**Architecture Document**

***<<*** ***La Famiglia Pizzeria >>***

1. **Acronym Explanations**
2. **SOLID:**  A set of design principles that aim to make software designs more understandable, flexible, and maintainable.

* **S – Single Responsibility Principle (RSP):** A class should have only one reason to change, meaning it should have only one responsibility
* **O – Open-Close Principle (OCP):** Software entities should be open for extension but closed for modification.
* **L – Liskov Substitution Principle (LSP):** Objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program.
* **I – Interface Segregation Principle (ISP):** Clients should not be forced to depend on interfaces they do not use
* **D – Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules, both should depend on abstractions.

1. **Keep It Simple, Stupid (KISS):**  This principle advocates for simplicity in design. Overly complex systems are harder to understand and maintain. By keeping things simple, the system becomes more robust, maintainable, and flexible.
2. **Don’t Repeat Yourself (DRY):** This principle stresses the importance of reducing the repetition of code patterns. Redundancy can lead to inconsistency and bugs, so having a single place for code logic or date can improve maintainability.
3. **You Aren’t Gonna Need It (YAGNI):** An extreme programming principle which suggests developers should only implement functionality when it’s necessary. It advocates against adding functionality until deemed necessary.
4. **Overview**

The Pizza Restaturant Application is a web-based solution that facilitates the ordering and delivery of pizzas. The application is designed using Java Spring Boot framework for the backend and React.js for the frontend . The data storage and management is handled using MySQL.

1. **Architecture Details**

The application follows a modular approach, segregating functionalities into different packages under the main package pizz.restaurant:

* **application:** Contains application-wide configurations.
* **domain:** Houses the business logic, entities, and database repositories.
* **presantation:** Manages the API endpoints and reques-response data transfer objects.
* **security:** Exclusively dedicated to user-related actions such as authentication, authorization, and more.

## Backend (Spring Boot)

The backend follows a layered architectural approach:

1. **Presentation Package:** This layer, in line with the Separation of Concerns principle, exclusively interacts with the client-side.

* **Controller Package:** Manages the incoming HTTP requests and sends appropriate responses, ensuring a clear and KISS (Keep It Simple, Stupid) approach.
* **Requests DTOs Package:**  Handles the validation of incoming controller requests, following the DRY (Don’t Repeat Yourself) principle.
* **Response DTOs Package:** Structures the data sent as a response to client requests.

1. **Domain Package:**  Represents the core of the application.

* **Services Package:** Contains the business logic. Abides by the Single Responsibility where each service has a specific task.
* **Entities Package:** Represents the data model or the entities of the application.
* **Repository Package:** Interacts with the database and handles data persistence, adhering to the Interface Segregation Principle.

1. **Application Package:** This is where the application configurations are stored, keeping configurations DRY and ensuring the application remains adaptable without YAGNI (You Aren’t Gonna Need It) extra complexities.
2. **Security Package:** Exclusively dedicated to user-related actions such as authentication, authorization and more.

**Security Package Overview**

* **aspect**

**LoginAspect**: - Responsible for saving JWT tokens to the database post-execution of the login method in the AuthenticationController.

**RefreshTokenAspect**: - Performs two main tasks:

1. Deletes the token from the database post-execution of the refreshToken method.
2. Saves the token in the database once the refreshToken method returns.

* **config**

**AccessControlList:** Defines permissions for different roles and the endpoints they can access.

**JwtAuthenticationEntryPoint:** Manages exceptions for unauthorized access attempts.

**SpringSecurityConfiguration:** Holds the configuration for Spring's security, determining access controls, filters, etc.

* **controller**

**AuthenticationController:** Provides login functionality for any user role and registration functionality exclusively for customers.

**UserController:** Exclusively used by employees to register new users.

* **entity**

**Customer, Token, User:** Represents the respective data models.

* **filter**

**CustomJwtAuthenticationFilter:** Intercepts requests to validate JWT tokens and set authentication in the security context.

* **repository**

**TokenRepository and UserRepository:** Manage CRUD operations for tokens and users.

* **request**

**CustomerRequest, UserRequest, LoginRequest:** Data transfer objects that shape the incoming request data.

* **response**

**UserResponse:** Data transfer object shaping the user data sent in the response.

* **schedule**

**ScheduledTasks:** Contains a cronjob that runs every day at midnight to delete expired JWT tokens.

* **service**

**CustomUserDetailService**: Fetches user details required for authentication.

**TokenService:** Manages operations related to JWT tokens.

**UserService:** Provides user-related operations.

* **util**

**JwtUtil:** Utility class offering JWT-related operations.

* **validator**

**UniqueUser and UniqueUserValidator:** Custom validation ensuring the uniqueness of usernames.

## Frontend (React)

The fronted, developed in React, encourages a component-based architecture aligning with the SRP where each component serves a distinct purpose

1. **Why Spring Boot?**

**Pros:**

1. **Rapid Development:** It’s convention over configuration approach significantly reduces development time, adhering to the KISS principle.
2. **Integrated Server:** Facilitating a YAGNI approach by providing only what’s necessary for deployment.
3. **Wide Ecosystem:** Ensures DRY by offering pre-built functionalities.
4. **Microservice Ready:** Demonstrates the Open-Close Principle, allowing for extension without modification.

**Cons:**

1. **Learning Curve:** Might challenge the KISS principle for beginners.
2. **Memory Consumption:** Can sometimes defy the YAGNI principle with unused embedded tools.

**Constraints:**

1. **Framework Dependency:** The application becomes heavily reliant on the Spring ecosystem.
2. **Configuration Overhead:** At times, contradicts the KISS principle
3. **Why React?**

**Pros:**

1. **Component-Based Architecture:** Directly correlates with the SRP, where each component has its purpose.
2. **Virtual DOM:** Provides and efficient and KISS method for updates and rendering.
3. **Strong Community Support:** Encourages Dry by utilizing community-driven solutions.
4. **Flexibility:** Aligns with the Open-Close Principle as it integrates seamlessly with various backends.

**Cons:**

1. **Learning Curve:** The KISS principle can be challenging due to JSX and component lifecycle complexities.
2. **Only the UI Part:** Requires additional libraries, potentially conflicting with YAGNI

**Constraints:**

1. **UI Focused:**  Ensuring Separation of Concerns by focusing solely on UI.
2. **Dependency on External Libraries:** Can sometimes breach the YAGNI principle.
3. **Why MySQL?**

**Pros:**

1. **Reliability:** Proven over years, keeping the KISS principle in mind.
2. **Open Source:** Reduces redundancy (abiding by DRY) by leveraging widely-supported tools.
3. **ACID Compliance:** Ensures that transactions are handled as atomic units, adhering to the Atomicity principle.
4. **Scalability:** Maintains performance without unnecessary complexities, in line with YAGNI.

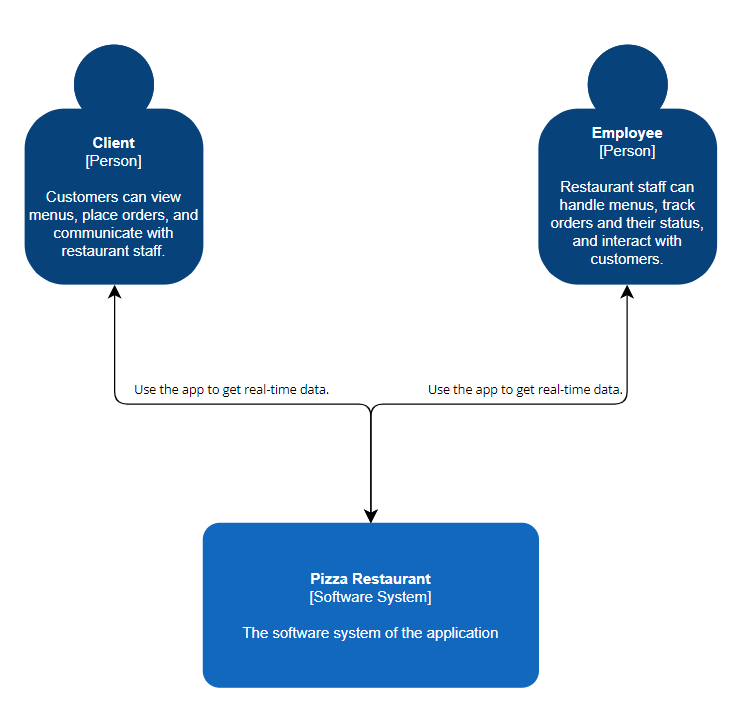
**Cons:**

1. **Not NoSQL:** Could defy the KISS principle for non-relational data.
2. **Complexity:** Without proper optimization, can challenge the KISS and YAGNI principles.

**Constraints:**

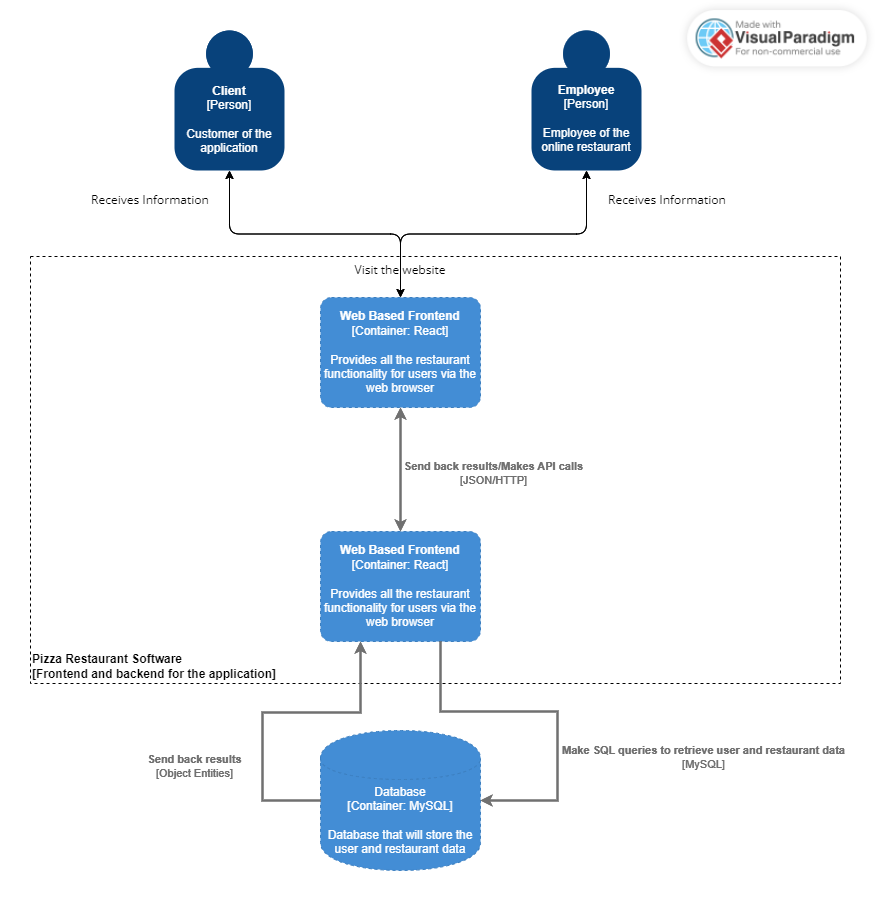
1. **SQL Injection:** Vulnerability if not secured, stressing the importance of Liskov Substitution Principle for secure implementations.
2. **Schema Changes:** Echoes the Open-Close Principle as changes can be challenging.
3. **C4 Diagrams Explanations**

## System Context Diagram



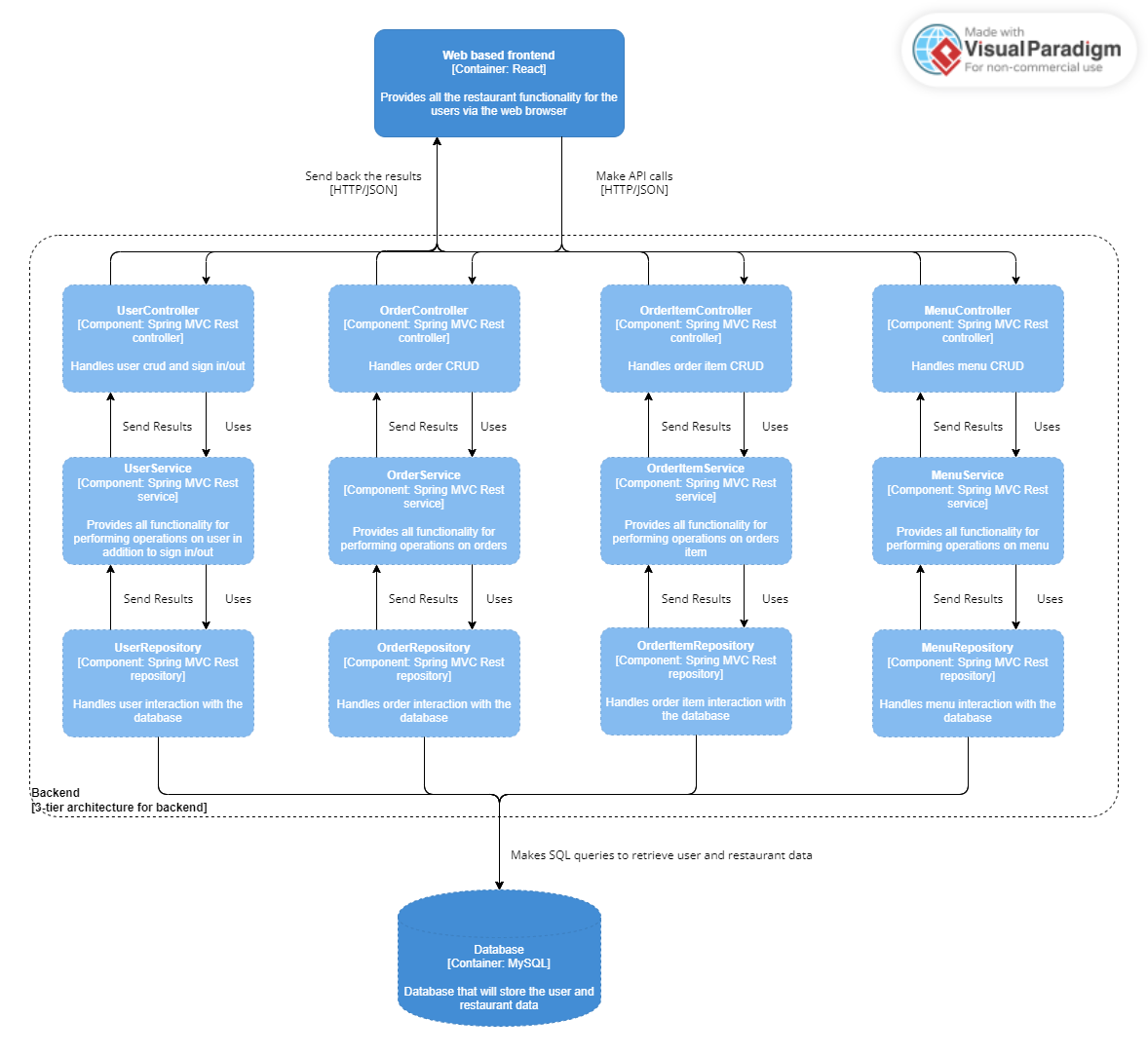
This diagram shows a software system for a Pizza Restaurant. It has three parts: customers, restaurant staff, and the software system. Customers can use the app to see menus, order, and talk to staff. Restaurant staff can manage menus, track orders, and talk to customers. The software system connects them and shares real-time data. Arrows connect the parts to show information flow.

**7.2 Container Diagram**



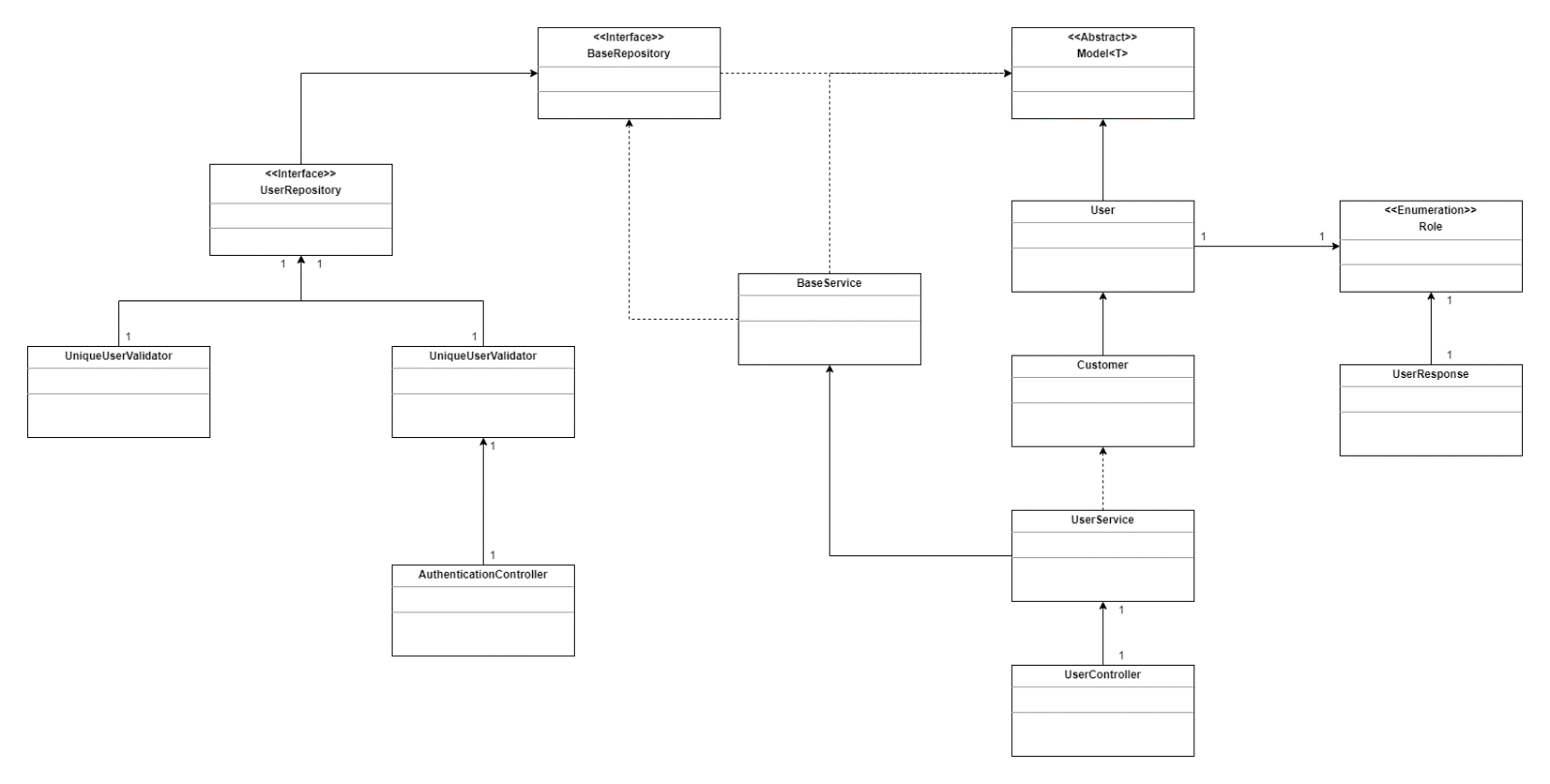
This is a flowchart for pizza restaurant software. It uses blue rectangles and arrows. Rectangles represent different software parts. The top left is "Customer," top right is "Employee," middle is "Web Interface," bottom left is "Software," and bottom right is "Database." Arrows show info flow. Customers and employees access the web, which connects to software and database.

**7.3 Component Diagram**



This is a flowchart of software architecture. It uses blue rectangles with white text and arrows to connect them. Rectangles represent different parts like "User Interface," "Database," "Web Server," "Email Server," and more. Arrows show how data and communication flow between these components.

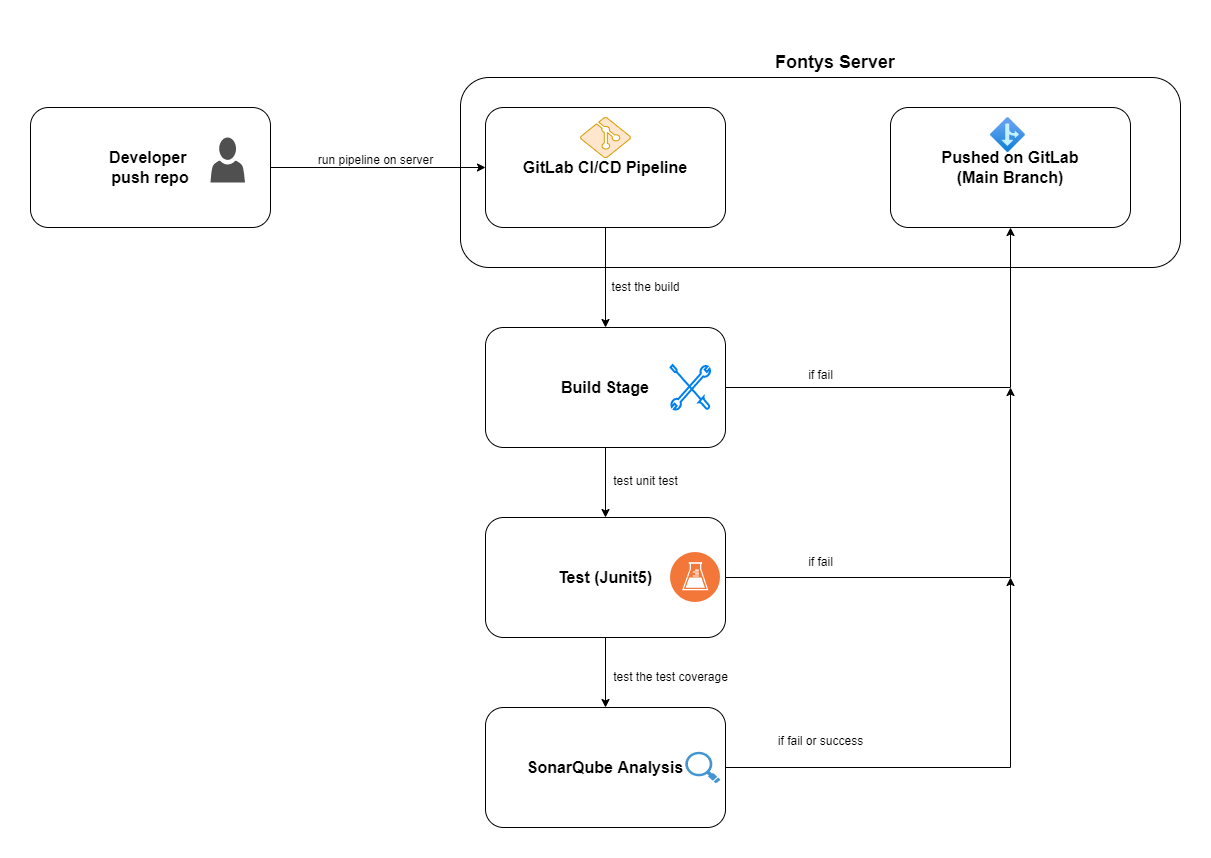
**7.4 UML Diagram**



In this photo, there are different parts of the system represented by things called "classes" and "interfaces." Some examples are "User," "Customer," "AuthenticationController," "UserService", etc.

The lines between these classes and interfaces show how they're connected. These lines can mean different things, like two classes working together (association), one class needing another (dependency), or one class being a special type of another (inheritance).

1. **CI/CD Diagram Explanation**



The diagram illustrates a simple software development and deployment process.

1. **Developer Push Repo:** The developer starts the process by uploading their code repository.
2. **GitLab CI/CD Pipeline:** The code undergoes a series of steps in this pipeline for continuous integration and deployment.

* **Build Stage:** The first stage checks if the code can be properly built.
* **Test (JUnit5):**Successful builds move to this stage where unit tests are performed.
* **SonarQube Analysis:**If unit tests pass, the code proceeds to this stage. SonarQube assesses test coverage and other metrics. The process continues regardless of success or failure in this stage.

**Pushed on GitLab (Main Branch):** In the end the code is added to the main branch on GitLab.

1. **Conclusion**

The Pizza Restaurant application, while adhering to foundational software design principles like SOLID, KISS, DRY and YAGNI, provides and efficient user-friendly, and principled solution. Proper safeguards and optimizations, when approached with these principles in mind, ensure the best performance and security.