This LLD provides a detailed breakdown of the modules, functions, scripts, and database structures necessary to implement the flight route optimization project

1. Data Ingestion and Storage:

1.1. CSV Data Cleaning:
- **Objective**: Clean and preprocess the flight route data from a CSV file.
- Script: `data_cleaning.py`
- Functions:
  - load_and_clean_data(input_file, output_file): This function reads the raw data from `input_file`, selects relevant columns (`Origin`, `Dest`, `Distance`), drops rows with missing values, and saves the cleaned data to `output_file`.

```[Pseudo Code]
function load_and_clean_data(input_file, output_file):
    read data from input_file into dataframe
    select columns Origin, Dest, Distance
    drop rows with null values in selected columns
    save cleaned dataframe to output_file
```

1.2. Database Storage:
- **Objective**: Store the cleaned flight route data in a PostgreSQL database.
- Script: `db_setup.py`
- Functions:
  - create_routes_table(): Creates the `routes` table in the database if it doesn't already exist.
  - insert_routes_from_csv(csv_file): Reads the cleaned data from `csv_file` and inserts each row into the `routes` table.

```[Pseudo Code]
function create_routes_table():
    connect to PostgreSQL database
    execute SQL to create routes table if not exists
    commit changes and close connection

function insert_routes_from_csv(csv_file):
    connect to PostgreSQL database
    read data from csv_file into dataframe
    for each row in dataframe:
        execute SQL to insert row into routes table
    commit changes and close connection
```

2. Route Planning:

2.1. Dijkstra's Algorithm:
- **Objective**: Calculate the shortest path between two airports using Dijkstra's algorithm.
- Module: `dijkstra.py`
- Functions:
  - dijkstra(graph, start, end): Implements Dijkstra's algorithm. It takes a graph representation of airports and routes, a start airport, and an end airport, and returns the shortest path and its cost.

```[Pseudo Code]
function dijkstra(graph, start, end):
    initialize priority queue with (0, start, [])
    initialize seen set
    while queue is not empty:
        pop (cost, node, path) from queue
        if node is in seen:
            continue
        append node to path
        add node to seen
        if node equals end:
            return (cost, path)
        for each neighbor of node in graph:
            if neighbor is not in seen:
                push (cost + distance to neighbor, neighbor, path) to queue
    return (infinity, [])
```

2.2. Graph Building:
- **Objective**: Build a graph representation of airports and routes from the database.
- Module: `graph_builder.py`
- Functions:
  - build_graph(): Connects to the PostgreSQL database, fetches route data, and constructs a graph where each airport is a node and each route is an edge with a weight representing the distance.

```[Pseudo Code]
function build_graph():
    connect to PostgreSQL database
    execute SQL to fetch all rows from routes table
    initialize empty graph
    for each row in fetched data:
        add route (destination, distance) to graph[origin]
```

```
    close connection
    return graph
```

3. Web Interface:

3.1. Flask Application:
- **Objective**: Provide a web interface for users to input origin and destination airports and get the shortest route.
- Script: `app.py`
- Functions:
  - home(): Renders the home page with an input form.
  - get_route(): Handles POST requests, reads origin and destination from the request, calculates the shortest route using Dijkstra's algorithm, and returns the result as a JSON response.

```[Pseudo Code]
function home():
    render index.html template

function get_route():
    read JSON data from request
    extract origin and destination from data
    if origin or destination is missing:
        return error response
    calculate cost and path using dijkstra(graph, origin, destination)
    if cost is infinity:
        return error response
    return JSON response with cost and path
```

3.2. HTML Template:
- **Objective**: Provide a user-friendly interface for inputting origin and destination airports.
- File: `templates/index.html`

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Flight Route Optimizer</title>
  </head>
  <body>
    <h1>Flight Route Optimizer</h1>
```

```
        <form id="routeForm">
          <label for="origin">Origin:</label>
          <input type="text" id="origin" name="origin">
          <label for="destination">Destination:</label>
          <input type="text" id="destination" name="destination">
          <button type="submit">Get Route</button>
        </form>
        <div id="result"></div>
        <script>
          document.getElementById('routeForm').addEventListener('submit', function(event) {
            event.preventDefault();
            const origin = document.getElementById('origin').value;
            const destination = document.getElementById('destination').value;
            fetch('/route', {
              method: 'POST',
              headers: {
                'Content-Type': 'application/json'
              },
              body: JSON.stringify({ origin, destination })
            })
            .then(response => response.json())
            .then(data => {
              if (data.error) {
                document.getElementById('result').innerText = data.error;
              } else {
                document.getElementById('result').innerText = `Cost: ${data.cost}, Path:
${data.path.join(' -> ')}`;
              }
            });
          });
        </script>
      </body>
    </html>
    ```
```

**4. Database Structure:**

**4.1. Table Definition:**

- Table: `routes`
  - Columns:
    - `id` (SERIAL PRIMARY KEY): Unique identifier for each route.
    - `origin_airport` (VARCHAR(3)): Airport code for the origin.
    - `destination_airport` (VARCHAR(3)): Airport code for the destination.
    - `distance` (INTEGER): Distance between origin and destination airports.
```

```sql
CREATE TABLE IF NOT EXISTS routes (
    id SERIAL PRIMARY KEY,
    origin_airport VARCHAR(3),
    destination_airport VARCHAR(3),
    distance INTEGER
);
```

5. Deployment:

5.1. Database:
- Ensure PostgreSQL is installed and running.
- Create and configure the `route_opti` database with necessary tables.

5.2. Application:
- Use a web server like Gunicorn for deploying the Flask app.
- Configure environment variables for database connection details.
- Ensure the web server is correctly configured to serve the Flask application.

6. Error Handling and Logging:

6.1. Error Handling:
- Implement try-except blocks around database connections and critical operations.
- Return meaningful error messages for invalid inputs or failed operations.

6.2. Logging:
- Log errors and significant events to a file or monitoring service for debugging and maintenance.

.