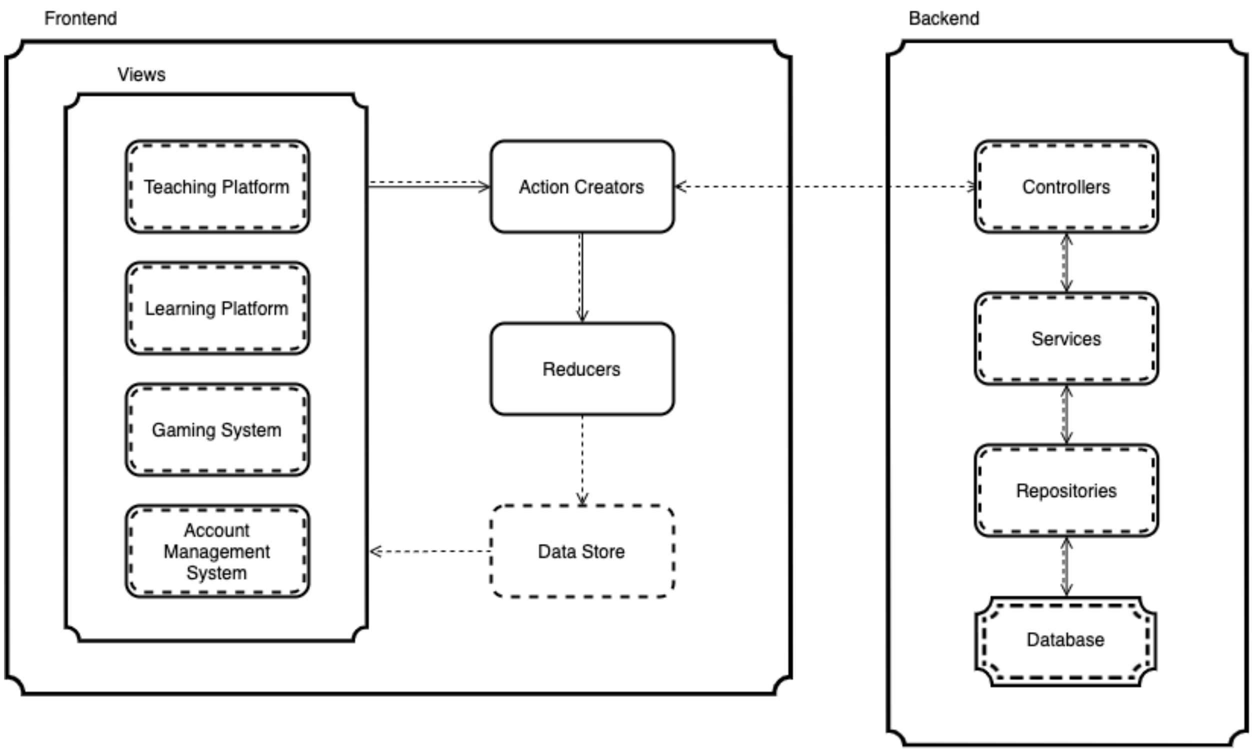
Evaluation of Architecture Alternatives

Architecture/ Attribute	**Communicating Processes**	**Redux**	**Layered System**	**Main Program and Subroutine**	**Implicit Invocation**
Reusability	+	+	+	-	+
Flexibility	+	+	+	+	+
Performance	+	-	-	+	-
Modularity	+	+	+	+	+
Maintainability	-	+	+	+	+
Integrity	-	+	+	-	-
Correctness	-	+	+	-	-

Candidate Architecture



Rationale

We decided to use a hierarchical heterogeneous style. This is to allow different architecture styles to be used for the subsystems depending on their individual needs.

At the top level, we used the communicating processes architecture because it allows us to decouple the computation into logic (backend) and presentation (frontend).

The implicit invocation architecture was also considered for the top level as it also allows for decoupling of computation. This architecture involves the frontend component subscribing to the backend for updates. However, the system requires the frontend to be able to request data from the backend as and when needed. Hence, the communicating processes architecture was deemed more appropriate.

Frontend

For the frontend component, we used the Redux architecture. The software quality attributes that are important to the frontend are correctness, modularity, flexibility and maintainability.

Correctness

The Redux architecture helps to achieve correctness by providing a single source of truth for the frontend system. The Data Store component contains the state for the entire application. As the state is immutable, every component in the frontend is able to access the same normalized data, achieving correctness throughout the frontend. To modify the states in the Data Store component, this is done by calling the Action Creators component which leverages on reducers to capture information of new states. By adhering to a unidirectional data-flow from Action Creators to Data Store, the Redux Architecture encourages data normalisation and ensures that there will not exist multiple independent copies of the same data that are unaware of each other.

Modularity

The Redux architecture is accompanied by the React framework to render the view of the frontend. This framework emphasises on the single responsibility principle by decoupling the frontend into several components. Each component only does one thing. In our project, the frontend is decomposed into 4 independent components, namely Teaching platform, Learning platform, Gaming System and Account Management System. This allows us to develop each subsystem separately for a faster development cycle.

Maintainability

The decomposition of frontend into independent components improves maintainability of code. As each component is self-contained, dependency issues are greatly reduced, allowing testing and debugging to be done more quickly. Reducer is also split up into slice reducers with each slice reducer being responsible for providing an initial value and calculating the updates to that slice of the state.

Flexibility

The redux architecture supports the flexibility of the application through the use of slice reducers. As slice reducers are functions used to create changes to a slice of an application's state, adding more slice reducers will generate a greater variety of actions performed by the application. Eventually, multiple slice reducers are combined into a single root reducer by using the CombineReducers in Redux library, allowing for greater extensibility.

Tradeoffs

The redux architecture has a low performance due to the overhead of updating data via the action creators and reducers, as opposed to updating the value directly. However, this tradeoff is made to achieve correctness as these overheads ensure that there is a single source of truth.

Alternatives

The main program and subroutine architecture was considered for the frontend component. However, it is difficult to achieve correctness because the global data is accessed directly. So, this may lead to unrelated subroutines having an impact on each other if they modify the same global data. Thus, the redux architecture was selected over the main program and subroutine architecture.

Backend

For the backend component, we used the layered architecture. The layered architecture allows us to satisfy maintainability and integrity software quality attributes which are crucial to the backend.

Maintainability

Each layer of the layered architecture has a single responsibility. This provides high cohesion and decouples each component and improves maintainability as changes made in one layer of architecture generally do not impact components in other layers.

Integrity

The layered architecture also supports the integrity quality attribute because a request must go through every layer above the database in order to access the database. Therefore, we can implement authorisation and other logic in the higher layers in order to prevent unauthorised access. Moreover, we are able to achieve abstraction by providing an API interface as the highest layer in the architecture to support frontend development. Thus, the frontend can communicate and make a request to the server without having to know the implementation details of the lower layers. This helps to accelerate the development process.

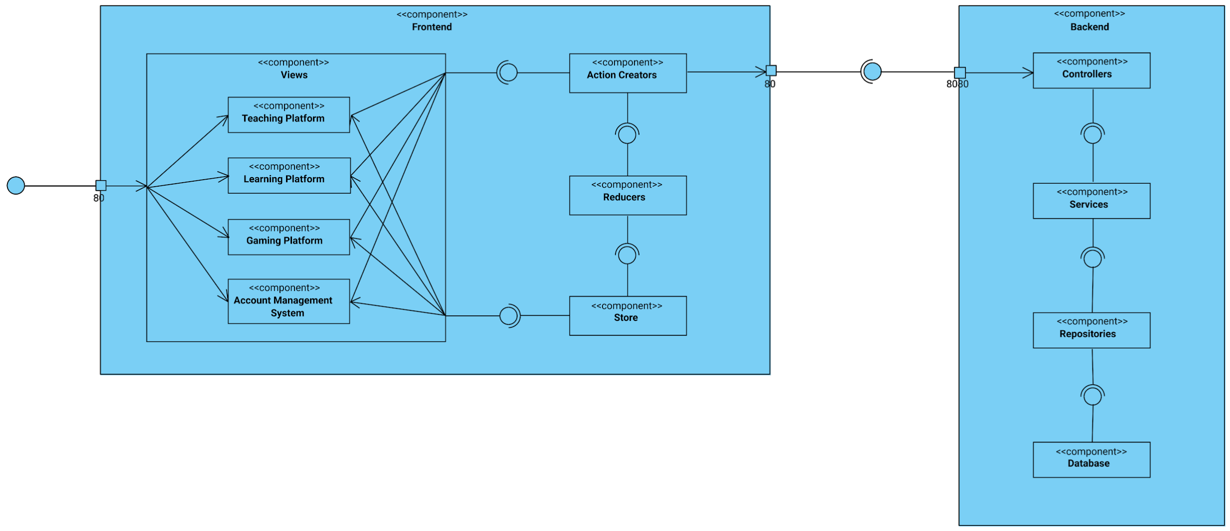
Tradeoffs

The layered architecture has a low performance due to the overhead of passing requests through multiple layers, instead of accessing the database directly. However, this tradeoff allows for more important quality attributes to be satisfied. For example, the system is more maintainable as each layer has a single responsibility, so changes made in one layer generally do not affect other layers.

Alternatives

The main program and subroutine architecture was considered for the backend. However, it is difficult to ensure integrity. Since the architecture allows subroutines to modify the global data directly, it is difficult to prevent unauthorised access. Hence, the layered architecture was chosen for the backend component.

Subsystem Interface Design



The system will be divided into frontend and backend.

The frontend communicates with the backend via HTTP requests. Components in the frontend send out HTTP requests to the backend. The intent may be to request for existing data in the database or services or to transmit user-created data to the backend. Upon receiving a request, the backend retrieves the requested data from its database and provides required service before responding to the request. Frontend-components receive responses, processes the received data, and uses it for the browser to render a viewable page.

Frontend

The frontend subsystem is responsible for the user-facing part of the system. This includes the presentation of information as well as the graphical user interface for the users to interact with the system.

The frontend subsystem will be further divided into the teaching platform, the learning platform, the account management system and the gaming system.

Teaching Platform

The teaching platform subsystem is responsible for the platform viewed by the teachers. This includes the creating, reading, updating, deleting and listing (CRUDL) of topics, levels, learning materials and questions.

Learning Platform

The learning platform subsystem is responsible for the platform viewed by the students. This includes the display of topics, levels and learning materials.

Gaming System

The gaming system is responsible for game-related activities. This includes the maze games and quizzes for students.

Account Management System

The account management subsystem is responsible for providing a platform for admin to manage user accounts. This includes the creating, reading, updating, deleting and listing (CRUDL) accounts. System administrators are also able to revoke active sessions in this subsystem. The subsystem is also responsible for user login and logout.

Backend

The backend exposes multiple endpoints to handle the requests sent by the frontend. The backend also contains authentication, authorization and business logic to validate requests sent by the front end before updating the database. This ensures that only users with the required permissions are allowed to create or edit the entities served by the backend, for example, students are not able to view other students' results. This also ensures that the data being stored in the entities before being persisted in the backend conforms to validation rules, for example, answers can only contain numbers and not characters.