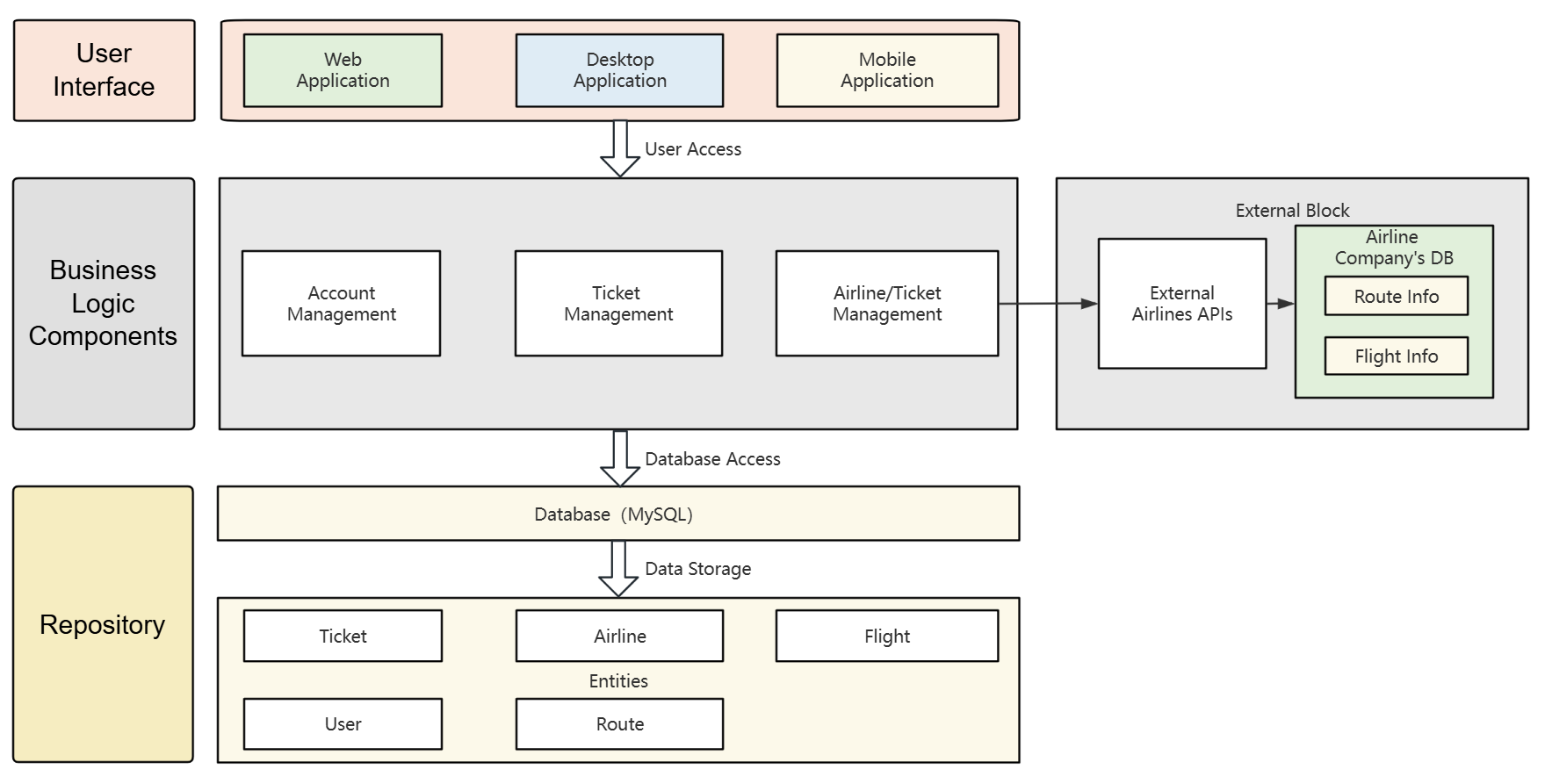
CPS-5301 Final Project

Architecture Design Documentation

Going Anywhere APP - Group 3

Cheng Bao(1335784), Xinin Qiu(1335785), Dinggen Zhang(1336574)

# Architectural Design Patterns



### 1.1. Layer Model（Main）

The Layered Model pattern decomposes a system into multiple levels, each representing a set of related functions. This pattern helps to separate concerns and allows different parts of the system to be developed and maintained independently.

* **UI Layer**：Includes Web Application, Desktop Application and Mobile Application components that handle the user interface and presentation logic.
* **Business Logic Layer**：Includes User Management, Ticket Management, and Airline Management components that handle business rules and logic.
* **Repository Layer**：Database component, responsible for data persistence and storage.

**Advantages**：

* **Modularity**: Each level can be developed and tested independently of the others.
* **Maintainability**: Changes at each level do not affect other levels, reducing maintenance costs.
* **Scalability**: New layers can be easily added or existing layers can be modified to suit new needs.

### 1.2. Repository Model

The Warehouse Pattern is a design pattern for data access that encapsulates data access logic in a centralized location, allowing the business logic layer to be decoupled from specific data access techniques.

In the User Management, Ticket Management, and Airline Management components, we use the Warehouse Pattern to encapsulate access to the database, so that these components do not need to know the specific details of the database implementation.

**Advantages**：

* **Decoupling**: The business logic layer is decoupled from the data access layer, which improves the maintainability and testability of the code.
* **Flexibility**: Facilitate database updates and maintenance without affecting the business logic layer.

# Components

Different Layers correspond to different components.

### 2.1. UI Layer

**Web Application**：Provides a browser-based user interface that allows users to access the system via the Internet.

**Desktop Application**：Provides a desktop application program interface that may need to be installed on the user's computer.

**Mobile Application**：Provides a mobile application program interface for smartphones or tablets.

### 2.2. Business Logic Layer

**User Management**：Handles user-related business logic such as registering, logging in, logging out, viewing and editing personal information, and managing passenger information.

**Ticket Management**：Handle business logic related to airline tickets, including searching for flights, viewing available tickets, locking/unlocking tickets, selecting passengers, purchasing tickets, refunding tickets, and changing tickets.

**Airline Management**：Handles business logic related to airlines, including viewing a list of connected airlines, adding airlines, viewing flight details, activating/deactivating flights, and deleting airlines. (The component needs to interact with APIs provided by external airlines to obtain real-time flight information)

### 2.3. Repository Layer

**Database**：Use a MySQL database to store the application's data.

**Entities**：Entities in the database, including Ticket, Airline, Flight, User, Route.

# Interface

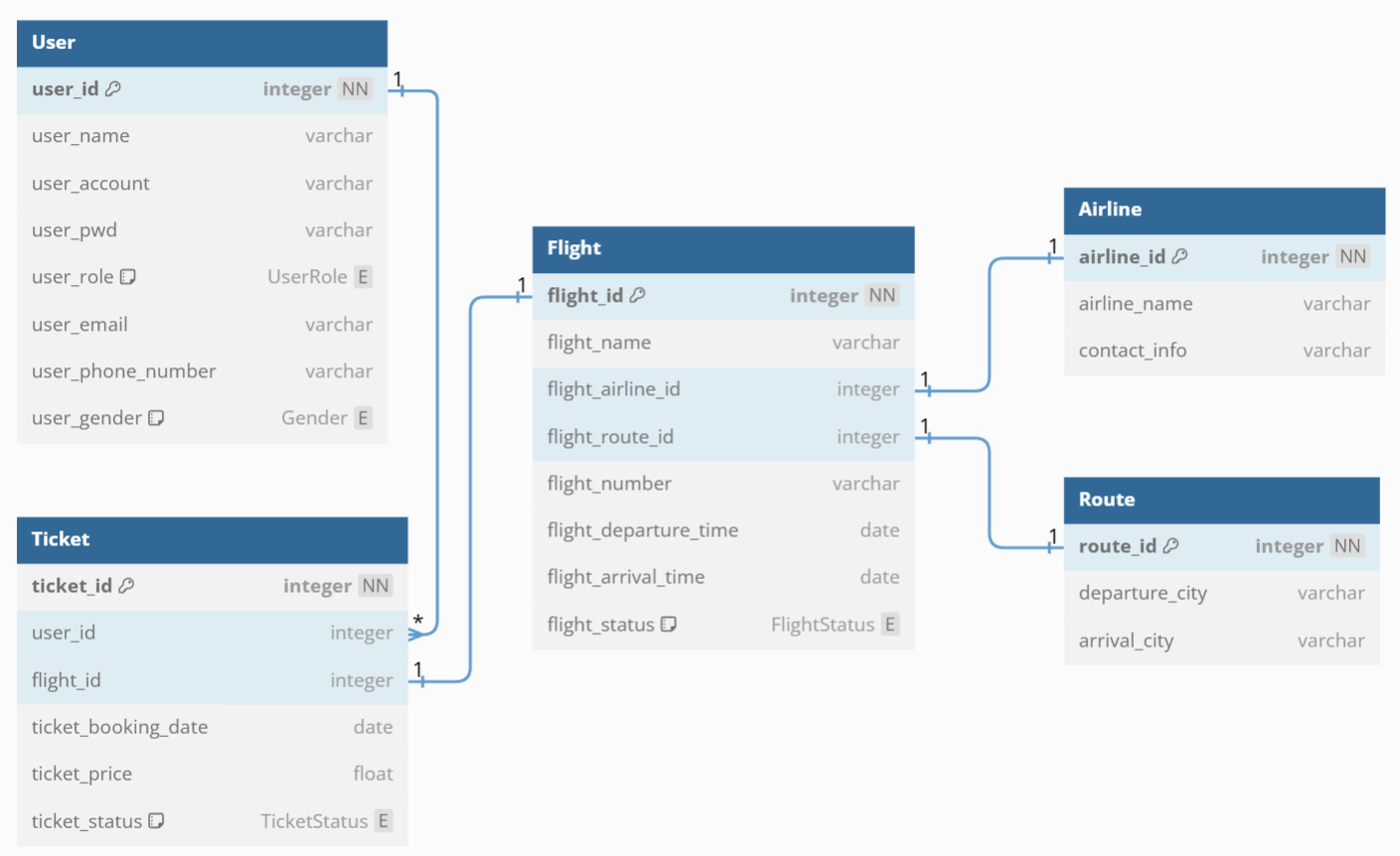
Clear interface design in the software architecture ensures that components can interact with each other seamlessly.

**UI Layer To Business Logic Layer**：This includes Web applications, desktop applications, and mobile applications, which interact with business logic components through the User Access Interface, a part that is mainly reflected in the front-end page interaction.

**Business Logic Layer to Repository Layer**：It includes user management, ticket management and airline management. User management and ticketing management interact directly with the database, while airline management obtains data through external airline APIs.

**Database To Entity** ：Use the interface provided by the MySQL database to store entity data such as tickets, airlines, flights, users and routes.

# Database Design



We use the DBML language to model the database

Five main Entity: User, Ticket, Flight, Airline and Route.

### 4.1. User

* **user\_id**: Primary key, integer type, non-null (NN).
* **user\_name**: User name, string type.
* **user\_account**: User account, string type.
* **user\_pwd**: User password, string type.
* **user\_role**: UserRole, enumerated type (UserRole), used to distinguish between user types.
* **user\_email**: User mailbox, string type.
* **user\_phone\_number**: User phone number, string type.
* **user\_gender**: The gender of the user, enumerated type (Gender).

### 4.2. Ticket

* **ticket\_id**: Primary key, integer type, non-null.
* **user\_id**: Foreign key, associated to the User table, representing the user who purchased the ticket.
* **flight\_id**: Foreign key, associated with the Flight table, indicating the flight to which the ticket belongs.
* **ticket\_booking\_date**: Ticket booking date, date type.
* **ticket\_price**: Airfare, floating point type.
* **ticket\_status**: Ticket status, enumerated type (TicketStatus).

### 4.3. Flight

* **flight\_id**: Primary key, integer type, non-null.
* **flight\_name**: Flight name, string type.
* **flight\_airline\_id**: Foreign key, associated with the Airline table, indicating the airline that operated the flight.
* **flight\_route\_id**: Foreign key, associated to the Route table, representing the route of the flight.
* **flight\_number**: Flight number, string type.
* **flight\_departure\_time**: Flight departure time, date time type.
* **flight\_arrival\_time**: Flight arrival time, date time type.
* **flight\_status**: FlightStatus, enumerated type (FlightStatus).

### 4.4. Airline

* **airline\_id**: Primary key, integer type, non-null.
* **airline\_name**: Airline name, string type.
* **contact\_info**: Contact information, string type.

### 4.5. Route

* **route\_id**: Primary key, integer type, non-null.
* **departure\_city**: Departure city, string type.
* **arrival\_city**: Arrival city, string type.

### 4.6. Codes that generated ER Diagram

// Use DBML to define database structure

// Docs: https://dbml.dbdiagram.io/docs

enum Gender{

M

F

None

}

enum UserRole{

User

Guest

Administrator

}

Table User {

user\_id integer [primary key, not null, increment]

user\_name varchar

user\_account varchar

user\_pwd varchar

user\_role UserRole

user\_email varchar

user\_phone\_number varchar

user\_gender Gender [default: 'None']

}

Table Flight {

flight\_id integer [primary key, not null, increment]

flight\_name varchar

flight\_airline\_id integer

flight\_route\_id integer

flight\_number varchar

flight\_departure\_time date

flight\_arrival\_time date

flight\_status FlightStatus [note: "Activated/Deactivated, default isActivated"]

}

Table Airline {

airline\_id integer [primary key, not null, increment]

airline\_name varchar

contact\_info varchar

}

Table Route {

route\_id integer [primary key, not null, increment]

departure\_city varchar

arrival\_city varchar

}

Table Ticket {

ticket\_id integer [primary key, not null, increment]

user\_id integer

flight\_id integer

ticket\_booking\_date date

ticket\_price float

ticket\_status TicketStatus [note: "Open/Locked/Purchased/Canceled/Changed/Refunded/Boarding/On-going/Completed, default is Open"]

}

Enum TicketStatus{

Open

Locked

Purchased

Canceled

Changed

Refunded

Boarding

OnGoing

Completed

}

Enum FlightStatus{

Activated

Deactivated

}

Ref: Ticket.flight\_id - Flight.flight\_id

Ref: Flight.flight\_airline\_id - Airline.airline\_id

Ref: Flight.flight\_route\_id - Route.route\_id

Ref: Ticket.user\_id > User.user\_id // many-to-one