**JetWayz**

**INTERNSHIP REPORT**

*Submitted by:*

**Patel Deep Nimeshbhai**

ENROLLMENT NO: 210280107060

*In partial fulfilment for the award of the degree of*

**BACHELOR OF ENGINEERING**

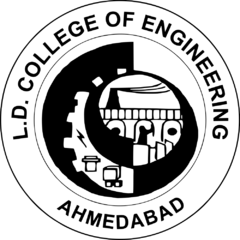
*In*

**Computer Engineering**

**L.D. COLLEGE OF ENGINEERING**

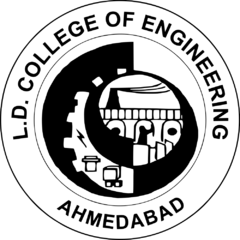
**AHMEDABAD-380015**

**GUJARAT**



**Gujarat Technological University,**

**Ahmedabad [April 2025]**

**L.D. College of Engineering**

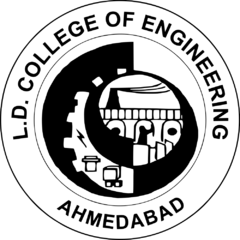
**120, Circular Road, University Area, Ahmedabad, Gujarat 380015.**

# CERTIFICATE

This is to certify that the project report submitted along with the project entitled “JetWayz’’ has been carried out by Gelatar Dharmesh Dhiraj lal under my guidance in partial fulfilment for the degree of Bachelor of Engineering in Computer Engineering, 8th Semester of Gujarat Technological University, Ahmedabad during the academic year 2024-25.

Prof. Nikunj Domadiya Dr. Chirag Thakkar

Internal Guide Head of Department

**L.D. College of Engineering**

**120, Circular Road, University Area, Ahmedabad, Gujarat 380015.**

# DECLARATION

We hereby declare that the Internship/Project report submitted along with the Internship/Project entitled “JetWayz’’submitted in partial fulfilment for the degree of Bachelor of Engineering in **Computer Engineering** to Gujarat Technological University, Ahmedabad, is a Bonafide record of original project work carried out by me at **OPL Pvt Ltd** under the supervision of **Deepak Vishwakarma** and no part of this report has been directly copied from any students’ reports or taken from any other source, without providing due reference.

Place: Ahmedabad

Date: 05 APRIL 2025

Name of the student Signature

**Patel Deep Nimeshabhai**

# ACKNOWLEDGEMENTS

I would like to show my deepest gratitude to **Dr. Nilay N. Bhuptani** , Principal, L.D. College of Engineering, **Dr. Chirag Thakkar**, Head of Department, Computer Engineering, Ahmedabad, for granting me permission to work as an intern in this esteemed IT company and for encouraging us to successfully complete our work.

I would like to express my indebtedness appreciation to our respected guide **Prof.** Pragnesh Patel**,** Assistant Professor, Computer Engineering, L.D. College of Engineering, Ahmedabad & **Mr. Deepak Vishwakarma**, **OPL Pvt. Ltd.** for his invaluable guidance and consistent support during the study. I am grateful to his detailed suggestions and to help clarifying concepts which helped me a lot during the course of this study.

I am grateful to all my **faculty members** of Computer Engineering department, L.D. College of Engineering, Ahmedabad, **my parents** and **my colleagues** for their support, special attention and motivation during the internship and training duration work.

# ABSTRACT

The theoretical foundation of software engineering education gains true value when applied in practical scenarios. Recognizing this principle, Gujarat Technological University mandates internships in the final semester to facilitate students' transition from academic knowledge to professional application. These internships serve as crucial bridges connecting classroom concepts to industry realities.

This report documents my comprehensive 12-week internship experience as an Associate Software Engineer Trainee at OPL Pvt. Ltd., Ahmedabad. During this period, I focused on developing "JetWayz," a sophisticated flight management system designed to streamline airline operations through an integrated digital platform. The system offers specialized interfaces for administrators, airline staff, and passengers, enabling efficient flight scheduling, booking management, and travel planning through an intuitive and responsive user experience.

The project leveraged a modern technology stack comprising Angular for frontend development, Spring Boot with Java for backend services, and MySQL for database management. Working within the MVC architectural pattern, my contributions spanned both frontend and backend development, including implementing secure RESTful APIs, developing robust exception handling mechanisms, and maintaining disciplined version control practices through Git.

This immersive experience provided invaluable insights into enterprise software development lifecycles, enhanced my technical and collaborative capabilities, and developed my adaptability in agile work environments. The internship strengthened my capacity to work both independently and as an effective team member while adhering to project timelines and deliverables.

This report represents a synthesis of academic theory and practical implementation, demonstrating how classroom knowledge transforms into scalable, real-world software solutions. The internship has been instrumental in not only reinforcing my technical foundation but also in shaping my professional identity as an emerging software engineer prepared for industry challenges.

# List of Figures

2.3.1 Office Environment…………………..………………………………………….………...17

2.3.2 Celebrations ...…...…....……….………………………….………………….……………18

2.3.3 Travel & Celebrations……………………………………………….…………………….18

2.3.4 Travel & Celebrations .……………………….…………………………….……….….….19

3.2.1 Spring Boot architecture ………….…………………………………….…………………32

3.2.2.1 Dependency Injection……………………………………………...………………….....33

3.2.2.2 Dependency Injection………………….……………………………………………..….34

3.2.7 Spring Cloud Gateway ……………………………………………….………………..…..39

3.2.8 JWT Authentication ……………………………………………………………..…….….40

3.3.1 Angular Overview …………………………………………………………….………..…43

3.3.2 Project Structure ………………………………………………………….…………….…44

3.3.4 Services and Dependency Injection……………………………………………………….46

3.3.5 Data Binding & Event Handling…………………………………………………………..47

3.3.7 Reactive Forms & Validation……………………………………………………………...49

3.3.9 Consuming JWT-Protected APIs ……………………………………………………....….50

4.3.1 Registration Page UI……………………………………………………………….…....…53

4.3.2 Login Page UI ………………………………………………….……………….……….…..54

4.5.1 Folder Structure ……..…………………………………………………………….…..…..55

4.8.1 Flight Search Page UI………………………………………………….…….………….…60

4.8.2 Flight Result Page UI…………………………………………………………….…..….…61

4.8.3 Passenger Details Fill-up Page UI……………………………………………………....…62

4.8.4 Payment Page UI………..………………………………………………….………….…..63

4.9.1 Use Case Diagram……………………………………………………………….………...64

4.9.2 Activity Diagram…………………………………………………………………………..66

4.9.3 Class Diagram …………………………………………………….………..……………..67

4.9.4 Sequence Diagram…….………………………….………………….………….…………68

# List of Tables

2.1 Job Profile Matrix…………………………………………………………………….……...16

2.3.5 Software & Tools Table……………………………………………………………………20

# TABLE OF CONTENT

Certificate…………………………………………………………………..……………………...2

Internship Joining Letter………………………………………………………………..…………3

Company Certificate………………………………………………………………..…………..…4

Declaration……………………………………………………………………………..…...……..5

Acknowledgement………………………………………………………………………….……..6

Abstract…………………………………………………………………………..…………….….7

List of Figures……………………………………………………………………….….…………8

List of Tables……………………………………………………………………………..……….9

Table of Contents……………………………………………………………..…………….……10

Chapter-1: About the company…………...………………………………………………….…..13

* 1. About……………………………………………………………………….…......13
  2. What we do? ……………………………………………………………………..13
  3. Industries……………………………………………………………………...…..14
  4. Mission………………………………………………………………….……...…14
  5. Vision………………………………………………………………………....…..14
  6. Website…………………………………………………………………………...14

Chapter-2: Internship Overview...…………………………………………………………….…15

2.1 Objectives………...…………………………………………...…………………15

2.2 Job Description……...……………………………………………...……………16

2.3 Work Environment….……………………………………………..…………….17

2.4 Software/tools table………………………………………………………..…….20

Chapter-3: Learning…………...…………………………………………………………………21

3.1 Programming Language Java 8 ………………….…………………………...21

3.1.1OOP Concepts...………………………………………………....…22

3.1.2 Lambda Expressions...………………………….…..………….......23

3.1.3 Stream API...…………………………………………………….…24

3.1.4 Functional Interfaces...……………………………....…………..…25

3.1.5 Method References...………………………………………..…..…26

3.1.6 Collections & Generics...……………………………………..……28

3.1.7 Exception Handling...………………………………………...……29

3.2 Backend Development–Spring Boot………………………………………..….31

3.2.1 Spring Boot Introduction……...……………..………….…...….…31

3.2.2 REST API Development…...…..………………………….….……32

3.2.3 Dependency Injection (@Autowired)……………..…………….…33

3.2.4 Spring Data JPA………………….…………………………...…....35

3.2.5 Microservices Architecture…………………………….…….....….36

3.2.6 Service Registration (Eureka)………………………………..…….37

3.2.7 API Gateway (Spring Cloud Gateway)………………………....….38

3.2.8 JWT Authentication & Authorization…………………………..….39

3.2.9 Exception Handling (@ControllerAdvice)……………….….....….41

3.2.10 Application Properties Configuration…………….…………...….41

3.3 Frontend Development – Angular.………...…………………..…………….…43

3.3.1 Angular Overview………………...…….…….……………...…….43

3.3.2 **Project Structure** ……………………………..………...……..…...44

3.3.3 **Components & Modules** ……………………….……...…….…….45

**3.3.4 Services and Dependency Injection** ………………………...………45  
 **3.3.5 Data Binding & Event Handling** ……………………..…………..…47  
 **3.3.6 Routing & Navigation** …………………..……………………….….48  
 **3.3.7 Reactive Forms & Validation** …………………..…….……….……48  
 **3.3.8 API Integration using HttpClient** ………………..……………….…49  
 **3.3.9 Consuming JWT-Protected APIs** ……………..…………….………49

Chapter-4: Project – JetWayz-Flight Management System ......................................................... 51

4.1 Introduction & Objective ................................................................................... 51

4.2 Tech Stack Overview ......................................................................................... 51

4.3 Project Features .................................................................................................. 52

4.4 MVC Architecture ............................................................................................. 54

4.5 Folder Structure Explanation ............................................................................. 55

4.6 Authentication & Authorization ..........................................................….......... 58

4.7 Middleware and Utilities .................................................................................... 59

4.8 Screenshots & Output Pages .............................................................................. 59

4.9 UML Diagrams .................................................................................................. 64

Chapter-5: My Skills & Work……………………………………………...……………………69

5.1 Skills….….….….….….….….……….….………….……………..……....…69

5.2 Work………………………………...……………………………..……....…70

Chapter-6: Conclusion…………………………...………………………...……………………72

Chapter-7: References……………………………………………………...……………………73

Chapter-8: Mentor’s Feedback.…………………………………………...………………….…74



# 

# **OPL.Pvt.Ltd.**

## **About:**

* OPL is a pioneering digital infrastructure and financial technology company committed to transforming the financial ecosystem through innovation and technology. With a strong focus on simplifying credit access, streamlining financial services, and supporting MSMEs, corporates, and large financial institutions, OPL delivers tailor-made, high-impact solutions across the digital lending and fintech space..

## **What we do?**

* Digital Lending Platforms:  
  Revolutionizing credit accessibility with end-to-end, high-efficiency platforms like PSB Loans in 59 Minutes, enabling fast-tracked application processing, automated creditworthiness scoring, and instant approvals.
* Custom Tech Solutions for Financial Institutions:  
  Empowering banks, NBFCs, and corporates with white-label technology products that automate and optimize loan disbursement, enhance customer service, and boost operational scalability.
* MSME & Retail Credit Support:  
  Dedicated tools and platforms to enable growth for micro, small, and medium businesses, providing access to Mudra loans, home loans, personal loans, and auto financing with seamless digital journeys.
* Government & Social Sector Integration:  
  Building national-level portals like *JanSamarth.in* and *JanSuraksha* to support government-linked credit and insurance schemes, streamlining citizen access to welfare and empowerment programs.
* Data-Driven Intelligence & Predictive Analytics:  
  Using real-time data, advanced analytics, and proprietary models like *Fit Rank* to help lenders make smarter, faster, and more informed decisions, while driving innovation in credit assessment and risk management.

## **Industries:**

* Finance
* Banking
* Fintech
* RegTech
* Education
* MSMEs

## **Mission:**

* At OPL, our mission is to revolutionize the financial ecosystem by delivering innovative, accessible, and intelligent digital solutions. We aim to bridge the gap between institutions and individuals by streamlining processes, accelerating credit access, and empowering growth across all sectors through advanced technology.

## **Vision:**

* We envision a world where innovation drives inclusion. OPL is committed to becoming a global leader in digital finance by continuously evolving our platforms and solutions to empower financial institutions, support underserved communities, and shape the future of finance with transparency, speed, and impact.

## **Website:**

* <https://www.oplinnovate.com>

# Internship Overview

## **Objective:**

* **To build strong programming fundamentals** by gaining hands-on experience in Core Java and Java 8 concepts, which are essential for a successful career in Java development.
* **To develop and integrate RESTful APIs** using Spring Boot, enabling effective communication between the application and database layers—an essential component of modern backend development.
* **To understand and design relational databases** using MySQL, learning to manage different types of relationships and create efficient, scalable data models.
* **To gain practical knowledge of frontend development** by building dynamic and interactive user interfaces using the Angular framework, including concepts like server-side rendering and component-based architecture.
* **To learn microservices architecture** by modularizing application code into independently deployable services and building inter-service communication while managing multiple databases.
* **To implement robust security features** including user authentication and authorization, and securing APIs and API gateways to ensure data privacy and application integrity.
* **To experience real-world software development practices**, working collaboratively with a team of developers on full-stack projects, applying principles like MVC and modular coding.

## **Job Description:**

**Java Software Developer** [OPL Innovate]

**The role**

* Backend Developer:  
  Responsible for designing, developing, and maintaining server-side logic using Java and Spring Boot. Involved in building scalable APIs, integrating databases, and ensuring high performance and responsiveness of backend services.
* Full-Stack Contributor  
  Collaborates across the stack to support frontend integration using Angular, contributes to RESTful API development, and ensures smooth communication between client-side and server-side components.

**Key responsibilities**

* Strong knowledge of Core Java, OOPs Concepts, and Collections Framework
* Hands-on experience in building RESTful APIs using Spring Boot and managing relational databases with MySQL
* Familiarity with microservices architecture, including modular design and inter-service communication
* Proficient in writing clean, maintainable code with version control using Git and following MVC architecture

**Job details**

|  |  |  |
| --- | --- | --- |
| Java Software Developer | 5 days working etc. | 8.5 hours/day |
| Full-time | ~~Part-time~~ | Ashwamegh Elegance - 3, Opp. SBI Corporate Office, SM Road, Ambawadi, Ahmedabad - 380015 | Fresher |
| ~~Overtime~~ | ~~travel~~ | ~~evening/weekend work~~ |

[2.1 Job Profile Matrix]

## **Work Environment:**

* At OPL, we create an environment where innovation and collaboration thrive, empowering our employees to reach their full potential. We believe in crafting groundbreaking solutions that address the challenges of the financial world, from simplifying credit access to transforming digital lending systems. If you're passionate about using technology to drive meaningful change and overcome real-world challenges, OPL is the place for you. Join our team and be a part of the transformation, as we work together to revolutionize the future of finance. Explore our current openings, and take the first step towards an exciting career with OPL



[2.3.1 Office Environment]

* My Company values diversity and inclusion and has a diverse workforce, including employees from different backgrounds and cultures. My Company encourages open communication and feedback, and employees are encouraged to speak up and share their ideas.



[2.3.2 Celebrations ]

* My Company also celebrates various events, festivals and employee birthday celebration, food parties, yearly vacations etc.



[2.3.3 Travel & Celebrations ]



[2.3.4 Travel & Celebrations ]

* My Company also provides opportunities for employees to grow and develop their skills through training programs, mentorship, and career advancement opportunities. My Company promotes work-life balance and offers flexible work arrangements, such as remote work options and flexible schedules.
* Overall, My Company work environment is positive, collaborative, and focused on growth and development, while also valuing employee well-being and work-life balance.

## **Software tool’s table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Tool Name | Logo/Image | Use Case |
| 1. | Visual Studio Code |  | Visual Studio Code is a free coding editor that helps you start coding quickly. Use it to code in any programming language, without switching editors. Visual Studio Code has support for many languages, including Python, Java, C++, JavaScript, and more. |
| 2. | **STS (Spring Tool Suite)** to run your **Spring Boot** backend |  | For backend development, used **Spring Tool Suite (STS)** to run and manage **Spring Boot** applications. STS provided an integrated development environment tailored for building enterprise-level Java applications with Spring. It helped streamline backend development, REST API creation, and seamless integration with the frontend built using Angular. |
| 3. | GitHub |  | GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere. |

[2.3.5 Software & Tools Table]

# Learning

# 3.1 Programming Language - Java 8

## **Theoretical Foundation**

Java 8, released in March 2014, represents a paradigm shift in Java's evolution by introducing functional programming concepts to a traditionally object-oriented language. This update was driven by several key industry trends:

1. **Multicore Processing**: As CPU manufacturers shifted toward multicore architectures instead of faster clock speeds, programming languages needed better tools for concurrent programming.
2. **Big Data Processing**: The explosion of data volumes required more efficient ways to process large collections of information.
3. **Code Maintainability**: The increasing complexity of applications demanded more expressive, less verbose coding patterns.
4. **Competing Languages**: The rise of functional languages and frameworks was challenging Java's dominance in enterprise development.

Java 8 addressed these challenges by introducing functional interfaces, lambda expressions, the Stream API, and method references—features that fundamentally changed how Java code is written while maintaining backward compatibility. This update balanced innovation with stability, allowing developers to adopt functional programming techniques gradually without breaking existing systems.

The importance of Java 8 cannot be overstated; it represents the most significant change to the language since its inception, enabling more concise, readable, and maintainable code while providing better support for parallel processing and asynchronous programming patterns essential for modern applications.

## **3.1.1 OOP Concepts**

### **Theoretical Foundation**

### Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" containing data and procedures. Java was designed as an OOP language from its inception, implementing four core principles that help manage complexity in software systems:

#### **Key Principles:**

1. **Encapsulation**: Bundles data with methods that operate on that data, hiding internal state and requiring all interactions to occur through well-defined interfaces. This creates information hiding and reduces system interdependencies.
2. **Polymorphism**: Allows objects to be treated as instances of their parent class, enabling a single interface to represent different underlying forms. This creates flexibility through dynamic method binding
3. **Inheritance**: Enables a class to inherit properties and behaviors from a parent class, establishing "is-a" relationships that promote code reuse and logical hierarchies.
4. **Abstraction**: Focuses on the essential qualities of an object rather than one specific implementation, allowing programmers to think at higher levels of conceptualization.

#### **Importance:**

* Reduces complexity through decomposition of large systems
* Improves maintainability through logical organization
* Enhances code reuse through inheritance hierarchies
* Provides better modularization for team development
* Models real-world domains more naturally

## **3.1.2 Lambda Expressions**

### **Theoretical Foundation**

### Lambda expressions fundamentally changed Java by introducing functional programming concepts to the language. They represent anonymous functions—code blocks that can be passed around as if they were data.

### At their core, lambda expressions are a more concise syntax for implementing single-method interfaces (functional interfaces). They allow developers to treat functionality as a method argument, enabling behavior parameterization without the verbose anonymous inner class syntax.

### Lambda expressions are based on lambda calculus, a formal mathematical system for expressing computation, developed by Alonzo Church in the 1930s. This system forms the theoretical foundation of functional programming languages.

#### **How Lambda Expressions Work:**

When you write a lambda expression in Java, the compiler:

1. Determines the target functional interface based on context
2. Verifies compatibility with the interface's method signature
3. Creates an instance of the functional interface with the lambda body as the method implementation
4. Uses invokedynamic bytecode to defer the actual implementation strategy to runtime

The JVM can optimize lambda execution through techniques like inlining and removing unnecessary object creation, often making lambdas more efficient than equivalent anonymous classes.

#### **Importance:**

* **Conciseness**: Dramatically reduces boilerplate code
* **Readability**: Makes code intention clearer
* **Behavior parameterization**: Enables passing functions as arguments
* **Functional programming**: Makes functional patterns practical in Java
* **Collection processing**: Powers the Stream API
* **Event handling**: Simplifies callback implementations
* **Parallel processing**: Facilitates concurrent programming patterns

## **3.1.3 Stream API**

### **Theoretical Foundation**

The Stream API represents a paradigm shift in how collections are processed in Java, introducing a functional approach to data manipulation. Unlike collections, which are data structures that store elements, streams are pipelines of computational operations that can transform data.

Conceptually, streams are built upon:

1. **Declarative programming**: Expressing what should be done rather than how
2. **Functional composition**: Building complex operations from simple ones
3. **Lazy evaluation**: Deferring execution until results are needed
4. **Data pipeline architecture**: Representing multi-step processing as a single flow

Streams leverage the functional interfaces introduced in Java 8 to accept behaviors as parameters, allowing transformations to be expressed as lambda expressions or method references.

#### **Stream Processing Model:**

A stream pipeline consists of:

1. **Source**: A data origin (collection, array, generating function, I/O channel)
2. **Intermediate operations**: Transformations that produce new streams (filter, map, etc.)
3. **Terminal operation**: Final processing that produces a result or side effect

The key insight is that intermediate operations are lazy—they only set up the pipeline but don't process any elements until a terminal operation is invoked. This enables significant optimizations:

* **Short-circuiting**: Stopping processing once a result is determined
* **Loop fusion**: Combining multiple operations into a single pass
* **Parallelization**: Splitting processing across multiple threads automatically

#### **Importance:**

* **Expressive code**: Implements complex data transformations clearly
* **Reduced complexity**: Eliminates explicit iteration and temporary collections
* **Parallelism**: Simplifies concurrent processing without explicit thread management
* **Optimization**: Enables runtime optimizations impossible with imperative code
* **Functional composition**: Encourages reusable, composable data transformations
* **Memory efficiency**: Supports processing large or infinite data sets

## **3.1.4 Functional Interfaces**

### **Theoretical Foundation**

Functional interfaces serve as the cornerstone of Java's implementation of functional programming. A functional interface is an interface that contains exactly one abstract method, making it suitable as a target type for lambda expressions and method references.

The concept bridges object-oriented and functional paradigms by using Java's existing interface type system to represent functions. This design choice was crucial for maintaining backward compatibility while introducing functional programming concepts.

Functional interfaces can be viewed as "function types" in Java. While traditional object types describe what an object is, functional interface types describe what an object can do—its behavior.

#### **How Functional Interfaces Work:**

When you use a lambda expression or method reference, the Java compiler:

1. Identifies the target functional interface from context
2. Verifies the lambda's parameter types and return type are compatible with the interface method
3. Creates an implementation of that interface with the lambda/method reference as the implementation of the abstract method

The @FunctionalInterface annotation is a marker that ensures an interface meets the requirements of having exactly one abstract method. While optional, it helps detect errors at compile time if additional abstract methods are added accidentally.

#### **Importance:**

* **Type safety**: Provides compile-time verification for lambdas
* **API design**: Enables behavior parameterization in APIs
* **Composition**: Allows functions to be composed into more complex functions
* **Reusability**: Standard interfaces encapsulate common functional patterns
* **Integration**: Bridges between OOP and functional programming paradigms
* **Expression**: Creates a vocabulary for common computational patterns

**3.1.5 Method References**

### **Theoretical Foundation**

Method references provide a shorthand notation for lambda expressions that invoke existing methods. They represent a significant enhancement to Java's functional programming capabilities by allowing direct references to methods or constructors using the double colon (::) syntax. This mechanism elegantly bridges the object-oriented nature of Java with functional programming concepts, reducing verbosity while preserving semantic clarity. Method references streamline code by eliminating the need to explicitly declare parameters when the method being called has the same signature as the functional interface's abstract method.

#### **How Method References Work:**

The Java compiler translates method references into appropriate functional interface

implementations by:

1. Determining the target functional interface from context
2. Matching parameter types between the interface method and referenced method
3. Handling any necessary type conversions
4. Creating an implementation that forwards calls to the referenced method

There are four distinct types of method references:

* Static method references (ClassName::staticMethod)
* Instance method references on specific objects (objectInstance::instanceMethod)
* Instance method references on arbitrary objects of a particular type (ClassName::instanceMethod)
* Constructor references (ClassName::new)

#### **Importance:**

* **Readability:** Creates more concise and expressive code than equivalent lambdas
* **Intent clarity:** Explicitly shows that existing functionality is being reused
* **Code reuse:** Promotes utilizing established methods rather than reimplementing logic
* **Type inference:** Leverages Java's type inference system for parameter matching
* **Consistency:** Provides uniform syntax for referencing different kinds of methods
* **Documentation:** Self-documents code by directly referencing named methods

**3.1.6 Collections & Generics**

### **Theoretical Foundation**

The Collections Framework combined with generics forms the backbone of Java's data structure ecosystem. Collections provide type-safe, high-performance implementations of fundamental data structures, while generics enable compile-time type safety and elimination of explicit casts. This pairing represents a sophisticated type system that supports parametric polymorphism, allowing algorithms to be written independent of concrete types while maintaining strong type guarantees.

#### **How Collections & Generics Work:**

The Java Collections Framework is organized around a coherent set of interfaces, implementations, and algorithms that operate on collections:

1. Core interfaces (Collection, List, Set, Map) define abstract behaviors
2. Concrete implementations provide various performance characteristics
3. Generic type parameters specify element types at compile time
4. Type erasure converts generics to raw types at runtime
5. Bounded wildcards (? extends T, ? super T) support flexible subtyping relationships

Modern collections support functional operations through stream APIs, enabling declarative transformation and processing of data structures using map, filter, reduce and other higher-order functions.

#### **Importance:**

* **Type safety:** Eliminates ClassCastExceptions through compile-time verification
* **Code reuse:** Enables implementation of algorithms independent of concrete types
* **Performance:** Provides optimized implementations for different use cases
* **Expressiveness:** Allows precise modeling of type relationships and constraints
* **Maintainability:** Reduces duplication through reusable, type-safe components
* **Composition:** Supports powerful composition of collections and operations

**3.1.7 Exception Handling**

### **Theoretical Foundation**

Exception handling represents Java's systematic approach to managing runtime anomalies and error conditions. Based on the concept of separating normal execution flow from error-handling logic, Java's exception mechanism provides a structured way to detect, propagate, and recover from exceptional conditions. The system builds on an exception class hierarchy that distinguishes between checked exceptions (anticipated, recoverable conditions) and unchecked exceptions (programming errors or catastrophic failures), establishing a contract between API designers and consumers about expected failure modes.

#### **How Exception Handling Work:**

Java's exception handling architecture operates through a multi-layered mechanism:

1. Exception objects encapsulate error information and context
2. The try block encloses code that might throw exceptions
3. The catch blocks define handlers for specific exception types
4. The finally block guarantees execution regardless of exception occurrence
5. The throws clause declares checked exceptions a method might throw
6. Try-with-resources automatically manages resource cleanup

The exception handling system leverages the type system through exception hierarchies, allowing catch clauses to handle specific exceptions or broader categories according to the Liskov Substitution Principle.

#### **Importance:**

* **Separation of concerns:** Isolates error-handling code from normal business logic
* **Centralization:** Enables handling of errors at appropriate abstraction levels
* **Robustness:** Prevents system crashes through controlled error recovery
* **Documentation:** Makes API failure modes explicit through checked exceptions
* **Resource safety:** Ensures proper resource cleanup even when errors occur
* **Context preservation:** Maintains error context through exception stack traces
* **Type safety:** Leverages the type system to categorize and handle different error conditions

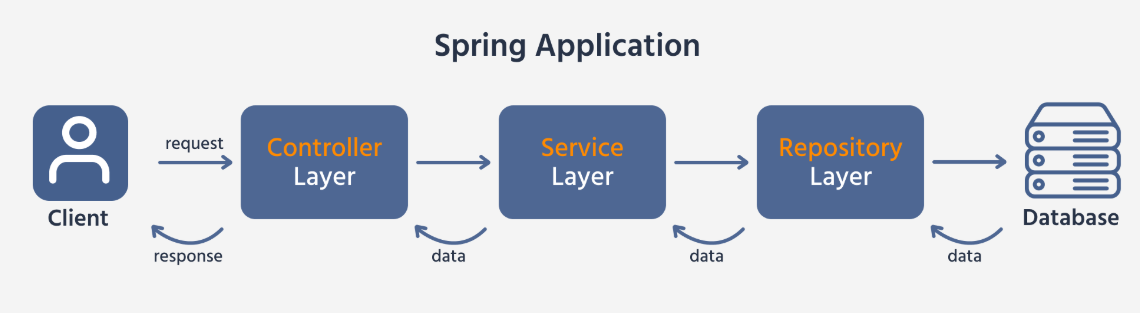
**3.2 Backend Development – Spring Boot**

**3.2.1 Spring Boot Introduction**

Spring Boot is an extension of the Spring Framework that simplifies the development of Java-based applications by providing an opinionated approach to configuration. It eliminates much of the boilerplate code and configuration that characterized traditional Spring applications, allowing developers to focus on business logic rather than infrastructure setup.

* **Key features of Spring Boot include**:
* **Auto-configuration**: Spring Boot automatically configures your application based on the dependencies you've added to your project.
* **Standalone**: Spring Boot applications can be run without requiring an external web server deployment, thanks to embedded servers like Tomcat, Jetty, or Undertow.
* **Opinionated defaults**: Sensible default configurations that can be overridden when necessary.
* **Production-ready**: Built-in features for health checks, metrics, monitoring, and externalized configuration.
* **Spring ecosystem integration**: Seamless integration with the broader Spring ecosystem, including Spring Data, Spring Security, and Spring Cloud.
* **The Spring Boot architecture follows a layered approach:**

1. **Presentation Layer**: Controllers that handle HTTP requests and responses
2. **Service Layer**: Business logic implementation
3. **Data Access Layer**: Repository interfaces that interact with databases
4. **Database Layer**: Persistent storage of application data



[3.2.1 Spring Boot architecture]

**3.2.2 REST API Development**

RESTful API development is one of Spring Boot's core strengths. Spring Boot leverages Spring MVC to create RESTful web services that follow REST architectural principles.

* **Key Components of REST API Development:**

1. **Controllers**:

Classes annotated with @RestController that handle HTTP requests and define API endpoints.

1. **Request Mapping**:

Annotations like @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping to map HTTP methods to controller methods.

1. **Response Entity**:

The ResponseEntity class for full control over the HTTP response, including status codes, headers, and body.

1. **Path Variables and Request Parameters**:

Extracting data from URLs with @PathVariable and @RequestParam.

1. **Request Body**:

Using @RequestBody to convert JSON/XML payloads to Java objects.

1. **Content Negotiation**:

Supporting multiple media types for request/response (JSON, XML, etc.).

1. **HATEOAS**:

Hypermedia as the Engine of Application State for self-descriptive APIs.

* **REST API endpoints typically follow standard conventions:**
* GET /resource: Retrieve a list of resources
* GET /resource/{id}: Retrieve a specific resource
* POST /resource: Create a new resource
* PUT /resource/{id}: Update an existing resource
* DELETE /resource/{id}: Delete a resource*]*

**3.2.3 Dependency Injection (@Autowired)**

Dependency Injection (DI) is a core principle of the Spring framework that Spring Boot inherits. It's a design pattern that implements Inversion of Control (IoC), allowing dependencies to be injected into components rather than having components create or find their dependencies.

* **Dependency Injection Methods in Spring Boot:**

****

[3.2.2.1 Dependency Injection]

****

[3.2.2.2 Dependency Injection]

* **Constructor injection is generally preferred as it:**
* Ensures required dependencies are not null (immutability)
* Makes testing easier
* Explicitly declares dependencies
* **Spring Bean Scopes:**
* **Singleton**: Default scope; one instance per Spring container
* **Prototype**: New instance created each time requested
* **Request**: One instance per HTTP request
* **Session**: One instance per HTTP session
* **Application**: One instance per ServletContext
* **Websocket**: One instance per WebSocket

**3.2.4 Spring Data JPA**

Spring Data JPA simplifies database operations by reducing the boilerplate code required for data access. It's an abstraction on top of JPA (Java Persistence API) that provides repository support for the data access layer.

* **Key Features of Spring Data JPA:**

1. **Repository Interfaces**: Define repository interfaces that extend Spring Data interfaces like JpaRepository, CrudRepository, or PagingAndSortingRepository.
2. **Method Name Queries**: Create queries by simply defining method names following a specific pattern (e.g., findByLastName(String lastName)).
3. **JPQL Queries**: Write queries using Java Persistence Query Language with the @Query annotation.
4. **Native SQL Queries**: Execute native SQL queries when needed.
5. **Pagination and Sorting**: Built-in support for paginated queries and sorting results.
6. **Specifications**: Type-safe criteria queries using the JPA Criteria API.
7. **Auditing**: Automatic tracking of who created or modified entities and when.

* **Entity Relationship Mappings:**
* **@OneToOne**: One-to-one relationship between entities
* **@OneToMany/@ManyToOne**: One-to-many/Many-to-one relationship
* **@ManyToMany**: Many-to-many relationship
* **Transaction Management:**

Spring Data JPA integrates with Spring's transaction management, providing:

* Declarative transaction management using @Transactional
* Programmatic transaction management using TransactionTemplate
* Support for transaction propagation, isolation levels, and timeout settings

**3.2.5 Microservices Architecture**

Microservices architecture is an approach to application development where a large application is built as a suite of small, independent services that communicate over a network. Spring Boot, combined with Spring Cloud, provides an excellent foundation for building microservices.

* **Characteristics of Microservices:**

1. **Decentralized**: Each service can be developed, deployed, and scaled independently.
2. **Domain-Driven**: Services are organized around business capabilities.
3. **Autonomous**: Services can function and evolve independently of other services.
4. **Polyglot**: Different services can use different technologies and data storage solutions.
5. **Resilient**: Failure in one service doesn't cascade to others.
6. **Scalable**: Individual services can be scaled independently based on demand.

* **Microservices Patterns Supported by Spring:**

1. **Service Discovery**: Finding service instances (Eureka)
2. **API Gateway**: Single entry point for clients (Spring Cloud Gateway)
3. **Circuit Breaker**: Handling service failures gracefully (Resilience4j)
4. **Configuration Server**: Centralized configuration management (Spring Cloud Config)
5. **Distributed Tracing**: Tracking requests across services
6. **Event-Driven Communication**: Asynchronous messaging between services (Spring Cloud Stream)
7. **Client-Side Load Balancing**: Distributing requests across instances (Spring Cloud LoadBalancer)

**3.2.6 Service Registration (Eureka)**

In a microservices architecture, service instances can dynamically change due to auto-scaling, failures, or upgrades. Service Discovery allows services to find and communicate with each other without hardcoding location information.

* **Spring Cloud Netflix Eureka:**

Eureka is a service registry that enables service discovery in a microservices environment.

* **Key Components:**

1. **Eureka Server**: The registry that maintains information about all client service applications.
2. **Eureka Client**: Services that register with Eureka to announce their availability.

* **Features:**

1. **Self-Registration**: Services register themselves with the Eureka server.
2. **Health Monitoring**: Eureka monitors the health of registered services.
3. **Load Balancing**: Integration with client-side load balancers.
4. **Multiple Zones and Regions**: Support for geographical distribution of services.
5. **Self-Preservation Mode**: Protection against network partitions.

* **Implementation:**

To implement Eureka in a Spring Boot application:

1. Create a Eureka Server application with @EnableEurekaServer
2. Configure client applications with @EnableDiscoveryClient
3. Use Spring Cloud's DiscoveryClient or load-balanced RestTemplate for service-to-service communication

**3.2.7 API Gateway (Spring Cloud Gateway)**

An API Gateway acts as a single entry point for all clients, routing requests to the appropriate microservice, handling cross-cutting concerns, and providing a unified API for the client applications.

* **Spring Cloud Gateway:**

Spring Cloud Gateway is a library for building API gateways on top of Spring WebFlux, providing a reactive, non-blocking approach.

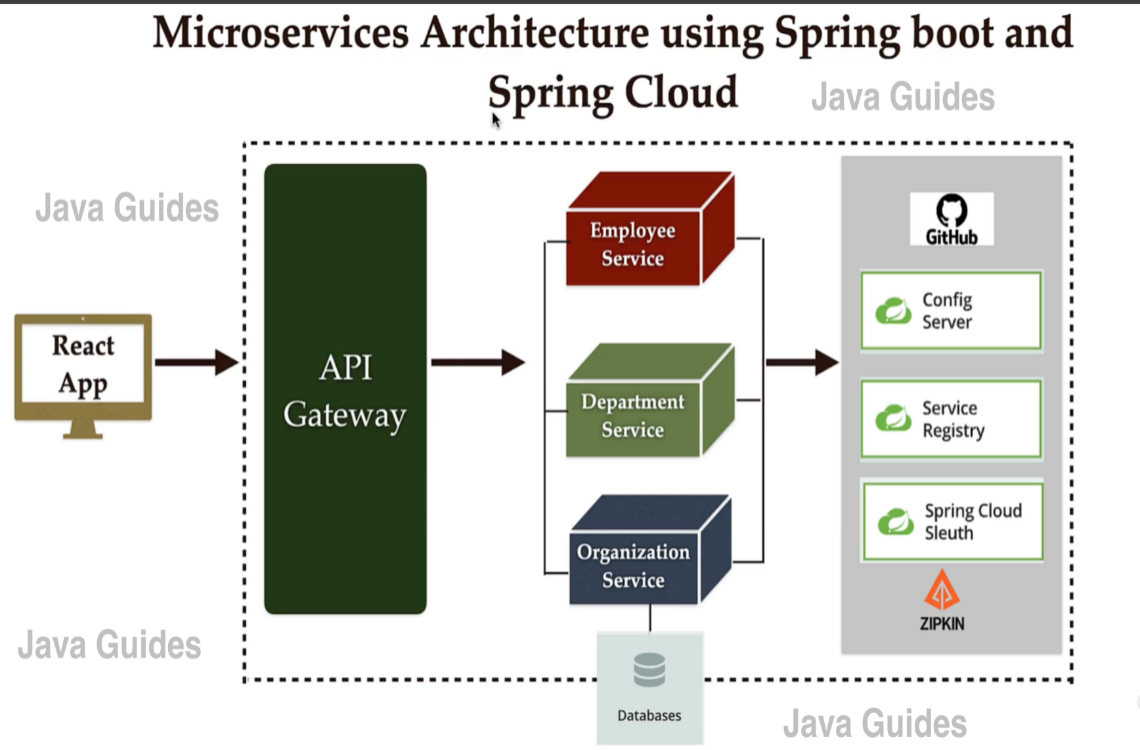
* **Key Features:**

1. **Routing**: Route requests to different services based on various predicates.
2. **Filtering**: Modify requests and responses before or after sending them to the backend services.
3. **Circuit Breaking**: Integration with circuit breakers for fault tolerance.
4. **Rate Limiting**: Protect services from overuse.
5. **Path Rewriting**: Modify the request path before forwarding.
6. **Load Balancing**: Distribute requests across multiple instances.
7. **Security**: Authentication and authorization at the gateway level.

* **Implementation:**

Routes can be configured using:

* Java configuration with RouteLocatorBuilder
* Properties in application.yml
* Discovery client integration for automatic route creation



[3.2.7 Spring Cloud Gateway]

**3.2.8 JWT Authentication & Authorization**

JSON Web Tokens (JWT) provide a compact, self-contained means for securely transmitting information between parties as a JSON object. They are commonly used for authentication and authorization in microservices architectures.

* **JWT Components:**

1. **Header**: Contains token type and signing algorithm
2. **Payload**: Contains claims (user data, permissions, expiration time)
3. **Signature**: Ensures the token hasn't been altered

* **Implementation in Spring Boot:**

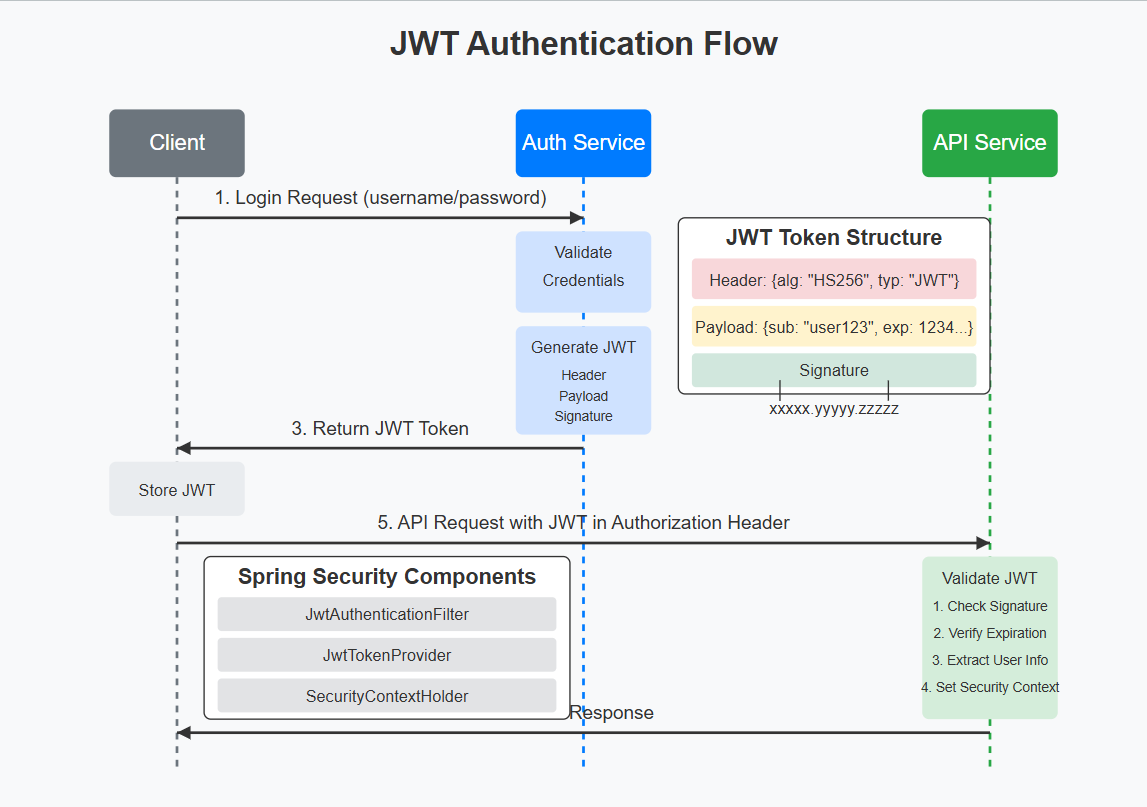
1. **Authentication**: Validate user credentials and generate a JWT
2. **Token Storage**: Client stores the token, typically in local storage or cookies
3. **Authorization**: Each request includes the JWT in the Authorization header
4. **Token Validation**: Verify the token's signature and expiration
5. **User Details Extraction**: Extract user information and authorities from the token

* **Spring Security Integration:**

1. **Authentication Filter**: Intercepts requests to validate JWTs
2. **UserDetailsService**: Loads user-specific data
3. **PasswordEncoder**: Securely hashes passwords
4. **SecurityContext**: Stores the authenticated user details

* **Best Practices:**

1. Use HTTPS to protect token transmission
2. Set appropriate token expiration times
3. Store sensitive user information in token claims judiciously
4. Implement token revocation strategies

****

[3.2.8 JWT Authentication]

**3.2.9 Exception Handling (@ControllerAdvice)**

Proper exception handling is crucial for creating robust REST APIs. Spring Boot provides mechanisms to handle exceptions gracefully and return appropriate error responses to clients.

* **Global Exception Handling with @ControllerAdvice:**

@ControllerAdvice is a specialized @Component that allows you to handle exceptions across the whole application, not just within a single controller.

* **Key Components:**

1. **@ControllerAdvice**: Annotation to create a global exception handler
2. **@ExceptionHandler**: Annotation to handle specific exception types
3. **ResponseEntity**: Class to customize the HTTP response
4. **Custom Error Response**: POJO representing error details

* **Best Practices:**

1. Create custom exception classes for different error scenarios
2. Return appropriate HTTP status codes
3. Provide meaningful error messages
4. Include error codes for client reference
5. Log exceptions for debugging purposes
6. Consider security implications (don't expose sensitive information)

**3.2.10 Application Properties Configuration**

Spring Boot provides a flexible mechanism for configuring applications through properties files, YAML files, environment variables, and command-line arguments.

* **Configuration Sources (in order of precedence):**

1. Command-line arguments
2. Java System properties
3. Application properties outside the packaged application
4. Application properties packaged inside the application

* **Property Files:**

1. **application.properties/application.yml**: Default configuration file
2. **application-{profile}.properties**: Profile-specific configuration
3. **bootstrap.properties**: Configuration loaded before the main application context

* **Configuration Types:**

1. **Application Configuration**: Custom application settings
2. **Spring Boot Configuration**: Built-in properties that control Spring Boot behavior
3. **Logging Configuration**: Control logging behavior and levels
4. **Web Properties**: Configure embedded servers, MVC, etc.
5. **Data Properties**: Database connections, JPA settings, etc.
6. **Actuator Properties**: Monitoring and management endpoints

* **Externalized Configuration:**

1. **@Value**: Inject property values into beans
2. **@ConfigurationProperties**: Bind properties to structured objects
3. **Environment**: Programmatically access properties
4. **Profiles**: Environment-specific configurations

* **Cloud-Native Configuration:**

1. **Config Server**: Centralized configuration with Spring Cloud Config
2. **Vault**: Secret management

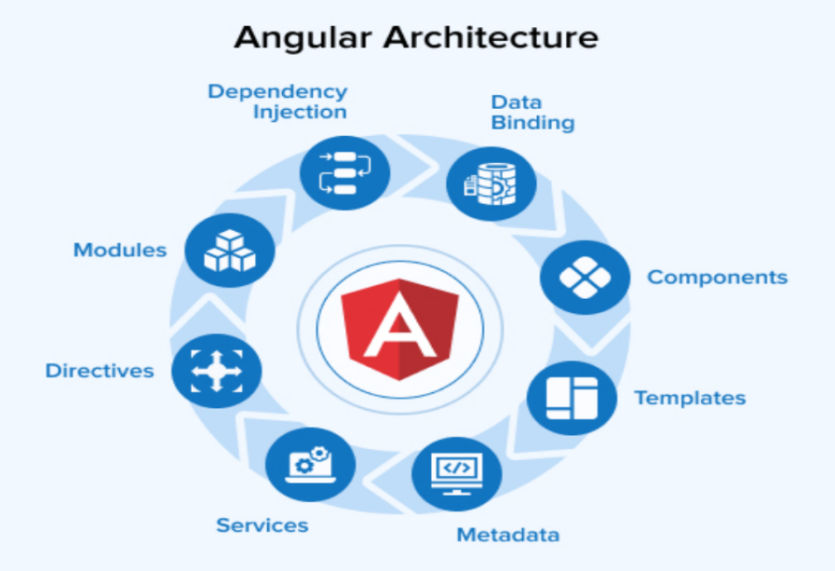
**3.3 Frontend Development – Angular**

**3.3.1 Angular Overview**

Angular is a robust platform and framework developed by Google for building dynamic client-side applications. It utilizes HTML, CSS, and TypeScript to create responsive and feature-rich web applications. Angular follows a component-based architecture that promotes code reusability, maintainability, and testability.

The core architectural concepts of Angular include:

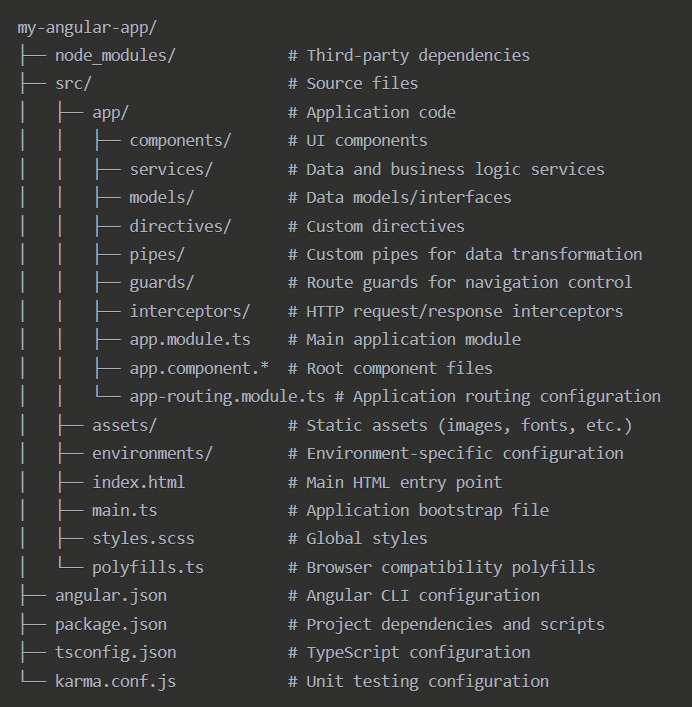
* **Components**: Self-contained building blocks that control specific portions of the user interface
* **Templates**: HTML-based views enhanced with Angular's directives and binding syntax
* **Services**: Specialized classes that encapsulate business logic, data manipulation, and external API communications
* **Modules**: Organizational units that group related components, directives, pipes, and services
* **Dependency Injection**: A design pattern that enhances efficiency by providing components with their required dependencies

**

[3.3.1 Angular Overview]

**3.3.2 Project Structure**

Angular applications follow a standardized project structure that promotes organization and scalability. The Angular CLI automatically generates this structure when creating a new project:



[3.3.2 Project Structure]

This structure enforces separation of concerns and modular development, making the application easier to maintain as it grows in complexity.

**3.3.3 Components & Modules**

* **Components**

Components are the fundamental building blocks of Angular applications. Each component encapsulates:

1. **Component Class**: A TypeScript class decorated with @Component that contains the component's logic
2. **HTML Template**: Defines the component's visual representation
3. **Styles**: CSS, SCSS, or other styling specific to the component
4. **Metadata**: Configuration information about the component provided through the @Component decorator

Components follow a hierarchical structure, with a root component (typically AppComponent) that hosts child components, which may in turn host their own children, creating a component tree.

**3.3.4 Services and Dependency Injection**

* **Services**

Services in Angular are singleton objects that provide functionality across the application. They serve several key purposes:

1. **Data Access**: Communicating with external APIs and servers
2. **Business Logic**: Implementing application-specific logic
3. **State Management**: Maintaining application state
4. **Cross-Component Communication**: Enabling components to share data

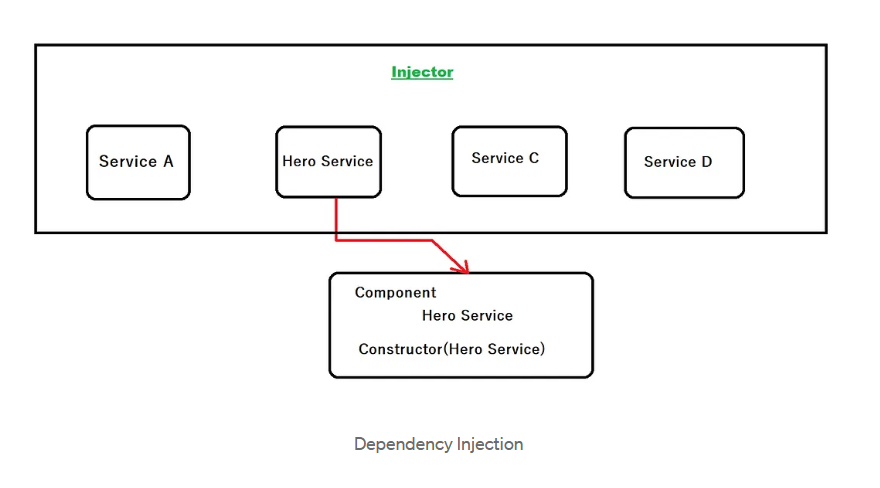
Services are class-based and typically decorated with @Injectable(), indicating they can be injected as dependencies.

* **Dependency Injection System**

Angular's dependency injection (DI) system is a powerful pattern that:

1. Provides declared dependencies to classes when instantiated
2. Reduces coupling between components and their dependencies
3. Enhances testability by allowing dependencies to be mocked
4. Promotes the Single Responsibility Principle

The DI system operates through a hierarchy of injectors that follow the component tree structure, allowing for efficient resource management.



[3.3.4 Services and Dependency Injection]

**3.3.5 Data Binding & Event Handling**

Angular offers various mechanisms for communication between component classes and their templates:

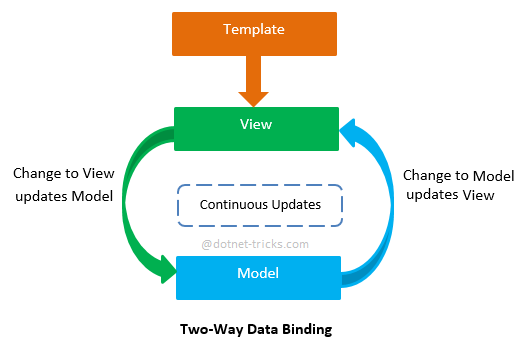
* **Data Binding Types**

1. **Interpolation** {{expression}}: Displays component property values in the view
2. **Property Binding** [property]="expression": Sets properties of HTML elements or directives
3. **Event Binding** (event)="handler()": Responds to user actions and events
4. **Two-way Binding** [(ngModel)]="property": Combines property and event binding for bidirectional data flow

* **Event Handling**

Event handling in Angular allows components to respond to user interactions:

1. **Event Binding**: Capture DOM events like clicks, mouse movements, and keyboard actions
2. **Event Objects**: Access event details through $event parameter
3. **Custom Events**: Create and emit component-specific events using EventEmitter
4. **Event Propagation**: Control event bubbling and capturing phases



[3.3.5 Data Binding & Event Handling]

**3.3.6 Routing & Navigation**

Angular Router enables navigation between different views in single-page applications:

* **Key Routing Concepts**

1. **Route Configuration**: Define application routes, each mapping a URL path to a component
2. **Router Outlet**: A directive that marks where the router should display the component for the active route
3. **Router Links**: Directives for navigating between routes through HTML elements
4. **Programmatic Navigation**: Navigate via code using the Router service
5. **Route Parameters**: Pass and retrieve data through URL segments
6. **Query Parameters**: Pass optional parameters through the URL
7. **Route Guards**: Protect routes based on conditions like authentication status

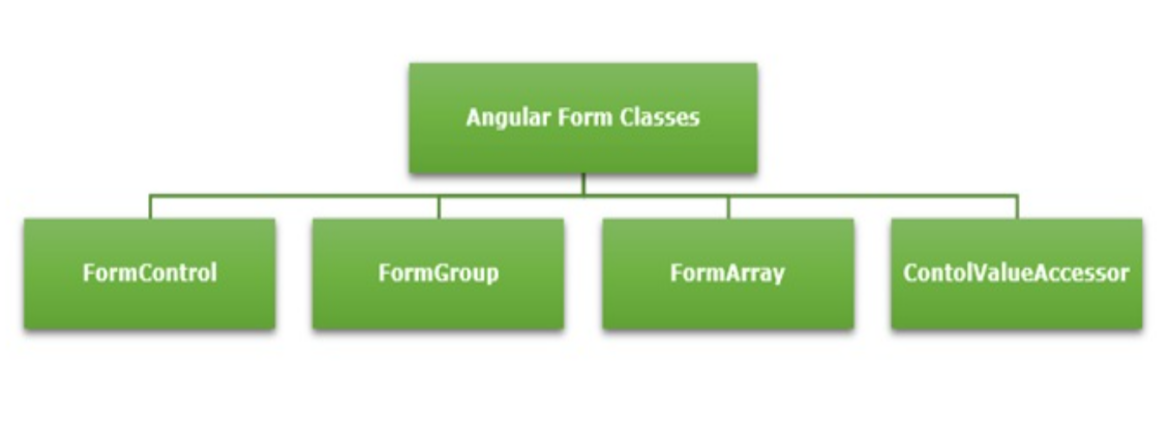
**3.3.7 Reactive Forms & Validation**

Reactive forms provide a model-driven approach to handling form inputs with explicit form control objects:

* **Form Building Blocks**

1. **FormControl**: Manages the value and validation state of an individual form control
2. **FormGroup**: Tracks the value and validity state of a group of FormControl instances
3. **FormArray**: Manages a variable number of form controls
4. **FormBuilder**: A service that simplifies the creation of complex form structures

* **Validation Approaches**

1. **Built-in Validators**: Required, minLength, maxLength, pattern, email, etc.
2. **Custom Validators**: Functions that implement specific validation logic
3. **Asynchronous Validators**: Validators that perform asynchronous operations like API calls
4. ****Cross-field Validation**: Validate related fields against each other

[3.3.7 Reactive Forms & Validation]

**3.3.8 API Integration using HttpClient**

Angular's HttpClient provides a modern, powerful client for making HTTP requests:

* **Key HttpClient Features**

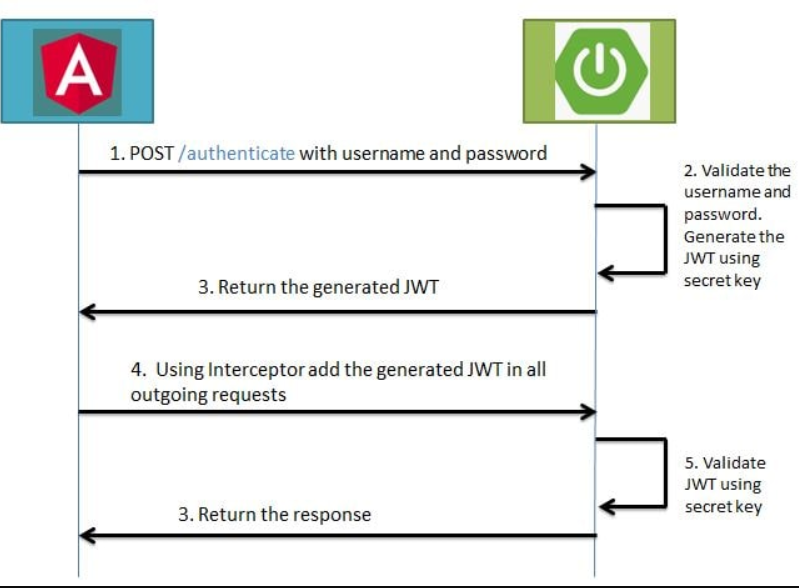
1. **HTTP Methods**: Support for GET, POST, PUT, PATCH, DELETE, and other HTTP methods
2. **Request Configuration**: Options for headers, parameters, response type, etc.
3. **Typed Responses**: Type-safe HTTP operations with generic typing
4. **Observables**: Asynchronous handling of HTTP operations using RxJS
5. **Error Handling**: Centralized error processing
6. **Interceptors**: Modify or transform HTTP requests and responses globally

**3.3.9 Consuming JWT-Protected APIs**

JSON Web Tokens (JWT) provide a secure method for authentication and authorization in Angular applications:

* **JWT Implementation Components**

1. **Authentication Service**: Manages user login, token storage, and logout operations
2. **Token Storage**: Securely store JWT tokens using browser storage mechanisms
3. **HTTP Interceptors**: Automatically attach JWT tokens to outgoing HTTP requests
4. **Authentication State**: Track and expose the user's authentication status
5. **Token Expiration**: Handle token refreshing or session expiration
6. **Authorization Guards**: Protect routes based on authentication status and user roles

**

[3.3.9 Consuming JWT-Protected APIs]

**Project : Jetwayz-Flight Management System**

## **4.1 Introduction & Objective:**

Jetwayz is a comprehensive full-stack web application designed to streamline flight management operations, similar to commercial airline booking platforms. It enables users to search for flights, make bookings, and manage their travel itineraries. The application also provides an administrative interface for airline staff to manage flights, schedules, and passenger information. The primary goal is to deliver a secure, efficient, and user-friendly platform for both travelers and airline personnel with a responsive and intuitive UI.

## **4.2 Tech Stack Overview:**

Jetwayz employs the following technologies and tools:

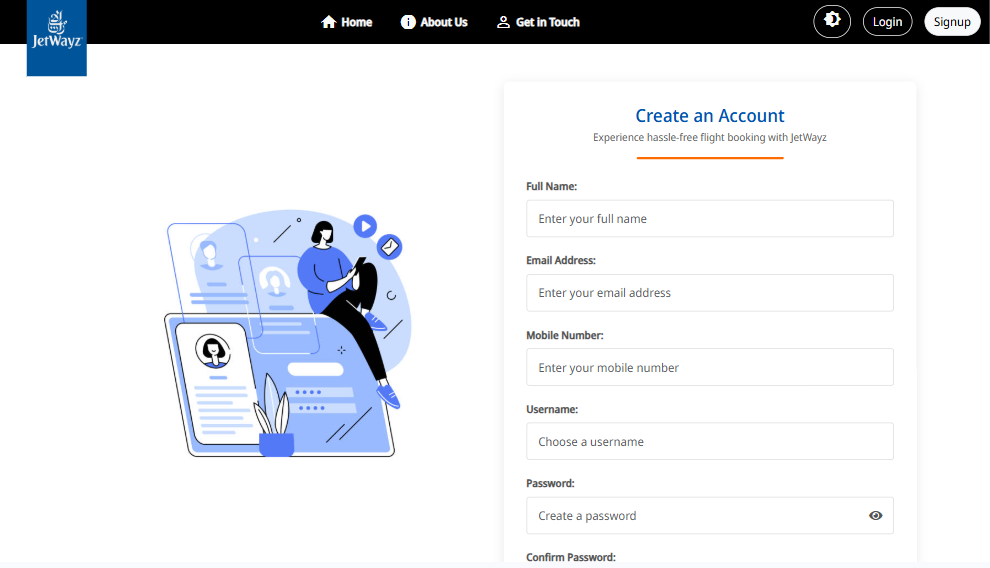
* Frontend:
* Angular: Single-page application framework for dynamic and responsive UI
* Angular Material: Component library for consistent design language
* SCSS: For advanced styling capabilities and UI refinement
* RxJS: For reactive programming and handling asynchronous operations
* Backend:
* Spring Boot: Java-based framework for creating stand-alone, production-grade applications
* Spring Security: Authentication and authorization framework
* Spring Data JPA: Data access layer abstraction
* RESTful API: For client-server communication
* Database:
* MySQL: Relational database to store user data, flight information, and bookings
* Hibernate: ORM for database mapping and entity relationships
* Authentication & Security:
* JWT (JSON Web Tokens): For stateless authentication
* Spring Security: Role-based access control and request filtering
* Password encryption with BCrypt
* Other Tools:
* Maven: Dependency management and build automation
* Swagger UI: API documentation

## **4.3 Project Features:**

* User Registration and Secure Login
* Flight Search with Multiple Filters
* Booking Management System (Create, View, Cancel)
* Seat Selection Interface
* Payment Processing Integration
* Booking Confirmation and E-Ticket Generation
* Administrative Dashboard for Flight Management
* Real-time Flight Status Updates
* Form validation and error handling
* Responsive layout across devices

### **User Registration Page UI:**

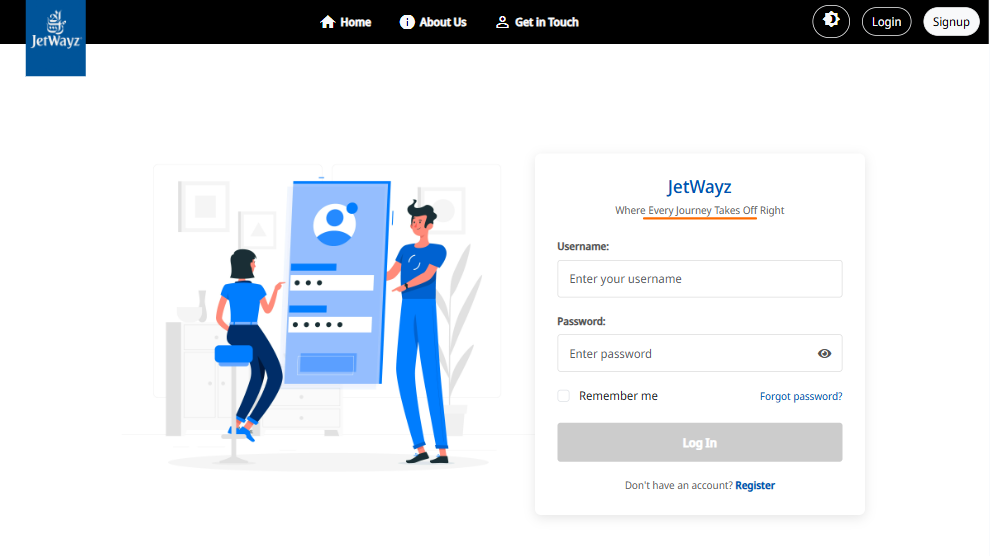
* The Registration Page allows new users to create an account on the platform. Users enter their personal details including name, email, password, phone number, and address. The data is validated client-side using Angular's reactive form validators and then submitted to the Spring Boot backend.
* On the server side, the system validates the data again and encrypts the password using BCrypt before storing it in the MySQL database. The system implements Spring Security for authentication, ensuring only properly registered users can access protected areas of the application.
* Email verification is implemented to confirm user identity. Upon successful registration, a confirmation email is sent to the user with a verification link. Once verified, users can log in and access the flight booking features.



[4.3.1 Registration Page UI]

### **Login Page UI:**

* The Login Page provides a secure authentication gateway for registered users. Users enter their email and password, which are validated against the database using Spring Security authentication providers.
* Upon successful authentication, a JWT token is generated and returned to the client. This token is stored in the browser's local storage and used for subsequent API requests to verify the user's identity.
* The page implements security features such as account lockout after multiple failed login attempts and password recovery options. If authentication fails, appropriate error messages guide the user on how to proceed.



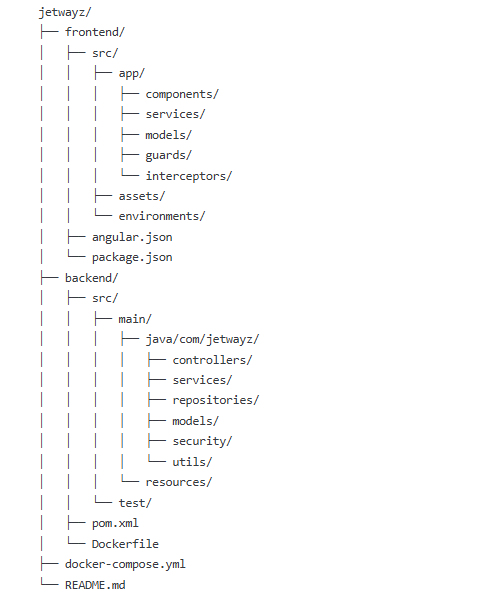
[4.3.2 Login Page UI]

## **4.4 MVC Architecture:**

* The project follows the Model-View-Controller design pattern with a clear separation of concerns:
* Model: Represents data objects like users, flights, and bookings using JPA entities.
* View: Angular components render dynamic content with Material Design elements.
* Controller: Spring Boot REST controllers handle client requests and application logic.

Example: FlightController.java → Processes requests, interacts with services, and returns responses to the Angular frontend that displays the data in flight-list.component.html.

## **4.5 Folder Structure Explanation:**



[4.5.1 Folder Structure]

### **frontend/src/app/**

The frontend layer is built with Angular framework and follows a component-based architecture.

* **components/:** Contains all Angular components organized by feature.

1. flight-search/: Components for searching and filtering flights.
2. booking/: Components for creating and managing bookings.
3. user-profile/: Components for user profile management.
4. admin/: Components for administrative functions.

* **services/:** Contains Angular services that handle data fetching and business logic.

1. auth.service.ts: Manages authentication state and user sessions.
2. flight.service.ts: Handles flight-related API calls.
3. booking.service.ts: Manages booking operations and state.

* **models/:** Contains TypeScript interfaces representing data structures.

1. user.model.ts: Defines user data structure.
2. flight.model.ts: Defines flight information structure.
3. booking.model.ts: Defines booking data structure.

* **guards/:** Contains route guards for protected navigation.

1. auth.guard.ts: Prevents unauthorized access to protected routes.
2. admin.guard.ts: Ensures only admin users can access administrative routes.

* **interceptors/:** Contains HTTP interceptors for request/response handling.

1. token.interceptor.ts: Adds authentication tokens to outgoing requests.
2. error.interceptor.ts: Handles common error responses.

### **backend/src/main/java/com/jetwayz/**

The backend is built with Spring Boot and follows a layered architecture pattern.

* **controllers/:** Contains REST API controllers that handle HTTP requests.

1. FlightController.java: Handles flight-related endpoints.
2. BookingController.java: Manages booking operations.
3. UserController.java: Handles user-related requests.
4. AdminController.java: Provides administrative endpoints.

* **services/:** Contains business logic implementation.

1. FlightService.java: Implements flight-related business logic.
2. BookingService.java: Handles booking operations logic.
3. UserService.java: Manages user-related operations.

* **repositories/:** Contains data access interfaces.

1. FlightRepository.java: Provides database operations for flights.
2. BookingRepository.java: Data access for bookings.
3. UserRepository.java: User data operations.

* **models/:** Contains JPA entity classes.

1. User.java: Entity representing user data.
2. Flight.java: Entity representing flight information.
3. Booking.java: Entity representing booking records.

* **security/:** Contains security configurations and utilities.

1. WebSecurityConfig.java: Spring Security configuration.
2. JwtTokenProvider.java: JWT generation and validation.
3. UserDetailsServiceImpl.java: User details service implementation.

* **utils/:** Contains utility classes and helpers.

1. ApiResponse.java: Standardized API response format.
2. DateTimeUtils.java: Date manipulation utilities.
3. EmailService.java: Email notification service.

### **resources/**

Contains application configuration files and static resources.

* **application.properties:** Application configuration.
* **data.sql:** Initial database seeding script.

**4.6 Authentication & Authorization:**

Authentication and authorization in Jetwayz are handled using Spring Security and JWT tokens, ensuring secure access to protected resources.

### **Registration & Login:**

* Registration: Users register with personal details and password. The password is encrypted using BCrypt before storage in the database.
* Login: During login, Spring Security authenticates the user and, if successful, generates a JWT token containing the user's identity and roles.

### **Token Management:**

* JWT tokens are signed with a secret key and include an expiration time for security.
* Tokens are included in the Authorization header for all API requests.
* The TokenInterceptor in Angular automatically adds the token to outgoing requests.

### **Protected Routes:**

* Angular Route Guards: Prevent unauthorized navigation to protected routes in the frontend.
* Spring Security Filters: Validate JWT tokens and enforce role-based access control on backend endpoints.

### **Authorization Levels:**

* USER: Standard access for flight search and booking management.
* ADMIN: Enhanced access for flight administration and user management.

### **Authorization Flow:**

* When a user logs in, a JWT token with roles is generated.
* Each API request includes this token in the Authorization header.
* The JwtAuthenticationFilter validates the token and sets up Spring Security context.
* Method-level security annotations (@PreAuthorize) control access to specific operations.

**4.7 Middleware and Utilities:**

* **Custom Middleware:**
* JwtAuthenticationFilter – Validates tokens and sets up security context.
* GlobalExceptionHandler – Centralized exception handling with appropriate HTTP responses.
* LoggingInterceptor – Logs request/response details for debugging and auditing.
* **Utility Services:**
* EmailService: Sends booking confirmations and notifications.
* PdfGeneratorService: Generates e-tickets and boarding passes.
* PaymentGatewayService: Handles payment processing and validation.
* **Angular Interceptors:**
* TokenInterceptor: Automatically adds authentication tokens to requests.
* ErrorInterceptor: Handles API errors consistently across the application.

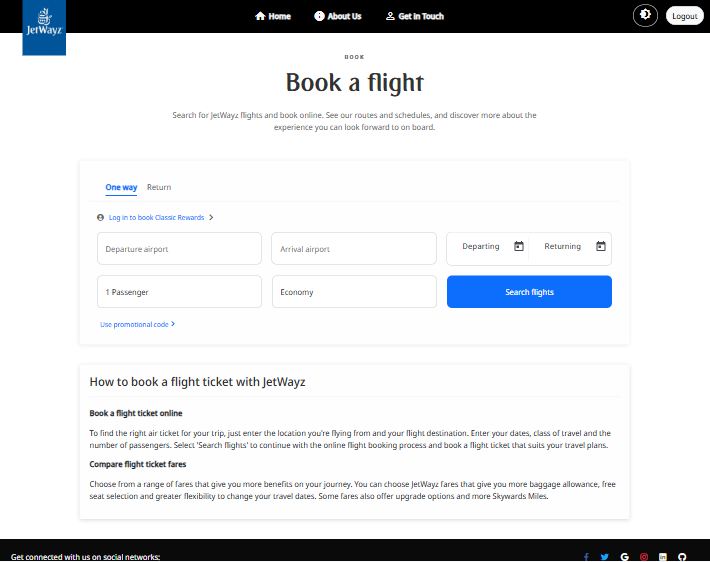
**4.8 Screenshots & Output Pages**

### **Flight Search Page UI**

The Flight Search Page allows users to search for available flights based on origin, destination, date, and passenger count. The page features an intuitive form with autocomplete for airports, calendar for date selection, and real-time validation.

The search results are displayed in a sortable and filterable table showing flight numbers, times, prices, and available seats. Users can apply filters for direct flights, specific airlines, or price ranges. The responsive design adapts seamlessly to different screen sizes, providing an optimal experience on both desktop and mobile devices.

Upon finding a suitable flight, users can proceed to the booking process with a single click. The page communicates with the Spring Boot backend via RESTful API calls and displays real-time availability of flights.



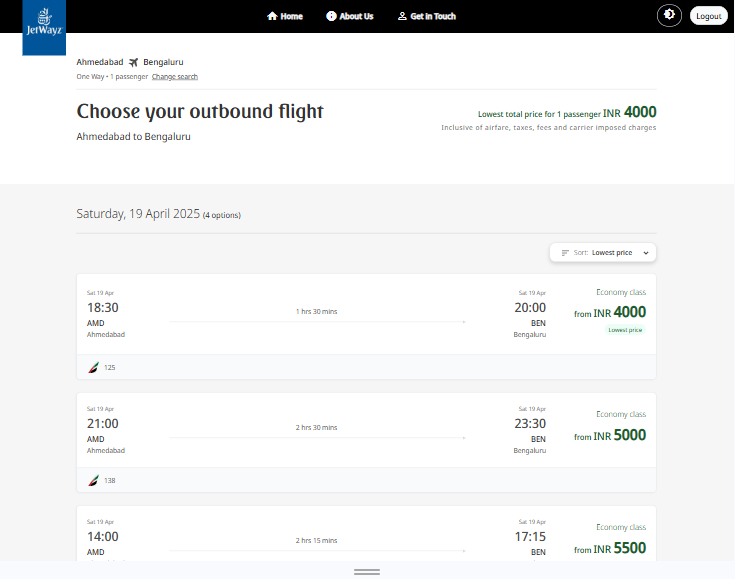
[4.8.1 Flight Search Page UI]

### **Flight Result Page UI**

The Flight Results page presents a clean layout showing available flights matching user search criteria. The header displays the journey route with an arrow icon, along with trip type and passenger count. Below, users see a title section with "Choose your outbound flight" and route details, plus pricing information showing the lowest fare available.

The main content shows the current date and number of options, with a sorting dropdown to organize results by different criteria. Each flight appears in a clickable card displaying departure and arrival details, flight duration, pricing with cabin class, and airline information.

The responsive design ensures optimal viewing across all devices with consistent styling throughout the interface. Users can easily select their preferred flight by clicking on the entire card to proceed with the booking process.



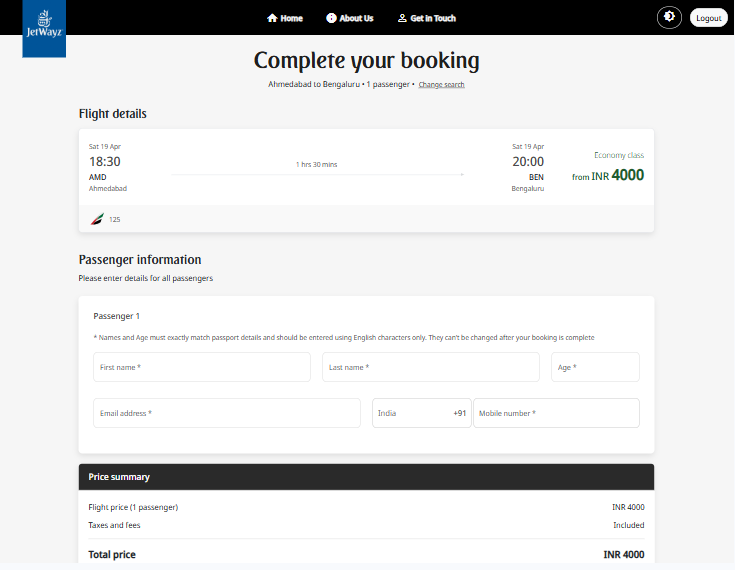
[4.8.2 Flight Result Page UI]

### **Passenger Details Fill-up Page UI**

The Passenger Details page collects required information for all travelers in the booking. The interface features a clean form with sections for personal information (name, date of birth), contact details (email, phone), and travel document information (passport/ID number, expiration date).

Real-time validation highlights completed fields and errors with appropriate guidance. Returning users can select previously stored passenger profiles. Special request options allow passengers to indicate meal preferences, mobility needs, or other accommodations.

The page includes a booking summary showing flight details, selected seats, and total cost. Once all required fields are validated, the "Continue to Payment" button becomes active, allowing users to proceed to the final step.



[4.8.3 Passenger Details Fill-up Page UI]

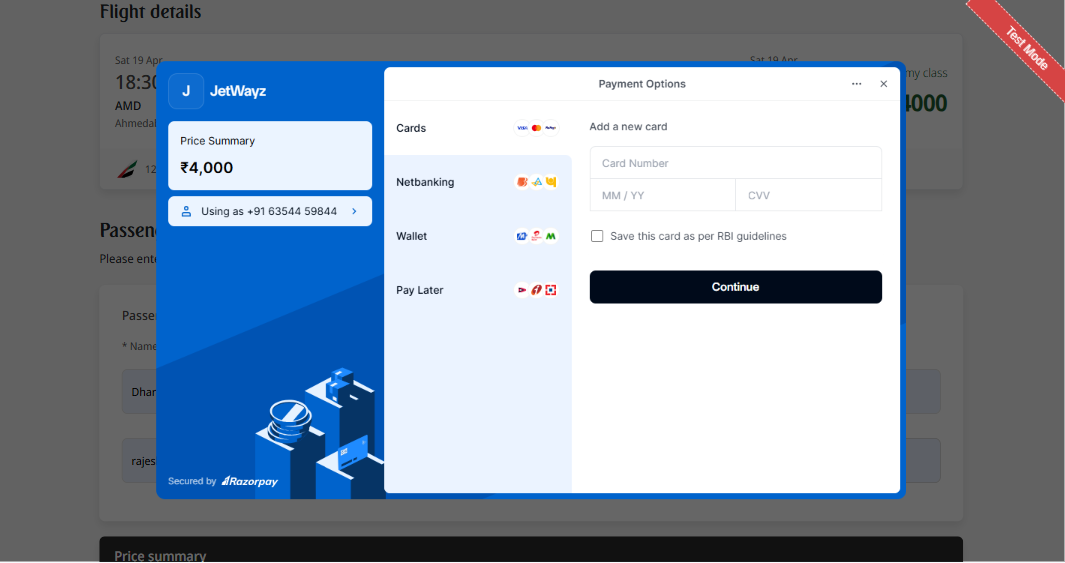
### **Payment Page UI**

The Payment Page provides a secure interface for completing the booking transaction. At the top, a booking summary recaps flight details, passenger names, selected seats, and a price breakdown showing base fare, taxes, fees, and total amount.

The payment section offers multiple options including credit/debit cards, digital wallets, bank transfers, and airline vouchers. For card payments, the interface includes fields for card number, cardholder name, expiration date, and CVV with real-time validation.

Security features are prominently displayed, including encryption indicators and security badges. Users can enter billing address details or select "Same as contact address" to reuse previously entered information.

A "Complete Booking" button initiates the payment process. Upon success, users receive a booking reference number and e-ticket details with options to download, print, or email the confirmation.

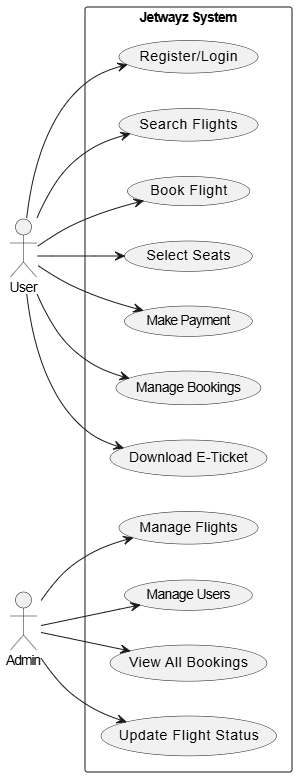


[4.8.4 Payment Page UI]

**4.9 UML Diagrams**

### **Use Case Diagram:**

A use case diagram for Jetwayz illustrates the interactions between different actors (passengers, administrators) and the system. It highlights the core functionalities such as flight search, booking management, and administrative operations.



[4.9.1 Use Case Diagram]

Key actors include:

* Passengers (searching flights, making bookings, managing profiles)
* Administrators (managing flights, handling user accounts)

The diagram clearly delineates the boundaries of the system and shows how different user types interact with various features, providing a high-level overview of system functionality and user interactions.

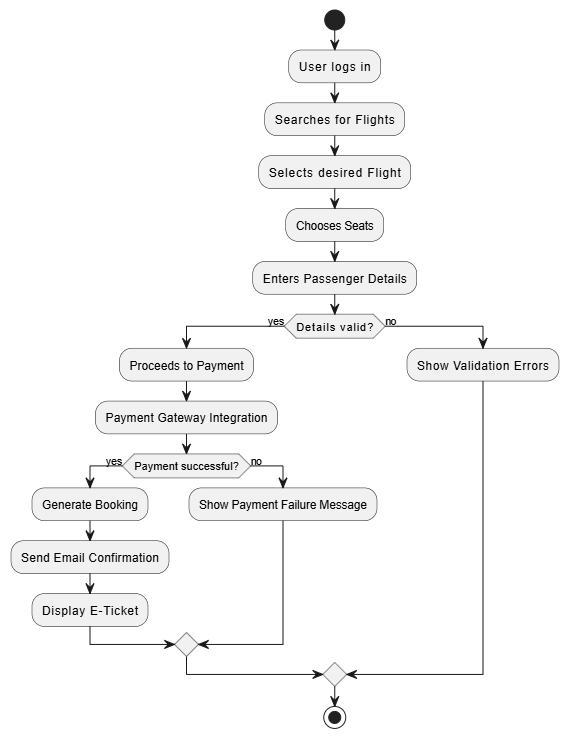
### **Activity Diagram:**

The activity diagram for the flight booking process illustrates the sequential flow of activities from initial flight search to booking confirmation. It shows decision points, parallel activities, and the interaction between user actions and system processes.

Key activities include:

1. User searches for flights
2. System displays available flights
3. User selects a flight
4. User chooses seats
5. User enters passenger details
6. System validates information
7. User completes payment
8. System processes payment
9. System generates booking confirmation
10. User receives e-ticket

The diagram helps visualize the complete booking workflow and identify potential bottlenecks or optimization opportunities in the process.



[4.9.2 Activity Diagram]

### **Class Diagram:**

The class diagram for Jetwayz illustrates the static structure of the system, showing classes, their attributes, methods, and relationships. It provides a blueprint of the system's object-oriented design and highlights key entity relationships.

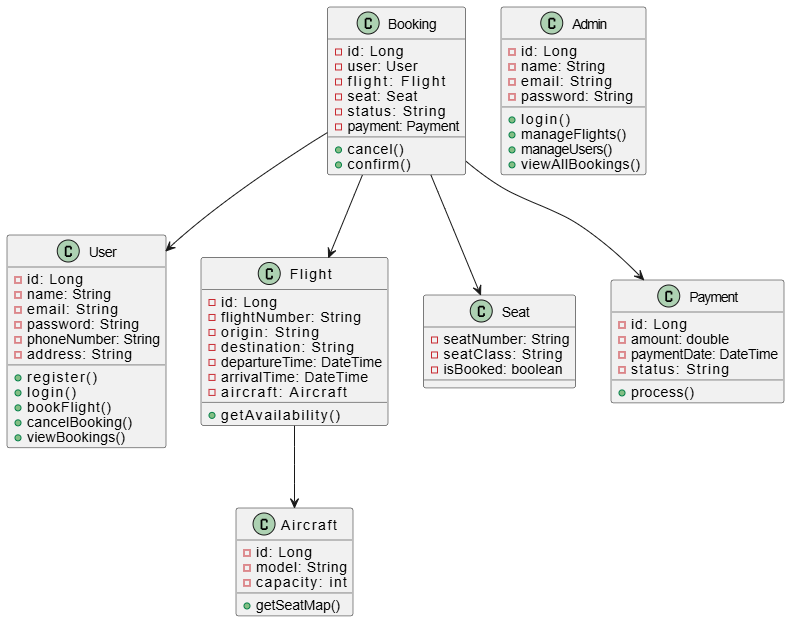
Major classes include:

* User (with subclasses Passenger, Administrator)
* Flight
* Booking
* Aircraft
* Payment
* Seat

Relationships demonstrate how these entities interact, such as:

* A Booking belongs to a User
* A Booking references a Flight
* A Flight has many Seats
* A Payment is associated with a Booking

The diagram helps developers understand the system architecture and data model, facilitating consistent implementation across the codebase.



[4.9.3 Class Diagram]

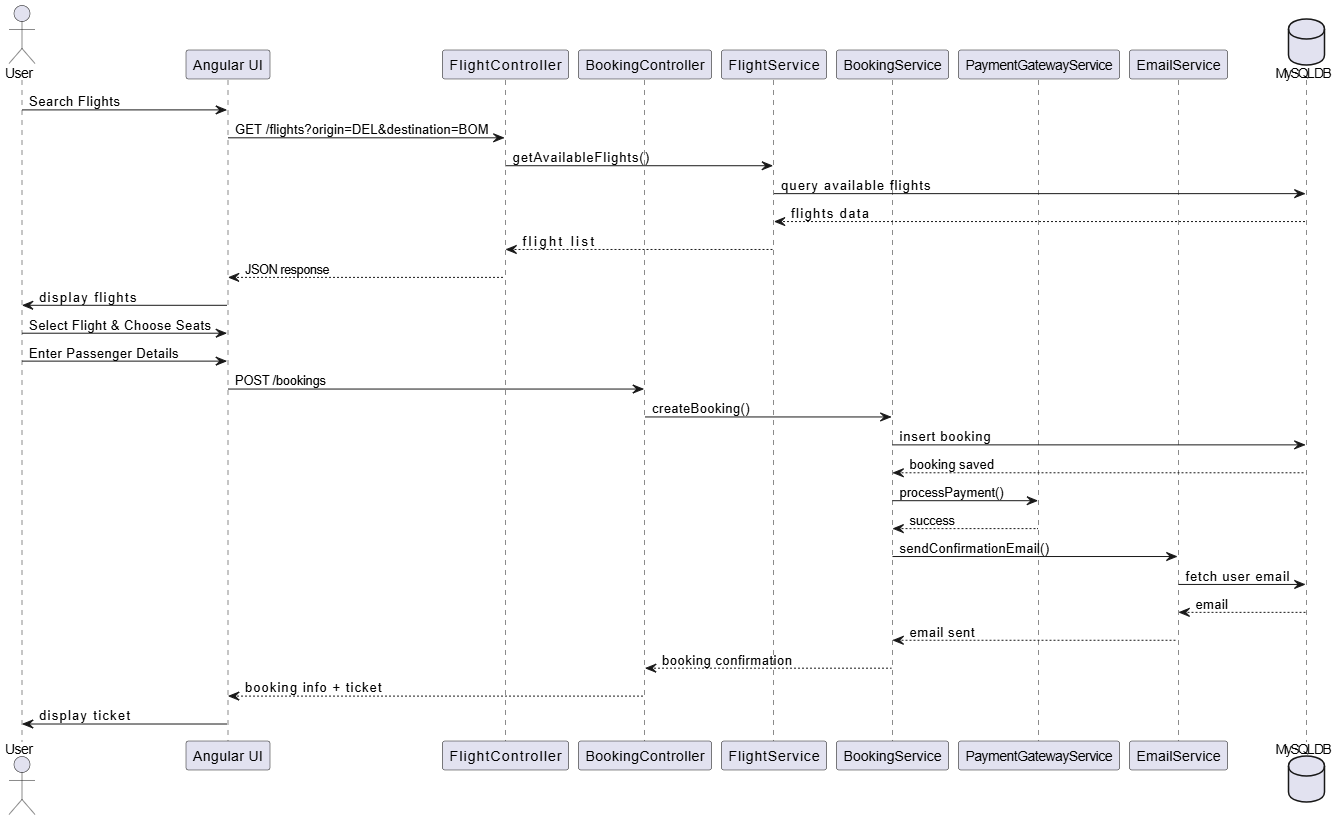
### **Sequence Diagram:**

The sequence diagram for the flight booking process illustrates the time-ordered interactions between different components of the system. It shows how objects communicate with each other during the booking workflow.

The diagram demonstrates:

* How the Angular frontend communicates with Spring Boot controllers
* The flow of data between controllers, services, and repositories
* Database interactions for persisting booking information
* External service calls for payment processing and email notifications

This visualization helps understand the dynamic behavior of the system and identify potential performance issues or race conditions in the processing sequence.



[4.9.4 Sequence Diagram]

# My Skills & Work

## **Skills:**

* **Flight Management System Development:**

1. Full-Stack Web Development a. Angular Framework b. Component Architecture c. Responsive Design Principles d. State Management e. UI/UX Implementation f. Dynamic Form Handling
2. Backend Development a. Spring Boot Application Development b. RESTful API Design and Implementation c. Authentication & Authorization d. Database Integration e. Exception Handling f. Service Layer Implementation g. Java 8+ Features
3. Database Management a. MySQL Database Design b. SQL Query Optimization c. Entity Relationship Modeling d. Data Normalization e. Transaction Management
4. Development Workflow a. Git Version Control b. Branch Management
5. System Integration a. Payment Gateway Integration b. Email Notification Services c. Flight Data API Integration d. Map Services Integration

* **Web Development:**

1. HTML5
2. CSS3/SCSS
3. TypeScript
4. JavaScript ES6+
5. Bootstrap/Material Design

* **Soft Skills:**

1. Communication
2. Teamwork
3. Time Management
4. Problem Solving
5. Critical Thinking
6. Adaptability
7. Attention to Detail
8. Work Ethic

## **Work:**

"During my internship, I contributed significantly to the development of the JetWayz flight management system, focusing on both frontend and backend components. My primary responsibilities included implementing the flight booking module with interactive seat selection capabilities and developing the user authentication system with role-based access control for customers, airline staff, and administrators. I created a comprehensive reservation management system that handled the entire booking lifecycle, from search to confirmation, and integrated secure payment processing with multiple gateway options. Additionally, I designed and implemented the admin dashboard for monitoring system metrics and managing flight operations, along with a responsive customer profile management interface. On the research side, I investigated industry best practices for airline API integration, secure payment processing workflows, and efficient flight data caching strategies to enhance system performance. During deployment phases, I configured the Spring Boot application for various environments, documented API endpoints, conducted thorough cross-browser compatibility testing, and implemented automated testing for critical system components. Throughout the project, I adhered to agile methodologies and collaborated closely with senior developers to ensure code quality and timely delivery of features."

## **Tasks Completed During Internship:**

* **Development:**

1. Flight booking module implementation
2. User authentication and role-based access control
3. Reservation management system development
4. Payment processing integration
5. Booking confirmation and e-ticket generation
6. Flight search and filtering functionality
7. Admin dashboard implementation

* **Research:**

1. Research on airline API integration best practices
2. Research on secure payment processing workflows
3. Research on booking cancellation and refund policies
4. Research on flight data caching strategies
5. Research on performance optimization for search results

* **Deployment:**

1. Deployment of Angular components and services
2. Configuration of Spring Boot application for different environments
3. Database schema migration and management
4. Documentation of API endpoints and system architecture

# Conclusion

* After completing my internship at OPL Pvt. Ltd., I am pleased to conclude that my experience working on the JetWayz flight management system has been incredibly valuable and enriching. This comprehensive project has provided me with practical insights into enterprise software development that no classroom experience could match.
* Throughout my time working on JetWayz, I had the opportunity to develop and refine a diverse set of technical skills across the full development stack. From implementing complex flight booking modules to integrating secure payment processing systems, each challenge expanded my capabilities as a developer. The experience of building a responsive, user-friendly interface for multiple stakeholders—from administrators to airline staff to passengers—taught me the importance of understanding diverse user needs when designing digital solutions.
* I was fortunate to work alongside experienced professionals who offered guidance and constructive feedback, helping me improve both my coding practices and problem-solving approaches. Their mentorship significantly enhanced my understanding of software architecture and best practices in the industry. OPL's collaborative environment encouraged open communication and idea sharing, making it possible to overcome technical hurdles efficiently and learn continuously from my peers.
* The agile methodology adopted by OPL allowed me to experience real-world software development cycles, from planning and development to testing and deployment. This practical exposure to project management has given me valuable insights into how teams coordinate to deliver complex systems on schedule.
* In conclusion, my internship experience working on the JetWayz project at OPL Pvt. Ltd. has been transformative in my journey as a software engineer. The technical expertise, collaborative skills, and industry knowledge I've gained have prepared me well for future challenges in the field. I am confident that these experiences will serve as a strong foundation as I embark on my professional career in software development.

# References

* **Documents:**
* Angular Documentation
* Spring Boot Documentation
* MySQL Documentation
* RESTful API Design Guidelines
* **Online articles:**
* Baeldung: [https://www.baeldung.com/](https://www.baeldung.com/" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://www.baeldung.com/)
* Medium - Angular In Depth: [https://medium.com/angular-in-depth](https://medium.com/angular-in-depth" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://medium.com/angular-in-depth)
* Spring Framework Guru: [https://springframework.guru/](https://springframework.guru/" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://springframework.guru/)
* Stack Overflow Developer Survey 2024
* **Websites:**
* OPL Pvt. Ltd. (n.d.). About Us, Mission, Vision, Logo, etc... Retrieved from
* [https://www.oplinnovate.com/](https://www.oplinnovate.com/" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://www.oplinnovate.com/)
* [https://angular.io/docs](https://angular.io/docs" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://angular.io/docs)
* [https://docs.spring.io/spring-boot/docs/current/reference/html/](https://docs.spring.io/spring-boot/docs/current/reference/html/" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://docs.spring.io/spring-boot/docs/current/reference/html/)
* [https://dev.mysql.com/doc/](https://dev.mysql.com/doc/" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://dev.mysql.com/doc/)
* [https://s3-ap-southeast-1.amazonaws.com/gtusitecirculars/uploads/Internship\_\_\_Project\_Report\_Guidelines\_\_3180701\_\_661242.pdf](https://s3-ap-southeast-1.amazonaws.com/gtusitecirculars/uploads/Internship___Project_Report_Guidelines__3180701__661242.pdf" \t "C:\Users\DHARME~1\AppData\Local\Temp\_blank" \o "https://s3-ap-southeast-1.amazonaws.com/gtusitecirculars/uploads/internship___project_report_guidelines__3180701__661242.pdf)

# Mentor’s Feedback (Company Feedback)