Introduction

Background

During the development of our application, it was necessary to clarify the architecture that was required in order to properly develop our application. During our discussions, specific concerns were raised, and these concerns were noted down so that we could properly address them.

Purpose

This report is written for the justification of the selected architecture system that our team has chosen, as well as highlight the quality attributes that our candidate architecture is able to address. Additionally, we explain the rationale behind our choice of system architecture, and elaborate on the alternatives and why we did not use them. Lastly, we highlight the different subsystems within each layer of our application, and elucidate each subsystem with its individual component diagram.

Scope

- Top-level System Architecture
- Quality Attributes
- Architecture Rationale
- Consideration of Alternative Architectural Styles
- Subsystem Design

System Architecture

Top - Level System Architecture

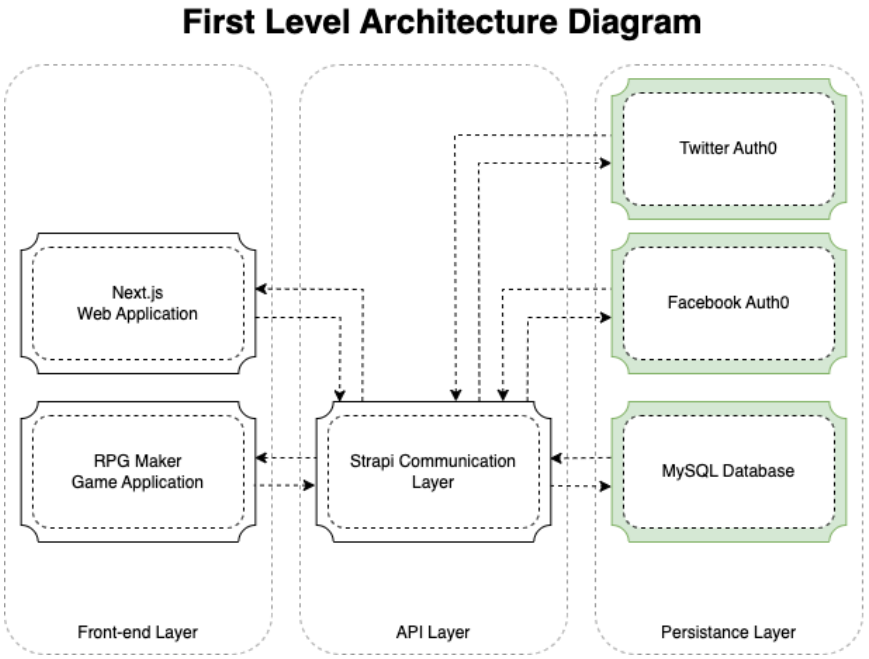

Figure 1 : First Level Architecture Diagram

Ultrareign is an educational game application platform that is composed of two key applications, the web application and game application. The web application serves as the main game selection lobby and world/level/question creation user interface, and the game application serves as the interactive and engaging component for increasing user attention and engagement.

Our overall system adopts a **layered** architecture, which is formed by three key layers:

1. Front-end layer
2. API Layer
3. Persistence Layer

Subsystems within the Front-end layer handle user interactions and interfaces. All user input is done through this layer and is translated into data, and flows to the API layer.

The subsystem in the API layer handles business logic and the create, update, read, delete operations of data. This layer serves the intermediary layer, which communicates data between the Front-end and Persistence layers.

Lastly, subsystems within the Persistence Layer handle the security and availability of data. This includes the databases and third-party authentication servers.

Each layer will be explained in detail within their respective sections.

Quality Attributes

A layered architecture addresses a number of quality attributes that we have deemed important in our software engineering process, and is therefore architecture of choice for our application design.

Maintainability is addressed as separating our entire software package into different layers and subsystems allows us to effectively isolate and work on specific functionalities. This makes our application modular, and fixes or new features can be added without disrupting other layers.

Testability is addressed as the clear distinctions between different functionalities allow us to better identify and troubleshoot portions of code.

Portability is addressed as we can host our application across different environments, by creating and replacing the subsystems within the layers, according to the requirements of each specific operating environment.

Flexibility is addressed as a layered architecture enhances the modularity of the application, and different layers and subsystems can be swapped out depending on the requirements of the device, platform or operating system.

Security is addressed as we are able to keep the Persistence layer, which contains sensitive user data, separate from the front-facing interfaces and business logic. Separating the Persistence layer allows us to keep it closed from external users, and enhances the overall security of our application.

Interoperability is addressed using a layered architecture as we are able to make changes to a layer or subsystem according to the communication protocols of the operating system, which is supported by the modularity of the architecture.

Installability is addressed as a layered architecture allows for future support for other platforms. For example, we can move from a web-based system to a mobile application.

Usability is addressed as we are able to separate the user interfaces away from the complexity of the business logic and databases, allowing us to focus on creating a good experience for users through our front-end user interface

Efficiency is addressed as a layered architecture has well-defined subsystem, with clearly established communication protocols and plug-ins created to effectively communicate between layers, which therefore enhances the efficiency of our entire application

Architecture Rationale

When conceptualising for *Ultrareign*, the main concern for our team was that we wanted to create distinction between the front-end and back-ends. Our intention for this distinction was to make sure that each member would be able to contribute to the development according to our expertise. Additionally, an added advantage is that identifying and isolating issues within our application becomes easier, and we can better troubleshoot and test the code later on in development.

As such, the logical architecture of choice in our case is a **layered** one, since it is able to fulfil our main developmental concern.

A layered architecture follows the **Call-and-Return** architectural style, meaning that the system's components are divided up into distinct layers or subsystems and grouped according to their functionalities, and components send subroutine calls between each other to pass control and data.

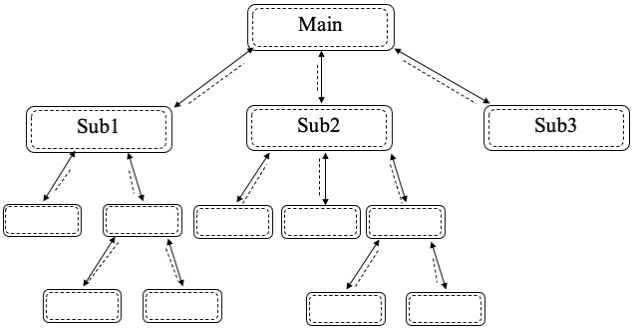

Figure 2 : First Level Architecture Diagram

Given that we wanted to use a layered architecture in our application, the Call-and-Return architectural style was the most obvious and natural style of architecture to adopt.

Consideration of Architecture Alternatives

The main alternative architectural styles consist of the Independent Component and Data Flow styles. Based on our developmental concerns, as well as to ease up the division of workload, we decided that there was a need to separate the different components of our system.

The Independent Component architecture was another strong candidate for our system's architecture. However, we felt that by grouping up similar components, we would be able to more easily divide up our work, and make the development feel more manageable.

Next, the Data Flow architectural style, which has architectures such as Batch Sequential Processing or Pipe-and-Filter, were deemed not appropriate for our developmental concerns, as it tends to lead to tight coupling of components.

Batch Sequential Processing

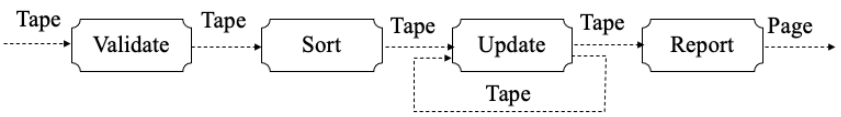

Figure 3 : First Level Architecture Diagram

Pipe-and-Filter

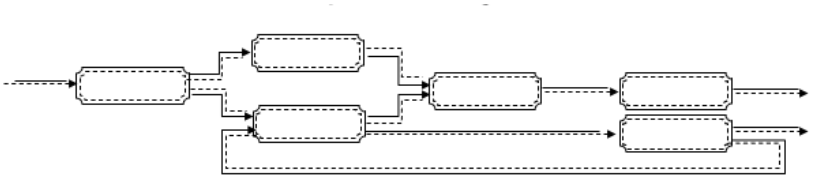

Figure 4 : First Level Architecture Diagram

Lastly, within the Call-and-Return architectural style, there are other alternative architectures such as the Data Centre Architecture. However, the requirements of the entire game application meant that we needed an architecture that had a front-end UI which could interact with other components containing the business logic and database.

Data Centre Architecture

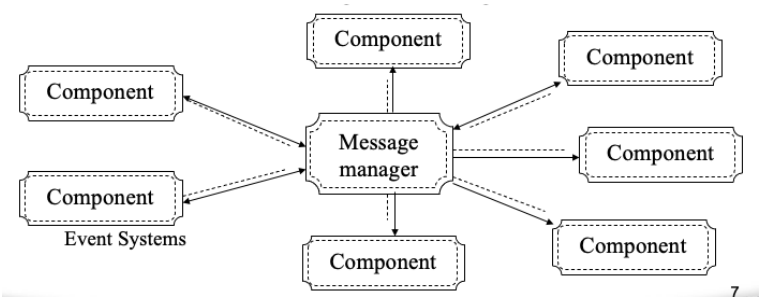

Figure 5 : First Level Architecture Diagram

Subsystem Designs

In our education game application, there are well-defined subsystems that reside within each layer.

Front-end layer:

1. Web application
2. Game Application

API layer

1. Strapi Communication Layer

Persistence Layer

1. Twitter Auth0
2. Facebook Auth0
3. MySQL Database

We will be covering the subsystem design of the Front-end and API layers through the use of component diagrams.

Web Application (Front-end Layer)

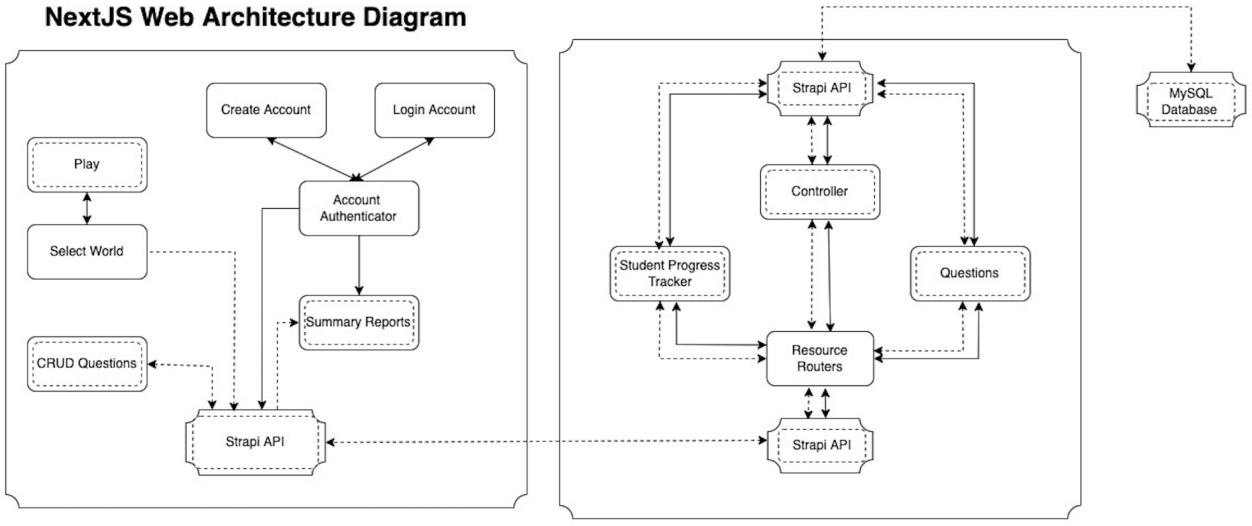

Figure 6 : Next.JS Web Architecture Diagram

Login Subsystem

The system would allow students and professors to create an account and/or login to the web interface. For creation of an account, the account authenticator would verify whether the user's username exists in the database and whether the password is valid (i.e. minimum 7 characters). When successful, the data would be stored in the database. For login to account, the account authenticator would check if the username corresponds to data in the database and checks that the password corresponds to the username. If authentication is successful, the user would be brought to the 'play' page for the student user and the 'summary report' page for the professor.

Play Subsystem

The student would select the world and level they want to play. After students have chosen, the controller would fetch the corresponding level details and bring the student into the game. Once they have completed, the student's game statistics would be stored in the database.

CRUD Questions Subsystem

The student or professor are able to perform certain actions (i.e. CRUD) to the questions in the different world and level. The controller would obtain the data from the database and present it to the user. The controller would register the action that the user wants to perform and update the database according to the action taken and final details entered.

Summary Report Subsystem

The professor can request for a summary report for all their classes or for specific classes and students. The controller would obtain the data, according to the specifications by the professor, from the database and present it in graphical format in the webpage.

Web

<Component Design>
This subsystem contains the user interface that the Students or Professors will interact with, and consists of the UI components that allow the user to interface with the back-end.

API

<Component Design>
This subsystem receives the data from both the Web UI components and MySQL database through Strapi API, and manages and controls the exchange of information between these components.

Database Communication

<Component Design>
This subsystem consists of the MySQL database, which stores the data of the game's questions

Game Application (Front-end Layer)

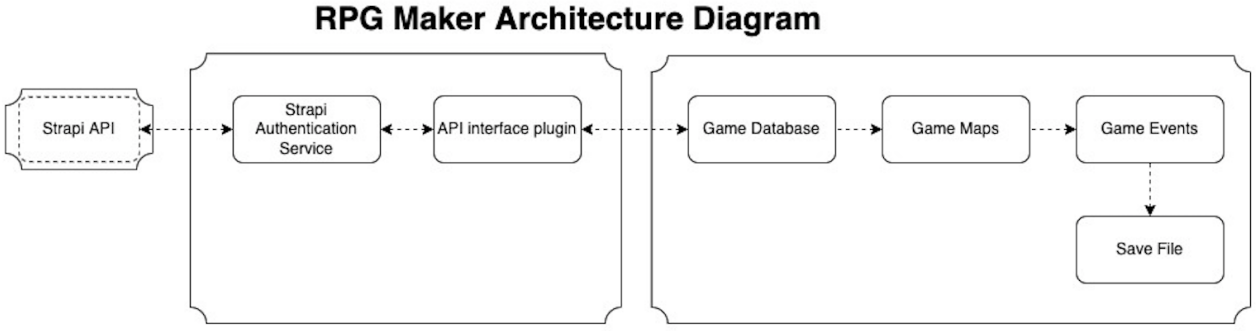

Figure 7 : RPG Maker Game Client Architecture Diagram

API

<Component Design>
This subsystem consists of an authentication service as well as an API plugin. The purpose of this subsystem is to provide communication between the online database system as well as the game software through the interface plugin.

Game Component

<Component Design>
This subsystem consists of a database, maps, events as well as a save file function. The purpose of this subsystem is to provide game flow and creation. The game database is inclusive of key items, characters and materials that will be used in the game. Game events are created and are attached to game maps to create a flow to the game.

Strapi Headless CMS (API Layer)

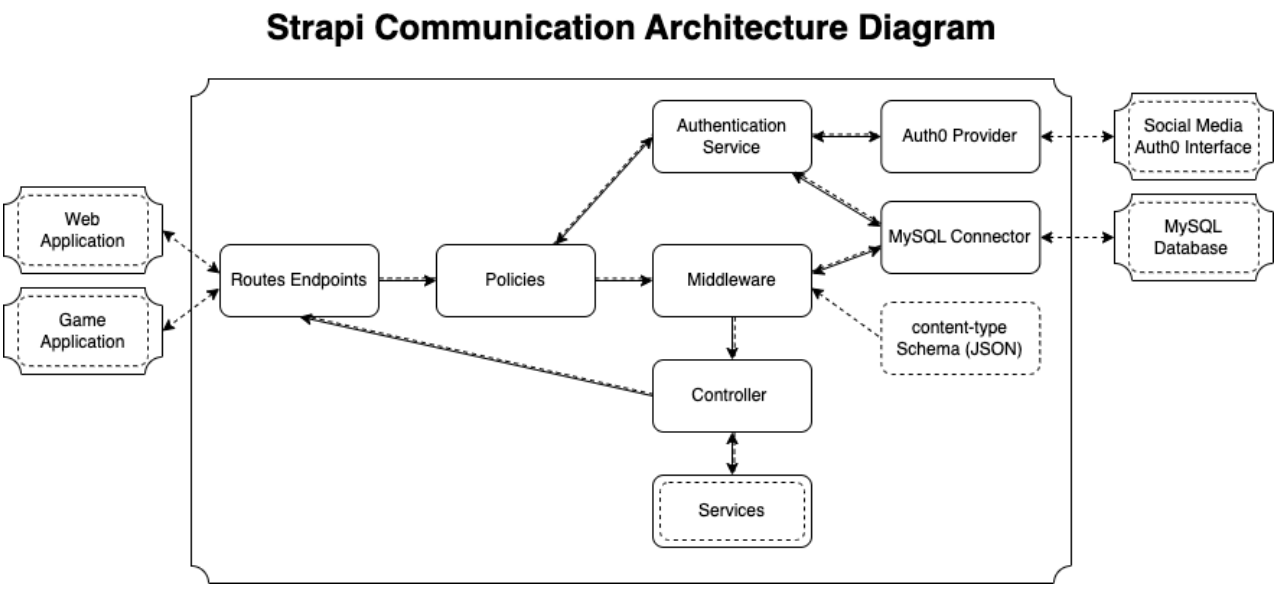

Figure 8 : Strapi Headless CMS Subsystem Architecture Diagram

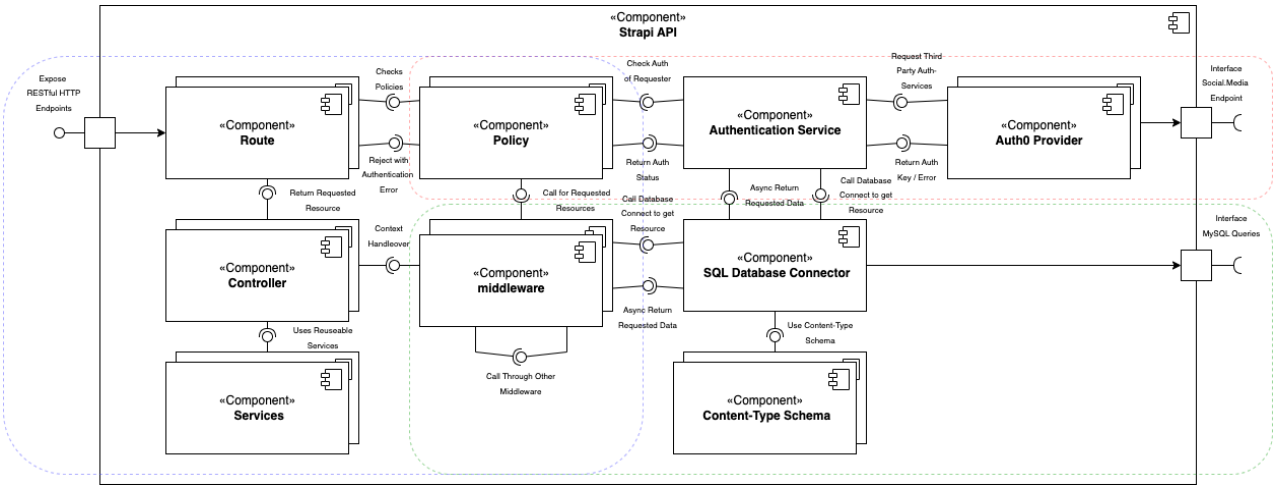

Figure 9 : Strapi Headless CMS Subsystem Component Diagram

Request-Response

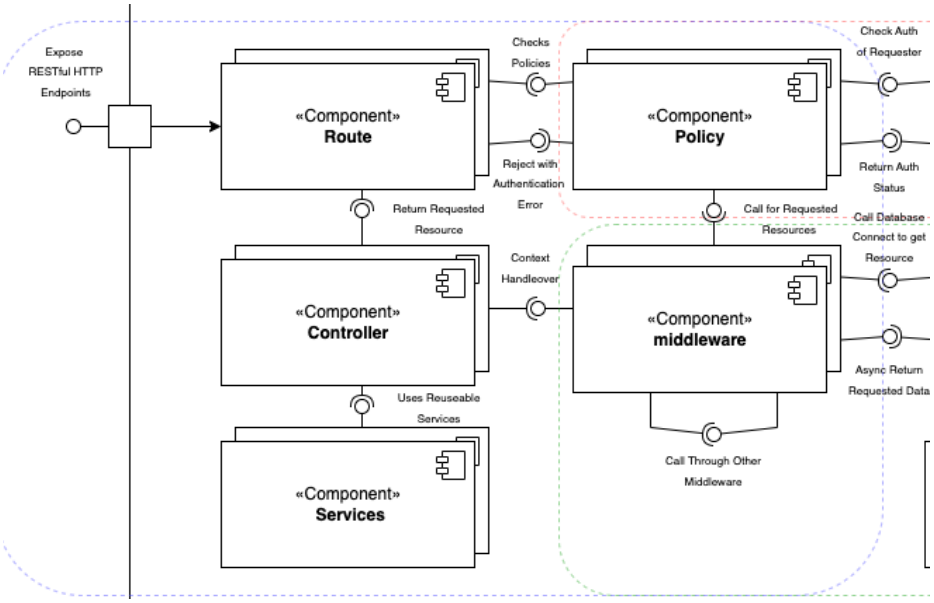

Figure 10 : Request Response Component Diagram

This subsystem consists of the Route Endpoint, Policies, Middleware, Controller and Services components. The purpose of this component system is to receive incoming HTTP requests from route endpoints, operate on data from the persistence layer and return a response.

Authentication

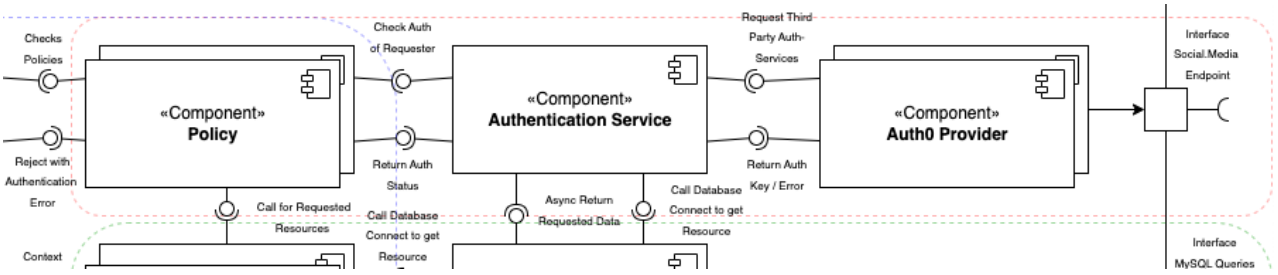

Figure 11 : Authentication Component Diagram

This subsystem consists of the Policy, Authentication Service, Ath0 Provider and Social Media Auth0 Interface components. The purpose of this component system is to take auth requests from the policies component and communication with third auth providers to authenticate users in the access and modification of data in the database.

Database Communication

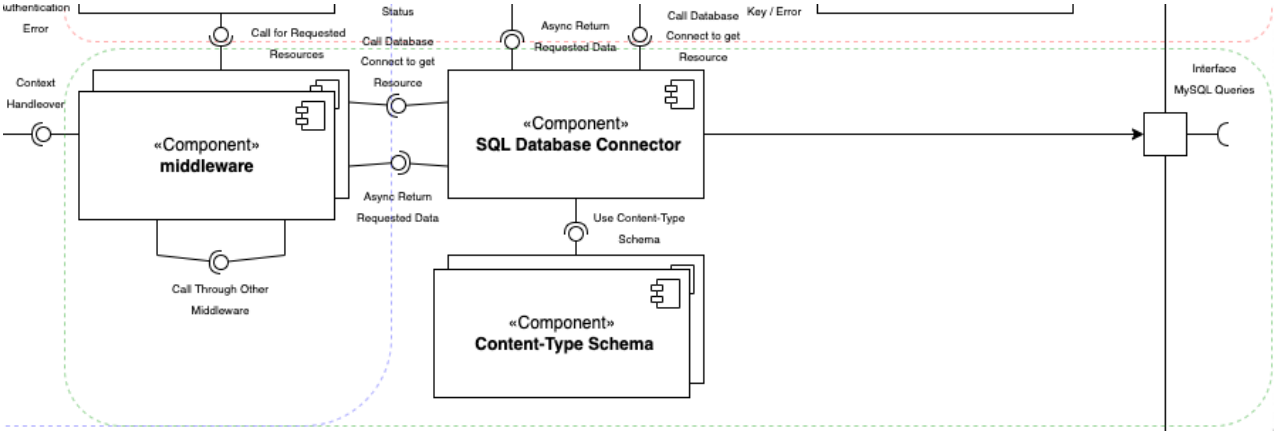

Figure 12 : Database Component Diagram

This subsystem consists of the MySQL Database, MySQL Connector and Middleware components. The Purpose of this component system is to store query persistence data onto the SQL via programmatically generated queries developed by strapi using our designed content-type Schema.