Janeeb
Solutions

ARCHITECTURAL SPECIFICATION

Extended Planning Instrument for Unpredictable Spaces and Environments

University Of Pretoria
janeeb.solutions@gmail.com

Table Of Contents

Architectural design strategy:

Background

For our logistics optimization system, we aimed to develop a comprehensive solution that addresses the unique needs of managing and optimising the placement of goods in logistics trucks. Our system leverages dynamic algorithms and learning models to enhance efficiency and space management. This application will assist logistics managers in predicting office capacities and managing desk allocations to prevent overbooking. Given the diverse and demanding requirements of such a system, we have focused on ensuring high performance, scalability, security, reliability, and usability.

Architectural Decisions

The architectural decisions have been based on the quality requirements of the system. We have used the quality requirements (listed in our System Requirements Specification: SRS) to identify our quality requirements which were used to identify our architectural patterns.

Architectural quality requirements:

NF1. Performance:

1.1. The application shall have fast response times for user interactions.
1.2. The application's database operations, implemented with Supabase, should have efficient query execution times to ensure quick retrieval and storage of data.
1.3. The system should return the optimal packing route within 2-5 minutes of the provided configuration/input parameters.
1.4. The real time 3D render should reflect optimizations in real time without any delay or buffering.

NF2. Scalability:

2.1. The backend infrastructure, particularly the database layer hosted on Supabase, should be capable of handling increasing loads as the user base expands.
2.2. The system should be scalable in the sense the algorithm could potentially be used for packing cargo trains, or shipping containers

NF3. Security:

3.1. All user authentication and authorization processes, including registration, login, and password reset, must follow the best practices for safe transfer and storage of sensitive user information.
3.2. Data stored in the Supabase database must be encrypted to protect user privacy and comply with relevant data protection regulations.

NF4. Reliability:

4.1. Error handling mechanisms shall be implemented to handle erroneous input and provide helpful feedback to users in case of unexpected errors.
4.2. Continuous integration and deployment pipelines (CI/CD) shall be set up using GitHub Actions to ensure reliable software releases.

NF 5. Usability:

5.1. The application shall have an intuitive and user-friendly interface to ensure ease of use for all user types, including logistics managers, truck drivers, and warehouse staff.

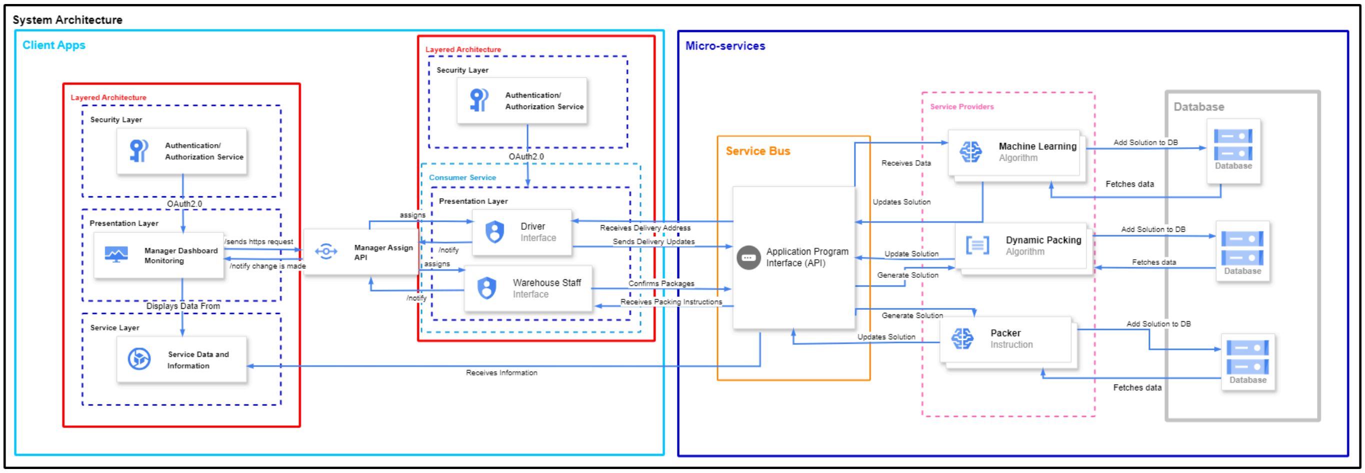
5.2. The interface should be accessible and provide a consistent user experience across various devices and screen sizes.

5.3. The system shall provide clear instructions and feedback to users, guiding them through the process and informing them of any errors or required actions.

5.4 The system shall provide real-time updates to users, ensuring they are always informed of the current status and any changes in the packing process.

Architectural Strategies:

The Extended Planning Instrument for Unpredictable Spaces and Environment is coupled with the following architectural patterns; layered, api gateway and service oriented architecture of the software. The different architectural patterns are shown in the following diagram:



The system is a compilation of a three tier layered architecture, which is formed by the security, presentation and service layers of the system. Each of the layers are responsible for a different task to which their layer entails.

The Security Layer is primarily responsible for handling Authentication and Authorization processes, ensuring that only legitimate users have access to the system while safeguarding sensitive data. This layer acts as a protective shield for both the Presentation Layer and the clients' information. It forms the first point of interaction between the user and the system, validating user credentials and permissions before any access to services or data is granted.

In our system, we utilise the following authentication and authorization mechanisms:

1. Google Authentication and Authorisation: This method leverages Google's secure OAuth 2.0 protocol, enabling users to authenticate using their Google accounts. The system ensures that users are who they claim to be, and it grants access based on their role and permissions within the platform. Google's identity services offer enhanced security features, such as multi-factor authentication (MFA) and adaptive risk-based access control
2. Github Authentication and Authorization: Similar to Google, GitHub's OAuth 2.0 is employed to authenticate users via their GitHub accounts. This is particularly useful for platforms targeting developers and technical users who are already part of the GitHub ecosystem. GitHub provides role-based access control (RBAC) to manage permissions at a granular level, ensuring that users have access only to the resources they need.
3. Supabase Authentication and Authorization: Supabase, a backend as a service (BaaS) platform, provides a comprehensive suite of security features including email/password authentication, third-party providers (e.g., Google, GitHub), and JWT-based authorization. It ensures seamless integration with the system's database and other services, securing access through token-based mechanisms and applying user roles to control resource access.

The Security Layer is crucial for protecting the integrity of the Presentation Layer, which is directly exposed to the user. It also communicates securely with the Service Layer, ensuring that the instructions processed by the system's backend microservices are only executed by authorised entities.

The Presentation Layer serves as the user interface of the system. It is responsible for displaying data and interactions processed by the backend microservices to the user. This layer ensures an intuitive and responsive user experience, relaying instructions from the security layer to the service layer, and ultimately presenting the results back to the user.

At the heart of the system, the Service Layer comprises the core business logic and microservices that handle the actual processing of data and execution of tasks. The layer interacts with the security and presentation layers, receiving and executing authorised instructions, and providing services such as data retrieval, updates, and other backend functions.

By maintaining a clear separation of concerns across these three layers, the system ensures both security and scalability, providing a robust platform for handling complex business processes while keeping user data safe.

Another architecture pattern that is found is the Service Oriented architecture, where the Service Consumers would be the Driver and Warehouse Staff Interfaces. Furthermore, the Service Providers would be the Learning Algorithm, Dynamic Packing Algorithm, and Packer Instructions. The Service Bus in the service oriented architecture would be the API, which serves as gateway from the service provider and the relevant service consumer.

The last architecture pattern we found is the API Gateway Architecture. The different interfaces in the system are the actual Client Application and the Microservices are the different algorithms used to solve the problem. As a gatekeeper that ensures only authorised requests get through, an API is used to connect the Client Application and the Microservices of the system together.

The API serves as a centralised entry point into the Learning Algorithm, Dynamic Packing Algorithm, Packer Instruction. This acts as a mediator between the clients of the different interfaces and the microservices, providing a unified interface to access them. This kind of approach performs several key functions, including security enforcement, request routing, protocol translation, load balancing, and monitoring of the system.

Architectural Patterns and Non-Functional Requirements:

Mitigating the architectural pattern with our non-functional requirements of the system is crucial. This ensures that the system is not only functionally correct, however it also meets the desired quality attributes. Here is how our three architectural patterns, Layered, API Gateway and Service Oriented Architecture affect our systems Non-Functional Requirements:

Performance:

Having a three tier layered architecture enables us to optimise each of the three layers to its specific performance. For example, the Security Layer of the architecture uses efficient authentication and authorization. By separating concerns of the system, a layered architecture reduces the overall complexity of each component. The simplification of each component leads to more efficient code and better performances when it comes to developing the system from scratch. When developing the software, we focused on optimising specific aspects of the system.

The separation of concerns allows each layer to focus on specific tasks, which can lead to more efficient code and optimised performance. For example, the Security Layer can implement efficient authentication and authorization mechanisms without affecting the performance of the Presentation and Service layers.

The API Gateway pattern of the system manages request routing, load balancing and caching to improve the overall performance of the system. By this, efficient performance between the Client Application and the Microservices is improved by the efficient management of the system's requests. Henceforth, the performance of the API is critical to ensure that it responds quickly and efficiently to user requests.

Managing the API in a system that has different sets of interfaces and endpoints can be time-consuming. Using API Gateway we saved heaps of time and effort by centralising the API controls by merging all the different interfaces under one umbrella solution offering greater flexibility and efficiency for various use cases.
Routing requests through a single entry point has worked wonders in simplifying our system, especially when dealing with multiple backend services. By unifying the different interfaces of the system and grouping multiple microservices in the backend, it takes a lot of load off developing and using the application.

SOA allows for the decomposition of complex systems into smaller, manageable services. This enables independent optimization of each service, potentially improving the overall system performance.

Scalability:

Changes in one layer (e.g., updating the security protocols) do not directly affect other layers, providing a stable environment where updates and maintenance can be handled more effectively.

The API Gateway serves as a vital tool for efficiently managing an API by distributing the workload effectively. By spreading incoming requests across multiple instances of our system API, it prevents overloading and boosts scalability and availability. This, in turn, enhances user experience by ensuring consistent and reliable service delivery.

Additionally, an API Gateway acts as a central point for authentication, monitoring, and traffic management, streamlining the process of handling API requests and responses. Its role in simplifying complex architectures and facilitating communication between different microservices makes it a key component in building robust and high-performing applications that can be easily scaled up or down depending on adding or removing microservices in the actual system.

The independence of services means that changes or failures in one service do not necessarily affect others, contributing to the system's stability.

Security:

A dedicated Security Layer enhances security by centralising all security-related functions, ensuring consistent enforcement of security policies across the system. This layer is important to the Presentation Layer which holds our manager interfaces. This is important to the security of the system as the manager interface has access to every interface in the system. We use supabase built in OAuthcallback which uses a combination of JWT and Key authentication features to restrict users access to pages based on their roles, for example drivers cannot access any other page except the driver pages and same for packers etc.

Furthermore, the API Gateway acts as an effective gatekeeper within our system. Using an API, we have a single point of entry to which we effectively enforce security policies such as ensuring only authorised requests reach the backend services. Building the system, we understand that a possible weak point for any application is the handling of data. Henceforth, the API Gateway acts as an efficient tool to safeguard against cyber threats by processing any incoming traffic through low latency traffic routing before it reaches its ultimate destination.

In our system, the API has the service role key so it is the only thing that is able to make CRUD operations to the database, all client side services query the api ensuring there is no access to the database from the front end. We are protected against SQL injection attacks as all calls to the API are checked with string validation and regex and validated to ensure they are valid queries. All API keys that are needed are all defined in an env.local file so that our API keys cannot be found anywhere on our github.

The security of the system is further enhanced with the SOA pattern, whose objective is to control the use of the services to avoid security problems. The security of the individual working components within the architecture, the identity and authentication procedures are related to the individual components, and the security of the connections between the components of the architecture.

Reliability:

By isolating functionalities into different layers, the system becomes more reliable as failures in one layer do not necessarily propagate to others.

By centralising error handling and providing fallback mechanisms, the API Gateway can improve the reliability of the system. It can also retry requests or provide alternative responses in case of failures.

The decoupled nature of services in SOA enhances reliability, as services can be independently scaled, replaced, or upgraded without disrupting the entire system.

Usability:

The layered approach can improve usability by allowing for a more focused and clear design of the user interfaces in the Presentation Layer, while other layers handle more complex processing.

It provides a unified and consistent interface for clients, making it easier for developers to interact with the system. This can lead to better user experiences and easier integration with other systems.

It allows for flexible integration of new services and can provide tailored interfaces for different types of users or clients, improving usability.

Design Patterns Used

Facade

The Manager Dashboard Interface will act as the facade, providing a simplified and unified interface for the logistics manager to interact with the underlying subsystems like the Dynamic Packing Algorithm Service,Learning Algorithm, Packer Instructions Service, and Delivery Instructions Service.

The facade will expose methods for the manager to input constraints, preferences, and priorities for the packing algorithm. It will communicate with the Dynamic Packing Algorithm Service, translating the user input and retrieving the generated packing solution for presentation.

The facade will also provide an interface for the manager to review and provide feedback on the packing solutions provided from the learning model. It will act as a way to monitor the algorithm's performance.

Furthermore, the facade will allow the manager to access packing instructions for warehouse staff and delivery instructions for truck drivers by communicating with the respective services and presenting the information in a user-friendly format.

By acting as the facade, the Manager Dashboard Interface will handle the complexities of communicating with the subsystems, promoting better code organisation, readability, and separation of concerns. It also facilitates easier maintenance and extensibility, as changes to the subsystems won't affect the facade's interface as long as the contracts remain unchanged.

Singleton

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to that instance. This is particularly useful for managing shared resources and maintaining consistent state across the system.

A singleton database connection can efficiently manage database connections, optimising resource utilisation and maintaining connection limits for improved performance and stability.

Security services managing authentication and authorization also benefit from the Singleton Pattern by providing a consistent and universally accessible point of control, enhancing overall security.

Architectural constraints

Data Constraints:

The system must utilise specific datasets provided in the linked documents for developing and training the learning algorithm. These datasets form the foundation for the algorithm's learning and optimization processes. All data must be handled in compliance with relevant data protection regulations. Sensitive information must be encrypted, and access must be restricted to authorised personnel only.

Technology choices

Backend:

- Languages & Frameworks: Typescript
- Database: PostgreSQL on Supabase
- Learning: Typescript
- APIs: RESTful APIs

Frontend:

- Languages & Frameworks: JavaScript, Vue, PrimeVue, Tailwind CSS
- Visualisation: Three.js.

DevOps:

- Version Control: Git, GitHub
- CI/CD: GitHub Actions
- Containerization: Docker
- Cloud Platforms: Netlify

Security:

- Auth/Access Control: OAuth 2.0
- Encryption: SSL/TLS

Collaboration:

- Communication: Discord, WhatsApp, Google Drive
- Project Management: GitHub Project Board
- Documentation: Google Docs.