4+1 Architecture View Model

Architecture — Project-Based Course Notes

https://ieeexplore.ieee.org/abstract/document/469759: The_41_View_Model_of_architecture.pdf

---

Plant UML code repository: https://github.com/AvantSwift-Solutions/COMP30022-Documentation

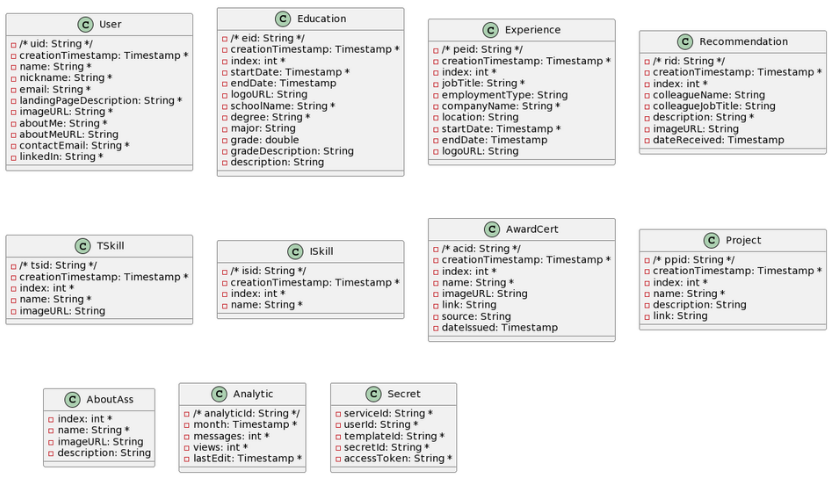
draw.io files: https://drive.google.com/drive/folders/1lYlrqnuB0qmtwOin3nWs9IlsPTgDmi44

Logical View

> "The logical view is concerned with the functionality that the system provides to end-users. Domain and class diagrams are used to represent the logical view.
>
> Examples of diagrams that can be used to support the logical model are **domain** and **database** models."

Database Schema/Collections

Note: Since this is a single user application, it is implied that all collections correspond to the same user - the client. Hence, there are no foreign key. Also, these are evolving schemas where new ones are developed depending on changing client requirements and/or technical difficulties/benefits.



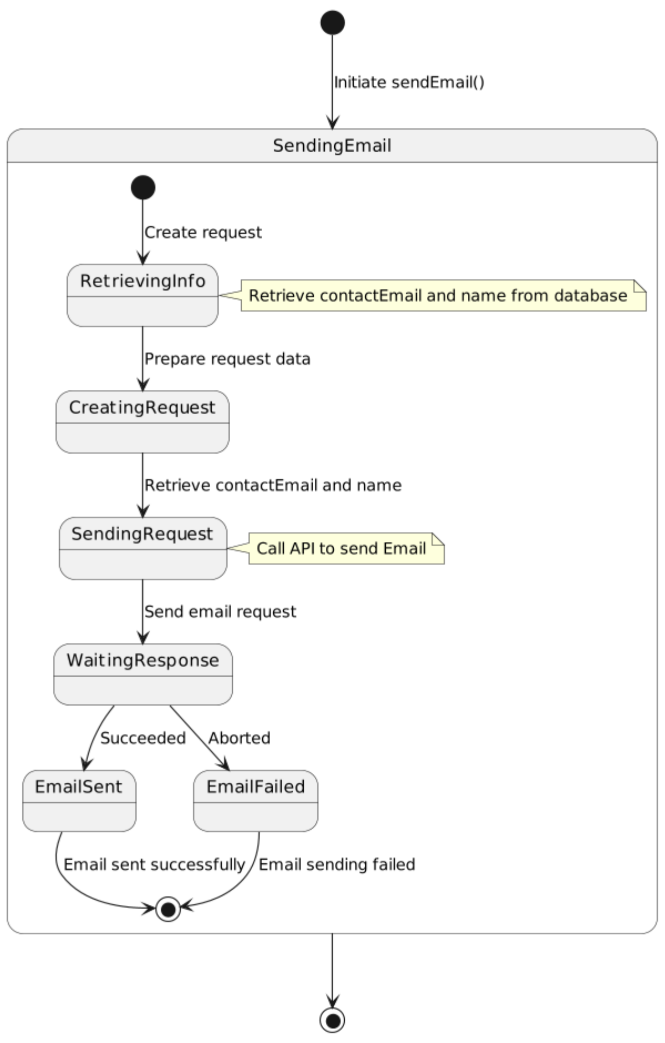
Note: * indicates a mandatory field

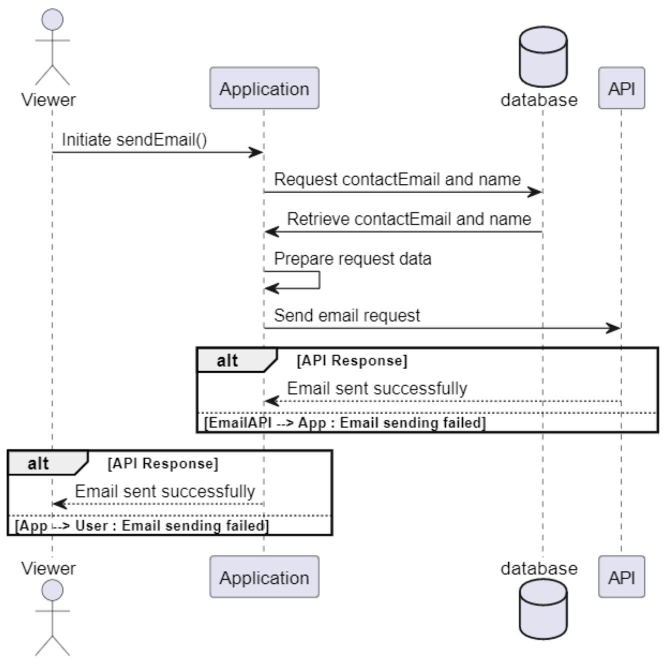
Process View

> "The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system. **Sequence state diagrams** are used to represent this view."

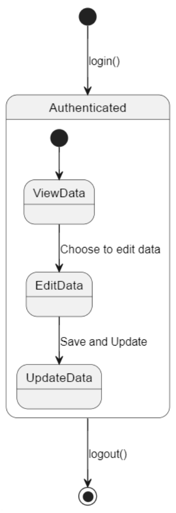
Sequence + State Diagrams

Viewer Contacting Admin via Email





Admin Authentication and Editing



See **Scenario → Sequence Diagram** for a more detailed diagram of the admin interacting with parts of the system and the Firestore database.

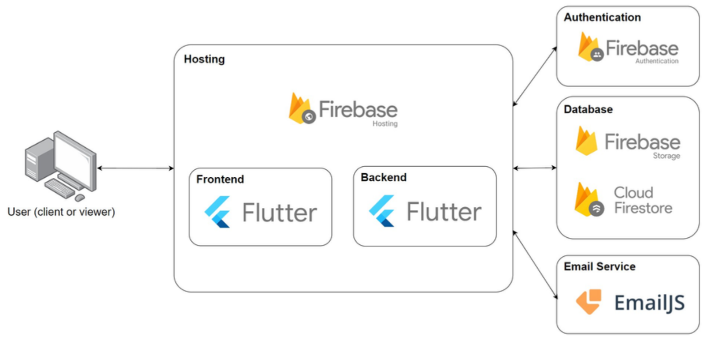
Development View

> "This view is represented by the package diagram and illustrates a system from a programmer's perspective and is concerned with software management. To demonstrate the development view, your team could describe the **architectural goals and constraints**, as well as **system diagrams**, and **API descriptions** (if any)."

Architectural Goals & Constraints

Requirement	What	Why	How	Priority
Authentication	Portfolio should be able to authenticate a single user, the client, and no one else.	Only the client should have the ability to add/edit/delete portfolio sections and information.	• Have a publicly accessible, but hidden, login page • Use Firebase authentication	HIGH
Security	Portfolio should be able to hide certain private and/or sensitive information from unauthenticated visitors.	There is some information that only the client and/or developers should have access to.	• Use authentication as discussed above • Use HTTPS and SSL certificate: chose a suitable hosting provider • Store information securely: Firebase Firestore and Storage	HIGH
Contact	Visitors should be able to contact the client securely and privately.	The portfolio should be able to convert visitors to potential clients/employers.	• Include an email form • Use an email service that has built-in security and privacy: EmailJS	HIGH
Updatability	The portfolio should be able to save text and media. The system should be able to retrieve data again later.	The client should be able to add/edit/delete portfolio sections and information.	• Use a database: Firebase Firestore and Storage	HIGH
Concurrency	The portfolio should be able to support multiple concurrent users on the web app.	Multiple visitors can browse the portfolio at the same time.	• Use a suitable hosting service: Firebase Hosting • Deploy at a publicly accessible domain	HIGH
Performance	The portfolio should load quickly even with lots of concurrent users.	Visitors have a seamless experience and leave with a positive impression.	• Use a high-performant data layer: Firebase • Store media and assets only at the required resolution • Optimize codebase and system	HIGH
Navigation	The portfolio should be easily navigated and items should be able to be searched for.	Visitors have a easy time finding relevant and interesting information.	• Good UI/UX design • Use a suitable database: Firestore	MEDIUM
Low Maintenance	The portfolio should be able to operate with minimal maintenance required.	The client does not need to spend a lot of time on the portfolio.	• Populate portfolio with default images and information • Implement intuitive admin view	MEDIUM
Future Proofing	The portfolio should be usable and fulfil the requirements even after completion of the subject	The client can still use the project as their portfolio after the semester.	• Use suitable hosting and domain as discussed above • Transfer ownership to the client after the semester	MEDIUM

System Diagram



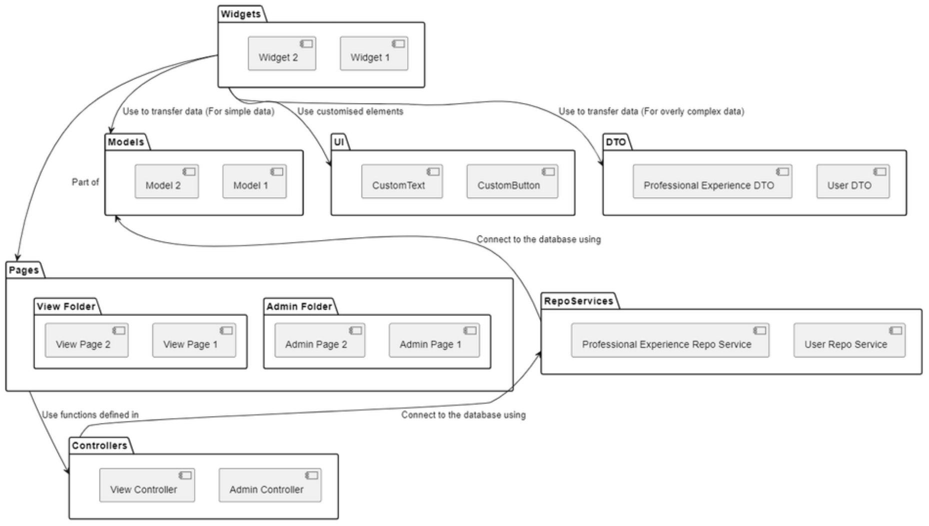
API References

- Firebase Firestore Database: Firestore | Firebase
- Firebase Storage: Cloud Storage for Firebase
- Firebase Authentication: Firebase Authentication
- Firebase Hosting: Firebase Hosting

- EmailJS (REST API): /send API | EmailJS

Note: Firestore is the database while Firebase Storage is used to store files such as images and assets.

Component Relationship



Pages: These pages are accessible by administrators for making changes and edits to the application. Broken into 2 folders, one for admin and one for view

Widgets: Widgets are modular components that can be reused across different pages. They help in breaking down complex pages into smaller parts.

Controllers: Controllers manage the functionality of the application. We have separate controllers for both admin and view pages to handle their respective logic.

Models: Models represent the structure of data within the application. They are used to maintain consistency and structure while handling data.

RepoServices: Used to communicate with the database, such as retrieving or sending complex queries.

UI: Custom UI components, such as buttons or specialized UI elements, are stored here for consistency and reusability.

Assets: All constant images and static assets are stored in this folder for easy management.

DTO: Used to transfer data across the packages/classes for complex models, to simplify the process and only transfer the information needed (e.g. landing page UI does not require full user object, but only the title and description associated with the model).

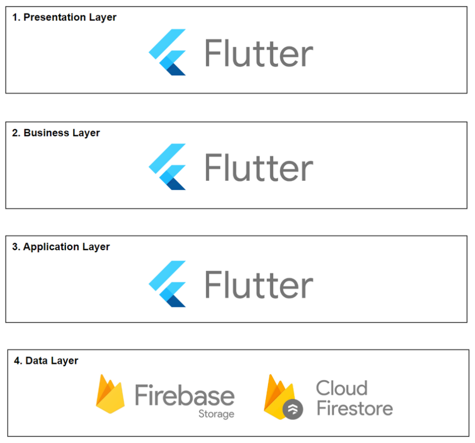
Physical View

> "The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components, and it represented using the deployment diagram.
>
> Diagrams that can support the physical view are **deployment diagrams**."

Software Architectural Pattern

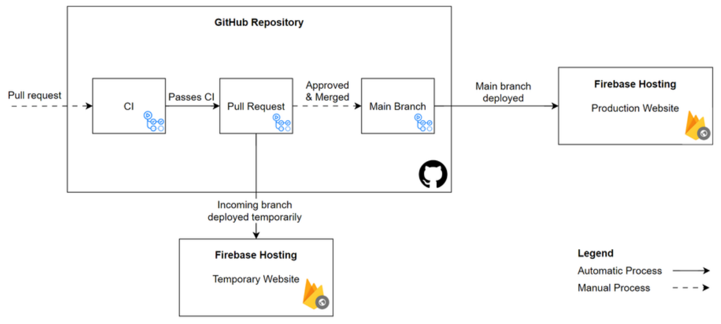
The **layered architecture pattern** was chosen for this application for the following reasons:

- The simplicity of the pattern to match the requirements of the project
- Easy to separate tasks between developers
- Our chosen framework, Flutter, works well for the whole stack
- Ability to test separate layers independent of others
- Ease of modification of separate layers, fitting of the agile methodology



We decided to use Flutter since it is a free, full-stack development framework. It also allows us to easily build a phone application along side a web app. Additionally, it integrates well with Firebase which was chosen due to the reasons outlined here: 4+1 Architecture View Model | Architectural Goals & Constraints.

Deployment Pipeline



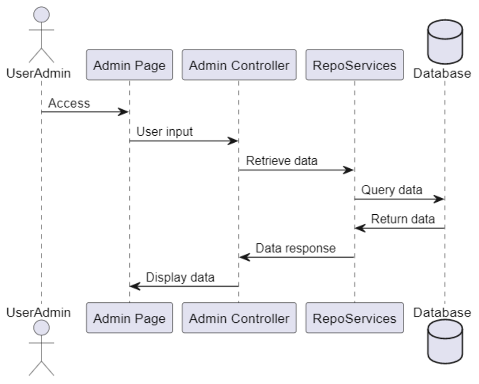
Scenario/Use Case View

> “Shows a subset of important use cases and is represented using a use case diagram. Your team should select use case(s) of architectural significance to demonstrate in the **use case descriptions and diagrams**, as well as a **sequence diagrams**.”

Sequence Diagram

The processes followed by the Admin and Viewers are similar, with their corresponding layers.

Admin Process

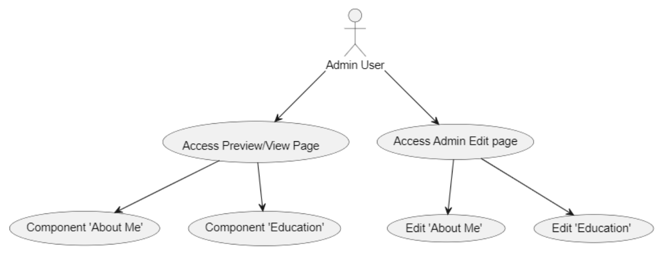


o input required

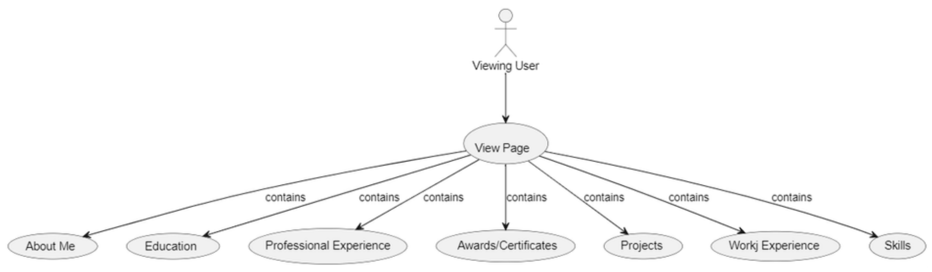
Use case Flowcharts

The manner in which a user (admin and viewer) interact with the system.

Admin use case



Viewer use case


Viewing User
View Page
contains
contains
contains
contains
contains
contains
contains
About Me
Education
Professional Experience
Awards/Certificates
Projects
Workj Experience
Skills