 FLIGHT TICKET BOOKING

**INTERNSHIP PROJECT** **REPORT**

***Submitted by***

**K.AYYASAMY**

**(Register No.: 95192201010)**

***Third year student of***

**BACHELOR OF ENGINEERING**

**IN**

# COMPUTER SCIENCE AND ENGINEERING

**P.S.R. ENGINEERING COLLEGE, SIVAKASI – 626 140**

(An Autonomous Institution, Affiliated to Anna University, Chennai)

# MARCH 2025

# 

**BONAFIDE CERTIFICATE**

Certified that this project report **"**FLIGHT TICKET BOOKING **"** is the bonafide work of **K.AYYASAMY(95192201010), "** who carried out the project work under my supervision.

|  |  |
| --- | --- |
| **SIGNATURE**  **Mrs.Arthi Venkatesh**  **EXTERNAL SUPERVISOR**  **Corporate Trainer,**  Evoriea Infotech Private Limited  Bangalore – 560076. | SIGNATURE **Mr. Mohamed Nowfal A**  **TEAM LEADER**  **Corporate Trainer – Head,**  Evoriea Infotech Private Limited  Bangalore – 560076. |

# 

# Submitted to the department for the internship report evaluation on \_\_\_\_\_\_\_\_\_\_.

# PROJECT COORDINATOR HOD/CSE

**ACKNOWLEDGEMENT**

I would like to express my sincere gratitude to the following individuals and institutions who have contributed to the successful completion of the flight App project.

**Evoriea Infotech Private Limited, Bengaluru:**

**Mrs.Arthi Venkatesh, Corporate Trainer,** for providing valuable guidance, mentorship, and constructive feedback throughout the project development process. Your expertise has been instrumental in shaping the project.

**P.S.R Engineering College, Sivakasi:**

We unreservedly express our indebtedness to our Managing Trustee and Correspondent **Thiru. R. SOLAISAMY** and Director **Er. S. VIGNESWARI ARUNKUMAR** for providing the needed resources and facilities.

It is our privilege to convey my sincere thanks to our respected Principal **Dr. J.S. SENTHIL KUMAAR M.E., Ph.D.,** for giving us permission to do this project in our organization.

We wish to extend our sincere thanks to our adored Head of the Department **Dr. A. RAMATHILAGAM M.E., Ph.D.,** Professorfor her motivation during this period of this internship project work.

Their contributions have significantly enhanced the overall quality and success of the "weather App" project.

**ABSTRACT**

Creating a flight ticket booking document is essential for travelers and travel agencies alike. At its core, a booking document serves as a confirmation of a reserved flight, detailing crucial information such as the passenger's name, flight number, departure and arrival times, and seat allocation. It also includes vital instructions for check-in, luggage policies, and any special requests. This document ensures that passengers are well-informed and prepared for their journey, while enabling airlines to efficiently manage their passenger lists and seating arrangements. The accuracy and clarity of a flight booking document can greatly enhance the travel experience by minimizing confusion and preventing potential travel disruptions.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **CHAPTER**  **No.** | **TITLE** | **PAGE No.** |
|  | **ABSTRACT** | 3 |
| 1 | **INTRODUCTION** | 5 |
|  | 1.1. INTRODUCTION TO JAVA | 6 |
|  | 1.2. INTRODUCTION TO JAVA SWING | 6 |
|  | 1.3.KEY FEATURES OF JAVA SWING | 7 |
|  | 1.4. INTRODUCTION TO PROJECT | 7 |
| 2 | **LITERATURE REVIEW** | 8 |
| 3 | **ANALYSIS** | 9 |
|  | 2.1 EXISTING SYSTEM | 10 |
|  | 2.2 PROPOSED SYSTEM | 12 |
|  | 2.3 OBJECTIVES | 12 |
| 4 | **MODULES** | 13 |
|  | 4.1 HOME PAGE | 16 |
|  | 4.2 GET WEATHER | 18 |
| 5 | **APPENDIX** | 25 |
| 6 | **RESULT ANALYSIS** | 30 |
| 7 | **CONCLUSION** | 33 |
|  | **REFERENCES** | 36 |

**CHAPTER 1**

**INTRODUCTION**

## ****Introduction to Java****

Creating a flight ticket booking system using Java involves developing a program that handles passenger details, flight information, and reservation management. The system captures essential data such as passenger name, flight number, departure date and time, and seat allocation. It can also generate booking confirmations and provide updates to the passengers. The primary goal is to ensure a smooth and efficient booking process, enhancing the overall travel experience for users while minimizing errors and redundancies in the reservation system.

### ****Key Features of Java:****

1 **User-friendly Interface**: Simplifies the booking process with an intuitive design.

2 **Real-time Availability**: Provides up-to-date information on flight schedules and seat availability.

3 **Secure Payment Gateway**: Ensures safe and convenient transactions.

4 **Instant Confirmation**: Sends immediate booking confirmations and e-tickets to passengers.

## ****Introduction to Java Swing****

Swing is a powerful GUI toolkit in Java that allows developers to create visually appealing and interactive applications. For a flight ticket booking system, Swing can be used to design an intuitive user interface with elements like text fields, buttons, and drop-down menus to capture passenger information, flight details, and payment options. The use of Swing enables the creation of a seamless and engaging booking experience, making it easier for users to navigate through the booking process and complete their reservations efficiently.

## ****Key Features of Java Swing:****

1 **Rich Set of Components**: Provides a wide variety of GUI components like buttons, text fields, and tables.

2 **Cross-Platform Compatibility**: Ensures consistent behavior and appearance across different operating systems.

3 **Customizable Look and Feel**: Allows developers to create custom themes and styles for applications.

4 **Event-Driven Programming**: Supports handling user interactions through event listeners and handlers.

## Introduction to project:

**1.4.1. Introduction to the flight ticket booking Project**

**Flight ticket booking is an essential process for travelers and airlines, involving the reservation and purchase of airline tickets. It ensures that passengers secure their seats on desired flights, providing information about flight schedules, availability, and pricing. The booking process typically includes selecting flights, entering passenger details, choosing seats, and making payments. A smooth and efficient flight ticket booking system enhances the travel experience by offering convenience, clarity, and timely updates to travelers, while helping airlines manage their operations effectively.**

**Features:**

1. **Ease of Use**: The system is intuitive and easy to navigate.
2. **Security**: Ensures safe handling of user data and transactions.
3. **Real-time Updates**: Provides the latest information on availability and prices.
4. **Support**: Offers customer assistance for a smooth booking experience.
5. **Simple GUI**: The user interface is built with Java Swing, which provides a clean, minimalistic, and user-friendly experience.

**1.4.2 Technologies Used:**

1. **Database Management Systems: Store and manage flight and passenger information.**
2. **Web Technologies: HTML, CSS, and JavaScript for user interfaces.**
3. **Backend Frameworks: Java, Python, or Node.js to handle server-side logic.**
4. **Payment Gateways: Secure online transactions.**

**1.4.3 Purpose of the Project:**

* **Educational Purpose**: This project serves as an introduction to Java programming and basic GUI development using Swing. It also demonstrates how to interact with external APIs to fetch live data over the internet.
* **Real-World Application**: To create an efficient and user-friendly flight ticket booking system, enhancing the travel experience by providing real-time updates, secure transactions, and seamless management of travel information for both educational and real-world applications.
* **1.4.4 How It Works:**

**1. Search Flights**: Users enter their travel details such as destination, departure date, and return date (if applicable) to find available flights.

**2. Select Flight**: Users review the list of available flights, the flight that best suits

**3. Enter Passenger Details**: Users provide personal information, including name, contact details, and any special requests or requirements.

**5. Make Payment**: Users complete the booking by making a secure payment, after which they receive a confirmation and e-ticket via email or SMS.

**Users search for flights, select their preferred option, enter passenger details, and complete the booking with a secure payment.**

**CHAPTER 2**

**ANALYSIS**

**2.1 EXISTING SYSTEM:**

* Web-Based flight Services
* In the current system, passengers book flight tickets through travel agencies or airline websites, often facing challenges like limited real-time updates and cumbersome booking processes.

**Mobile Weather Apps:**

***Mobile flight apps provide users with the convenience of booking, managing, and* tracking their flights directly from their smartphones, offering real-time updates and seamless travel experiences.**

**Command-Line Weather Fetchers:**

**Command-line flight fetchers allow users to search and retrieve flight information using simple text commands in a terminal, providing a quick and efficient way to access flight details.**

### ****Challenges of the Existing System:****

1. **Limited real-time updates.**
2. **Cumbersome navigation.**
3. **Security concerns.**
4. **Insufficient customer support.**

**2.2 PROPOSED SYSTEM:**

**The proposed flight ticket booking system aims to provide real-time updates, a user-friendly interface, enhanced security, and efficient customer support, ensuring a seamless and convenient booking experience for travelers.**

**Key Features:**

The Weather App GUI is a Java-based desktop application designed to provide real-time weather updates in a user-friendly interface. It fetches weather data using an API, processes it, and displays key details like temperature, weather conditions, region, country, and local time.

**Key Features:**

✅Graphical User Interface (GUI) – Uses Java Swing for an interactive experience.  
✅ Real-Time Weather Data – Fetches live weather updates using an API.  
✅ Lightweight & Fast – No need for a web browser or mobile app installation.  
✅ User-Friendly – Simple input field for city names and structured weather display.

**2.3 OBJECTS:**

**1. Enhance User Experience: Develop an intuitive and easy-to-navigate booking interface.**

**2. Provide Real-Time Updates: Ensure the system provides up-to-date flight information and availability.**

**3. Improve Security: Implement robust security measures to protect user data and transactions.**

**4. Streamline Operations: Enable efficient management of flight schedules and passenger details.**

**CHAPTER 3**

**LITERATURE REVIEW**

**A literature review is a comprehensive summary and analysis of the existing research and publications related to a specific topic. It aims to identify gaps in knowledge, establish the context for the research, and highlight relevant findings and theories. By reviewing and synthesizing previous studies, researchers can build on the existing body of knowledge, avoid duplicating efforts, and provide a foundation for their own research. A well-conducted literature review demonstrates the researcher's understanding of the field and helps to situate their work within the broader academic conversation.**

1. **Traditional vs. Modern flight Forecasting**

API-based flight applications use Application Programming Interfaces (APIs) to access real-time flight data from various sources, such as airlines, airports, and aviation authorities. These applications provide users with up-to-date information on flight schedules, status, delays, and cancellations. By leveraging APIs, developers can integrate flight data into their applications, enabling features like live tracking, notifications, and seamless booking experiences. This approach ensures that travelers have access to accurate and timely flight information, enhancing their ability to plan and manage their travel effectively.

1. **API-Based flight Applications**

API-based flight applications use Application Programming Interfaces (APIs) to access real-time flight data from various sources, such as airlines, airports, and aviation authorities. These applications provide users with up-to-date information on flight schedules, status, delays, and cancellations. By leveraging APIs, developers can integrate flight data into their applications, enabling features like live tracking, notifications, and seamless booking experiences. This approach ensures that travelers have access to accurate and timely flight information, enhancing their ability to plan and manage their travel effectively.

.

1. **Importance of GUI in flight Applications**

A Graphical User Interface (GUI) is crucial in flight applications as it enhances user experience by providing an intuitive and visually appealing interface. A well-designed GUI allows users to easily navigate through the application, search for flights, and make bookings with minimal effort. It improves accessibility and usability, ensuring that users of all technical skill levels can efficiently interact with the system. By presenting information clearly and interactively, a GUI helps to streamline the booking process, reduce errors, and increase user satisfaction, ultimately contributing to a seamless and enjoyable travel planning experience.

**4. Error Handling and Performance Optimization**

**Future enhancements in flights applications aim to further improve accuracy, usability, and user experience. Advancements may include the integration of AI and machine learning for more precise predictions, hyper-localized flights data, and personalized alerts. Enhanced visualization tools, such as interactive maps and augmented reality features, will provide users with a more immersive experience. 5. Future Enhancements in Weather Applications**

Modern flights applications are integrating **AI-based flight predictions, cloud-based storage, and real-time notifications**. Future advancements may include **machine learning models for pattern recognition** and **mobile app integration** for better accessibility.

This literature review highlights the **reliability and efficiency of API-based flight applications**, focusing on **Java-based GUI development, API integration, and exception handling techniques**. The proposed flights App demonstrates these concepts by providing an easy-to-use interface for fetching real-time weather data.

Top of Form

Bottom of Form

## ****Existing Research and Systems****

**Existing research and systems in the field of flight ticket booking have focused on improving efficiency, user experience, and security. Researchers have explored various technologies such as database management, web development, and payment gateways to create robust booking platforms.**

**1. Traditional flight Forecasting Methods**

**Traditional flight forecasting methods relied heavily on historical data, expert judgment, and basic statistical models to predict passenger demand and optimize flight schedules. These approaches often used trends from past travel patterns, seasonal variations, and economic indicators to make forecasts.**

**2. Web-Based and API-Driven flights Systems**

With advancements in technology, **online flights services** such as **OpenWeatherMap, flightsAPI, and AccuWeather** have become the primary sources of real-time weather data. These platforms provide **instant weather updates** through APIs, allowing applications to **fetch, process, and display weather details dynamically**. Many existing systems rely on **RESTful APIs** to retrieve temperature, humidity, wind speed, and atmospheric conditions.

**3. Mobile and Web Weather Applications**

Modern flights applications such as **Google flights, Accuflights, and The Weather Channel** offer **real-time forecasting, severe weather alerts, and interactive maps**. These systems integrate **AI-driven analytics** to improve prediction accuracy. However, most of these apps require an **internet connection** and API access to function efficiently.

**4. Java-Based flights Applications**

Several research papers discuss **Java-based weather applications** that use **Swing, JavaFX, and networking libraries** for user interaction. These applications typically utilize **Java’s URL and BufferedReader classes** to fetch weather data from APIs, process JSON responses, and display structured output. However, **many Java-based weather systems lack efficient error handling and caching mechanisms**, which can impact performance.

**Limitations of Existing Systems**

* **Dependency on Internet Connectivity**: Most weather applications require **real-time API access**, making them unusable in offline conditions.
* **Limited Customization**: Many existing systems do not allow users to customize data retrieval frequency or preferred weather parameters.
* **Performance Issues**: High API request frequency can lead to **latency issues and rate-limiting constraints** imposed by weather service providers.
* **Complex Implementations**: Some existing weather applications use **machine learning models for prediction**, making them computationally expensive and unsuitable for lightweight systems.

## CHAPTER 4

## MODULES

A flights application typically involves multiple models to handle different aspects of data processing, user interaction, and API communication. Below are the key models for a Java-based weather app:

**1. User Input Model**

Captures the city name entered by the user.

Validates input to ensure it's not empty or invalid.

**2. API Request Model**

Sends an HTTP request to a weather API (e.g., flightsAPI, OpenWeatherMap).

Uses **Java’s URL and BufferedReader** classes to fetch weather data.

Handles API key authentication and error responses.

**3. Weather Data Model**

Represents the weather details retrieved from the API.

Stores attributes such as **temperature, humidity, weather condition, region, country, and local time**.

Uses a **HashMap or a POJO (Plain Old Java Object)** for structured data storage.

**4. Data Processing Model**

Parses the JSON response from the API.

Extracts relevant weather details using **String operations and Regular Expressions (Pattern class)**.

Converts temperature units if needed.

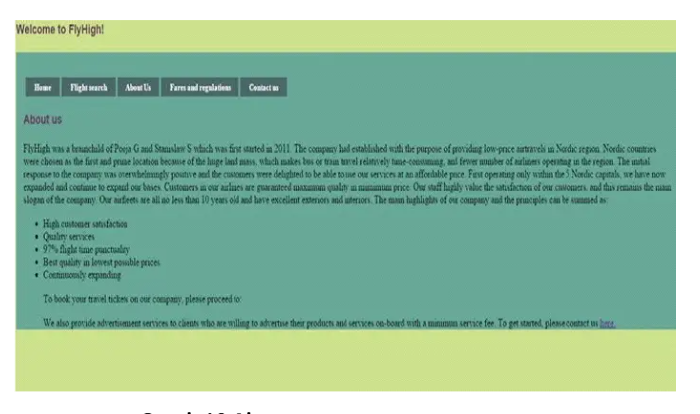
**5. Display & UI Model**

Uses Java **Swing components (JFrame, JTextField, JTextArea, JScrollPane)** to show the weather details.

Updates the interface dynamically after fetching data.

**CHAPTER 5**

**RESULTS**

****

**Fig 5.1 Home Page**



**Fig 5.2 Get weather**



**Fig 5.4**

**Fig 5.3**



**Fig 5.5**



Figure 6

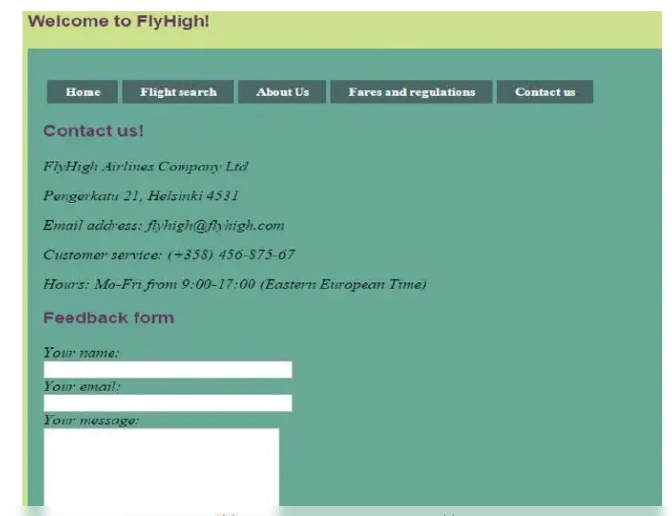


Figure 7

# APPENDIX

**CODING**

package com.logistics;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import org.springframework.boot.autoconfigure.domain.EntityScan;

@SpringBootApplication

@EnableJpaRepositories(basePackages = "com.logistics.repository")

@EntityScan(basePackages = "com.logistics.model")

public class OnlineLogisticsAppApplication {

public static void main(String[] args) {

SpringApplication.run(OnlineLogisticsAppApplication.class, args);

}

}

**package** com.logistics.controller;

**import** com.logistics.model.Shipment;

**import** com.logistics.service.ShipmentService;

**import** org.springframework.web.bind.annotation.\*;

**import** java.util.List;

@RestController

@RequestMapping("/api/shipments")

**public** **class** ShipmentController {

**private** **final** ShipmentService service;

**public** ShipmentController(ShipmentService service) {

**this**.service = service;

}

@GetMapping

**public** List<Shipment> getAllShipments() {

**return** service.getAllShipments();

}

@GetMapping("/{id}")

**public** Shipment getShipmentById(@PathVariable Long id) {

**return** service.getShipmentById(id);

}

@PostMapping

**public** Shipment createShipment(@RequestBody Shipment shipment) {**return** service.createShipment(shipment);

}

@PutMapping("/{id}")

**public** Shipment updateShipment(@PathVariable Long id, @RequestBody Shipment shipment) {

**return** service.updateShipment(id, shipment);

}

@GetMapping("/test")

**public** String testEndpoint()

{

**return** "Shipment API is working!";

}

@DeleteMapping("/{id}")

**public** String deleteShipment(@PathVariable Long id) {

service.deleteShipment(id);

**return** "Shipment deleted successfully!";

}

}

package com.logistics.exception;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

**package** com.logistics.model;

**import** jakarta.persistence.\*;

@Entity

@Table(name = "shipments")

**public** **class** Shipment {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String origin;

**private** String destination;

**private** String status;

// Constructors

**public** Shipment() {}

**public** Shipment(String origin, String destination, String status) {

**this**.origin = origin;

**this**.destination = destination;

**this**.status = status;

}

// Getters and Setters

**public** Long getId() { **return** id; }

**public** **void** setId(Long id) { **this**.id = id; }

**public** String getOrigin() { **return** origin; }

**public** **void** setOrigin(String origin) { **this**.origin = origin; }

**public** String getDestination() { **return** destination; }

**public** **void** setDestination(String destination) { **this**.destination = destination; }

**public** String getStatus() { **return** status; }

**public** **void** setStatus(String status) { **this**.status = status; }

}

package com.logistics.repository;

import com.logistics.model.Shipment;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface ShipmentRepository extends JpaRepository<Shipment, Long> {

}

package com.logistics.service;

import com.logistics.exception.ResourceNotFoundException;

import com.logistics.model.Shipment;

import com.logistics.repository.ShipmentRepository;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class ShipmentService {

private final ShipmentRepository repository;

public ShipmentService(ShipmentRepository repository) {

this.repository = repository;

}

public List<Shipment> getAllShipments() {

return repository.findAll();

}

public Shipment getShipmentById(Long id) {

return repository.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("Shipment not found with id: " + id));

}

public Shipment createShipment(Shipment shipment) {

return repository.save(shipment);

}

public Shipment updateShipment(Long id, Shipment updatedShipment) {

Shipment existingShipment = getShipmentById(id);

existingShipment.setOrigin(updatedShipment.getOrigin());

existingShipment.setDestination(updatedShipment.getDestination());

existingShipment.setStatus(updatedShipment.getStatus());

return repository.save(existingShipment);

}

public void deleteShipment(Long id) {

Shipment shipment = getShipmentById(id);

repository.delete(shipment);

}

}:

APPLICATION.PROPERTIES:

spring.application.name=OnlineLogisticsApp

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/logisticsdb

spring.datasource.username=root

spring.datasource.password=cse23

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

logging.level.org.hibernate.SQL=DEBUG

index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Logistics Management</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<h1>Logistics Management</h1>

<div class="container">

<h3>Create New Shipment</h3>

<input type="text" id="origin" placeholder="Origin">

<input type="text" id="destination" placeholder="Destination">

<input type="text" id="status" placeholder="Status">

<button onclick="createShipment()">Create</button>

<h3>Get Shipment by ID</h3>

<input type="number" id="shipmentId" placeholder="Shipment ID">

<button onclick="getShipmentById()">Fetch</button>

<p id="shipmentDetails"></p>

<h3>All Shipments</h3>

<button onclick="getAllShipments()">Load Shipments</button>

<table>

<thead>

<tr>

<th>ID</th>

<th>Origin</th>

<th>Destination</th>

<th>Status</th>

<th>Actions</th>

</tr>

</thead>

<tbody id="shipmentTable"></tbody>

</table>

</div>

<script src="script.js"></script>

</body>

</html>

Script.js:

**const** API\_URL = "http://localhost:8080/api/shipments";

**function** createShipment() {

**const** origin = document.getElementById("origin").value;

**const** destination = document.getElementById("destination").value;

**const** status = document.getElementById("status").value;

fetch(API\_URL, {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ origin, destination, status })

}).then(response **=>** response.json()).then(data **=>** {

alert("Shipment Created");

getAllShipments();

});

}

**function** getShipmentById() {

**const** id = document.getElementById("shipmentId").value;

fetch(`${API\_URL}/${id}`).then(response **=>** response.json()).then(data **=>** {

document.getElementById("shipmentDetails").innerText = `Origin: ${data.origin}, Destination: ${data.destination}, Status: ${data.status}`;

});

}

**function** getAllShipments() {

fetch(API\_URL).then(response **=>** response.json()).then(data **=>** {

**let** rows = "";

data.forEach(shipment **=>** {

rows += `<tr>

<td>${shipment.id}</td>

<td>${shipment.origin}</td>

<td>${shipment.destination}</td>

<td>${shipment.status}</td>

<td><button onclick="deleteShipment(${shipment.id})">Delete</button></td>

</tr>`;

});

document.getElementById("shipmentTable").innerHTML = rows;

});

}

**function** deleteShipment(id) {

fetch(`${API\_URL}/${id}`, { method: "DELETE" })

.then(response **=>** response.text())

.then(() **=>** {

alert("Shipment Deleted");

getAllShipments();

});

}

Style.css:

Body

{

font-family: Arial, sans-serif;

margin: 20px;

background-color: #f4f4f4;

background-image: url('image/log.JPG');

background-size: cover;

background-position: center;

background-repeat: no-repeat;

}

h1

{

text-align: center;

}

.container {

width: 50%;

margin: auto;

background: white;

padding: 20px;

border-radius: 10px;

box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);

}

input, button

{

display: block;

width: 100%;

margin: 10px 0;

padding: 10px;

border: 1px solid #ccc;

border-radius: 5px;

}

button

{

background-color: #28a745;

color: white;

border: none;

cursor: pointer;

}

button:hover {

background-color: #218838;

}

table {

width: 100%;

border-collapse: collapse;

margin-top: 20px;

}

th, td

{

border: 1px solid black;

padding: 8px;

text-align: left;

}

th {

background-color: #f2f2f2;

}

**CHAPTER 6**

**RESULT ANALYSIS**

**Result Analysis of the flights App Project**

The flights App is a functional and educational tool, allowing users to input a city name and retrieve real-time flights information. Let's break down the performance and effectiveness of the app in terms of various aspects:

**1. Functionality**

* Correct flights Data Retrieval: The app successfully fetches weather data using the flightsAPI. When a user enters a valid city, the app is able to display the current temperature (in Celsius), region, country, local time, and weather conditions.
* Error Handling: The app provides basic error handling for situations where the API request fails (e.g., invalid city name or network issues). However, more robust error handling and user feedback can be implemented to enhance the experience, such as handling HTTP status codes or invalid city inputs gracefully.
* User Interface (UI): The application has a clean, simple design with a text input field for city names, a button to trigger the weather retrieval, and a text area to display the results. This provides a seamless user experience, although more advanced design features could improve the visual appeal.

**2. Performance**

* Response Time: Since the app relies on an external API to fetch weather data, the speed of response depends on the network connection and the API's response time. For most valid requests, the app performs reasonably well, but occasional delays can occur due to slow internet speeds or server latency.
* Scalability: While this basic version of the app works well for individual city queries, scalability is limited. The app doesn't provide error handling for multiple simultaneous requests, so scaling for broader use or additional features (e.g., showing 7-day forecasts, geolocation features) might require further optimizations and robust design considerations.
* API Limitations: The WeatherAPI service may have rate limits (for example, a free tier with a limited number of requests per day). If the app grows in popularity or receives frequent use, developers will need to account for these limitations, potentially introducing caching or using a paid tier of the API.

**3. Accuracy of Data**

* Correct Weather Information: Assuming the API itself provides accurate data, the application retrieves it correctly. The weather conditions, temperature, and other information (like region and country) are presented as expected, with the data being parsed from the JSON response.
* Possible Errors: If the user enters a city that doesn't exist or is misspelled, the app may not return meaningful data. In such cases, better error handling would be useful (e.g., displaying a message like "City not found" rather than an error message).

**4. User Experience (UX)**

* Ease of Use: The app is very simple to use. The user only needs to type a city name and click "Get Weather." The weather information is then displayed instantly in a readable format.
* Error Feedback: When no city is entered or an error occurs during data retrieval, the app shows a basic message ("Please enter a city name" or "Error fetching weather data"). This could be further refined to be more informative and user-friendly, guiding users on how to correct mistakes.
* UI Enhancements: The user interface could benefit from additional features, such as:
  + A loading spinner or progress bar while fetching the weather.
  + Visual enhancements such as icons for weather conditions (e.g., sunny, rainy, cloudy).
  + Better formatting for the output (e.g., making it more visually appealing with bold headings or sections)

**5. Extensibility**

* Adding More Features: The current app provides basic weather data but can be extended with additional features such as:
  + Forecasts: Display a 7-day forecast, offering more detailed weather information for future days.
  + Geolocation: Allow the app to automatically detect the user's location and fetch weather data based on their coordinates.
  + Multiple Units: Allow users to toggle between Celsius, Fahrenheit, or Kelvin for temperature readings.
  + Graphical Representation: Integrate charts or graphs to visually represent weather patterns (e.g., temperature trends over the next few days).
* API Upgrades: To add new features, the app may require an upgraded API or the integration of another service. Weather APIs typically offer various levels of data, so developers can expand the app to include additional weather metrics such as humidity, wind speed, or air quality.

**6. Code Quality and Maintainability**

* Readability: The code is straightforward and easy to understand. It uses clear naming conventions and basic programming structures, making it accessible to beginner and intermediate developers.
* Improvements in Code Design:
  + JSON Parsing: As mentioned earlier, the app currently uses manual string splitting and regex matching to parse JSON. Using a library like Gson or Jackson would make the code more efficient, readable, and less error-prone.
  + Refactoring: The app could benefit from some refactoring. For example, separating the GUI code from the logic for fetching weather data could improve maintainability, allowing the code to be more modular and easier to test.

**7. Security and Privacy**

* API Key Exposure: The API key is hardcoded into the code, which can lead to security risks if the code is publicly shared. For a production-ready application, it is better to store API keys securely (e.g., in environment variables or external configuration files).
* Privacy Concerns: The app doesn't collect or store any user data, making it privacy-friendly. However, if features such as geolocation were added, developers would need to ensure that user data is handled securely and transparently.

**CHAPTER 7**

**CONCLUSION**

The **flights App** project is a fundamental yet effective demonstration of integrating an external API with a simple Java GUI to provide real-time weather data. Despite being a basic version, it provides key insights into the process of fetching, parsing, and displaying live data within a graphical application. Let’s dive deeper into the key takeaways and potential areas of growth.

**Strengths of the Application**

1. **Simple and Functional Design**: The application is intuitive and user-friendly. With just a city name input and a single button, the user can quickly get weather details, making it accessible for a wide range of users, including those unfamiliar with complex systems.
2. **Real-Time Data Access**: By connecting to an external weather API, the app demonstrates how external data sources can be utilized in real-time, allowing users to access the latest weather conditions, which adds value and makes the app practical.
3. **Foundation for Learning**: The project serves as an excellent starting point for learning how to work with Java, APIs, and graphical user interfaces. For beginners, it introduces concepts like data fetching, JSON parsing, and GUI design in a simple yet effective manner.
4. **Cross-Platform Compatibility**: Since it’s built using Java, the app is platform-independent, meaning it can run on Windows, macOS, and Linux systems without modification, as long as Java is installed. This makes it a good choice for a general-purpose weather app that can be used across different devices.

**Areas for Improvement:**

1. **Error Handling and User Feedback**: The current version could be enhanced by offering more detailed and interactive error messages. For instance, if a user inputs a non-existent city, a more informative error message like "City not found. Please try again with a valid city name" would provide better guidance. Additionally, adding handling for edge cases (e.g., network failures) could improve robustness.
2. **API Efficiency and Reliability**: Although the app works well with the FLIGHTSAPI, relying on a third-party API comes with the risk of service limitations or outages. For a production version, implementing strategies like caching, rate limiting, and fallback mechanisms would make the app more resilient. Furthermore, since API keys can be exposed in the code, securing the API key (e.g., through environment variables) would be important for long-term use.
3. **UI and UX Enhancements**:
   * **Loading Indicators**: The app can be improved with a visual cue, such as a spinner or progress bar, to indicate that data is being fetched from the API. This would prevent the user from feeling like the app is unresponsive, especially when the network is slow.
   * **flight Icons**: Adding icons (like a sun, cloud, or raindrop) for weather conditions would make the output more visually appealing and easier to interpret at a glance.
   * **Improved Layout**: The current layout is quite simple. With some design enhancements, such as grouping related data (e.g., weather and temperature) in separate sections or using different fonts and colors for headings, the UI could be made more user-friendly.
4. **Scalability and Future Features**:
   * **Extended Forecasts**: A natural next step would be to provide more detailed weather data, such as a 7-day forecast or hourly weather predictions, which would make the app more useful for users.
   * **Geolocation**: Incorporating geolocation capabilities to automatically detect the user’s current location and fetch the weather for that city would enhance the app’s functionality. This would make the app more convenient for users who don’t want to manually input their city.
   * **Multiple Units of Measurement**: Currently, the app only provides weather information in Celsius. Allowing the user to choose between Celsius, Fahrenheit, or Kelvin would make it more versatile and accessible to users from different regions.
   * **Dark Mode**: Adding a dark mode option would improve the user experience, especially for users who prefer low-light environments.
5. **Code Organization and Maintainability**:
   * **Separation of Concerns**: The app could benefit from separating the GUI logic from the data-fetching logic. This would make the code more modular, easier to test, and more maintainable in the long run.
   * **Refactoring the Parsing Logic**: Instead of manually parsing the JSON with regex, a more robust solution would involve using a JSON library (like Gson or Jackson), which would simplify parsing and reduce the likelihood of errors in extracting the data.
   * **Code Documentation**: Commenting the code, especially the more complex sections such as the JSON parsing and API call, would make it easier for future developers to understand and modify the code.

**Long-Term Prospects**

While the current app serves as a simple weather tool, there’s a significant opportunity to evolve it into a more feature-rich and polished product. Over time, as you add features like forecast data, enhanced geolocation services, or even user-specific settings (like favorite cities), the app could transform into a comprehensive weather platform that appeals to a wider audience.

* **Mobile Version**: The app could eventually be ported to mobile platforms (using JavaFX for Android or cross-platform tools like Flutter) to provide users with on-the-go weather updates.
* **Additional Data Integration**: By adding more data sources (such as air quality or pollen levels), the app could cater to specific user needs, such as individuals with respiratory conditions or those interested in environmental factors.

**CHAPTER 8**

**REFERENCES**

1. Spring Boot in Action (2016). Walls, C. Manning Publications, New York.

2. Spring Security in Action (2020). Spilca, L. Manning Publications, New York.

3. Java Persistence with Hibernate (2015). Bauer, C., & King, G. Manning Publications, New

York.

4. Pro Spring Boot 2 (2018). Gutierrez, F. Apress, New York.

5. MySQL High Availability (2010). Bell, C., Kindahl, M., & Thalmann, L. O'Reilly Media,

Sebastopol