# Menu Memorizer App - Architecture Document

Contents

[Menu Memorizer App - Architecture Document 1](#_Toc191190789)

[Overview 1](#_Toc191190790)

[System Architecture 2](#_Toc191190791)

[Architecture Style 2](#_Toc191190792)

[High-Level Components 2](#_Toc191190793)

[Technology Stack 2](#_Toc191190794)

[System Components 2](#_Toc191190795)

[Frontend (Vue.js) 2](#_Toc191190796)

[Backend (Node.js + Express) 3](#_Toc191190797)

[Database (PostgreSQL) 4](#_Toc191190798)

[Future Enhancements 5](#_Toc191190799)

[Conclusion 5](#_Toc191190800)

[Technical Architecture 6](#_Toc191190801)

[Frontend 6](#_Toc191190802)

[Backend 7](#_Toc191190803)

[DB 7](#_Toc191190804)

[User flow: 8](#_Toc191190805)

[How It Works 8](#_Toc191190806)

[Application Flow & State Management 8](#_Toc191190807)

[Scalability & Performance Questions 9](#_Toc191190808)

[What if your app suddenly has 1,000,000 users? 9](#_Toc191190809)

[How would you make the app work offline? 10](#_Toc191190810)

[How would you optimize the frontend for slow connections? 10](#_Toc191190811)

[Technical Questions 10](#_Toc191190812)

[How does the app handle user input and interactions? 10](#_Toc191190813)

[How do you manage state in your app? 10](#_Toc191190814)

[How does the text-to-speech feature work? 10](#_Toc191190815)

[What security measures are in place? 10](#_Toc191190816)

[How would you test your app? 10](#_Toc191190817)

## Overview

The Menu Memorizer App is designed to help waitstaff and restaurant employees memorize menus.

The application currently consists of:

* Flashcards for memorizing menu items
* Quizzes with multiple question types
* Category selection for focused learning
* Audio playback for descriptions and questions

## System Architecture

### Architecture Style

The application follows a **Client-Server Architecture** with a **RESTful API** for backend communication.

### High-Level Components

1. **Frontend** - Vue.js (SPA)
2. **Backend** - Node.js with Express
3. **Database** - PostgreSQL with Prisma ORM
4. **Authentication** - TBD
5. **Hosting** – TBD

## Technology Stack

|  |  |
| --- | --- |
| Component | Technology |
| Frontend | Vue.js (Vue 3) |
| State Management | Provide/Inject API |
| Routing | Vue Router |
| Styling | Tailwind CSS |
| Backend | Node.js with Express |
| ORM | Prisma ORM |
| Database | PostgreSQL |
| API Format | RESTful APIs |
| Hosting | TBD |
| Audio Playback | Web Speech API |

## System Components

### Frontend (Vue.js)

**Key Features:**

* **Category Selection Page**: Allows users to select a menu category before entering flashcards or quizzes.
* **Flashcards**: Display dishes, their descriptions, and ingredients.
* **Quizzes**: Various question formats, including multiple choice and multi-select.
* **Audio Playback**: Uses Web Speech API for text-to-speech functionality.

**Key Files:**

* **App.vue -** Main layout and navigation.
* **router/index.js** - Vue Router setup**.**
* **components/CategorySelector.vue** - Category selection.
* **components/Flashcards.vue** - Flashcards UI.
* **components/Quiz.vue** - Quiz interface.
* **components/AudioPlayer.vue** - Text-to-speech component.

#### Why vue.js

Vue.js offers a lightweight, reactive framework that makes UI development efficient. The flashcards and quiz components require smooth state management, which Vue’s reactivity handles very well. Additionally, Vue’s component-based architecture makes it easy to organize and scale the application. Since the app doesn't require the complexity of a framework like React or Angular, Vue provided a balance between simplicity and power.

### Backend (Node.js + Express)

**Key Features:**

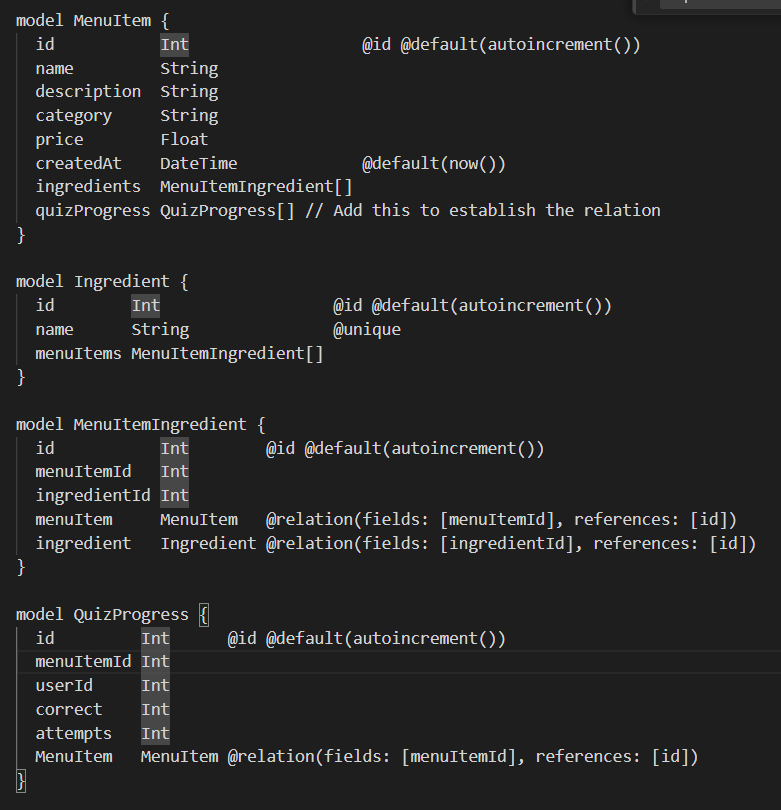
* REST API for fetching menu data
* Prisma ORM for database interactions
* Category-based filtering for flashcards and quizzes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | |  |  | | --- | --- | | **Endpoint** |  | | Description |
| GET | /menu | Retrieve all menu items |
| GET | /menu?category=xyz | Get menu items by category |
| GET | /quiz?category=xyz | Generate a quiz for a category |
| GET | /categories | Fetch available menu categories |

**Key Files:**

* **index.js** - Main Express server file.
* **prisma/schema.prisma** - Database schema definition.

### Database (PostgreSQL)

**Schema Overview:**

#### Why PostgreSQL

Since the app involves structured relationships between menu items, ingredients, and categories, a relational database was the best choice. PostgreSQL provides excellent support for complex queries, transactions, and data integrity, which ensures that menu relationships remain consistent. Additionally, Prisma ORM makes working with PostgreSQL even easier, allowing for a smooth developer experience.

A NoSQL solution like MongoDB wasn’t ideal because menu items, ingredients, and categories naturally fit into a structured relational model.

## Future Enhancements

* Add restaurant-based division
* STT quiz questions
* Drag and drop quiz questions
* **Leaderboard & Scoring**: Track user performance.
* Add Ui for data base inserting
* Image recognition for menu scanning into data base
* Spaced Repetition: Reinforce items the user struggles to remember.
* Search and filter: Quickly find dishes by name, category, or keyword
* Provide insights like accuracy rates or most-missed items.
* Allow users to add personal notes

## Conclusion

This document provides an overview of the design, technologies, and implementation details of the Menu Memorizer App. The app is designed to be scalable and modular, utilizing Vue.js for the frontend and Node.js for the backend, with PostgreSQL as the database. Planned future enhancements aim to improve interactivity and user engagement.

Creating this app was a solution for 2 problems I faced:

* 1. Wanting to keep coding even when unemployed
  2. Struggling with memorizing menu at my temporary waitressing job

So, I figured why not creating a combined solution to fix them both, while expanding my programming skills while I’m seeking a job as a developer.

As a backend developer I thought it will be right to make this a full stack project so I can upgrade my frontend skill set, and with that – work hard on the backend side while learning new technologies and improving the knowledge in the technologies I already work with.

Originally, the idea was to enable a user to update a menu (csv, text, or image), and allowing manual input and categorization.

# Technical Architecture

## Frontend

Framework: **Vue.js**

Vue.js offers a lightweight, reactive framework with a component-based architecture, making it ideal for building a structured and scalable application. Vue’s reactivity system ensures efficient updates to the UI, and Vue Router provides seamless navigation between flashcards and quizzes. Additionally, Vue’s single-file components improve code organization by encapsulating HTML, CSS, and JavaScript within a single file, enhancing maintainability.

* **Better Documentation**: *Vue’s official documentation is well-structured and easier to follow compared to React and Angular, making development smoother.*
* **Performance & Efficiency**: *Vue applications tend to be smaller* [*(10-30% less than React apps)*](https://alokai.com/blog/vue-vs-react) *due to the built-in feature set, reducing the need for external dependencies.*
* **Readability & Maintainability**: *Vue’s declarative template syntax makes it more intuitive, and its single-file component structure keeps the codebase modular and easy to manage.*

**Why Vue.js Over React?**

*While React is powerful, it often requires additional libraries for features like state management and routing. Vue, on the other hand, includes Vue Router and a well-integrated reactivity system, making it more lightweight and efficient for a project of this scale.*

**Why Vue.js over Angular?**

Vue.js is known for its simplicity, ease of learning, and fast development. It has a small learning curve compared to Angular, making it suitable for smaller projects or developers who prefer a lightweight framework.

Conversely, Angular is a comprehensive framework that offers robust features and scalability for larger, enterprise-level applications.

## Backend

I chose **Node.js** for the backend because it’s lightweight, scalable, and designed for handling asynchronous operations efficiently. Since the Menu Memorizer app primarily serves API requests for menu data and quizzes, Node.js, with its non-blocking I/O model, ensures fast and smooth performance. Additionally, using Express makes it easy to build a RESTful API, keeping the architecture flexible for future enhancements.

**Why Node.js Over Python/Django or Java/Spring Boot?**

For this type of application, Django or Spring Boot could have been alternatives, but Node.js offers faster development speed, a lightweight architecture, and better real-time capabilities if I ever extend the app with live updates or WebSockets. Additionally, since the frontend is built with Vue.js, using JavaScript for both frontend and backend ensures consistency across the stack.

## DB

I chose **PostgreSQL** because it’s a powerful, open-source relational database with strong support for structured data. Since the app involves structured relationships between menu items, ingredients, and categories, a relational database was the best choice.

PostgreSQL provides excellent support for complex queries, transactions, full ACID compliance, and data integrity, which ensures that menu relationships remain consistent. Additionally, Prisma ORM makes working with PostgreSQL even easier, allowing for a smooth developer experience.

**Why using Prisma ORM instead of writing raw SQL queries?**

Prisma provides a modern, type-safe way to interact with the database while reducing the chances of SQL injection and syntax errors. It also simplifies complex joins and relationships, making database queries more readable and maintainable. Since the app requires managing multiple related entities (menu items, ingredients, categories), Prisma makes it easier to handle these relationships while maintaining flexibility for future expansion.

**If this app needed to scale, what changes would you make?**

To scale the app, I’d focus on optimizing database queries, implementing caching, and improving load balancing. Right now, menu items are fetched dynamically, but for high-traffic scenarios, I’d introduce Redis caching for frequently accessed categories. For the backend, I’d consider using a managed PostgreSQL service to handle database scaling. On the frontend, I’d implement lazy loading to improve initial load times.

# User flow:

When a user enters the app, they can choose to study all dishes or focus on a specific category. If a user selects **"Quiz"** without choosing a category, the quiz will include all menu items. However, if they first select a category and study the corresponding flashcards, the quiz will be limited to that specific category.

## How It Works

1. Default Behaviour:
   * If no category is selected, the quiz includes all dishes.
2. Category-Specific Behaviour:
   * If a category is selected, both the flashcards and quiz content are filtered accordingly.
3. Routing:
   * The selected category is passed as a query parameter in the URL, allowing for seamless navigation.
4. Dynamic Filtering:
   * Both flashcards and quizzes dynamically adjust their content based on the selected category.

## Application Flow & State Management

1. Global State Handling:
   * The App.vue component maintains the selected category as a global state.
2. Navigation & Query Parameters:
   * Navigation links dynamically update to include the category in the query string.
3. Category Selection:
   * The CategorySelector.vue component updates the selected category using the setCategory method in App.vue.
4. Flashcards & Quiz Integration:
   * The Flashcards.vue and Quiz.vue components automatically filter their content based on the selected category.

# Scalability & Performance Questions

## What if your app suddenly has 1,000,000 users?

If my app had 1 million users, I’d focus on making it faster by improving the database, optimizing the backend, caching frequently used data, and ensuring the frontend loads efficiently. I’d also set up multiple backend servers and use cloud hosting to handle the extra traffic.

would look into specific tools like Redis for caching and cloud auto-scaling, but I’d prioritize making sure my API and database queries are optimized first.

**1. Database Optimization (Making the Database Faster)**

**Problem**: Right now, every user request goes straight to the database, which can slow things down.

**Solution**: I would make the database work more efficiently by:

* Indexing frequently searched data so queries run faster.
* Limiting how much data is sent at once (for example, loading only 10 flashcards at a time instead of all at once).
* Using database caching to store frequently used data in memory instead of making the database fetch the same data over and over.

**2. Backend Optimization (Handling More Users at the Same Time)**

**Problem**: If too many users request quizzes at once, the server might get overloaded.

**Solution**: I would improve this by:

* Using multiple backend servers instead of just one, so requests are spread out.
* Setting a limit on API requests to prevent spam or excessive traffic.
* Optimizing how data is processed by making sure the backend only loads what’s necessary.

**3. Caching (Reducing Repeated Work)**

**Problem**: If every user asks for the same menu data, the backend keeps doing the same work repeatedly.

**Solution**: I would store frequently used data in memory (e.g., Redis) so it loads instantly instead of querying the database every time.

**4. Frontend Optimization (Making the App Faster for Users)**

**Problem:** If the app loads too much data at once, it can become slow.

**Solution:** I would:

* Load only the necessary components when needed (lazy loading).
* Use pagination (loading a few items at a time) to keep things running smoothly.

**5. Scaling the Infrastructure (Using Cloud Services)**

**Problem:** If millions of people use the app at once, a single server won’t be enough.

**Solution:** I would:

* Use **cloud-based hosting** that automatically adds more power when needed.
* Consider **serverless functions** for lightweight API calls.

## How would you make the app work offline?

I could use Service Workers to cache menu data so users can still view flashcards and quizzes even when they are offline.

## How would you optimize the frontend for slow connections?

I would enable lazy loading for components, optimize images, and use a CDN to serve assets faster.

# Technical Questions

## How does the app handle user input and interactions?

The frontend makes API calls to the backend, which retrieves data from the PostgreSQL database. Vue then updates the UI with the fetched menu items and quiz questions. I use query parameters to filter content dynamically.

## How do you manage state in your app?

Since this is a small app, I use Vue’s built-in provide/inject system to manage the selected category globally. If the app grows, I could introduce Vuex or Pinia for more structured state management.

## How does the text-to-speech feature work?

use the Web Speech API to convert text into speech. When a user clicks the 'Play' button, the browser reads the text aloud using built-in voice synthesis.

## What security measures are in place?

Since this app doesn’t have user authentication yet, security risks are minimal. However, I ensure that API requests are properly structured to prevent errors or excessive load. If I add user logins, I would implement JWT authentication and protect against common threats like SQL injection.

## How would you test your app?

I would write unit tests for Vue components using Jest or Vue Testing Library and test backend routes using Postman or automated API tests with Jest and Supertest.