From : Sandip Kothari

[Sandip.kothari@gmail.com](mailto:Sandip.kothari@gmail.com)

Mobile : +91-9885887175

This document covers part 4 and Part 5 of the exercise. The source code correspondent to this document contains in zaniaV3.zip. Although it is extended from Part 1and Part 2 of the exercise, this particular implementation is not backward compatible with Part1 and Part 2. Only Part 3 and Part 4’s execution can be tested. Part 5 has no source code.

Part 1: Front End & Part 2 : Back End sections are not modified, but they are not exactly same as what is required for Part 3 and Part 4 of the exercise.

Contents

[Part 1: Front End & Part 2: Back End 4](#_Toc171080225)

[Step 1: Set up the Backend 4](#_Toc171080226)

[1. First, create a virtual environment and install the necessary packages. 4](#_Toc171080227)

[2. Install necessary dependencies. 4](#_Toc171080228)

[3. Create flask app 4](#_Toc171080229)

[Step 2: Set up the Frontend 4](#_Toc171080230)

[1. Initialize the React Project 4](#_Toc171080231)

[2. Install Necessary Dependencies 5](#_Toc171080232)

[3. Create a JSON File for Static Data 5](#_Toc171080233)

[4. Create Necessary Components 5](#_Toc171080234)

[5. Index Files 5](#_Toc171080235)

[6. Add Thumbnails 5](#_Toc171080236)

[How to run? 5](#_Toc171080237)

[1. Start the Backend Server 5](#_Toc171080238)

[2. Start the Frontend Server 5](#_Toc171080239)

[3. Open the Application in a Browser 5](#_Toc171080240)

[4. Test the Drag-and-Drop Functionality: 6](#_Toc171080241)

[Part 3 : Tying it up! 7](#_Toc171080242)

[Enhancements in Backend 7](#_Toc171080243)

[Enhancements in Frontend 7](#_Toc171080244)

[Overall Explanation 7](#_Toc171080245)

[Modified Code Structure 7](#_Toc171080246)

[Testing the objectives of Task 3 exercise 8](#_Toc171080247)

[1. Call the API to Display the Grid 8](#_Toc171080248)

[2. CORS Configuration 8](#_Toc171080249)

[3. Saving Every Five Seconds 9](#_Toc171080250)

[4. Display Loading Spinner During Save 9](#_Toc171080251)

[5. Display Time Since Last Save 9](#_Toc171080252)

[6. Avoid Saving if No Changes 9](#_Toc171080253)

[Detailed Steps for Testing (for users who doesn’t want to go in details) 9](#_Toc171080254)

[1. Load the Application 9](#_Toc171080255)

[2. Observe Network Requests 9](#_Toc171080256)

[3. Check UI Elements 9](#_Toc171080257)

[4. Confirm No Unnecessary Saves 10](#_Toc171080258)

[Example Workflow 10](#_Toc171080259)

[1. Initial Load 10](#_Toc171080260)

[2. Make Changes 10](#_Toc171080261)

[3. Idle State 10](#_Toc171080262)

[Part 4 : Deployment 11](#_Toc171080263)

[How to meet the objective (Blue print) 11](#_Toc171080264)

[Architecture 11](#_Toc171080265)

[API Design 11](#_Toc171080266)

[Implementation Details 11](#_Toc171080267)

[Folder Structure 11](#_Toc171080268)

[Part 5 : General questions 13](#_Toc171080269)

[Base URL 13](#_Toc171080270)

[Endpoints 13](#_Toc171080271)

[1. Get All Documents 13](#_Toc171080272)

[2. Get a Single Document 13](#_Toc171080273)

[3. Create a New Document 14](#_Toc171080274)

[4. Update an Existing Document 14](#_Toc171080275)

[5. Delete a Document 15](#_Toc171080276)

[API Design Considerations 15](#_Toc171080277)

[Example Error Response 15](#_Toc171080278)

# Part 1: Front End & Part 2: Back End

To build a simple full-stack application with React, Python, and PostgreSQL, we need to complete the following main steps:

1. Set up the backend with Python (using Flask) and PostgreSQL.
2. Create a frontend with React and implement the required functionality.
3. Add a README file to explain how to run the application.

High-level overview of the steps required to accomplish this:

## Step 1: Set up the Backend

We are going to create backend with Python/Flask and PostgreSQL:

### First, create a virtual environment and install the necessary packages.

Below steps are mentioned assuming windows 11 laptop and executed on powershell.

mkdir document\_backend

cd document\_backend

python -m venv venv

.\venv\Scripts\Activate

### Install necessary dependencies.

In document\_backend directory, create a file requirements.txt file and add below lines (text) in this file. This file we can keep on updating in future if there will be a need of new packages.

flask

flask-cors

psycopg2-binary

Now install this dependencies

pip install -r requirements.txt

### Create flask app

Create an app.py file (stored in document\_backend)

## Step 2: Set up the Frontend

We are going to create a full-stack application with React for the frontend.

### Initialize the React Project

Below steps are mentioned assuming windows 11 laptop and executed on powershell.

npx create-react-app document-app

cd document-app

### Install Necessary Dependencies

npm install react-dnd react-dnd-html5-backend react-bootstrap bootstrap

### Create a JSON File for Static Data

Create a file public/documents.json (stored in document-app/public/). This file has static data which is mentioned in the exercise.

### Create Necessary Components

* Create a file named App.js to set up the main application component.
* Create a file named DocumentCard.js for the document card component.
* Create a file named DocumentOverlay.js for the overlay component.
* We need to show the spinner while the image is loading, for that create a file named SpinnerComponent.js and modify the DocumentCard.js accordingly.

### Index Files

Following files are required in the document-app/src directory to bootstrap React application. These files are generated while initializing react project.

* index.js
* index.css : We can add global styles here
* App.css : This file has additional defined styles. By importing App.css file in App.js, the styles defined in App.css will be applied to React components.

### Add Thumbnails

Place thumbnail images for the document types in the public/thumbnail directory. Ensure each image file is named according to the type property in the documents array. Below thumbnails are added.

* bankdraft.png
* bill-of-lading.png
* invoice.png
* bank-draft-2.png
* bill-of-lading-2.png

# How to run?

### Start the Backend Server

cd zania\document\_backend

flask run

### Start the Frontend Server

cd zania\document-app

npm start

### Open the Application in a Browser

Open a web browser and navigate to <http://localhost:3000>.

### Test the Drag-and-Drop Functionality:

* Open your browser and go to http://localhost:3000.
* Try dragging and dropping the cards to reorder them.
* The cards should move to the new positions as you drag and drop them.

# Part 3 : Tying it up!

## Enhancements in Backend

For implementing this, backend needs to be added with endpoints to serve the documents and save the document order. For that Add below API Endpoints. (This modifications are required in app.py)

1. Serve Documents: Add an endpoint to get the documents.
2. Save Document Order: Add an endpoint to save the document order.

## Enhancements in Frontend

Update the frontend to fetch documents from the API, periodically save the document order, and show a loading spinner while saving. This modifications are required in App.js. Also Spinner component (spinner.js) is added. Certain modification are done in app.css to add methods for spinner.js.

## Overall Explanation

* Backend: Added /documents GET and POST endpoints to fetch and save document order.
* Frontend:
  + Fetching Documents: Use fetch to get documents from the backend when the component mounts.
  + Saving Documents: Use setInterval to save documents every 5 seconds if changes are made.
  + Spinner: Display a loading spinner while saving.
  + Last Saved Time: Display how long has passed since the last save.

With these changes, application periodically save the document order to the backend and display appropriate feedback to the user.

## Modified Code Structure

zania/

│

├── document\_backend/

│ ├── app.py

│ ├── requirements.txt

│ └── setup\_database.sql

│

├── document-app/

│ ├── public/

│ │ ├── thumbnails/

│ │ │ ├── bankdraft.png

│ │ │ ├── bill-of-lading.png

│ │ │ ├── invoice.png

│ │ │ ├── bank-draft-2.png

│ │ │ └── bill-of-lading-2.png

│ │ └── ...

│ ├── src/

│ │ ├── components/

│ │ │ ├── CardComponent.js

│ │ │ ├── DocumentCard.js

│ │ │ ├── DocumentOverlay.js

│ │ │ ├── SpinnerComponent.js

│ │ │ ├── SpinnerComponent.css

│ │ │ ├── OverlayComponent.js

│ │ │ ├── OverlayComponent.css

│ │ └── App.js

│ │ └── App.css

│ │ └── index.js

│ │ └── index.css

│ ├── package.json

└── ...

└── README.md (This document PersonalLearningV2.docx)

## Testing the objectives of Task 3 exercise

To test that your objectives are met, follow these steps:

### Call the API to Display the Grid

* **Step:** Open your application in a browser by navigating to http://localhost:3000.
* **Check:** Verify that the documents fetched from the API (http://localhost:5000/documents) are displayed in a grid layout. The documents should be rendered using the CardComponent and should match the data from the API response.

### CORS Configuration

* **Step:** Ensure that your backend API allows requests from any domain and port. Since you mentioned not spending time on this, your backend should already be configured to handle CORS properly.
* **Check:** If you are not encountering any CORS errors in the browser console when your frontend makes API calls to http://localhost:5000, it indicates that CORS is configured correctly.

### Saving Every Five Seconds

* **Step:** Make some changes to the document grid (e.g., reordering the cards).
* **Check:** Open the browser's developer tools (usually by pressing F12 or Ctrl+Shift+I) and go to the "Network" tab. Observe the network requests being made. You should see a POST request to http://localhost:5000/documents every five seconds, provided there are changes to save.

### Display Loading Spinner During Save

* **Step:** Make some changes to the document grid and observe the UI.
* **Check:** Verify that the loading spinner (SpinnerComponent) is displayed whenever a save operation is in progress.

### Display Time Since Last Save

* **Step:** Make some changes to the document grid and let the save operation complete.
* **Check:** Verify that the UI displays how many seconds have passed since the last save. This should update in real-time.

### Avoid Saving if No Changes

* **Step:** After the initial load, do not make any changes to the document grid.
* **Check:** Observe the network requests. There should be no POST requests to http://localhost:5000/documents if no changes have been made.

## Detailed Steps for Testing (for users who doesn’t want to go in details)

### Load the Application

* + Start your frontend application with npm start.
  + Open http://localhost:3000 in your browser.
  + Verify that the document grid is populated with data from the API.

### Observe Network Requests

* + Open browser developer tools and navigate to the "Network" tab.
  + Verify the initial GET request to http://localhost:5000/documents.
  + Make some changes to the document grid (e.g., drag and drop a card to reorder).
  + Verify that a POST request is made every five seconds if there are changes.

### Check UI Elements

* + Make changes to the grid and check for the loading spinner.
  + Observe the "Last saved" time and ensure it updates correctly.

### Confirm No Unnecessary Saves

* + Ensure no changes are made to the grid after the initial load.
  + Verify that no POST requests are made if no changes occur.

## Example Workflow

### Initial Load

* + Open the app.
  + Check that the grid displays documents.

### Make Changes

* + Drag and drop a document to a new position.
  + Observe the network requests and the appearance of the spinner.
  + Confirm the "Last saved" time updates after saving.

### Idle State

* + Do not interact with the grid.
  + Ensure no POST requests are sent without changes.

By following these steps, we can ensure that all objectives are met and the application functions as expected.

# Part 4 : Deployment

## How to meet the objective (Blue print)

To meet the requirements of creating a docker-compose file to start all components as microservices and to write documentation, I will break it down into the following steps:

1. Create Dockerfiles for the Frontend and Backend Services
2. Create a docker-compose.yml File
3. Write Documentation

## Architecture

The application is divided into two main components:

1. Frontend: A React application that provides the user interface and interacts with the backend API to fetch and save documents.
2. Backend: A Flask application that serves the REST API for fetching and saving documents.

## API Design

1. GET /documents: Fetches the list of documents.
2. POST /documents: Saves the updated list of documents.

## Implementation Details

* The frontend periodically saves the state to the backend every five seconds if changes have been made.
* The backend is a simple REST API using Flask.
* The frontend is built using React and uses the react-dnd library for drag-and-drop functionality.

## Folder Structure

The directory structure should look like below

document-app/

│

├── backend/

│ ├── Dockerfile

│ ├── requirements.txt

│ └── app.py

│

├── frontend/

│ ├── Dockerfile

│ ├── package.json

│ ├── public/

│ └── src/

│

└── docker-compose.yml

# Part 5 : General questions

Below a hypothetical API design for a project that allows adding, removing, and updating elements. The design is RESTful, with endpoints for each of the required operations. The data model is based on the documents used in this project.

## Base URL

The base URL for the API can be /api/v1.

## Endpoints

### Get All Documents

* + Endpoint: GET /api/v1/documents
  + Description: Retrieves a list of all documents.
  + Response:

[

{

"id": 1,

"title": "Bank Draft",

"type": "bankdraft",

"position": 0

},

{

"id": 2,

"title": "Bill of Lading",

"type": "bill-of-lading",

"position": 1

},

...

]

### Get a Single Document

* + Endpoint: GET /api/v1/documents/{id}
  + Description: Retrieves a single document by ID.
  + Response

{

"id": 1,

"title": "Bank Draft",

"type": "bankdraft",

"position": 0

}

### Create a New Document

* + Endpoint: POST /api/v1/documents
  + Description: Adds a new document.
  + Request Body:

{

"title": "New Document",

"type": "new-type",

"position": 5

}

* + Response:

{

"id": 6,

"title": "New Document",

"type": "new-type",

"position": 5

}

### Update an Existing Document

* + Endpoint: PUT /api/v1/documents/{id}
  + Description: Updates an existing document by ID.
  + Request Body:

{

"title": "Updated Document",

"type": "updated-type",

"position": 2

}

* + Response:

{

"id": 1,

"title": "Updated Document",

"type": "updated-type",

"position": 2

}

### Delete a Document

* + Endpoint: DELETE /api/v1/documents/{id}
  + Description: Deletes a document by ID.
  + Response:

{

"message": "Document deleted successfully."

}

## API Design Considerations

1. **Consistency:** Follow RESTful principles, ensuring each endpoint has a clear and consistent purpose.
2. **Scalability:** Design the API to handle future additions without breaking existing functionality. Versioning the API (/api/v1) allows for this.
3. **Error Handling:** Include meaningful error messages and status codes (e.g., 400 Bad Request, 404 Not Found, 500 Internal Server Error) for robustness.
4. **Security:** Implement authentication and authorization mechanisms to protect the API endpoints.
5. **Validation:** Validate incoming data to prevent invalid data from being processed.
6. **Pagination:** For GET /api/v1/documents, consider adding pagination parameters to handle large datasets efficiently.
7. **Rate Limiting:** Implement rate limiting to protect the API from abuse and ensure fair usage.

## Example Error Response

For invalid data in a POST or PUT request:

{

"error": "Invalid data",

"message": "The 'title' field is required."

}