Author: Team C
Date: April 6, 2023
Topic: Alternative Architecture

Alternative Frontend Architecture

We could have made use of a JavaScript frontend framework such as **React**.

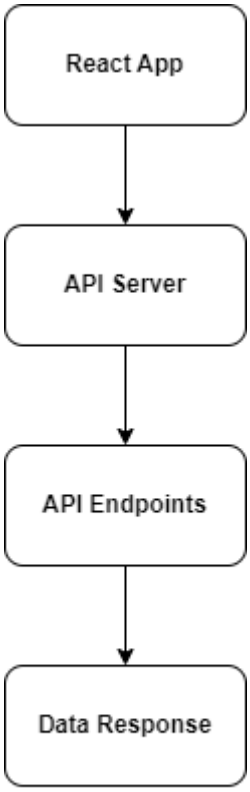
React is a very popular frontend library that allows one to build user interfaces using a component-based architecture. In this architecture, the user interface would be broken into smaller, reusable components, each with its own logic and rendering.

The main building block in React is the component. A component is a piece of code that defines a part of the user interface. Each component can have its own state and can interact with other components in the system.

React follows a unidirectional data flow architecture, which means that data flows from parent components to child components. In other words, data can only be passed down the component tree, and not back up. This helps to keep the application's data consistent and reduces the risk of bugs.

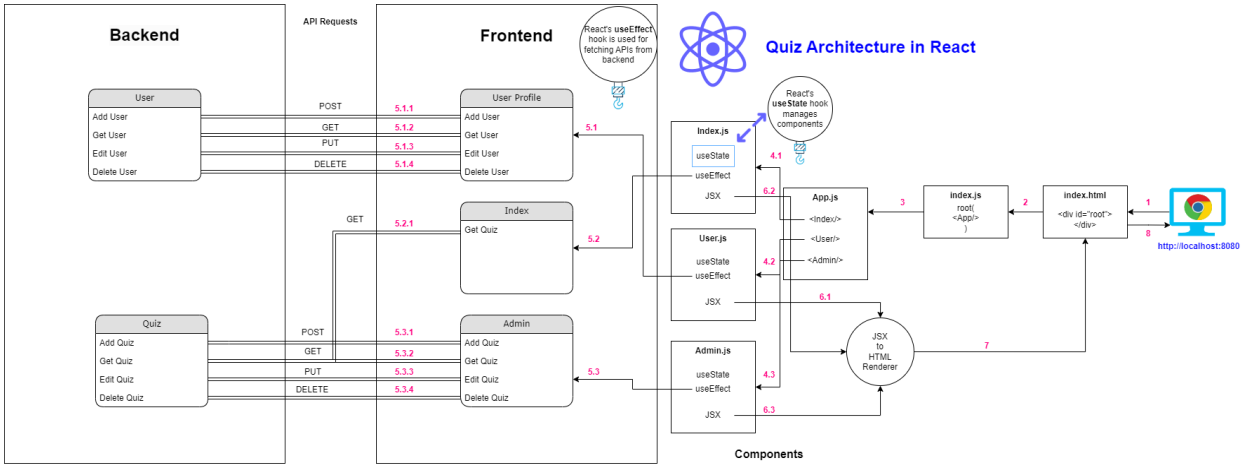
React also uses a virtual DOM (Document Object Model) to manage the user interface. The virtual DOM is a lightweight copy of the actual DOM, which React uses to track changes in the user interface. When a change occurs, React updates the virtual DOM, and then compares it with the actual DOM to identify the changes. This approach allows React to update the user interface more efficiently, resulting in faster rendering and better performance.

The use of APIs in React allows developers to build dynamic, data-driven applications that can interact with remote servers and retrieve the necessary data in real-time.



API Flow

Quiz App Frontend Architecture in React



The following architecture could have been implemented in case of using React in the Quiz App project. When the user requests the url at http://localhost:8080, the *index.html* of the react app will send the request to the *index.js* through *root* id. The *index.js* file contains the main component called *App* Component. This *App* Component is further decoupled into *Index, User and Admin* Components. The **useState** hook manages the components according to the request and thus makes the application a single page. It means that without reloading, the components can be constructed or destructed.

The **useEffect** hook will get the data from the backend to the frontend using REST APIs. Further, the template code in react is written in JSX which is then converted to HTML internally by react and rendered on the browser.

In summary, React's frontend architecture is component-based, with a unidirectional data flow and a virtual DOM for efficient rendering. Using these features, developers can build scalable, high-performance user interfaces that can adapt to changing requirements over time.

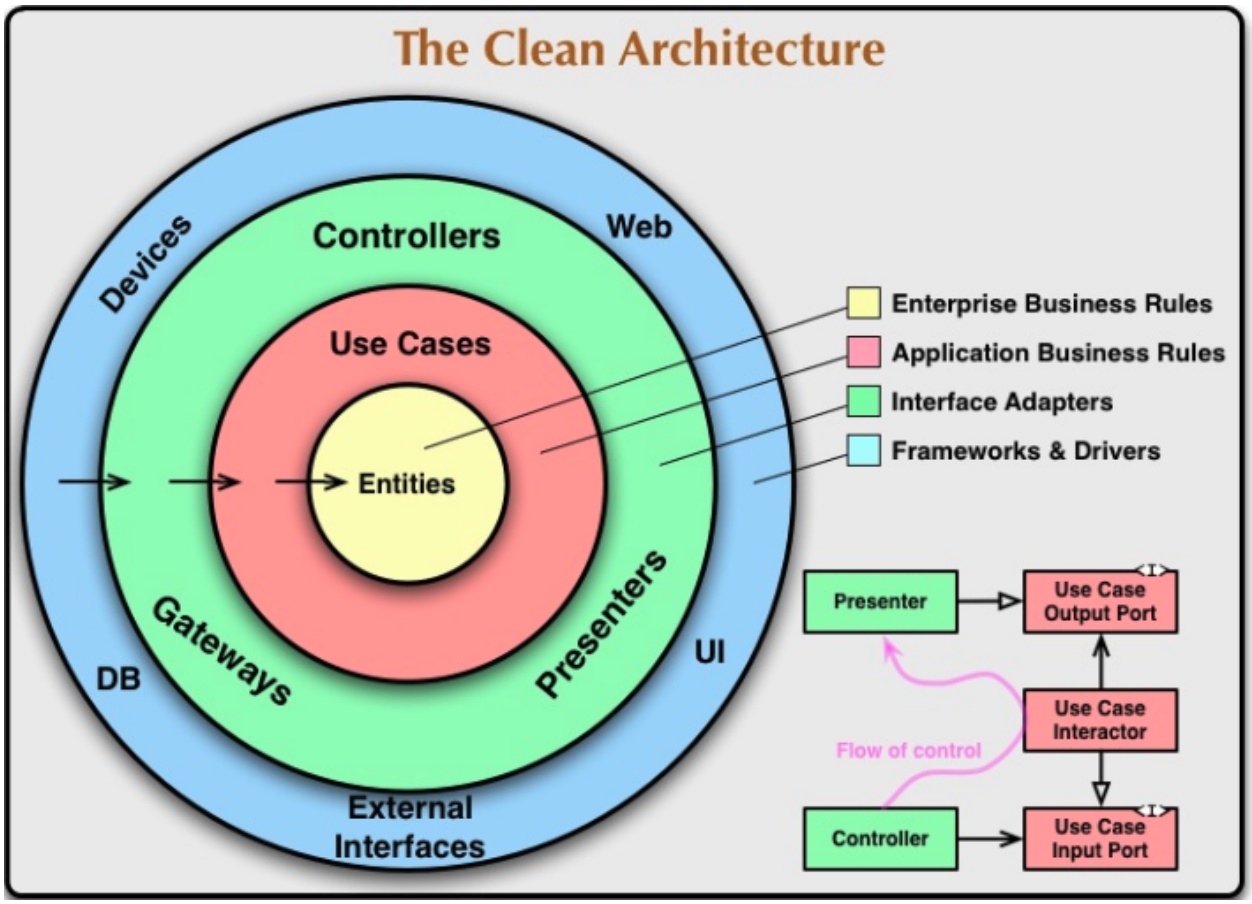
Alternative Backend Architecture

We could have made use of a JavaScript Backend framework NestJS instead of ExpressJS and rather than using MVC structure, we can use 'Clean Structure'.

NestJS is a TypeScript-based framework designed for creating high-performance and scalable server-side applications using Node.js. It leverages powerful HTTP server frameworks such as Fastify or Express while the philosophy of clean architecture involves dividing the components of a software design into concentric rings. Its main objective is to enable programmers to structure their code in a manner that isolates the business logic from the delivery mechanism, ensuring a clear separation of concerns.
The points mentioned below should be the software design principles:

- **The Single Responsibility** Principle dictates that a class should only have a single reason to undergo changes. The updated version suggests that a module must have responsibility to only one actor.
- **The Open-Closed** Principle requires that a class remains open for extension but closed for modification.
- **Liskov's Substitution** Principle states that objects should be replaceable with instances of their subtypes without affecting the program's correctness.
- **The Interface Segregation Principle** proposes that having multiple client-specific interfaces is preferable to having one general-purpose interface.
- **The Dependency Inversion Principle** advises to depend on abstractions rather than concretions.

Clean Architecture Diagram:



The Above image is a very high-level conceptual abstraction of the various layers of your application or service. It introduces class concepts like Repository, UseCase, and Presenter.

Why did we choose this framework and architecture as an alternative?
We choose NestJS because it provides the following benefits

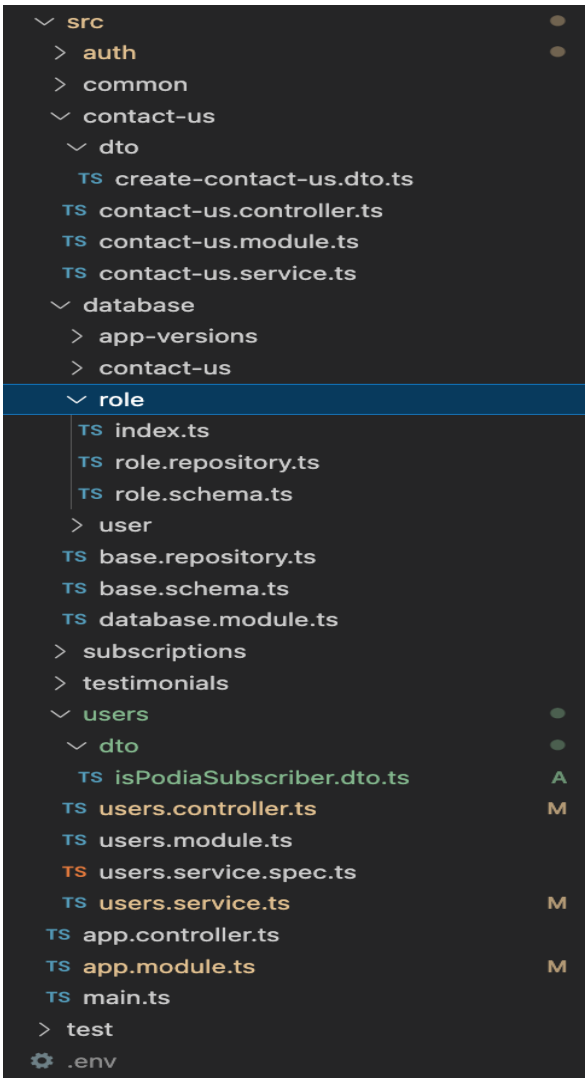
- **TypeScript Support:** NestJS is built entirely on TypeScript and offers developers the ability to write code with strong typing and type checking features.
- **Modular Architecture:** NestJS offers a modular architecture that allows developers to structure their application in a more organized and scalable way.
- **Dependency Injection:** NestJS offers a powerful dependency injection system that makes it easier to manage dependencies between different parts of the application.
- **Compatibility with Existing Libraries:** NestJS is compatible with popular Node.js libraries such as Express.js and Fastify, which makes it easy to integrate with existing systems.

- **Testing:** NestJS provides comprehensive testing utilities that make it easier to write and run tests for different parts of the application.
- **Scalability:** NestJS is designed to be highly scalable and can handle large and complex applications with ease.
- **Community Support:** NestJS has a growing community of developers who contribute to the development of the framework, provide support, and share their knowledge and expertise.

We choose Clean Architecture because it provides the solution for

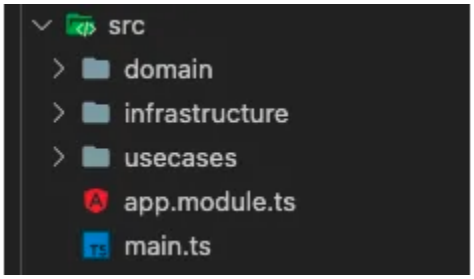
- making code resilient and simple to maintain.
- managing work distribution within a team.
- working effectively even when all specifications, databases, and other requirements are not yet available.

Example Project Structure in VS-Code:

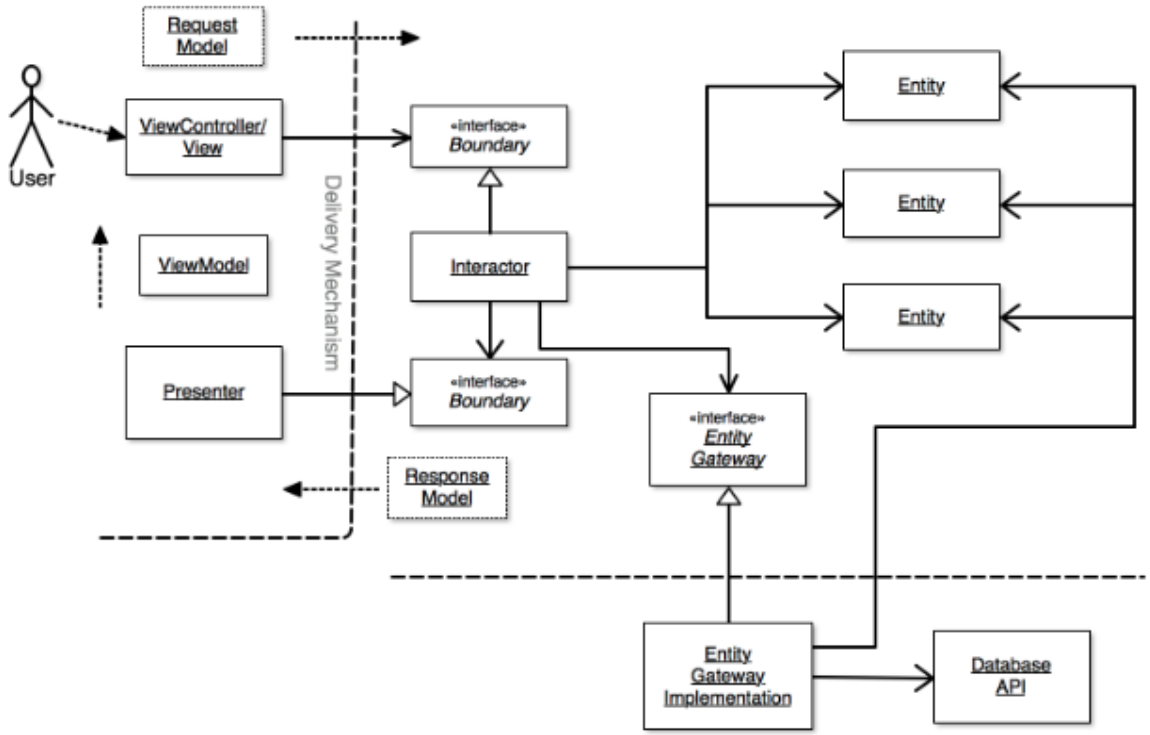


There are three primary packages in the software design: domain, use cases, and infrastructure.
Each of these packages must follow specific rules:

- **Domain package** should consist of business logic and related code and must not have any outward dependencies. It should not depend on frameworks like NestJS or any other packages like use cases or infrastructure.
- **Usecases** act as a conductor in the architecture and only depend on the domain package to execute business logic. Use Cases should not have any dependencies on infrastructure, including frameworks or NPM modules.
- **Infrastructure** package should contain all the technical details, configuration, and implementations like database, web services, NPM modules, etc. It must not contain any business logic and should have dependencies on domain, use cases, and frameworks.



Data Flow Diagram of Clean Architecture:



The Clean Architecture is composed of various components, each of which plays a specific role in the architecture. These components are:

Interactors: These represent the use cases and serve as the central objects in the Clean Architecture. For every use case, there is an interactor, named after the use case. Interactors are responsible for executing the use case by communicating with entities and gateways. Together, all the interactors contain the application-specific business logic and rules.

Entities: These represent the business objects and contain business logic that can be used in multiple applications. They are plain simple objects, such as structs in Swift, and do not inherit from frameworks like NSManagedObject or ActiveRecord. Entities do not have any knowledge of the database or network layer.

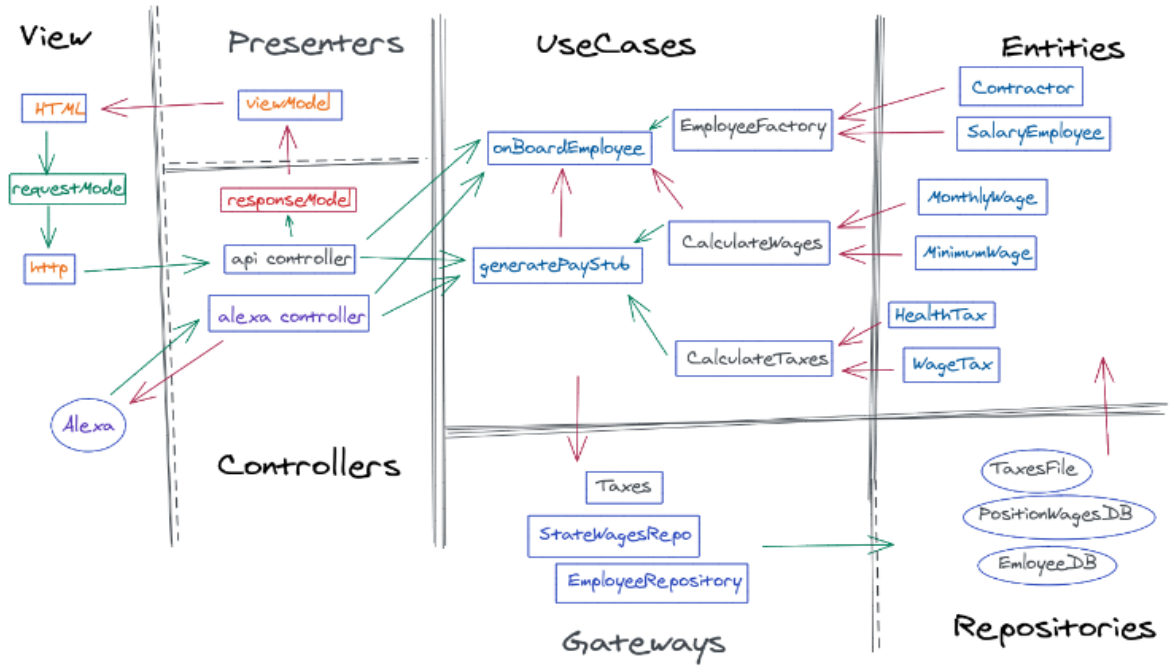
Presenters: These contain the presentation logic and are called by an interactor (through a boundary or protocol) to present information to the delivery mechanism. Presenters create a ViewModel and pass it to the view or view controller. They do not have any knowledge of UIKit or any other interface.

ViewModels: In the Clean Architecture, ViewModels contain simple data like strings, integers, or boolean values required by the view to display the information. They are simple structures that contain all the necessary data (prepared or formatted by the presenter) required by the view, which is very basic and only displays the data.

Boundaries: Boundaries are simply interfaces or protocols in Swift that all communication must pass through to talk to another layer or ring in the architecture.

Gateways: A gateway is another type of boundary that is used to communicate with the outside world, such as a database, network, service, or other devices that the core may need to interact with. Gateway is the term used here, but you may prefer to give them more specific names like DatabaseStore or NetworkAPIClient, which would be protocol names or abstract classes.

StakeOverFlow Image:



The green arrows in the diagram depict input, while the red arrows represent output. It is worth noting that the arrows in the first and third diagrams move in opposite directions. Data flows into the system from the view and passes through the controllers and databases before heading towards the entities at the center of the system in the first and third diagrams.

However, in the third view, all the entities are situated to one side. Repositories populate the entities, which are then injected into the controllers and use cases that require them. In this case, all the red arrows move in one direction, which is called dependency inversion.

It is important to keep in mind that within a region, inputs and outputs can flow in any direction. However, once a boundary is reached, green and red arrows can only move in one direction. Inputs go into a region, while outputs come out of a region. Although it may be tempting to make the repository an input to the generatePaySlip UseCase and directly obtain the data, it is preferred to make sure that Entity classes are the only source of data flowing into the UseCase and serving as their own output.

Conclusion

The goal of a good architecture is to minimize the number of decisions that need to be made. The Clean Architecture achieves this by establishing distinct boundaries between various components of your application in a manner that renders the implementation details behind those boundaries irrelevant to your application. This approach provides you with an unparalleled level of flexibility and independence, enabling you to incorporate diverse drivers of your app, such as UIs or tests, at any time. Likewise, you can seamlessly integrate other external dependencies of your app, such as a database or network, without worrying about the underlying implementation.

Attributions

1. React Documentation: https://react.dev/
2. Diagrams: https://www.diagrams.net/
3. NestJS Documentation: https://docs.nestjs.com/
4. Clean Architecture Intro and Explanation: https://rodschmidt.com/posts/the-clean-architecture-an-introduction/