**FlameBurger**

**Architectures Used:**

1. **Client-Server Architecture**
2. **Layered Architecture**

The goal of this project is to build a web application for to manage a Burger chain’s order process. Users will be able to browse the restaurant’s menu, add items of their choice, and finally place their orders. The main task of this application is to make the life of the customer easier by having a simple ordering process and the owners by having an effective system to manage and track the payments and orders.

**Selected Architecture: Client – Server Architecture**

**Requirement:**

Download Node.js and Postgres on the system. Install any required NodeJS elements using the command npm install.

**Implementation Platform:**

* Frontend: ReactJS, HTML, CSS
* Backend: NodeJS
* DB: Postgres

**Compilation instructions:**

Once the platform is setup, just enter the project directory and use the command: **npm start.**

By design of the application the backend and frontend are tightly coupled with each other. Thus, this will trigger and launch both the frontend and backend of the system.This will open the homepage of the application in the browser or Open http://localhost:3001 to view it in your browser.

Once the application is launched the browser, user is directed to the landing homepage where in the user can either login or register. Once successfully authenticated, the user would land on Menu-Order Page wherein they can order their favorite meal. They can then sign out of the application and complete the process.

**Unselected Architecture**

**Requirement:**

* -Python
* Run `pip3 install -r requirements.txt` to install Django.

**Implementation Platform:**

* Python, Django, SQLite 3

Compilation instructions:

To run this application on your laptop, run this command from your terminal:

`$ python manage.py runserver`

or `$ python3 manage.py runserver`

Copy paste the URL link to your web browser.

`Starting development server at [http://127.0.0.1:8000/`](http://127.0.0.1:8000/%60)

On this web app, users can:

* Register, login, browse the menu and items to their cart.
* If appropriate, they can add topping and extras.
* Once an order is complete, user can decide to place an order.

After submission, the items will be removed from the current cart and the site administrator can:

* See a new order on the admin interface that he/she can mark as completed.
* Add or update category and name of product if needed.

**Differences between Client-Server and Layered Architecture:**

**Client-Server Architecture:**

* Communication: Client-Server architectures involves a centralized server responsible for managing all the processing the application. It would accept a request from a client, process any business logics, also communicate with the DB for any required information and the finally send the response to the client.
* Scalability: Scalability is achieved by adding servers with better configurations or nodes for processing the information. Adding these independent nodes from a centralized server help scale and improve the performance of the application.
* Security: Security measures in this implementation are central, ensuring a consistent implementation is done right through the application. The central server controls the authentication, encryption, and the data movement in the application. Any necessary security measures can be enforced in the application by this central server.
* Implementation: We chose an end-to-end JS stack for this architecture as it correctly aligns with our structure. The different end points for the various set of functionalities seamless align with front end units in the architecture.

**Layered Architecture:**

* Communication: Layered application segregates the application into distinct layers like presentation, business logics etc. Each layers have a defined task to do, and it further communicates with its adjacent layers for processing.
* Scalability: Scalability here is achieved through independent development and optimization of layers separately. Every layered must be improved on performance to ensure the overall application is having an improvement. There is no centralized control.
* Security: Security measures are distributed across layers. Every layer might have an independent implementation. If not managed carefully, it might lead to inconsistencies. Thus, it has more of a distributed security control. Each layer might need its own security implementation, thus increasing the complexity of the system.
* Implementation: We chose Django (Python) as the base for this implementation. Django by structure follows a layered style of implementation. Thus, it was easy to maintain and separate the functionalities in various layers in this implementation.

**Pros and Cons of the architectural styles:**

**Client-Server architecture:**

*PROS*

* Centralized Management: Everything is under control of the central server. Easy to manage data, secure data and make any central changes.
* Scalability: Easy by adding higher end server to manage more clients.
* Resource Sharing: Resources like database, files and encryption techniques can be shared for every client.
* Security: Access controls, authentication and data encryption can be implemented by the central server, reducing any unauthorized access or data breaches.

*CONS*

* Single Point Failure: If the server goes down, all clients would be affected. This could lead to processing issues or a complete downtime too.
* Network Issues: A weak network could result in higher latency when relaying information. It can reduce the performance of the application.
* Cost: Setting up and maintaining higher end servers for applications with a huge client and data load can be expensive investments.

**Layered Architecture:**

*PROS*

* Simplicity and Understandability: By segregating the application into distinct layers (presentation, business logic, data access, etc.), it becomes easier to understand and maintain the codebase.
* Reusability: Components and services in each layer can be reused across the application. For example, the data access layer can be utilized by different parts of the business logic layer, promoting code reuse, and reducing redundancy.
* Independence of Development: Layers can be developed and updated independently as long as the interfaces between them remain consistent. This allows for parallel development and easier updates, potentially speeding up the development process.

*CONS:*

* Performance Overhead: The additional layers can introduce latency, as calls must pass through multiple layers before reaching their destination. This overhead can impact the performance of the application, particularly for operations that require rapid interaction between layers.
* Complexity in Inter-layer Communication: As applications grow, managing the interactions between layers can become complex. Ensuring that changes in one layer do not adversely affect another can require significant coordination and can introduce tight coupling between layers if not managed carefully.

**Rationale for choosing Client Server over Layered Architecture:**

1. **Centralized Server:** The architecture offers a centralized control that ensures all the tasks are monitored and performed in the right order. Any new changes can be implemented through this central server.
2. **Scalability:** Client-Server systems allow easy additions of additional support needed for supporting the server functioning. The nodes involved in processing can be gradually increased with the increasing load on the server.
3. **Security:** This architecture ensures the right security measures are in place for managing the sensitive information. Given the central server is secured, the right security measures would be implemented right through the application.
4. **Resource Sharing:** Client-Server model ensures efficient information sharing among different clients. This helps optimize the utilization and overall performance of the application.

In conclusion, we say that the client-server architecture better align with our project's requirements, offering centralized management, scalability, dependability, and strong security measures. These elements would be crucial for maintaining any business chain that want an online system for order management.

**Differences from the Proposed architectural plans**

We have mostly aligned with the architectural and design choices from our project proposal. We did not implement the external API call to Hartland POS as planned in our initial proposal. We had planned to integrate with an external call to this POS that could enable transaction management that is to be defined in our system. We moved away from this decision due to the following concerns:

1. We wanted to focus more on architectural implementation, understand the key differences when implementing different style and process the differences when managing these requests and take key architectural decisions Thus, we decided to keep all the calls internal to be managed by our server.
2. There were key changes done to the API configuration of the API endpoints. There were several restrictions placed on how we can integrate it with our style of implementation. This was another key reason to move away from this initial thought.