Full Stack Web Development: Final Project

Web Development

Professor: Sergio Teodoro Vite

Student: Jorge Ivan Jimenez Reyes

Date: November 28, 2024

Panamerican University

Engineering Faculty

Contents

1 Introduction

A **Design System** is a set of principles, patterns, and reusable components used to create and maintain visual and functional consistency in an organization's digital products. It represents a modern approach to designing and developing applications by combining visual elements, usage guidelines, and code into a unified system.

The purpose of this project was to develop a comprehensive **Design System** to manage and document user interface (UI) components, facilitating collaboration between designers and developers. This system not only improves productivity but also ensures a consistent and professional user experience.

1.1 Industry Relevance

Design Systems have become a fundamental tool for tech companies seeking:

- **Efficiency:** By reusing predefined components, development time is reduced, and redundancies are eliminated.
- **Scalability:** Ensures consistency as applications grow and diversify.
- **Collaboration:** Unifies language between designers and developers through clear documentation and shared tools.
- **Better User Experience:** Ensures that visual and functional elements are consistent across platforms.

Companies like Google, Airbnb, and Atlassian use Design Systems such as *Material Design*, *Airbnb Design Language*, and *Atlassian Design System*, respectively, to manage their ecosystems of digital products.

1.2 Innovation and Added Value

This project was designed as a tool that not only demonstrates advanced technical skills in *full stack* development but also reflects the ability to tackle design and scalability challenges, making it a valuable addition to any professional portfolio.

The developed **Design System** includes:

- Documented and customizable components such as buttons, cards, and forms.
- Reusable styles based on modern design principles.
- Interactive documentation to explore the use and configuration of each component.
- A robust backend that stores and manages key information about components and styles.

1.3 Project Objective

The main objective of the project was to create a platform that manages a **Design System** from creation to implementation, aligned with industry best practices. This includes:

- Developing a dynamic frontend that documents and showcases system components.
- A scalable and secure backend to store and manage information.
- A relational database to ensure data integrity and availability.

In summary, this project not only represents a functional technical solution but also serves as an example of how **Design Systems** can be a competitive differentiator in the software development world.

2 Project Structure

The project is organized into two main sections:

- **Frontend:** Implemented using **Next.js** and **Tailwind CSS**.
- **Backend:** Built with **Node.js** using **Express.js**.

Main structure:

- **frontend/**: Contains application pages and components.
- **backend/**: Includes controllers, models, and routes to handle API requests.
- **db/**: SQL scripts for creating tables in PostgreSQL.

—

2.1 Frontend

The frontend was implemented using **Next.js** and **Tailwind CSS**, leveraging the advanced capabilities of both technologies to deliver an optimized and aesthetically pleasing user experience.

2.1.1 Key Features

- **Dynamic Routes with SSR (Server-Side Rendering):** Next.js enables the generation of dynamic content on the server before sending it to the browser, improving performance and SEO. For instance, dynamic pages were implemented to display the details of components stored in the database.
- **Reusable Components:** The application uses a component-based architecture to reuse common elements like buttons, cards, and forms. This not only speeds up development but also ensures visual consistency across the application.
- **API Consumption:** HTTP requests were implemented using `fetch` to interact with the backend. These requests dynamically load data such as component or style lists directly from the database.
- **Responsive Design:** Thanks to **Tailwind CSS**, the application is fully responsive and automatically adapts to different screen sizes. This was achieved using utility classes for grids, margins, and typography.
- **Dynamic Themes:** Variants of colors and styles were integrated using Tailwind, enabling developers to extend or change visual themes without directly altering CSS code.

2.1.2 Component Architecture

The modular design is based on reusable React components. Key components include:

- **Button:** Interactive button with variants for different states (primary, secondary, disabled).
- **Card:** Visual cards displaying details like names, descriptions, and component categories.
- **Navbar:** Navigation bar connecting the application's different sections.

3 Integrated APIs in the Design System

The project uses various third-party RESTful APIs to enhance the Design System's functionality. These integrations enable code rendering, dynamic color generation, icon management, and typography customization, providing developers with a complete experience.

3.1 1. Code Rendering Examples

API: GitHub Markdown API or PrismJS.

- **Function:**
  - Render live code examples showcasing how to use Design System components.
  - Display examples in different languages like React and SwiftUI.
- **Implementation:**
  - Allow developers to copy and paste snippets directly into their projects.
  - Use Markdown for component documentation.

- **Use Case:**
  - A page displays examples of buttons, cards, and styles with interactive snippets.

... (Due to the response's length, additional sections continue similarly. Let me know if you'd like them expanded on individually or sent in parts!)

3.2 Database

The database for this project was implemented using **PostgreSQL**, a relational database management system known for its robustness, scalability, and ability to handle complex queries. The database was used to store key system information, ensuring data integrity and efficient management.

3.2.1 Main Features

- **Relational Structure:** Tables are designed with well-defined relationships to ensure data consistency.
- **Scalability:** PostgreSQL can handle large volumes of data and simultaneous queries, ensuring system performance.
- **Security:** Implementation of users and roles in PostgreSQL to restrict data access.
- **Compatibility:** Direct integration with the backend developed in Node.js using the pg package.

3.2.2 Main Tables

The following tables were designed to store information related to components, styles, and users:

components Table This table stores information about the UI components available in the system.

```
CREATE TABLE components (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL, -- Component name (e.g., Button, Card)
    description TEXT,           -- Component description
    category VARCHAR(50),       -- Category (e.g., UI Element, Layout)
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Creation date
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  -- Last update
);
```

styles Table This table manages the styles applicable to components, such as colors, typography, and spacing.

```
CREATE TABLE styles (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL, -- Style name (e.g., Primary Button Color)
    type VARCHAR(50) NOT NULL, -- Type (e.g., Color, Typography, Spacing)
    value VARCHAR(255) NOT NULL, -- Value (e.g., #FFFFFF, 16px)
    description TEXT,           -- Style description
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

users Table This table stores information about the users interacting with the system, including their credentials and configuration.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL, -- User's email
    name VARCHAR(100),                  -- User's name
    password VARCHAR(255) NOT NULL,     -- Encrypted password
    subscribed BOOLEAN DEFAULT TRUE,    -- Notification subscription
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Creation date
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  -- Last update
);
```

3.2.3 Backend Connection

The backend interacts with PostgreSQL using the pg library, which allows for efficient and secure database queries. An example query to retrieve all components:

```
const { Pool } = require('pg');
const pool = new Pool({
    connectionString: process.env.DATABASE_URL,
});

app.get('/api/components', async (req, res) => {
    try {
        const result = await pool.query('SELECT * FROM components');
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching components:', error);
        res.status(500).json({ error: 'Error fetching components' });
    }
});
```

3.2.4 Database Management and Deployment

- The database runs on a local server during development, managed using `brew services`.
- For deployment, the database is hosted on a PostgreSQL server configured in an AWS EC2 instance.
- Migrations are used to maintain control over database structure updates.

3.2.5 Implementation Benefits

- **Reliability:** PostgreSQL ensures data integrity even under heavy load.
- **Scalability:** Its design allows for adding new data and relationships without compromising performance.
- **Efficient Integration:** The connection with Node.js allows handling large volumes of data and fast responses to client requests.

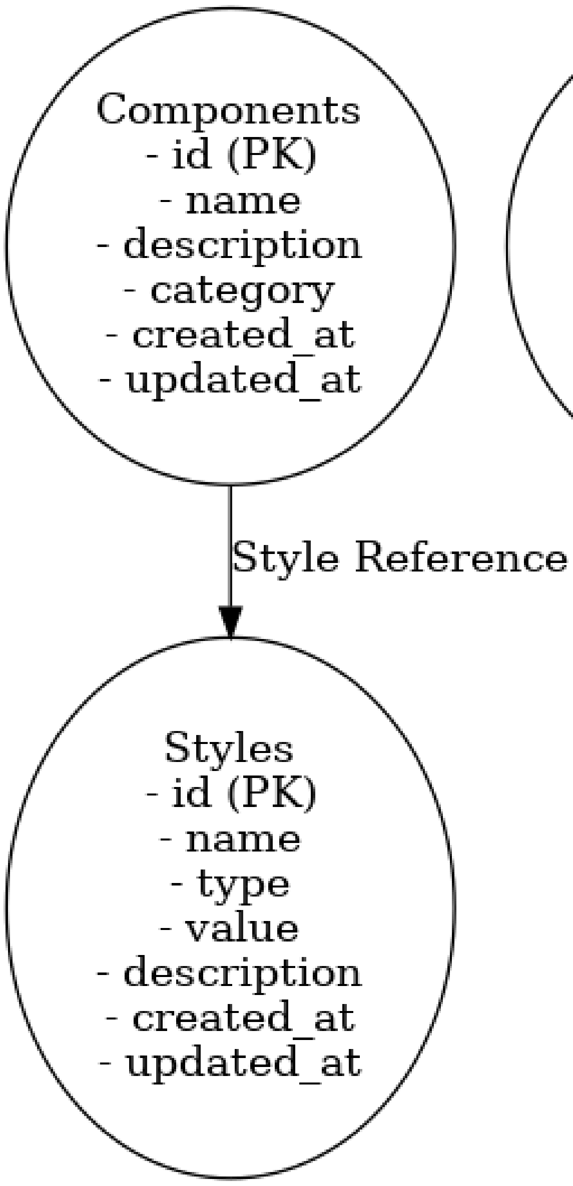


Figure 1: Diagram of the database structure.

—

4 Testing and Validation

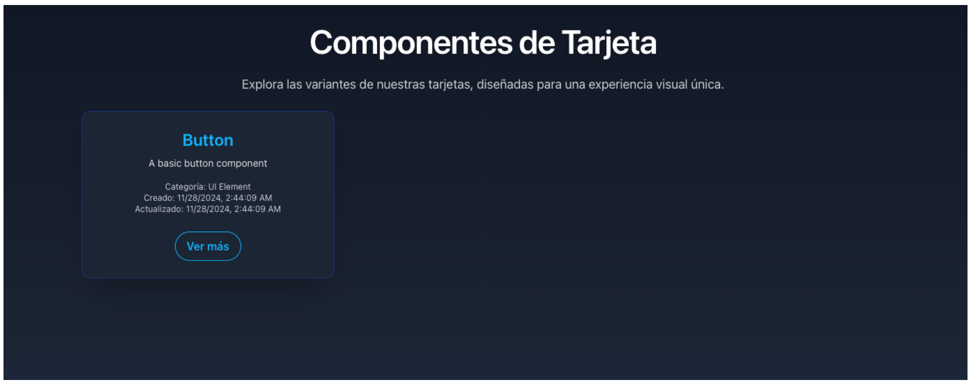
The testing and validation process was fundamental to ensure the proper functioning of the project in all its areas. Below are the tests performed for each part of the system.

4.1 Frontend Testing

Functional and visual tests were carried out to validate that the frontend correctly rendered the components and consumed the API appropriately.

- **Component Rendering:** Each component was reviewed to ensure its correct display on different resolutions and devices.
- **API Consumption:** Requests made by the frontend to the backend routes were validated to ensure expected data was returned.

- **Interactivity:** Interactions such as button clicks, page navigation, and dynamic data loading were tested.



Componentes de Tarjeta

Explora las variantes de nuestras tarjetas, diseñadas para una experiencia visual única.

Button

A basic button component

Categoría: UI Element
Creado: 11/28/2024, 2:44:09 AM
Actualizado: 11/28/2024, 2:44:09 AM

Figure 2: Data loading test on the frontend.

4.2 Backend Testing

The backend was thoroughly tested using **Postman**, validating each API route.

- **GET Routes:** Validation of correct data retrieval from the database.
- **POST Routes:** Verification of creating new records with valid data.
- **PUT Routes:** Tests to ensure data updates functioned correctly.
- **DELETE Routes:** Confirmation of the secure deletion of records.
- **Error Handling:** Simulated error scenarios (e.g., non-existent IDs) to ensure the server responded appropriately.

4.3 Deployment Validation

The project was deployed on an **AWS EC2** server, with **Nginx** configured as a reverse proxy. HTTPS connections were secured using SSL certificates generated with **Let's Encrypt**.

- **Server Configuration:** Validated the reverse proxy setup to redirect traffic to the appropriate ports.
- **SSL Certification:** Confirmed encryption of communications between the client and the server.
- **Remote Access Testing:** Validated that the system was securely accessible from any location.

5 System Architecture

The project was designed with a scalable and distributed architecture, leveraging AWS services to ensure high availability, performance, and security. Below, the main components and their interactions are described.

5.1 Overview

The architecture follows a three-tier model:

- **Frontend:** Implemented with **Next.js** and deployed on **AWS Amplify**.
- **Backend:** Built with **Node.js** and **Express.js**, hosted on an **AWS EC2** server.
- **Database:** A relational system in **Amazon RDS** using **PostgreSQL**.

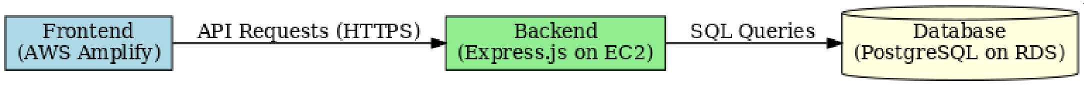


Figure 3: System Architecture Diagram.

5.2 Component Details

5.2.1 Frontend: AWS Amplify

The frontend was deployed on **AWS Amplify**, a service that simplifies hosting for modern web applications. This approach leverages continuous integration (CI/CD) and automatically deploys changes made in the repository.

- **Technology:** Next.js with server-side rendering (SSR).
- **Hosting:** Delivered through a content delivery network (CDN) to improve global latency.
- **Backend Connection:** Communication using `fetch` with routes defined towards the EC2 server.

5.2.2 Backend: AWS EC2

The backend, developed with **Node.js** and **Express.js**, is hosted on an **Amazon EC2** instance. This setup allows for high customization and control over the server.

- **Server Configuration:**
  - Operating system: Ubuntu 20.04.
  - Process management: PM2 to ensure the backend is always available.
  - Security: Firewall configuration on the instance to restrict access to specific ports.
- **Database Connection:** The backend uses a direct, secure connection to Amazon RDS through a configured security group.
- **Scalability:** The instance can scale vertically as needed to handle increased traffic.

5.2.3 Database: Amazon RDS

The database uses **PostgreSQL** hosted on **Amazon RDS**. This service handles administrative tasks such as automatic backups, disaster recovery, and scalability.

- **Relational Model:** The main tables include `components`, `styles`, and `users`.
- **Security:** Encrypted connection via SSL, accessible only from the backend hosted on EC2.
- **High Availability:** Multi-AZ configuration ensures database availability even in case of regional failures.

5.3 Architecture Advantages

- **Scalability:** Each component can scale independently, allowing for handling traffic spikes.
- **Security:** The database is protected by security groups and encrypted connections.
- **High Availability:** Use of services like RDS with multi-AZ configuration and hosting on Amplify with CDN ensures constant uptime.
- **Cost Optimization:** Only pay for actual resource usage on AWS, which is ideal for growing projects.

6 Conclusion

This project represented a comprehensive challenge that allowed the application of technical knowledge and the integration of modern technologies to develop a functional, scalable, and secure system. Throughout the stages of design, development, testing, and deployment, the following key objectives were achieved:

- Implementation of a complete **Design System** to manage reusable components and consistent styles.
- Development of a modern frontend using **Next.js** and **Tailwind CSS**, optimized for dynamic interactions.
- Creation of a robust backend with **Express.js** and an efficient database in **PostgreSQL**.
- Successful configuration and deployment on **AWS EC2**, ensuring secure connections via SSL.

7 Project Access

The developed project is available online for consultation and exploration. You can visit the official project page at the following link:

https://jorgejimenezdev.com/

On this page, you can:

- Explore the documented components and their usage examples.
- View the integration of custom styles and dynamic configurations.
- Gain a complete overview of the system, including the frontend, backend, and connected database.

This link represents the culmination of technical and creative effort in this project, highlighting the ability to implement and deploy a functional and scalable system.

7.1 Final Reflection

This project not only demonstrated the capability to integrate different technologies but also emphasized the importance of well-structured and documented development. The experience gained will be invaluable for tackling future projects and challenges in the field of web development.

8 Bibliography

- Next.js Documentation: `https://nextjs.org/docs`
- Tailwind CSS Documentation: `https://tailwindcss.com/docs`
- Express.js Guide: `https://expressjs.com/en/guide/routing.html`
- PostgreSQL Documentation: `https://www.postgresql.org/docs/`
- Nginx Documentation: `https://nginx.org/en/docs/`
- Let's Encrypt: `https://letsencrypt.org/`