Festo Devops Documentation

Frontend

In our *frontend* folder we have separated the code per page, *chatbox, codeeditor, home,* etc. All of the corresponding code for each page will be located in those folders. We also have a *redux* folder, which was used to send the generated code information between pages. Lastly, we have a *components* folder for *headers*

Front End Tech Stack

- **React** – Core JavaScript library for building UI components.
- **Vite** – Lightning-fast development server and bundler optimized for modern frameworks.
- **Redux** – State management library used to handle global application state, such as the active project and conversation history.
- **Tailwind CSS** – Utility-first CSS framework for rapid styling with responsive design.
- **React Router** – Handles client-side routing for seamless navigation between views.
- **Monaco Editor** – Provides an in-browser code editing experience similar to VS Code.
    - https://www.npmjs.com/package/@monaco-editor/react

- **StackBlitz WebContainer** – WebContainers are a browser-based runtime for executing Node.js applications and operating system commands, entirely inside your browser tab.
    - https://webcontainers.io/
    - https://webcontainer-tutorial.pages.dev/1-webcontainer-api/1-express-app/1-what-we-are-building/
    - https://webcontainers.io/api

Frontend Components

The UI is organized into modular React components, each handling a distinct part of the application's functionality:

- codeEditor/ – Components related to the in-browser code editing interface:

  - CodeEditorBox.jsx: Main Monaco-based editor window for viewing and editing code.
  - Terminal.jsx: (Optional) Terminal-style interface for logs or CLI interactions.
  - FileTree.jsx: Visual file explorer for selecting and navigating code files.
  - Toolbar.jsx: Controls like download/export, run, and layout actions.

- textArea/expandableTextArea.jsx – A dynamic multi-line input component that resizes automatically, used for sending messages in the chat interface
- header/ – Top navigation elements:

```
.
|___codeEditor
| |___CodeEditorBox.jsx
| |___Terminal.jsx
| |___FileTree.jsx
| |___Toolbar.jsx
|___textArea
| |___expandableTextArea.jsx
|___header
| |___Header.jsx
| |___EmptyHeader.jsx
```

Backend Code

Our *backend* folder follows a normal django style structure, where there is another *backend* folder nested inside the first *backend folder*.

```
.
|___db.sqlite3
|___backend
| |___asgi.py
| |___init__.py
| |___utils
| | |_____dynamodb.py
| | |_____openai.py
| |___test.py
| |___settings.py
| |___urls.py
| |___views.py
| |___wsgi.py
|___manage.py
```

Core Technologies

- **Framework**: Django
- **Database**: AWS DynamoDB
- **LLM**: OpenAI Completion APIs (via `openai_util`)
- **Deployment**: AWS Lambda (compatible views)

Key Endpoints

Endpoint	**Method**	**Description**
/api/create-project/	POST	Create a new project with code + conversation
/api/generate-project/	POST	Generate new code/message from chat history
/api/projects/	GET	List all project metadata
/api/project-get	GET	Fetch a project by `projectId`
/api/project-delete	DELETE	Delete a project by `projectId`
/api/add-message	POST	Append a message to a project

Database

Overview

This backend uses Amazon DynamoDB as its primary database. It stores all project data, including project metadata, code files, and chat-based message history. The backend is implemented in Django, and the views are Lambda-compatible via @csrf_exempt.

Table Schema

Table Name: ai_assistant_devops

Attribute	Type	Description
project_id	string	Primary key. UUID representing the unique ID of each project.
name	string	Name of the project. Default: "Untitled Project".
code	dict	JSON structure of all project files. Each file has a name and contents.
conversation	list	Ordered list of messages exchanged
createdAt	number	Unix timestamp in seconds (UTC) when project was created.
updatedAt	number	Unix timestamp in seconds (UTC) when project was last updated.

Backend DynamoDB Functions

1. create_project(name, code, conversation)
- Creates a new item with:
    - Auto-generated project_id (UUID)
    - Initial code and message list
    - createdAt and updatedAt fields

2. get_project(project_id)
- Fetches a single project by ID.

3. list_projects()
- Returns all project metadata in the table (excluding file content by default).

4. update_project_code(project_id, new_code)
- Merges new_code into the existing code map using a custom deep merge strategy.

5. add_message_to_conversation(project_id, role, content, type=None)
- Appends a new message with metadata to the conversation list.
- Example message:

```
json:
{
  "message_id": "uuid-1234",
  "timestamp": 1745519319,
  "role": "user",
  "content": "What should this app do?",
  "type": "code_generation" // optional
}
```

6. update_conversation(project_id, messages)
- Replaces the entire conversation list with a new one (used after regeneration).

7. delete_project(project_id)
- Permanently removes the project from the table.

Installation

Follow these steps to set up the project locally:

1. Clone the repository

```
git clone <repository-url>
```

2. Navigate to the frontend repository

```
cd frontend
```

3. Install dependencies

```
npm install
```

4. Add environment variables to the backend

```
cd backend
touch .env
```

5. Run the development servers (use 2 separate terminals)

Terminal 1 – Backend

```
python3 manage.py runserver
```

Terminal 2 – Frontend

```
cd frontend
npm run dev
```

Deployment

Deploying the frontend is fairly simple, you will need all the files in the *frontend* folder and will need to copy them into the non-company GitLab repo called "*Fst Big Data Platform / fst-students / AI_assisted_DevOp*s." Guilherme Mendes is the owner of this repository. We needed a separate repository as the GitLab yaml file was not working with the Festo GitLab. Once you have the *frontend* folder copied into that repository, it should automatically deploy to AWS Amplify. If you do not have the repository connected to AWS Amplify, create an app in Amplify and link the repository through GitLab.

Deploying the backend is more complicated and has inconsistencies with the API Gateway working. There are three main components to it, you have the AWS Lambda which stores the django backend code. You will need all the files which are located in the *backend* folder. In addition to this, since all the dependencies are over the file size limit, they are copied into an S3 bucket called *ai-assisted-devops-backend-zip*, what you need is the *lambda_layer.zip*, which needs to be connected to AWS Lambda through the *layers* functionality. Lastly, you need to add the endpoints in API Gateway, all these endpoints are located in the *backend* folder in *urls.py*. Currently, the gateways are not very consistent so the setup here might need some changes.