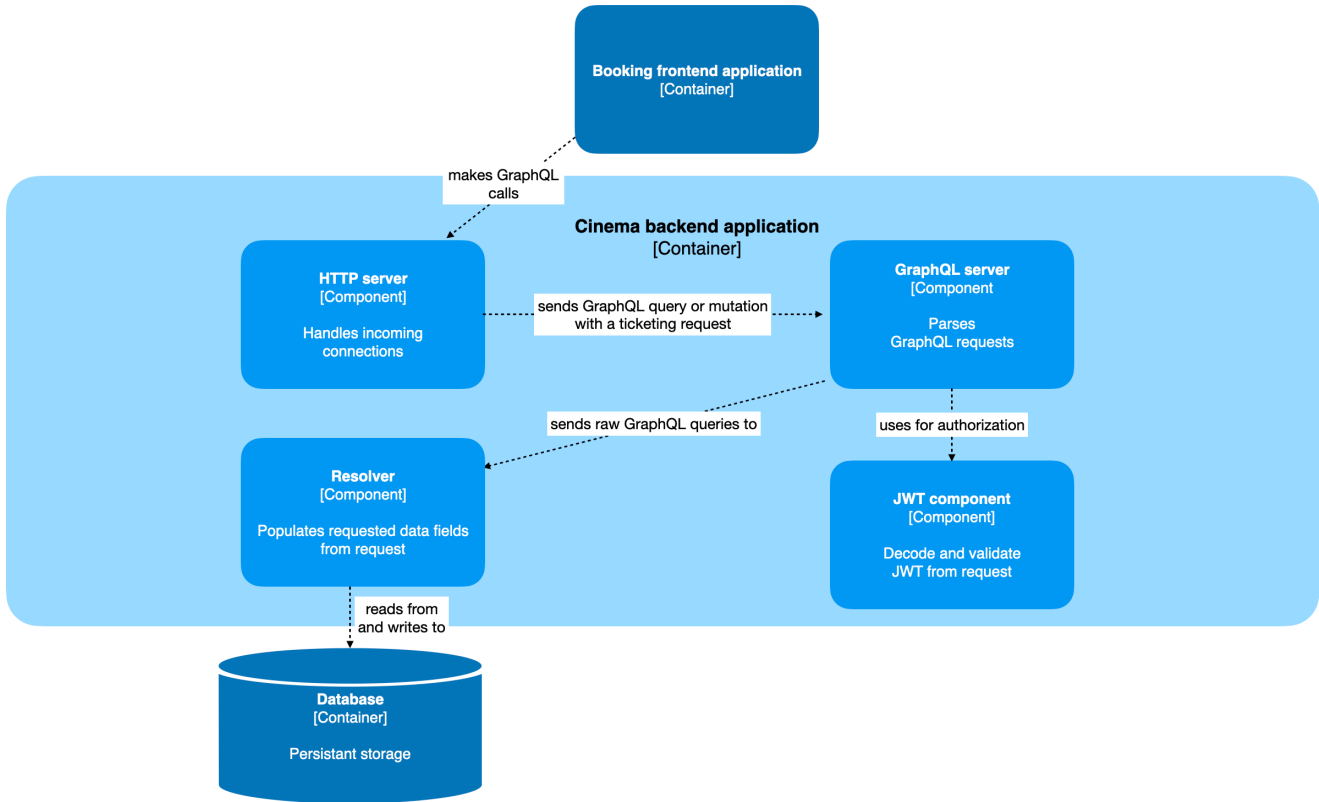


Figure 8: Component diagram C3

6.3 Sequence diagram

The following Figure 9 visualizes what happens when in the frontend a simple request is triggered until it receives the data. It also visualizes the technologies that are working with each other.

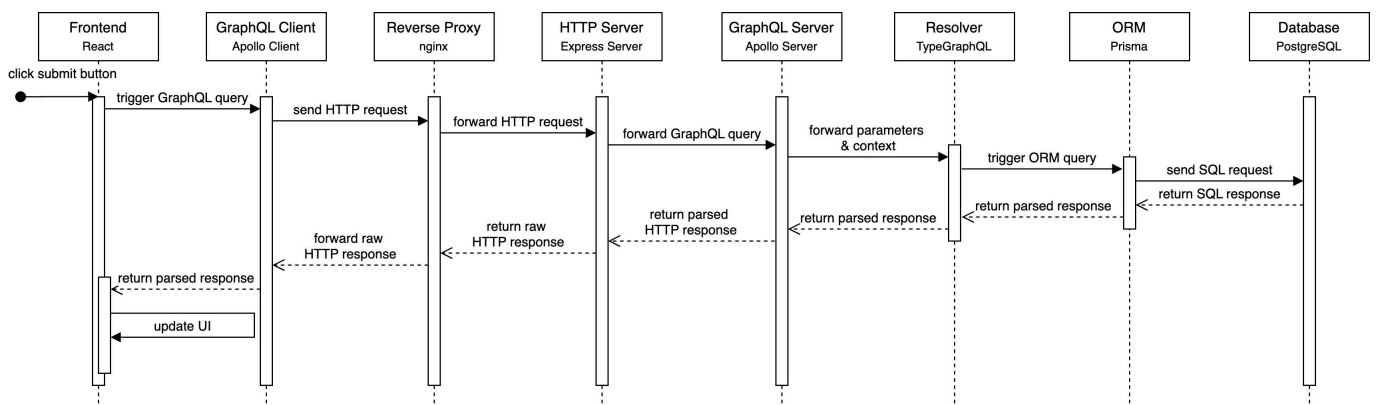


Figure 9: Sequence diagram

In the frontend a GraphQL query or mutation is formed and prepared with the required parameters. The frontend uses Apollo Client to send the GraphQL request to the GraphQL API. To reach the GraphQL API we send the request to our reverse proxy, which runs with nginx, that forwards the request to the HTTP server on our backend server. The Apollo Server, which is a GraphQL Server, uses Express as the HTTP server. The HTTP server parses incoming HTTP requests and forwards the GraphQL body to the Apollo Server. The Apollo Server then parses and

validates the GraphQL query and forwards the proper parameters to the correct resolver. The resolver is written with TypeGraphQL and is responsible to fulfill the query. It performs the business logic of the application and the associated database query. If you build a large scale application you can split up the resolver into different components so that the business logic can be separated into different parts. What approach you use is your responsibility and has to be chosen with your specific requirements and needs.

6.4 User interface sketches

In this section you can see our sketches and wire frames for our user interface (UI), we focused that the UI is self explanatory and intuitive to use so that we do not confuse the user or the developer with the example application but rather enabling them to use and extend the application with ease.



Figure 10: Landingpage sketch

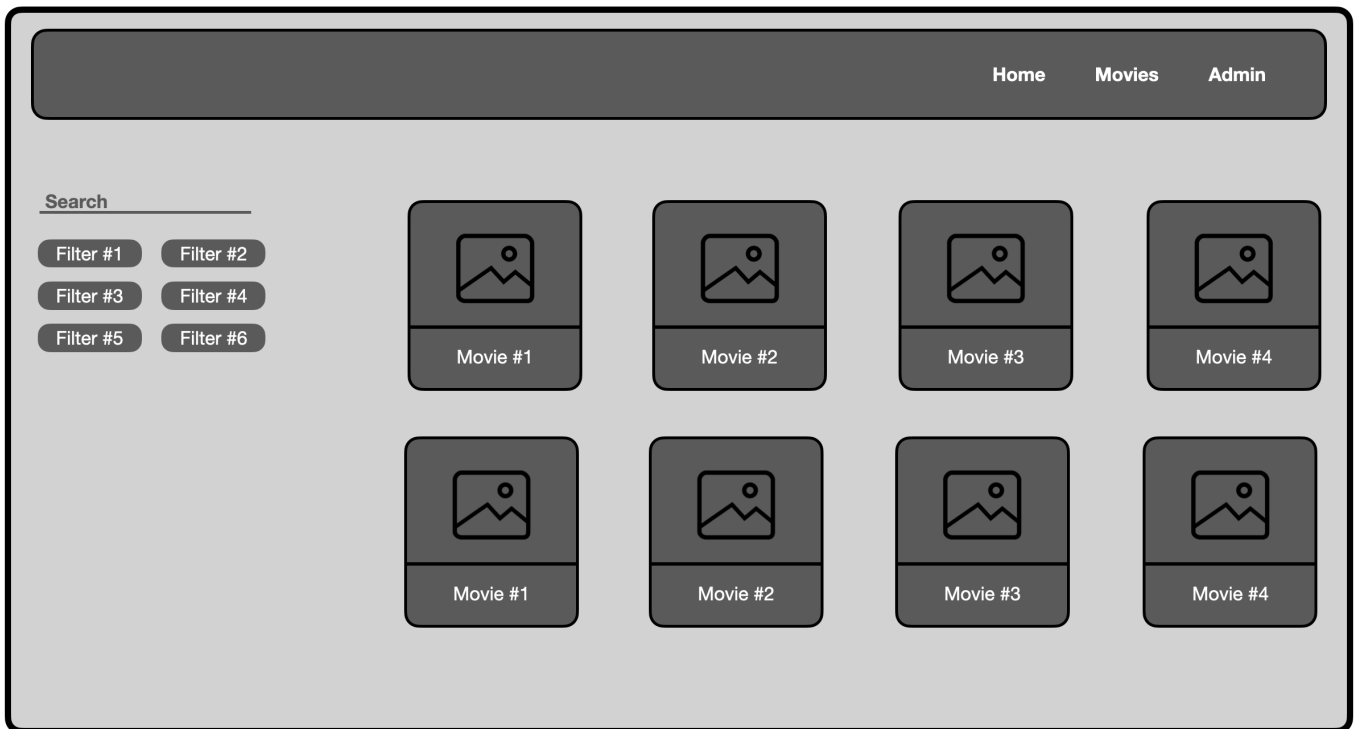


Figure 11: Movie search sketch

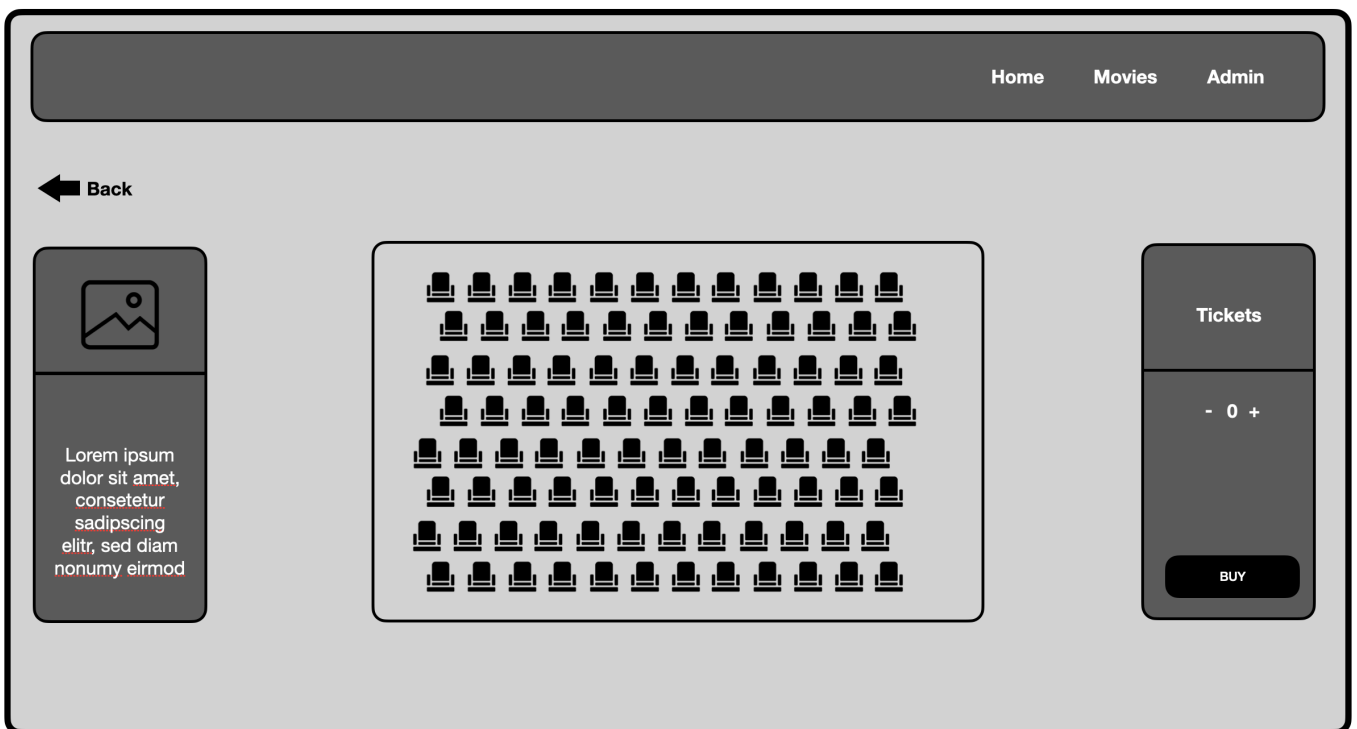


Figure 12: Ticket reservation sketch

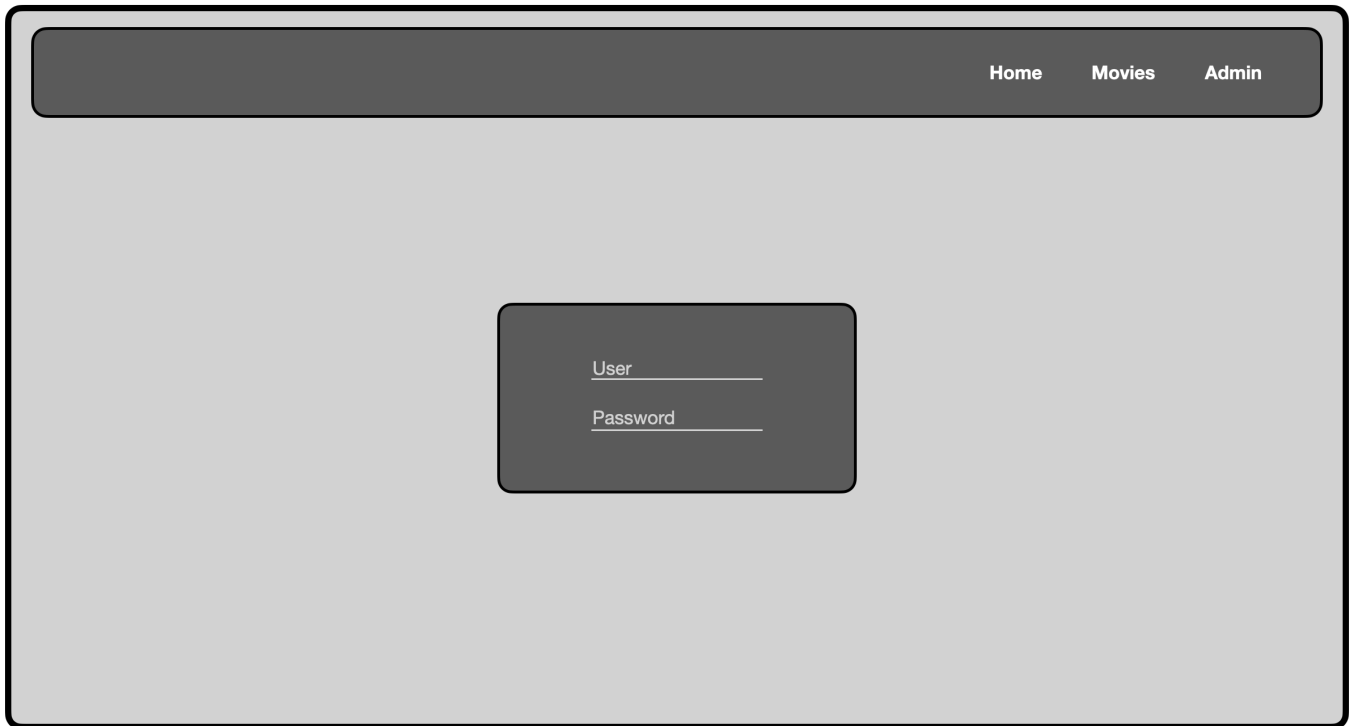


Figure 13: Login sketch

6.5 Folder structure

The following is a visualization of the folder structure of this project. Note that some files have been omitted because we do not want to explain each file but only the ones that might change and that are of significance.

```

apps/    -> Hosts all application code
├─ api/   -> Hosts the code for the GraphQL API
│  └─ Earthfile  -> Defines how to build the API container
│  └─ prisma/    -> Hosts the ORM definition
│     └─ schema.prisma  -> Single source of truth of the database
│     └─ migrations    -> Hosts all migrations scripts that were generated
│     └─ generated     -> Hosts all generated code from the Prisma schema like Prisma Client and
TypeGraphQL resolvers
│  └─ src      -> Hosts the source code of the API and any custom resolvers
│  └─ package.json  -> Defines the API application related dependencies
│  └─ tsconfig.json -> Defines the API application specific TypeScript configs
├─ documentation/ -> Hosts the code for the documentation application for the tutorial
│  └─ Earthfile  -> Defines how to build the documentation container
│  └─ docs      -> Holds the .MD files that contain the tutorial
│  └─ src      -> Hosts the code for the documentation application
│  └─ static   -> Hosts assets like images
│  └─ docusaurus.config.ts -> Configuration file for docusaurus
│  └─ sidebars.ts -> Configuration file for the order of the tutorial
│  └─ package.json  -> Define the documentation application related dependencies
│  └─ tsconfig.json -> Define the documentation application specific TypeScript configs
├─ frontend/   -> Hosts the code for the demo frontend application
│  └─ Earthfile  -> Defines how to build the frontend container
│  └─ src      -> Hosts the source code of the frontend and all its components
│  └─ codegen.ts -> Configuration file for the type generation of all GraphQL queries and
mutation responses
│  └─ package.json  -> Define the frontend application related dependencies
│  └─ tsconfig.json -> Define the frontend application specific TypeScript configs
│  └─ vite.config.ts -> Define the frontend application specific Vite configs
├─ proxy/      -> Hosts the config for the reverse proxy
│  └─ nginx.conf -> Define the reverse proxy settings
docs          -> Hosts the source code for the written documentation
gitlab-ci.yml -> Define the CI/CD pipeline
docker-compose.yml -> Defines the container orchestration
Earthfile    -> Defines how to build this entire project
eslint.config.mjs -> ESLint configs
format.sh    -> Small script to format all files
package.json -> Defines global dependencies for this project for linting and formatting
pnpm-workspace.yaml -> Defines the workspace of this monorepo
prettier.config.js -> Prettier configs

```

6.6 Project setup

For this project we opted for a monorepo to host the source code for all the different applications. We decided to do so, because the code fragments are tightly coupled to each other. This allows for rapid prototyping because the frontend code is highly aware of the structure of the api. It makes the code more unified since the formatting, linting and building can be streamlined with the same tools and configuration. For the configuration of the monorepo we opted for pnpm [6] with its workspace feature.

6.6.1 Linting and formatting

For formatting we opted for the opinionated styling of Prettier. It follows best practices on code formatting and allows less time to be spend on discussing code styles and more time on actually building the project. A formatter

alone will only help with indentation, spacing and so on, but it won't help with faulty code like unused variables or use before declarations. For these checks we need a linter. We opted for the most widely used linter in the TypeScript community ESLint.

The configuration for ESLint is based on the configuration generated by creating a new Vite project and follows the recommendation of the different rule set plugins like `typescript-eslint` or `eslint-plugin-react`. The configuration for prettier is adapted to the configuration of eslint so that they do not try to overwrite each other.

6.6.2 Containerization

We could just run the project locally, but there are several possible issues that can happen due to different machine configuration. To resolve these issues we want to run our application inside different Docker containers. This will allow us to build isolated and reproducible applications that work on all major operating systems.

After the development cycle finishes, it's common to deploy the code to a production system. The development variant and the production variant are not identical though because the development variant supports various features that make development more efficient but are not as performant as compiled production code without these development features. For that reason we also create a variant of the docker images usable for a production environment.

6.6.3 Docker orchestration

We use Docker Compose to define the multi-container application. It allows for a developer to start the project with a single command without knowing the entirety of the project. The configuration file ensures that the containers always start with the configurations and volumes that they require.

For a production variant we created a separate Docker Compose configuration that uses the production variant images and does not mount the code into the code. The configuration exists as a basis for a configuration of a production server. The configuration serves as an example and should be adapted to the requirements of the infrastructure of the production server.

6.6.4 Building

As already discussed in the [evaluation of build tools](#) we use Earthly in this project because we want reproducible builds that work independently of your machine configuration. This will allow us to reach more developers and allow them to test this project without having to spend time fixing a build that does not work on their machine.

The build process is defined as close to the code as possible, so the build configuration for the GraphQL API application is inside its folder in `/apps/api`.

Each build process also defines how to build a Docker image and what artifacts, from that build process, have to be saved to the local machine. This includes all `node_modules` folders, that contain the code dependencies, all the generated code like the Prisma Client and the generated TypeGraphQL resolvers.

The reason we have to save these artifacts locally is, that the IDE of the developer also understands the project and can provide useful recommendation and understands the types and their available properties.

Next to the development variant we also provide building instructions for a production variant. The production variant differs from the development variant because it compiles the code and does not support hot-reloading of the code like in the development variant. A challenge was the generated typings for the frontend because in the development mode the generator listens to the live backend but during a build in a pipeline it is not feasible to have a backend running. So we had to implement steps so that the schema can be generated manually and copied into the frontend container during build time so that we can support also generating the types from a static schema instead of a live backend.

6.6.5 CI/CD

We have a Gitlab CI/CD pipeline that continuously checks our source code. A merge request can only be merged when the entire pipeline succeeds. We have a job that checks that all lines conform the Prettier standard and a different one that checks if everything is satisfying the ESLint rules. Lastly whenever a tagged commit is pushed, the documentation is built into a PDF. For a production variant we also build the images in the production configuration and push them to the Gitlab container registry, so that they could be pulled from a production server.

6.7 Authentication / Authorization

As most applications we require authentication and authorization for our protected resources, this means that someone has to log in before he can access a certain resource, and for certain resources we even require him to be the corresponding owner else, he cannot view the data. For this we require some mechanism to verify if he has access to the resource or not.

By default, our TypeGraphQL-Prisma library will generate CRUD resolvers for all our data models, these CRUD resolvers are unprotected by default which allows any user (even not logged in) to perform the operation. We want to disallow that by leveraging the RLS (row level security) feature that comes with Postgres. This will allow us to create powerful policies for access restriction and enforce these securities even with custom resolvers that we might write at a later time, meaning we only have to think of our access to the model when creating the model and do not have to fear that in some later stages we forget to implement the same security restrictions on a newly created custom resolver that tries to implement some custom process.

An alternative to using RLS would be to create a wrapper around each resolver and expose the wrappers. In these wrappers you would perform your checks and then trigger the code of the generated resolvers. This is not the best solution because it requires a lot of manual work, because for a single model multiple resolvers will be generated, which all would have to be wrapped. When adding relationships you would always have to remember the access permissions of the related model and be sure to implement them accordingly and keep them in sync. So for a more robust and simpler version we opted for RLS.

6.7.1 Important note

In a real world application you should never do authentication and authorization or any identity management by yourself from scratch like we do it here in the example application. We do a simple mechanism since your company might already have a authentication/authorization mechanism and the focus point of this project is rapid prototyping rather than security. It serves as a simple and understandable mechanism for learning purposes. You should consider well vetted open source projects that are respected by the security community like [Keycloak](#) or [Zitadel](#). Make sure to self host these authentication applications rather than using their cloud to increase security even further since you do not have to give away your data.

6.7.2 Authentication

To verify if a person is who he claims to be, we implemented a simple login and register mechanism. This starts with the model of a user.

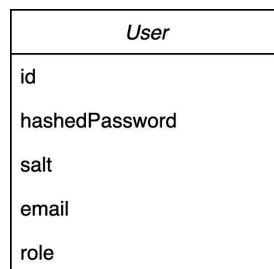


Figure 14: User data model

Before a user can log in, he has to register himself to the application. For this we created a custom resolver to resolve the incoming request. The resolver is located in the file `/apps/api/src/resolvers/auth/Register.resolver.ts`. See the following activity diagram.

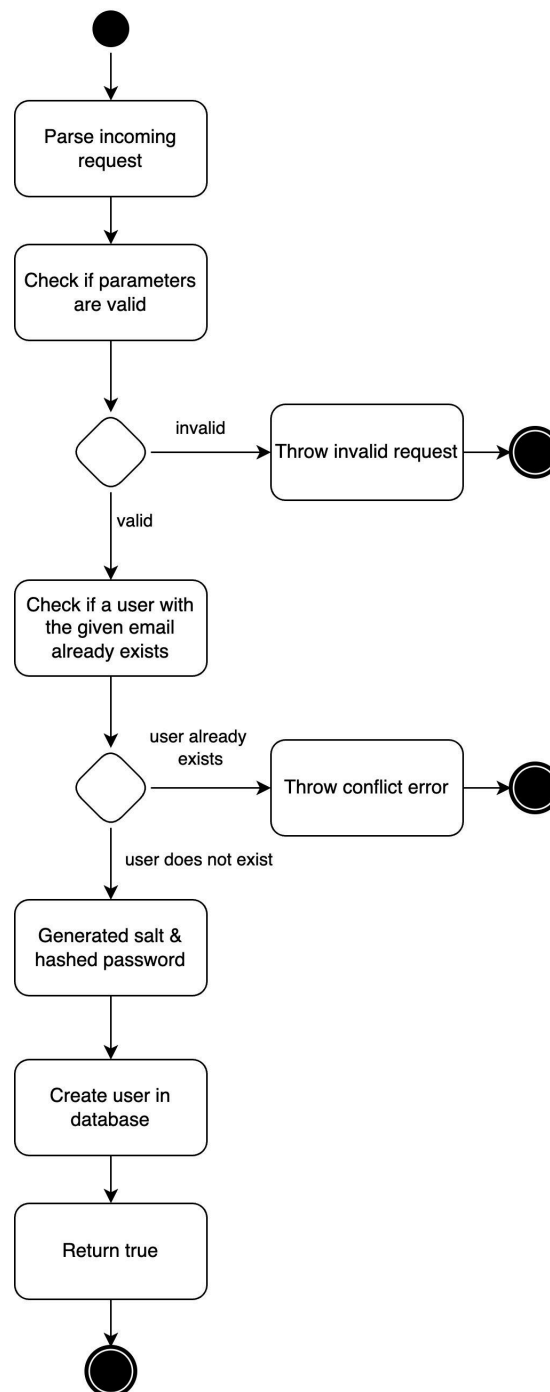


Figure 15: Register activity diagram

After the user is registered, a login can be performed. For this we also created a custom resolver to create the JWT (JSON Web Token). Note that here we return an access token without a refresh token. This is for simplicity only. In a real production environment, it would be quite annoying to be logged out after the short lived access token expires. Since the whole token management should not be done by yourself, but rather the authentication and authorization software like Keycloak, we do not show an example of how this could be done.

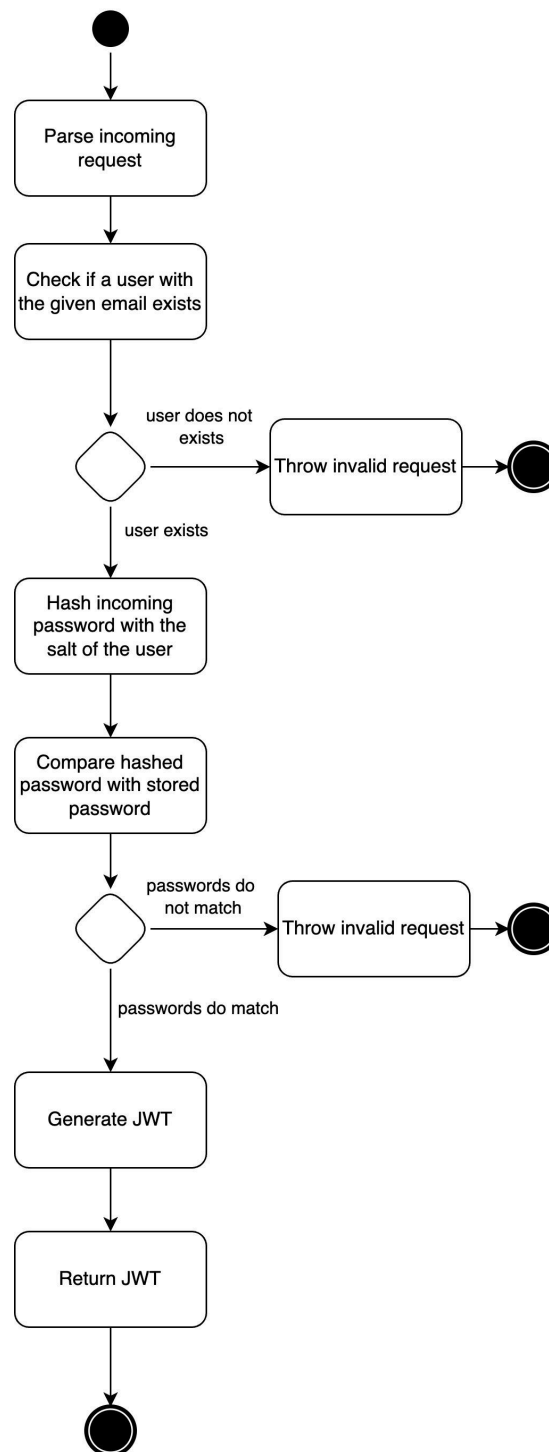


Figure 16: Login activity diagram

On the frontend the JWT is stored in the localstorage. Note that this is only for simplicity, if you would want to protect your token against malicious access from either third party library or XSS (Cross-site scripting) attacks you could consider to store the token in a secured session cookie, that is only accessible by the browser by setting the `HttpOnly` flag.

Now for every request done by the Apollo Client, the JWT will be added in a authorization header as a bearer token.

6.7.3 Authorization

Now with the JWT we have the ability to allow or deny access to certain resources. For this we leverage RLS. In order for the policies to gain access to the JWT body, we override all requests made by the Prisma Client. We set a custom config that is valid for the execution of the database transaction which contains the body of the JWT. With this, each policy can make powerful restriction based on the database row and the JWT.

```
CREATE POLICY access_to_own_transactions ON "transactions" TO api USING (  
  "user_id" = current_setting('jwt.claims.sub')  
);
```

This is a simple example policy that only allows access to a row in the fictional table `transactions`, when the value of the column `user_id` matches the one given in the JWT body.

The Prisma Client with the JWT information, that will be subject to the RLS policies, is used by default and is the only access to the database that the generated resolvers from TypeGraphQL-Prisma will have. For custom resolvers we also create a second Prisma Client instance with elevated privileges to perform certain database operations, that should still be performable by the backend but not directly through the API e.g. reading the password field of a user to validate the login.

To realize these two different Prisma Client instances we create two different PostgreSQL roles: `api` and `api_elevated`. The `api` role is used for all queries and default operations, which is subject to the policies. The `api_elevated` role will be used for the certain edge cases where we need higher access.

6.8 Error handling

Error handling in a GraphQL environment is not as straight forward as in REST. RESTful environments use HTTP status codes to communicate errors, while in GraphQL this is only used in a limited fashion. In the Apollo Server, our GraphQL server, only responds with a different status code than 200 in certain cases. [7]

- 500: Apollo Server hasn't correctly started up or is in the process of shutting down.
- 500: Apollo Server encountered an error while setting up the context.
- 500: Apollo Server encountered an unexpected error while processing the request.
- 405: Invalid HTTP method was used.
- 400: Apollo Server cannot parse the request body to a valid GraphQL document.

Because in GraphQL you can batch multiple requests into a single one, it is not always clear as to what the status code should be set.

Here is an example of multiple fictional queries being batched together:

```
query MoviesAndUsers {  
  movies {  
    title  
  }  
  users {  
    email  
  }  
}
```

If the request for `movies` is valid and data should be returned it would implicate that the status code should be 200. But the request to `users` is not allowed and a error is thrown, this would implicate that the status code should be 401/403. In the end the question remains which status code should be returned? To avoid this you have a `data` and a `errors` field in the GraphQL response where errors and response data can coexist while always using the status code 200 (expect for the above listed cases).

Note that the Apollo Client (that is sending the requests) has different error policies [8] that define how the response looks like if an error occurred in a resolver:

- **none (default):** If there was an error, the error is returned and the response data is set to undefined, even if the any resolvers return data.
- **ignore:** Errors are ignored and not returned, partial data will be returned as well as if no error occurred.
- **all:** Both error and partial results are returned.

6.8.1 Invalid JWT

We allow the user to provide a JWT, as a Bearer token, for his requests to the API. If a JWT is provided we parse and validate the token to check if it's a genuine and valid token. If not we do not process the request even when the requested source would not require a JWT. We also return the request with the status code 401 Unauthorized because the user provided invalid credentials.

6.8.2 RLS violations

Due to our RLS policies, unauthenticated or forbidden accesses to certain tables can be prohibited entirely. When still trying to access these resources the database will throw an error that is propagated to the Prisma Client and then to the GraphQL server which returns the following error:

```
"errors": [
  {
    "message": "\nInvalid `helpers_1.getPrismaFromContext)(ctx).user.findMany()` invocation
in\n/app/prisma/generated/type-graphql/resolvers/crud/User/UserCrudResolver.js:85:62\n\n 82 }\n
83 async users(ctx, info, args) {\n      84         const { _count } = (0,
helpers_1.transformInfoIntoPrismaArgs)(info);\n\n→ 85     return (0, helpers_1.getPrismaFromContext)
(ctx).user.findMany(\nError in batch request 1: Error occurred during query execution:
\nConnectorError(ConnectorError { user_facing_error: None, kind: QueryError(PostgresError { code:
\n\"42501\", message: \n\"permission denied for table User\", severity: \n\"ERROR\", detail: None,
column: None, hint: None })), transient: false })",
    "locations": [
      {
        "line": 5,
        "column": 3
      }
    ],
    "path": [
      "users"
    ],
    "extensions": {
      "code": "INTERNAL_SERVER_ERROR",
    }
  }
],
```

This error is too detailed and exposes too much of our server. So we implement a check before the error is returned, to catch these types of errors and return a nicer error:

```
"errors": [
  {
    "message": "insufficient credentials for requested sources provided",
    "extensions": {
      "code": "FORBIDDEN"
    }
  }
],
```

6.8.3 Client side error handling

On the client side you can either use operation error level handling, meaning that where you declare the operation you also handle possible errors:

```
const { loading, error, data } = useQuery(GET_MOVIES);  
  
if (error) return <p>Error occurred {error.message}</p>
```

Or you can handle errors on an application level, meaning that every response is checked if an error occurred and if so it will be dealt with in a common approach regardless of what operation triggered the error:

```
const errorLink = onError(({ graphqlErrors, networkError }) => {  
  if (graphqlErrors)  
    graphqlErrors.forEach(({ message }) =>  
      console.log(  
        `Error occurred: ${message}`  
      )  
    );  
  if (networkError) console.log(`Network error occurred: ${networkError}`);  
});  
...  
const client = new ApolloClient({  
  link: from([errorLink, authLink, httpLink]),  
  cache: new InMemoryCache(),  
})
```

In the demo application we use application level error handling for global errors that are not directly related with the operation. This is the Unauthorized error where the JWT is no longer valid and network errors. All operation related errors, like when a user try to register and the email was already in use, are handled on an operation level.

7 Quality measures

7.1 Test concept

We create our test concept based on the test quadrants from the agile testing methodology. The test concept is divided into the agile test quadrants (see below).

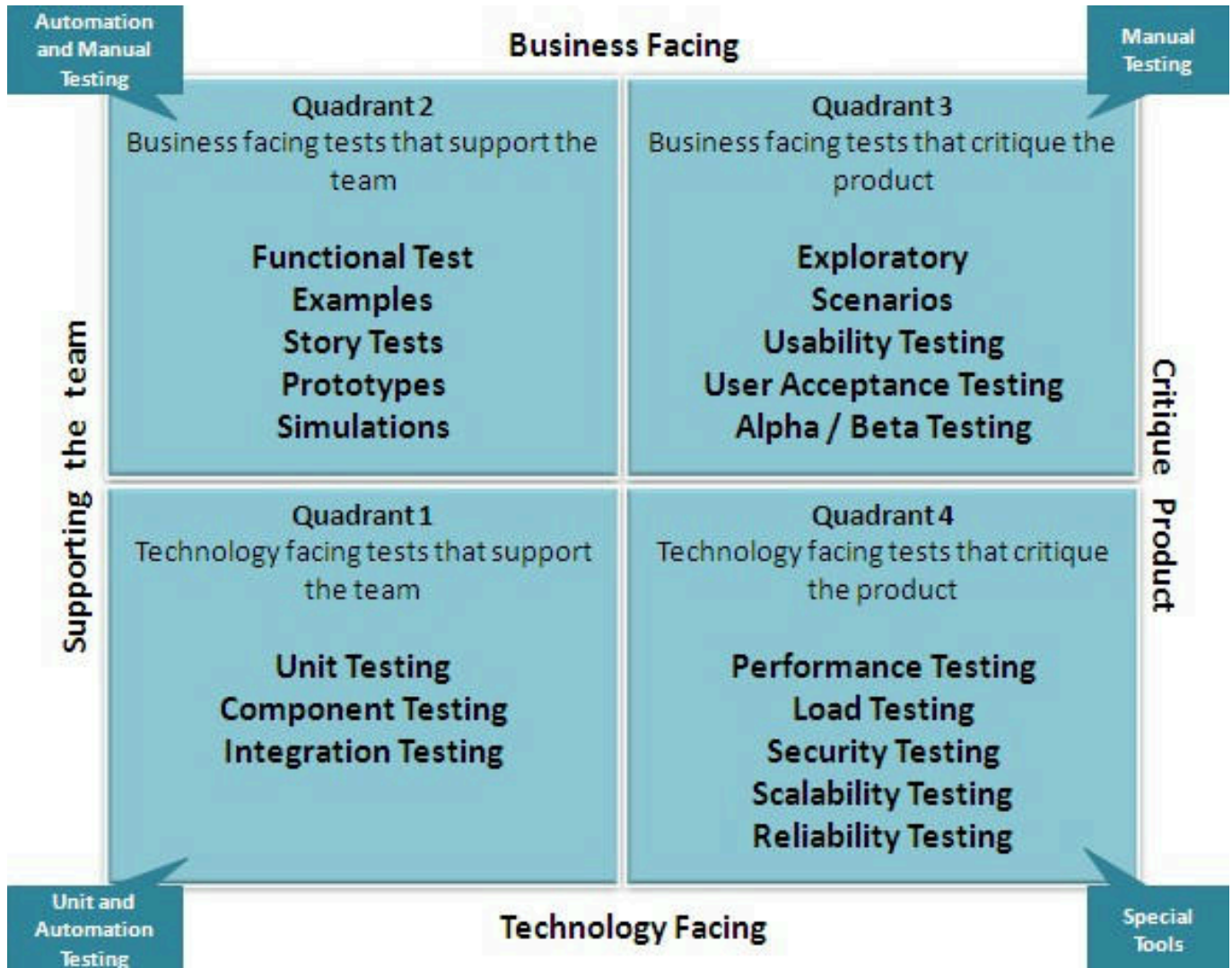


Figure 17: Agile testing quadrants [9]

7.1.1 Quadrant one

Unit tests would make sense for any product of scale, but for our project we focused on creating a useable and easy tutorial for rapid prototyping. Because of the time limit we opted to not setup a automated test suit since this would require a lot of initial setup work in order to have it runnable and working with our CI/CD pipeline. If you opt to copy our template and implement your own product we highly recommend to test all critical and breakable parts of your application. This would include all permissions (RLS), all your business logic from the backend (split them from the resolver to make them testable), and any other component critical for a smooth operation. As a testing library we would suggest Jest and for a mocking database [test containers](#). If you intent to have end-to-end testing we would recommend a sophisticated testing framework like [Cypress](#) or [Playwright](#).

7.1.2 Quadrant two

The second quadrant should assure that we are testing the code and the application continuously during the process, not just at the end. We will do manual integration and system tests to ensure that all parts of our system works correctly together. We will conduct the tests after every construction sprint and while developing when needed. See [chapter Test protocol](#) for the detailed manual tests. In the [appendix](#) is a test protocol overview with our manual and user tests that we have conducted.

7.1.3 Quadrant three

The third quadrant focuses on exploratory and usability testing. Since our proof of work for rapid prototyping should be easy to use by different developers we will do usability testing with external developers. This will give us feedback on how easy our app is to use and adapt with the given tutorial.

7.1.4 Quadrant four

As our project is too small for load and performance testing, we decided not to plan such tests.

7.1.5 Level of testing

Our main focus lies on manual user testing. With these kinds of tests, we will get the most benefit. Because real developers can give us the best feedback about the usability and complexity of our application. Despite that, as described, we will do (manual) integration tests to see the individual components working together.

7.1.6 Time of testing

- Manual unit tests: after every construction sprint.
- Integration test: after every construction sprint.
- User tests: after every construction sprint, maybe even during a construction sprint if needed. This will only affect the demo app and not the rapid prototyping.
- Testing of the rapid prototyping: We won't have a testable version of the rapid prototyping after every sprint. So we will start testing it after the second construction sprint is completed and then after each sprint
- Further functional tests: In the last regular construction sprint (construction sprint 3).

7.1.7 Summary

Our main testing strategy will be manual and user tests. The reason for that is the nature of our project. A big part is the understandability and simplicity. External developers should be able to use and adapt our application without too big of an effort.

7.2 Testing protocols

7.2.1 Manual tests

To assure that the tests will be executed the same way every time we wrote a testing protocol that refers to our NFRs.

7.2.1.1 Browser independent functionality (NFR1)

Description In this test the compatibility with different browsers is tested.

Preconditions The app is running on your local machine. The app should have been started as described in the README. The browsers Chrome, FireFox and Safari are installed.

Test steps

1. Open the app in the one of the above mentioned browsers.
2. Check all currently available movies.
3. Make a ticket reservation for a movie.
4. Repeat the last two tests for the remaining browsers. Make sure to make a reservation on a different movie.

Expected result The used account has three reservations for different movies. It is possible to check all the available movies in all of the browsers. There should not be a major difference in behavior during the process.

7.2.1.2 Operating system independent functionality (NFR2)

Description In this test the compatibility with different operating systems is tested. The main goal is to test if the app runs without any major disruptions.

Preconditions The app is cloned in a directory of your choice. If possible a VM with the operating systems MacOS, Ubuntu Linux and Windows is set up. Else the test needs to be done on multiple devices. Docker should also be installed on the different operating systems.

Test steps

1. Choose one of the operating systems.
2. Let the app run as described in the README.
3. Repeat this step for the other two operating systems.

Expected result The app should start with the steps described in the README. This should be the same for all of the stated operating systems.

7.2.1.3 Simple setup with minor effort (NFR3)

Description In this test the simplicity and functionality of the setup is tested. We build the app to be started with minor effort.

Preconditions The tester has docker installed on his machine.

Test steps

1. Open the README and exactly do what's stated in the README

Expected result The app should start as described in the README. The tester does not have to execute any other commands or tasks than as described in the README. After the steps the app should be running on his localhost.

7.2.1.4 The demo app is simple and compact (NFR4)

Description In this test, we want to assure that the demo application is not too complex and therefore has only a few screens.

Preconditions The app is running.

Test steps

1. Open the app on the overview screen.
2. Click on a movie of your choice.
3. Click on a screening.

Expected result You need 3 clicks to get to the seating plan of a screening.

7.2.1.5 Responsive user interface (NFR5)

Description In this test the responsiveness of the app is tested. The app should look good on different screen sizes.

Preconditions The tester uses a browser with developer tools. The app also should be running. The user will use the screen sizes: 640px, 780px, 1024px, 1280px, 2560px.

Test steps

1. Open the app on the overview screen.
2. In the developer tools start with the screen size 640px.
3. Check the whole screen for any errors.
4. Repeat the last step with the remaining screen sizes (780px, 1024px, 1280px, 2560px).
5. Repeat the last two steps with the screens: Login screen, ticket reservation screen, search screen.

Expected result The app looks good on every browser width. There shouldn't be any elements overlapping or missing. The user also will not have to scroll in width only in height.

8 API generation tool

As a secondary objective in our semester thesis, we want to conduct an analysis to identify requirements and critical success factor for a possible future API tool which can be used to define a schema and generate a CRUD API from it. The tool should be implemented as web-application with a simple drag and drop editor to create and edit a data schema. For every model of the data schema the respective CRUD operations should be fully generated and downloadable in order to allow customization of the generated code.

For this project you could leverage Prisma with the corresponding TypeGraphQL-Prisma generator. The data schema that would be created in the drag and drop editor could correspond to the same format of a Prisma schema. From the schema with the generator TypeGraphQL-Prisma you can generate CRUD resolvers for a GraphQL API which would have to be bundled together into a Apollo Server.

8.1 Requirements

The following requirements were defined for a generator of a GraphQL API but can also be adapted to a traditional REST API with some minor adjustments. So when you read a query it corresponds to a GET request and a mutation to either a DELETE, POST or PUT.

8.1.1 R1 Create and edit a data schema

It should be possible for a user to simply create a data model with all its properties:

- model name
- primary key
 - allow combined primary key
 - allow different types of primary key: UUID or just a simple auto incremental counter
- properties
 - customizable names
 - customizable data type (string, number, date, raw bytes ...)
 - simple constraints (uniqueness, min-/max-length, regular expression pattern)
- relationships
 - support for one-to-many, many-to-one and many-to-many relationships
 - circular relationships

After creating a model, it should still be possible to edit a saved data model with all its properties.

8.1.2 R2 Verify data schema

In order to prevent failures in generating the API, simple checks should already be performed during the creation of the data schema:

- uniqueness of names of models and their respective properties
- valid relationships (not pointing to a non existing model)

8.1.3 R3 Custom resolvers

Most applications require some custom processes and not just CRUD operations, so in the generator it should be possible to create a simple skeleton for custom either a query or a mutation. Make sure that the custom resolvers are also bundled into the final resulting API. For the custom queries and mutations you should be able to define the parameters in the editor.

8.1.4 R4 Save the data schema

The web-application should also provide persistence for the data schema. A user should be able to create an account and all the data schemas that he creates should be saved to his account. This allows him to continue the editing of the data schema at a later point rather than just in one instance and then having to start again from scratch.

8.1.5 R5 Error handling

The generated API should have some basic error handling implemented so that in case of faulty use it does not return an Internal Server Error but rather the respective error code with a meaningful description.

8.1.6 R6 Allow access constraints

As an advanced requirement access constraints could be implemented as well. This would allow the user to define who has access to certain models or even certain instances. An example for model access would be: a user with the role ADMIN has access to the model `billing` but no one else. An example for instance access would be: a user has access to all the posts where he is marked as the creator but should not be able to access any posts of a different creator.

8.2 What should get generated

From the finished schema the user should be able to generate the source code for a API tool that includes the following:

- the generated CRUD resolvers and the custom resolvers
- a server that bundles the different resolvers together
- SQL code to create a database according to the data schema
- everything that is needed to run the application (e.g. docker-compose configuration, TypeScript configuration, npm dependencies list, README with the required steps to start the application)
- basic automated test setup to demonstrate how the user can automatically test the application
- configuration for clean code (e.g. ESLint configs)

The user should be able to get the code running with minimal effort and all required manual steps should be described in great detail.

Extendable or adjustable parts should be clearly marked and communicated (e.g. how to add a custom logger, how to actually populate the custom resolvers ...).

Part III
Summary

9 Summary

A short summarization of our thesis.

9.1 Overall thoughts

We see that with this approach, you can save time in developing a full-stack application. You can develop features in a hasty manner without compromising on authorization / authentication security and still have type safety across the frontend and the backend. We do acknowledge that we have many dependencies on different core libraries which is not always a good thing. Also the learning curve for the first project might be quite steep. But when trying to build an application with this template for the second or third time, you will get acquainted with the way it all works together and will give you a efficiency boost.

9.2 Alternative ways

In order to cut down on dependencies an alternative to NodeJS could be used that was released during the development of this project: Deno V2. It is an alternative Javascript runtime that has many helpful tools integrated to the system itself like testing, formatting, linting, running TypeScript directly, enhanced security features and much more. With the newly released version it claims to also be NPM compatible which would allow us to still run our NPM packages that are responsible in this template for rapid prototyping. It would be interesting to try and test this template but using Deno V2 instead of all the other custom NPM dependencies that we use for these mentioned tasks.

Part IV
Appendix

10. Test protocols

For our tests we will mainly focus on integration tests, end to end tests and user tests. All these tests will be executed manually for simplicity.

10.1. Integration test template

Since a good user experience and an easy to understand software is crucial for this project, we will focus a lot on manual user tests. For this purpose we wrote three different type of tests. First we have the integration tests, they will assure that the whole setup works properly. Second there are the End-To-End (E2E) tests. These tests will be executed by ourself to assure that the key functionalities of our software are working. The last type of tests are the user tests. An external person that has little to no knowledge of this project will do these tests. That way we will get important information about how understandable our tutorials and software are.

Prerequisites [Docker v26+](#) and [Earthly v0.8.15+](#) are installed.

Tester -

Steps

Step	Description	Result
1	The project can be cloned from our public GitHub .	
2	The project can be built with Earthly.	
3	The project can be started with docker compose.	
4	All four applications (frontend, database, api, documentation) are running.	
5	The database migrations can be applied and the data gets generated.	
6	The demo application can be opened on localhost:8080.	

Table 1: Integration test (template)

10.2. End to End testing (E2E)

This section E2E tests. These tests will be done manually.

10.2.1. Login roundtrip

Preconditions Docker compose is running. The application started and can be opened on `localhost:8080`. You shouldn't be logged in with any account.

Tester -

Steps

Step	Description	Expected result
1	Open <code>localhost:8080</code> .	The tester gets redirected to <code>localhost:8080/login</code> .
2	Click on the link: Sign up at the bottom of the login form.	The tester gets redirected to <code>localhost:8080/register</code>
3	Create a new user with the email <code>testone@user.ch</code> and the password <code>SuperSecretPassword</code>	The tester gets redirected back to <code>localhost:8080/login</code>
4	Log in with your new created account. For E-Mail you use <code>testone@user.ch</code> and for the password <code>SuperSecretPassword</code>	The tester is able to log in and is getting redirected to <code>localhost:8080/</code> .
5	On the top right click the Logout button.	The tester is not logged in anymore. The button states Login and when the tester checks the local storage, there is no JWT anymore.

Table 2: E2E Test-One

10.2.2. Genre filter on overview page

Preconditions Docker compose is running. The application started and can be opened on `localhost:8080`. You should be logged in with the E-Mail `user1@user.com` and the password `SuperSecretPassword`.

Tester -

Steps

Step	Description	Expected result
1	Open <code>localhost:8080</code> .	The Movie-Overview-Page on <code>localhost:8080</code> gets displayed. There are multiple different Movies.
2	Click on the Genre-Filter to the left of the movies. Choose the filter Documentation.	The movies They Shall Not Grow Old and Midnight Family are getting displayed. The movie Pain & Gain should not be displayed.
3	Click on the Genre-Filter to the left of the movies. Add the filter Action. Now there should be the two filters Documentation and Action.	The movies They Shall Not Grow Old, Midnight Family and Pain & Gain are getting displayed.

Table 3: E2E Test-Two

10.2.3. Detail movie view

Preconditions Docker compose is running. The application started and can be opened on localhost:8080. You should be logged in with the E-Mail user1@user.com and the password SuperSecretPassword.

Tester -

Steps

Step	Description	Expected result
1	Open localhost:8080.	The Movie-Overview-Page on localhost:8080 gets displayed. There are multiple different Movies.
2	Click on the movie The Irish Man.	The tester gets redirected to localhost:8080/details/*(The id can vary because the id gets generated randomly). Screenings with different dates are displayed on the details page.
3	Click on the Location-Filter and choose a different one.	The screenings, that are displayed, show a different time and hall than the screenings on the previous location.

Table 4: E2E Test-Three

10.3. User tests

Since our goal is to make an easy to understand application that can be extended, user tests are really important. For a measurable progress we defined test scenarios a user has to do. The goal is to get feedback from the user on where we can improve and what parts are already good as they are.

Template demo application testing

Step	Description	Goal	Feedback
Setup	Set up the whole project according to the README.md	All four applications (frontend, database, api, documentation) are running.	
Login	Open localhost:8080 and create a new login.	The user is able to log in with his new created account.	
Filter	Head to the movie overview and filter for different movies.	The user gets displayed movies in the genre he filtered for.	
Screening	Open a seating plan for a specific screening of a movie.	The user sees the seating plan of the chosen movie.	
Logout	The user logs himself out.	The user is logged out	

Table 5: Demo-App-Test-Template

Template tutorial extending existing model

Step	Description	Goal	Feedback
Prisma Schema	Update the Prisma schema according to the tutorial.	metascore is added to the movie table	
Update Database	Update the Database according to the tutorial.	The metascore is added in the database. All four apps (frontend, database, api, documentation) are running with the updated schema.	
Playground	Create a GraphQL-Query to access the information about the metascores	The movies and their metascore are displayed on the Apollo Client.	
Adjust seeding	Adjust the seeding data according to the tutorial.	The metascore is added in the seed file.	
Query	Adjust the query in the frontend according to the tutorial.	The data of the query now also contains the metascore.	
Rendered information	Adjust the rendered information and add new texts in the translation file.	The metascore is displayed in the application.	

Table 6: Rapid-Prototyping-Test-Template

Template Create new model

Step	Description	Goal	Feedback
Prisma Schema	Update the Prisma schema according to the tutorial.	Reservation and ReservationToSeat is added in the schema.	
Update Database	Update the Database according to the tutorial.	The Reservation and ReservationToSeat are added in the database. All four apps (frontend, database, api, documentation) are running with the updated schema.	
Adjust seeding	Adjust the seeding data according to the tutorial.	The Reservation and ReservationToSeat are added in the seed file.	
Playground	Create queries according to the tutorial.	Empty reservations are displayed.	
Playground with authorization	Create queries according to the tutorial.	A reservation with an id is displayed.	
Create custom resolver	Create the resolver according to the tutorial.	A custom resolver got created.	
Expose resolver	Expose the resolver according to the tutorial.	Resolver got added to schema.ts	
Playground	Execute queries according to the tutorial.	A list with OccupiedSeats is displayed.	
Extend Frontend	Adapt the frontend according to the tutorial.	Frontend got extended with selectable seats.	
Create own page	Create a new page as according to the tutorial.	New page got created and the reservations can be viewed.	

Table 7: Rapid-Prototyping-Test-Template

Figures

Figure 1: Seating plan screen of the demo application	3
Figure 2: Screen of the first tutorial	4
Figure 3: Analysis level domain model (UML class diagram)	5
Figure 4: Use case diagram	6
Figure 5: Rendered Markdown pages	10
Figure 6: Getting started tutorials	11
Figure 7: System context model (C1) and Container diagram (C2)	17
Figure 8: Component diagram C3	18
Figure 9: Sequence diagram	18
Figure 10: Landingpage sketch	19
Figure 11: Movie search sketch	20
Figure 12: Ticket reservation sketch	20
Figure 13: Login sketch	21
Figure 14: User data model	25
Figure 15: Register activity diagram	26
Figure 16: Login activity diagram	27
Figure 17: Agile testing quadrants [9]	31

Bibliography

- [1] “Quick-Start-Guide.” Accessed: Sep. 23, 2024. [Online]. Available: <https://domainstorytelling.org/quick-start-guide>
- [2] Craig Larman, *UML and Patterns - Use Cases*. pp. 63–72. Accessed: Dec. 12, 2024. [Online]. Available: https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf
- [3] “ISO/IEC 25010.” Accessed: Dec. 12, 2024. [Online]. Available: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>
- [4] “adr/madr.” Accessed: Dec. 12, 2024. [Online]. Available: <https://github.com/adr/madr>
- [5] “Software Engineering Practices 2.” Accessed: Dec. 13, 2024. [Online]. Available: https://studien.ost.ch/allModules/public/40859_M_SEP2.html
- [6] “Fast, disk space efficient package manager.” Accessed: Sep. 20, 2024. [Online]. Available: <https://pnpm.io/>
- [7] “Apollo Server - Error handling.” Accessed: Nov. 15, 2024. [Online]. Available: <https://www.apollographql.com/docs/apollo-server/data/errors#setting-http-status-code-and-headers>
- [8] “Apollo Server - Error policy.” Accessed: Nov. 15, 2024. [Online]. Available: <https://www.apollographql.com/docs/react/data/error-handling#graphql-error-policies>
- [9] “Testing quadrants.” Accessed: Dec. 12, 2024. [Online]. Available: <https://tryqa.com/what-are-test-pyramid-and-testing-quadrants-in-agile-testing-methodology/>