**UNIEATS**

**divplusplus**

**Project Architecture and Development Environments for UniEats**

Designing UniEats, our innovative Dining Services App, required a thoughtful approach to architecture that would accommodate the complex and evolving needs of our university community. To achieve our goals, we chose a 3-tier client-server architecture, a model known for its clear separation of concerns, scalability, and maintainability. This essay explores the rationale behind our architectural decisions, the challenges we anticipate, and how we have structured our development and production environments to support UniEats effectively.

The 3-tier architecture we selected for UniEats divides the application into three distinct layers: the Presentation Layer, the Application Layer, and the Data Layer. Each of these layers plays a crucial role in ensuring that our app functions efficiently and is capable of scaling as our user base expands.

**1) Presentation Layer**

* At the top of this architecture is the Presentation Layer, also known as the client tier. This layer is the interface that users interact with—whether they are viewing dining menus, managing their dietary preferences, or making reservations. We chose React as the technology to power this layer, due to its ability to create dynamic and responsive user interfaces. React’s component-based architecture allows us to develop complex user interactions that are both intuitive and efficient, ensuring that our users have a seamless experience. The decision to use React is further justified by its support for single-page applications (SPAs), which improve performance by reducing the need for full page reloads. However, building an application with React is not without challenges. As our application grows, managing the state of various components can become increasingly complex. We need to implement a robust state management solution to prevent issues like inconsistent data flow and to maintain a clean, maintainable codebase. This will also ensure our UI remains consistent and reactive to user inputs and system changes. Additionally, while React enables us to create a dynamic front-end, it can pose challenges for search engine optimization (SEO) since content is rendered on the client side, rather than being pre-rendered on the server. Although SEO is not a primary concern for UniEats at this stage, it is something we must consider if the app evolves to require better visibility.

**2) Application Layer**

* Beneath the Presentation Layer lies the Application Layer, or business logic tier. This layer is the heart of UniEats, where all the core functionalities are processed. It acts as an intermediary between the user interface and the database, handling user requests, applying business logic, and ensuring that everything runs smoothly. For this layer, we selected Node.js with Express. Node.js’s non-blocking, event-driven architecture is particularly well-suited for an application like UniEats, which must handle multiple simultaneous requests, especially during peak dining hours. The use of Node.js also allows our development team to work consistently across the stack, using JavaScript for both client-side and server-side code. Express, a minimal and flexible web application framework, provides us with the tools to build robust APIs and handle routing effectively. However, working with Node.js also presents its own set of challenges. Managing concurrency—ensuring that multiple operations can occur simultaneously without conflict—is a critical concern. We must carefully structure our asynchronous code to avoid issues like race conditions, where the outcome of operations depends on the timing of events. Additionally, error handling in an event-driven architecture can be complex. It is crucial to implement comprehensive error-handling strategies to maintain the stability of the application and ensure a smooth user experience.

**3) Database Layer**

* The foundation of our 3-tier architecture is the Data Layer, which is responsible for storing and managing all of the data that UniEats relies on. For this layer, we chose MongoDB, a NoSQL database that offers flexibility and scalability. MongoDB’s document-oriented structure allows us to handle a variety of data types, which is essential given the diverse datasets we work with—from menu items to user preferences and transaction records. The choice of MongoDB is particularly advantageous because its schema-less design enables us to adapt to changes in data structure with minimal rework, an important feature as we continue to iterate on UniEats. However, this flexibility comes with trade-offs. One of the primary challenges we anticipate is ensuring data consistency across a distributed system. Unlike traditional relational databases, which typically enforce strong consistency, MongoDB operates on an eventual consistency model. This means that data changes may not be immediately visible across all parts of the system, which could pose challenges in scenarios where real-time data accuracy is critical, such as tracking meal credits. Additionally, while MongoDB is excellent for flexible data storage, performing complex queries involving multiple collections can be less efficient than in a relational database. To address this, we will optimize our data model and queries to maintain high performance as the amount of data grows.

Building UniEats within this 3-tier architecture requires careful consideration of both our development and production environments. In the development phase, we use tools and technologies that allow us to iterate quickly and efficiently.

To ensure the robustness and reliability of our code, we will use Jest as our primary testing framework for UniEats. Jest's comprehensive suite of features, including its ability to handle both unit and integration tests, allows us to thoroughly test our application's functionality. By writing test cases for individual components, API endpoints, and the various business logic functions, we can identify and resolve issues early in the development process. Jest's snapshot testing capabilities will be particularly useful for verifying that our React components render correctly over time, ensuring that the user interface remains consistent even as the application evolves. Overall, Jest will play a critical role in maintaining the quality and stability of UniEats as we continue to develop and refine the app.

When it comes to the production environment, our focus shifts to scalability, security, and reliability. UniEats is hosted on Microsoft Azure, which provides the infrastructure needed to handle the dynamic demands of campus dining operations. We will employ continuous integration and continuous deployment (CI/CD) pipelines to automate testing and deployment, reducing the risk of errors when new features are rolled out. Scalability is a key consideration, and our architecture supports horizontal scaling, which allows us to add more server instances as demand increases. Security is paramount, given the sensitive nature of the data we handle. We implement HTTPS for secure communication, use JSON Web Tokens (JWT) for user authentication, and leverage MongoDB’s built-in security features to protect our data. To ensure the reliability of the app, we have redundancy and failover mechanisms in place, along with monitoring tools that alert us to any potential issues. This setup ensures that UniEats remains available and responsive, even during peak usage times.

In conclusion, the decision to build UniEats using a 3-tier architecture has provided us with a framework that is both flexible and scalable, capable of supporting the complex needs of our university community. Each layer of the architecture—the Presentation Layer, Application Layer, and Data Layer—has been carefully chosen and configured to address specific requirements, while also anticipating the challenges that come with developing a comprehensive dining services application. As we continue to refine and expand UniEats, we remain committed to maintaining a robust architecture that will evolve with the needs of our users, ensuring that the app remains a reliable and essential tool for managing campus dining services.

**Deployment**

The project repositories are hosted on GitHub, and team members can access them using the following links:

* Frontend: <https://github.com/RohanChhika/UniEats_Frontend.git>
* Backend: <https://github.com/babaYonisali/sdpBackend.git>

We will be using Git for version control and GitHub for repository hosting. At each milestone, a new release will be made with the latest version of the software. Leveraging these tools allows for seamless collaboration, version tracking, and efficient code management. Additionally, this setup facilitates continuous integration (CI) and streamlined deployment processes, ensuring that new code is automatically tested and integrated into the main branch with minimal friction.

**Continuous Integration (CI):**  
To further enhance our development workflow, we will integrate a CI pipeline into our process. Continuous integration will automatically build and test our code every time a team member commits changes to the repository. This approach helps identify and fix bugs early in the development cycle, ensuring that the main branch remains stable and deployable at all times. The CI pipeline will include automated tests that cover key functionality, reducing the risk of introducing defects into the main codebase.

**Branching Strategy:**

* **Main Branch:** This is the functional branch where fully functioning code will be pushed to deployment. Only code that has passed all tests and code reviews will be merged into the main branch, ensuring that it is always in a deployable state.
* **Testing Branch:** This is where code to be tested will be pushed. All testing will be conducted in this branch before the features are merged into the main branch for deployment. This branch serves as a staging area for features that are ready for final validation.
* **Additional Branches:** If the need for other branches arises (e.g., for feature development or hotfixes), the repository will be updated accordingly. These branches will follow a similar process, with changes first being tested before they are merged into the main branch.

**Pushing/Committing Strategy:**  
To avoid conflicting code, team members will pull from the main branch before starting work to ensure they are working on the latest version of the code. Regular commits will be encouraged, and team members will be required to push changes to the appropriate branch (e.g., the testing branch) where automated CI tests will run. Only after passing these tests will the code be eligible for merging into the main branch.

**Microsoft Azure Integration:**  
For deployment and cloud services, we will be utilizing Microsoft Azure. Azure offers robust support for CI/CD (Continuous Integration/Continuous Deployment) pipelines, which we will leverage to automate our deployment process. With Azure, we can automatically deploy new releases of our application to the cloud, ensuring that the latest version is always available to users. Additionally, Azure’s scalable infrastructure allows us to easily manage our application’s performance, security, and availability, enabling us to deliver a reliable and efficient service.

**Deployment Diagram**

