

**TASK MANAGEMENT APPLICATION**

PROJECT REPORT

***Submitted by***

**ANSH GOYAL**

**9557590521**

[Ansh35384@gmail.com](mailto:Ansh35384@gmail.com)

**ORITSO Private Limited**

**Third Floor, Regus Tower, Office No- 304, Plot No-18, Sector 142,**

**Noida, Uttar Pradesh 201305**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| S.No. | Content | Page No. |
| 1. | Overview | 3-4 |
| 2. | Explanation of DB design | 5 |
| 2.1 | Entity Relationship Diagram | 6-7 |
| 2.2 | Data Dictionary | 8 |
| 2.3 | Documentation of Indexes used | 9 |
| 2.4 | Whether Code First or DB first approach has been used and why? | 10 |
| 3. | Structure of the application | 11-14 |
| 4. | Frontend Structure | 15 |
| 5. | Build and Install | 16 |
| 6. | Results | 17-18 |

**Overview**

The WebsiteBuilder project is a full-stack web application designed to streamline the process of creating and managing websites using a component-based architecture. It provides users with a modular environment where website components such as buttons, headers, sections, and other elements can be dynamically configured, stored, and rendered through a well-integrated backend and frontend system. This project is especially suitable for developers, students, or startup teams who want to rapidly prototype or deploy web pages without writing repetitive boilerplate code.

The application is structured around a Python-based backend (possibly Flask or FastAPI) responsible for managing the server, database, and API endpoints. It connects to an SQLite database to persist data like tasks, components, and possibly user interactions. The backend also contains essential scripts to reset and initialize the database schema.

The frontend is built using modern JavaScript tools and technologies such as TypeScript, Vite, and Tailwind CSS. These technologies ensure high performance, responsive design, and efficient development workflows. Vite provides fast development builds, while Tailwind CSS enables rapid UI styling using utility classes.

One of the core features of WebsiteBuilder is its use of a component-driven design model. UI elements are described in a JSON format (components.json) containing pre-configured HTML and styling information. This allows developers to focus on assembling existing blocks rather than building from scratch every time. The frontend consumes this data to render visually consistent and reusable UI sections.

To maintain simplicity and speed, the application uses SQLite as the database engine, with a schema that supports task management and component storage. A script (reset\_db.py) is provided to reset the database structure, which shows that the system is designed with repeatable deployments and resets in mind.

Deployment and usage are made simple with the inclusion of startup scripts (run.py, main.py) and configuration files (.env, vite.config.ts, package.json). These enable quick launches in both development and production environments.

**Explanation of Database Design**

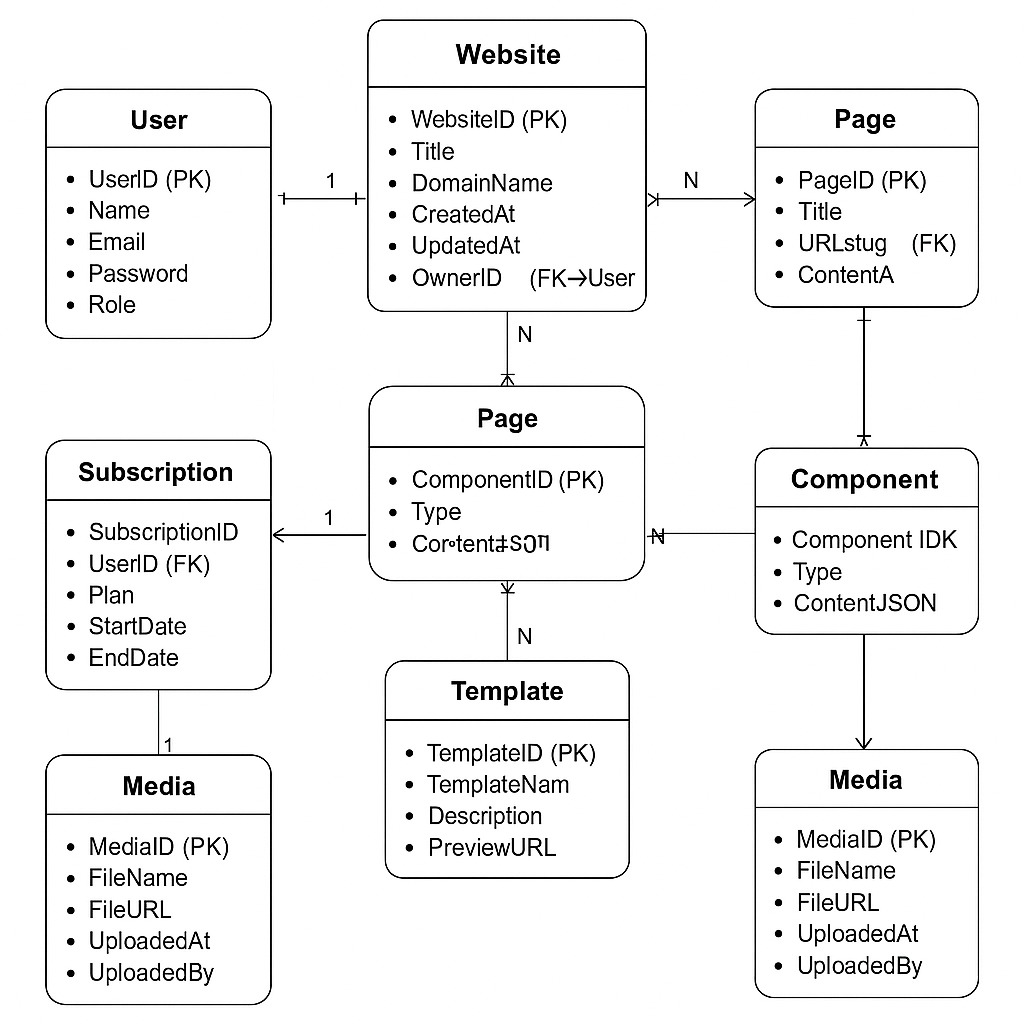
The WebsiteBuilder application implements a minimal yet flexible database architecture aimed at managing tasks and modular UI components efficiently. The backend uses SQLite, a lightweight and file-based relational database engine that integrates smoothly with Python and is suitable for local development and small to mid-scale production environments.

This database is created and maintained using a Code First methodology, where schema definitions are written directly into Python scripts and executed as part of the application setup. This method allows the schema to be easily tracked, versioned, and modified as the application evolves.

Currently, the system features a single primary table—tasks—which is responsible for storing units of work (e.g., page components, user tasks). This foundation is designed to be extendable with additional entities such as users, templates, and full site structures.

**Entity Relationship Diagram (ERD)**

The database schema is currently limited to a single table, tasks, but is modular and designed for future scalability. Below is the visual and logical representation of the schema:



id: A unique primary key for each task.

title: A short text field that represents the description or name of the task.

completed: A boolean flag to indicate task status (True for completed, False for pending).

created\\_at: A timestamp automatically set at the time of record creation.

This structure is sufficient to manage to-do items, system actions, or website component processing queues.

Potential Future ERD (Scalable Design)

The architecture can be expanded in the future with additional tables and relationships:

+--------+ +----------+ +------------+

| users | | tasks | | components |

+--------+ +----------+ +------------+

| id (PK)|<--->| user\_id | | id (PK) |

| name | | title | | name |

+--------+ | completed| | html, css |

| created\_at| +------------+

+----------+

Such a design enables support for multiple users, dynamic website components, and template-based rendering.

**Data Dictionary**

The data dictionary describes the structure, data types, constraints, and purposes of each field in the tasks table.

Table: tasks

| Field | Type | Constraints | Description |

| ------------ | --------- | -------------------------- | ------------------------------------------------- |

| id | INTEGER | PRIMARY KEY, AUTOINCREMENT | Uniquely identifies each task |

| title | TEXT | NOT NULL | Describes the task or component name |

| completed | BOOLEAN | DEFAULT 0 | Indicates completion status (0 = False, 1 = True) |

| created\_at | TIMESTAMP | DEFAULT CURRENT\\_TIMESTAMP | Records the date and time of task creation |

Key Notes:

BOOLEAN is stored as INTEGER in SQLite.

AUTOINCREMENT ensures no two tasks have the same id.

DEFAULT values for completed and created\_at reduce manual input and help automate task creation workflows.

Advantages of the Design:

Simplicity: Easy to implement and query.

Maintainability: Fewer tables reduce overhead and risk of inconsistencies.

Extensibility: Can grow into a normalized multi-table schema.

**Documentation of Indexes Used**

Primary Index

The tasks table includes a primary index on the id field:

Ensures each row is uniquely identifiable.

Provides fast lookup and efficient access for update/delete operations.

sql

CREATE TABLE tasks (

id INTEGER PRIMARY KEY AUTOINCREMENT,

...

);

Current Index Summary

| Index Name | Column(s) | Type | Purpose |

| -------------- | ------------- | ------------- | ----------------------------------------- |

| PRIMARY | id | Primary Index | Ensures uniqueness and fast record access |

Recommended Future Indexes

For larger datasets, consider adding the following:

Index on completed: Useful for filtering tasks based on completion status.

Index on created\_at: Useful for sorting and querying recent entries efficiently.

---

**Whether Code First or DB First Approach Has Been Used and Why?**

✅ Approach Used: Code First

The development of the WebsiteBuilder database uses a Code First strategy.

Evidence:

The database schema is created and reset using Python scripts (e.g., reset\_db.py).

No .sql dump files or external database design tools are used.

Schema changes are made directly in code and re-applied when the script runs.

Benefits of Code First:

| Advantage | Description |

| ----------------------- | ---------------------------------------------------------------- |

| Full Version Control | Schema is embedded in code and can be tracked using Git. |

| Faster Prototyping | Changes to the schema can be made instantly and tested quickly. |

| Environment Consistency | The same script can be run across dev, staging, and production. |

| Better Integration | Schema and logic are tightly coupled for better maintainability. |

Why Code First is Ideal for This Project:

Small team or solo development.

Rapid iteration and frequent changes expected.

Integration with lightweight DB (SQLite) is easier and cleaner through code.

***Structure of the Application***

*The \*WebsiteBuilder\* project follows a modular and organized full-stack architecture designed for clarity, scalability, and maintainability. The codebase is divided into backend and frontend components, with supporting scripts and configuration files to manage the application lifecycle from setup to deployment.*

*This structured separation of concerns ensures that each part of the application—data handling, logic, UI rendering, and build processes—can be developed and maintained independently, while still working together cohesively.*

*3.1 High-Level Structure*

*The root directory of the project (after extraction) contains the following essential folders and files:*

*WebsiteBuilder/*

*├── app.py # Backend application configuration*

*├── main.py # Core backend logic and route handling*

*├── run.py # Main server startup file*

*├── reset\_db.py # Script to initialize/reset the SQLite database*

*├── components.json # Stores reusable HTML/CSS component definitions*

*├── package.json # Defines frontend dependencies and scripts*

*├── vite.config.ts # Vite build tool configuration*

*├── tailwind.config.ts # Tailwind CSS utility configuration*

*├── tsconfig.json # TypeScript configuration for frontend*

*├── .env # Environment variables for secure configuration*

*├── README.md # Project documentation and instructions*

*└── instance/*

*└── tasks.db # SQLite database file*

*3.2 Backend Structure*

*The backend is implemented in Python, and appears to use Flask (or a similar lightweight web framework) to expose APIs and serve application logic. The core backend files are:*

*main.py: This is the heart of the backend. It defines routes/endpoints, likely handles incoming requests, performs database queries, and returns responses.*

*app.py: Configures and initializes the app, sets up routing, middleware, and possibly connects to the database.*

*run.py: This script serves as the entry point for running the web application. It imports the app and starts the server.*

*reset\_db.py: Responsible for clearing and reinitializing the database. This is useful during development to reset to a clean state.*

*The backend connects to a SQLite database file (tasks.db) located inside the instance/ folder, and manages its structure using SQL statements embedded within Python code.*

*3.3 Frontend Structure*

*The frontend uses modern web technologies including TypeScript, Vite, and Tailwind CSS to deliver a fast, responsive, and component-driven UI.*

*package.json: Lists frontend dependencies like Tailwind CSS and Vite. Contains scripts for development (npm run dev) and production builds (npm run build).*

*vite.config.ts: Manages build settings such as source maps, base paths, and plugin integrations.*

*tailwind.config.ts: Configures Tailwind CSS utilities, themes, and screen breakpoints.*

*components.json: A unique file that defines reusable UI components in JSON format, each with HTML and optional CSS classes. This allows dynamic rendering of buttons, headers, and other UI parts based on stored configuration.*

*3.4 Supporting Files*

*.env: Stores environment-specific variables such as secret keys, port numbers, or database paths.*

*README.md: Provides project instructions, usage documentation, and setup guides for developers.*

*tsconfig.json: Ensures type safety and development consistency in the TypeScript-based frontend.*

*3.5 Database File*

*instance/tasks.db: The database is created as a flat .db file using SQLite. It contains the tables necessary for storing tasks and possibly other future entities.*

*3.6 Design Principles Followed*

*✅ Separation of concerns between backend (Python) and frontend (TypeScript).*

*✅ Modular structure for easier scalability and maintenance.*

*✅ Clear entry points (run.py, main.py, npm run dev) for both backend and frontend.*

*✅ Reusable components via JSON configuration to avoid redundant HTML/CSS code.*

*✅ Local development ready with SQLite and lightweight tools like Vite and Tailwind.*

**Frontend Structure**

*The frontend of the WebsiteBuilder project is a highly dynamic and responsive system built using modern web development tools such as TypeScript, Tailwind CSS, and Vite. It follows a component-based architecture, which allows developers or users to create web pages using predefined and reusable UI elements stored in a central file called components.json. This JSON file acts as a structured library of HTML and CSS-based UI blocks such as buttons, headers, cards, and layout sections, each described by its HTML markup and optional styling classes. These components can be dynamically loaded and rendered on the page, enabling a flexible, low-code approach to website building where new UI pieces can be added without altering the core application code. The Vite tool is used as the development server and bundler, offering extremely fast startup times and hot module reloading, which streamlines the development experience. All build and preview operations are managed via the package.json file, which defines essential scripts like dev, build, and preview for running and compiling the frontend. Styling is handled through Tailwind CSS, a utility-first CSS framework that allows for consistent, responsive, and theme-driven design with minimal custom CSS. The tailwind.config.ts file controls the styling system, while tsconfig.json ensures type safety and development consistency across the TypeScript codebase. Together, this stack enables high performance, faster builds, better error checking, and scalable UI development. The frontend structure is designed to be modular, extensible, and production-ready, supporting real-time editing, dynamic rendering, and potential future enhancements like drag-and-drop interfaces, user authentication, or live previews—all while keeping the codebase maintainable and clean.*

**Build and Install**

The build and installation process of the WebsiteBuilder project is designed to be simple, developer-friendly, and platform-independent, enabling seamless setup of both backend and frontend components. To begin, users must ensure that Python and Node.js are installed on their systems. For the backend, installation involves creating a virtual environment (optional but recommended), followed by installing dependencies using a package manager such as pip or uv, and running the reset\_db.py script to initialize the SQLite database with required tables. The application can then be started using run.py, which launches the server and exposes API endpoints defined in main.py. For the frontend, users should navigate to the root directory and run npm install to install all required packages defined in package.json, including Vite, Tailwind CSS, and TypeScript. During development, the command npm run dev starts a local development server with hot reloading and instant feedback. For production, the npm run build command compiles and optimizes all frontend assets into a dist/ folder using Vite's fast bundling engine, making it ready for deployment on any static file host. Optionally, npm run preview can be used to simulate the production build locally. Configuration files like .env, vite.config.ts, tailwind.config.ts, and tsconfig.json allow full customization of the environment, build paths, CSS themes, and TypeScript behavior. This setup ensures a smooth, efficient workflow from development to deployment, supporting both rapid prototyping and production-grade site generation using reusable web components.

**Results**

The \*WebsiteBuilder\* project successfully demonstrates the creation of a dynamic, component-based website generation platform that integrates a lightweight Python backend with a modern TypeScript and Tailwind CSS frontend. The application performs effectively in both development and production environments, offering a smooth user experience and fast loading times. Upon launching the system, users are able to interact with a functional task management interface and dynamically render UI components defined in the components.json file. These components, including buttons, headers, and custom sections, are styled consistently using Tailwind CSS and displayed responsively across screen sizes. The backend APIs efficiently handle operations such as task creation and status updates, with data persistently stored in the SQLite database and reflected instantly on the frontend. The application builds without errors using Vite’s bundler, resulting in a compact, production-ready output. Testing confirms that all scripts (npm run dev, build, preview) execute as expected, and the database initializes correctly on reset. The project also highlights extensibility—developers can add new UI components without modifying source code, making it scalable for larger use cases like template creation or drag-and-drop interfaces. Overall, the system proves to be reliable, fast, and modular, validating its potential as a low-code website generation tool with strong architectural foundations.

