Food Wars Presentation

TDDP3
Group 1

Contents

01

Introduction

Problems in Traditional Classrooms

Stimulation

Difficult for teachers to stimulate the learning interests of students

Interaction

Lacks an interactive learning activity for students to do after class

Assessability

Difficult for teachers to continuously assess students' mastery

Project Task | Food Wars

Learn

They will know if their theories learnt are applied correctly from the feedback of the questions

Compete

Multiplayer mode to compete with friends. Leaderboard aspect to see how they rank among their peers

Assess

Progressive levels with increasing difficulty. Full mastery of course if they clear all the levels

Technologies we have used

Front End

unity

Back End

node JS

Database

Firebase

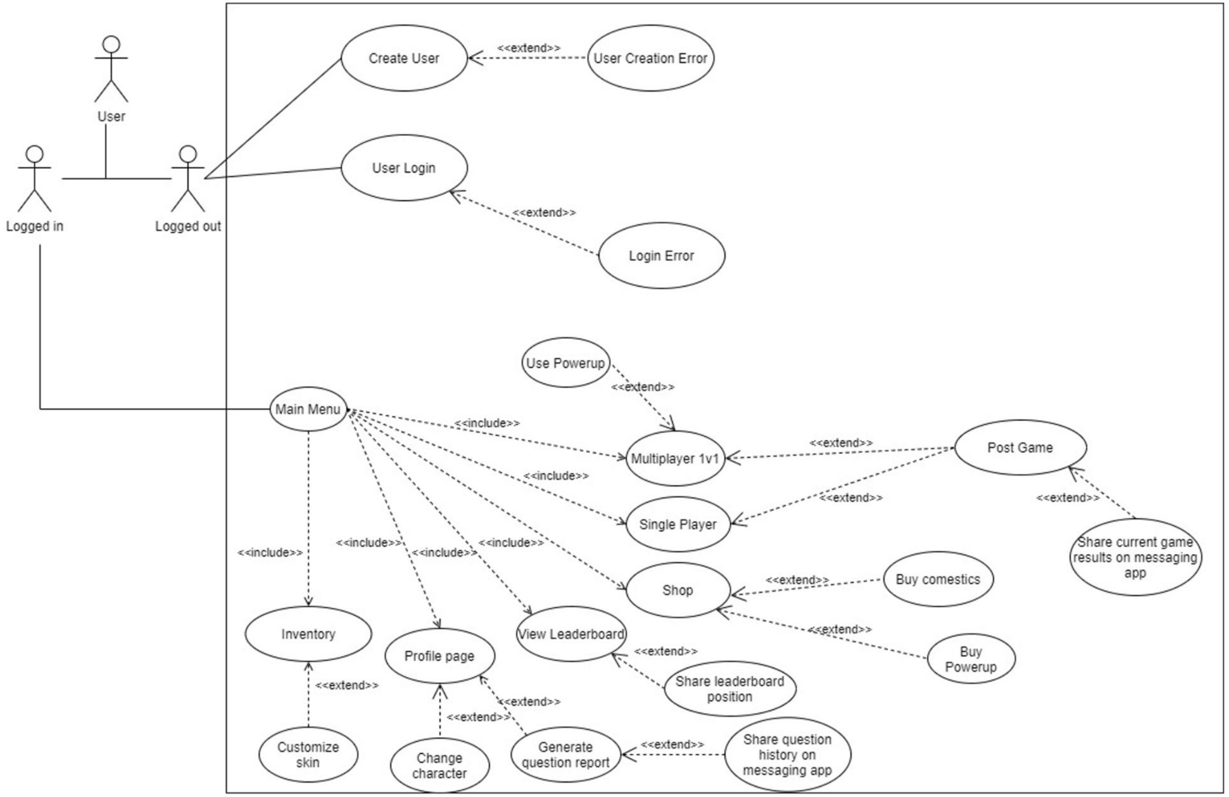
02

Analysis Models

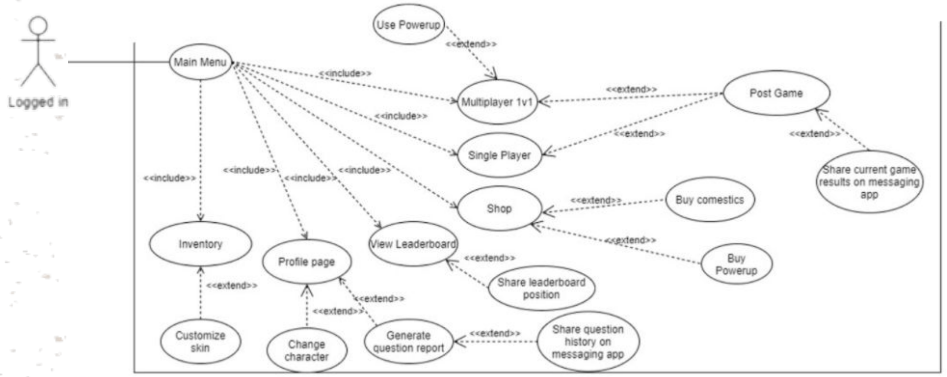
Use case Diagram

Dialog Map

Use Case Diagram



Use Case Diagram

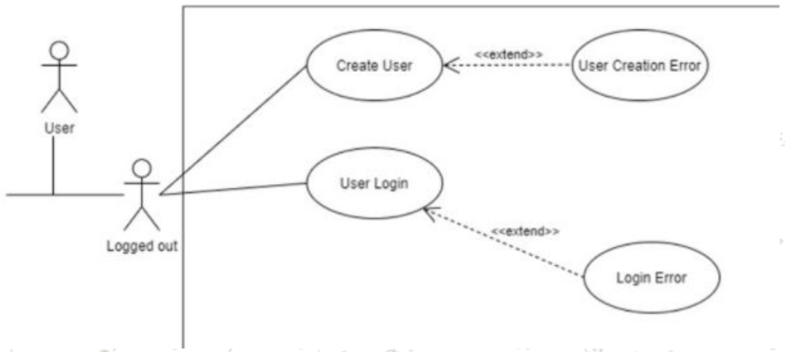


Logged in User

- ➔ Logged user:
  - ◆ Menu Page

- ➔ Menu Page
  - ◆ Profile Page
    - ● Change character
    - ● Generate report
  - ◆ Single Player / Multiplayer Mode
  - ◆ View leaderboard
    - ● Share leaderboard

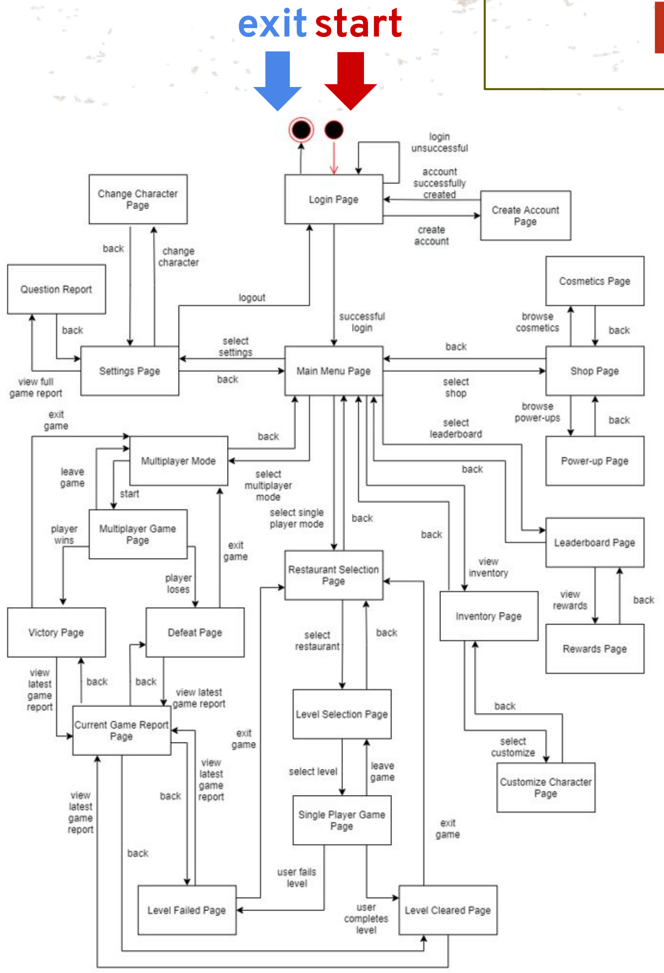
Use Case Diagram



Logged out User

- Logged out user:
  - Login
    - Wrong credential
      - Prompt error message
- New user:
  - Register an account
    - If the user entered his/her detail wrongly
      - Prompt error message

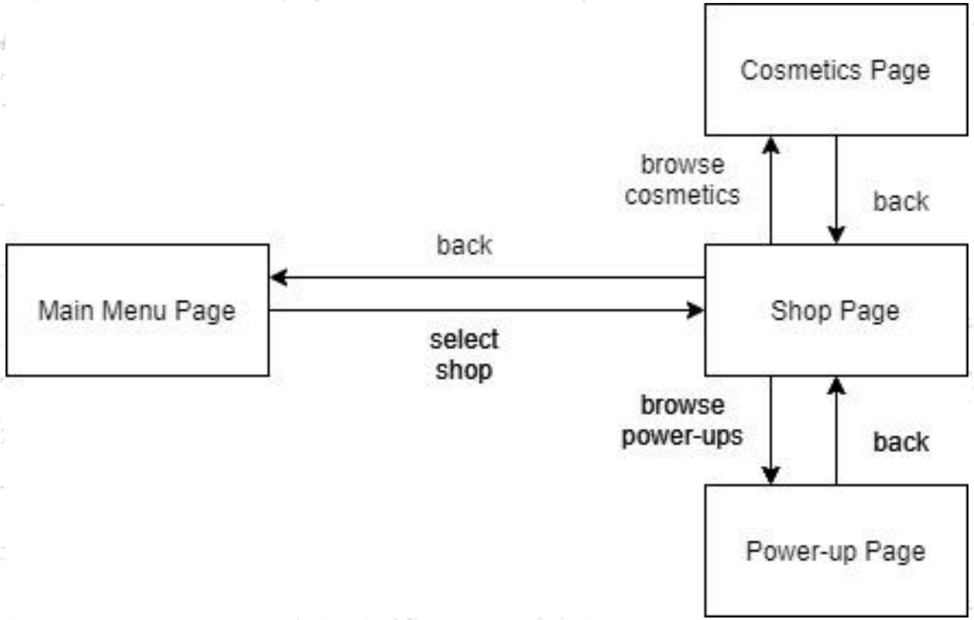
Dialog Map



- Show the flow of the game
- Transition between screen

1. User will first start from Login Page
2. Successful login will direct the user to Menu Page, where the user can access other pages. For example, Single Player Page, Multiplayer Page, Shop and etc.

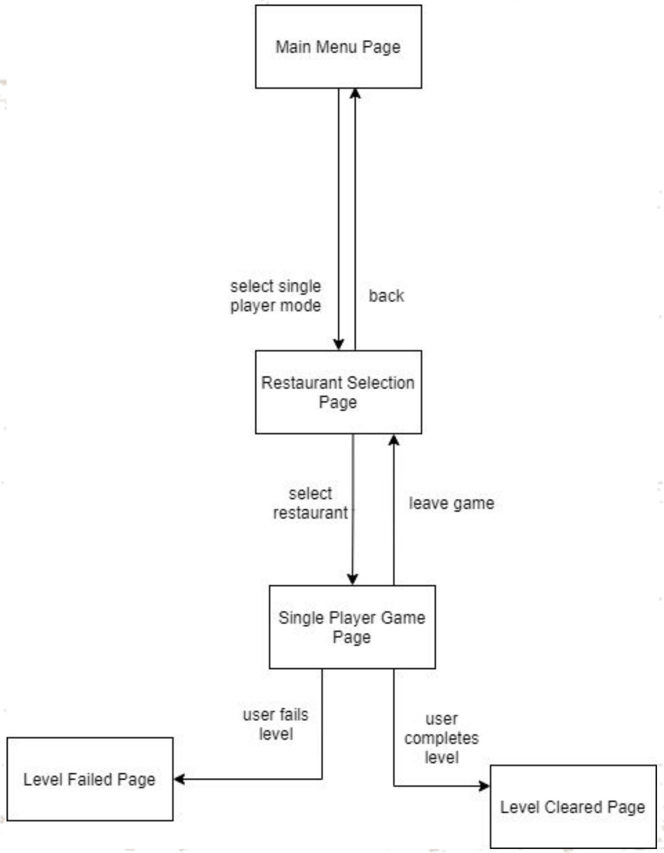
Dialog Map



1. Main Menu → Shop

- Browse and Purchase Cosmetics and Powerups
-To after purchase press back to exit Main Menu

Dialog Map

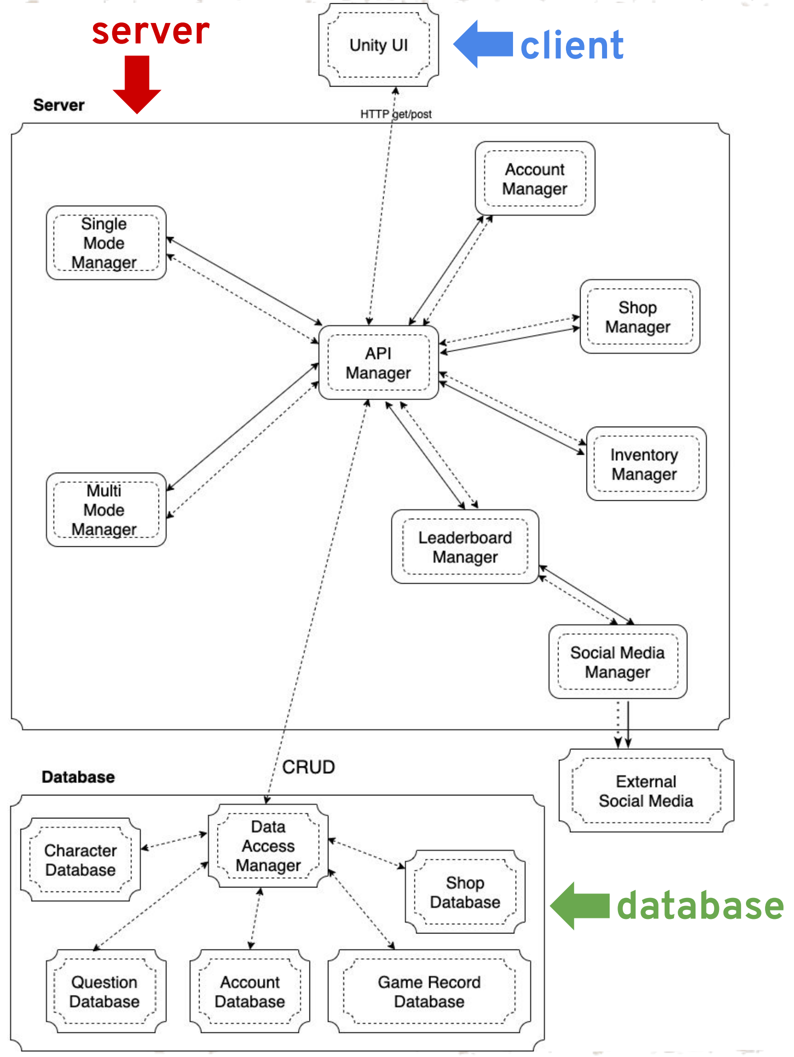


1. Main Menu → Restaurant Selection page
2. Restaurant Selection page → Single Player Game page
3. Single Player Game page → Level Cleared Page
   OR
   Single Player Game page → Level Failed Page

03

Architecture Design

By Guo Wanyao



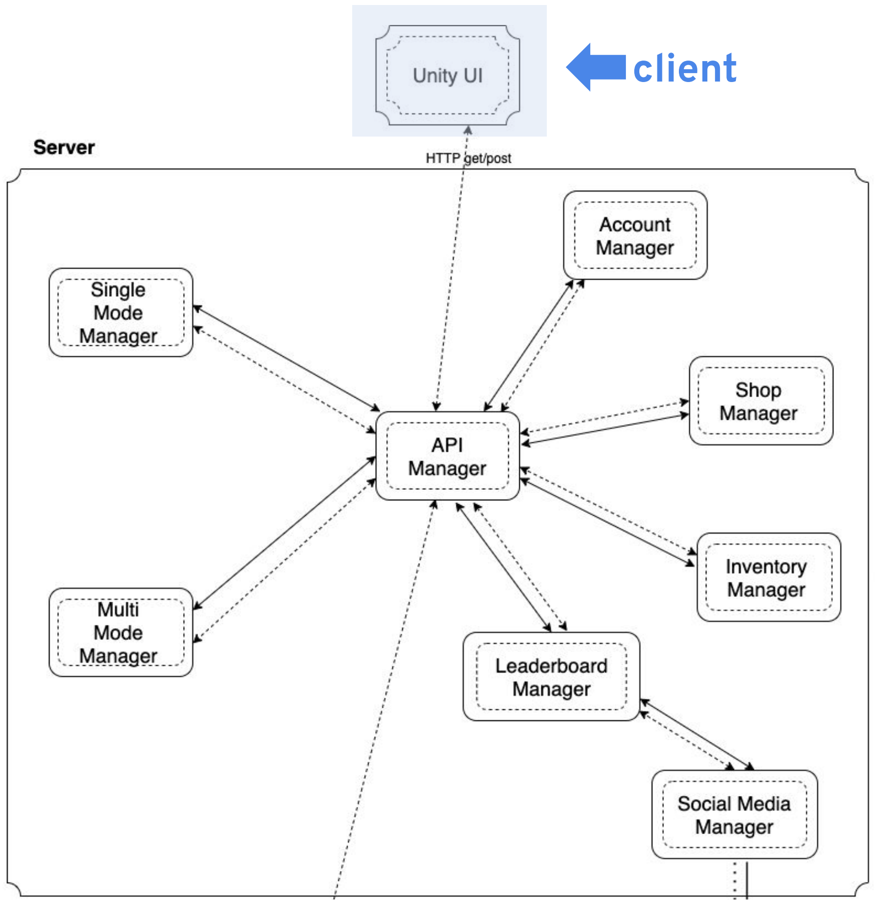
Client Server Architecture

➔ Micro-services? Not really...

➔ Model-View-Controller? Nah...

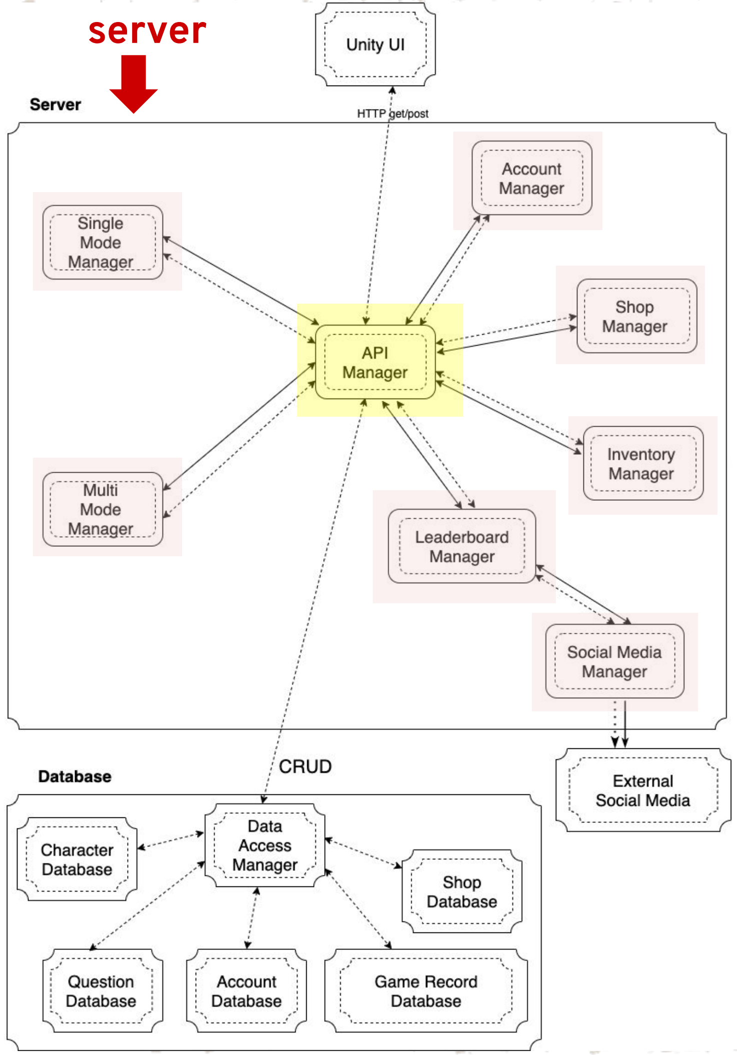
➔ Layered architecture? Nope...

➔ **Client server? YES!**



Client - Unity UI

- ➔ Provide an interface for users
- ➔ Request services of the server
- ➔ Display results that the server returns

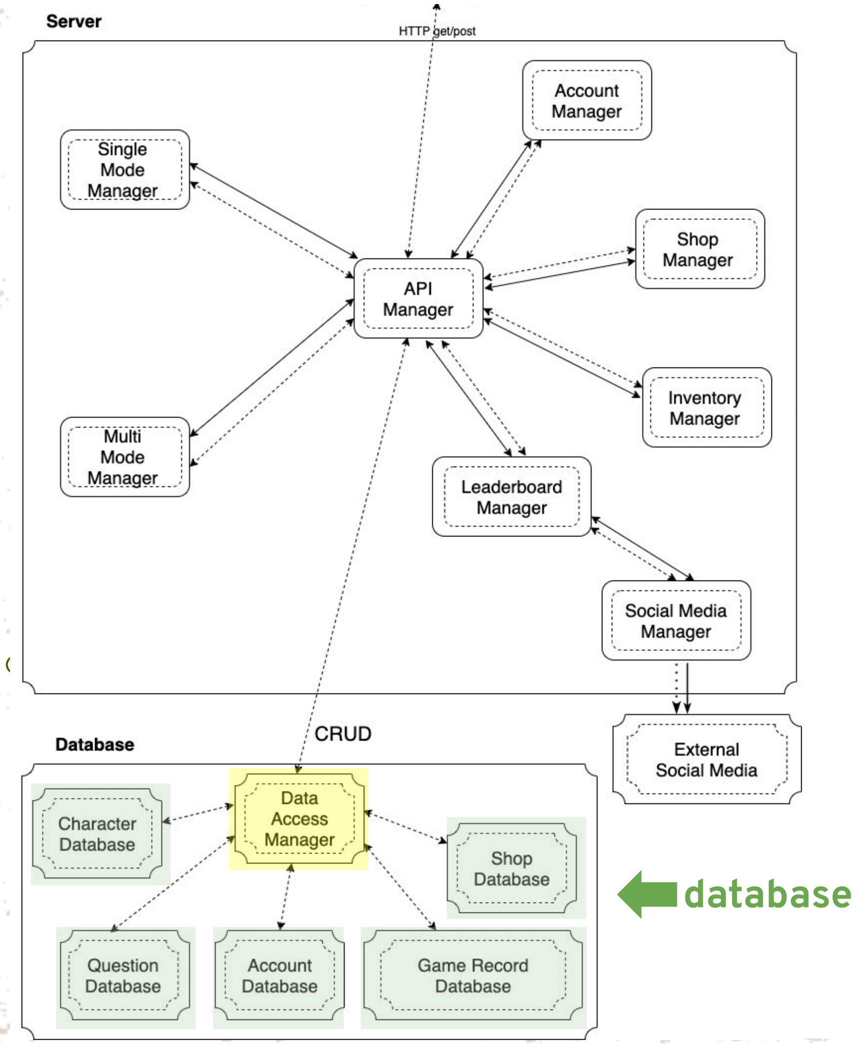


Server - Game Logic

➔ Central **API manager** directly interacts with Unity

➔ Each **logic manager** manages different sub-functionalities

➔ E.g. **Single Mode Manager:** control logic of single player game after receiving control from the **API Manager**



Backend Database

- The centralized manager: **Data Access Manager**
- **Data Access Manager** processes CRUD and distributes to specified database
- Each **specified database** manages data for one sub-functionality and completes the CRUD

Strengths of Client-Server Architecture

Scalability

1

- High scalability
- Support horizontal & vertical scaling

Parallel Execution

2

- Different processes can execute simultaneously and independently

Decoupling

3

- Separation of components & loose coupling
- Single Responsibility Principle

Flexibility

4

- Easily add new components into respective subsystem

Evaluation of Client Server Architecture

Pros

VS

Cons

Client-server architecture has weaknesses like:

- **Traffic congestion**: multiple clients make requests from the server, cause the connection to fail or slow down
- **Lack of robustness**: Client Server Architecture is centralized, in the event that the primary server fails, the entire network will be interrupted

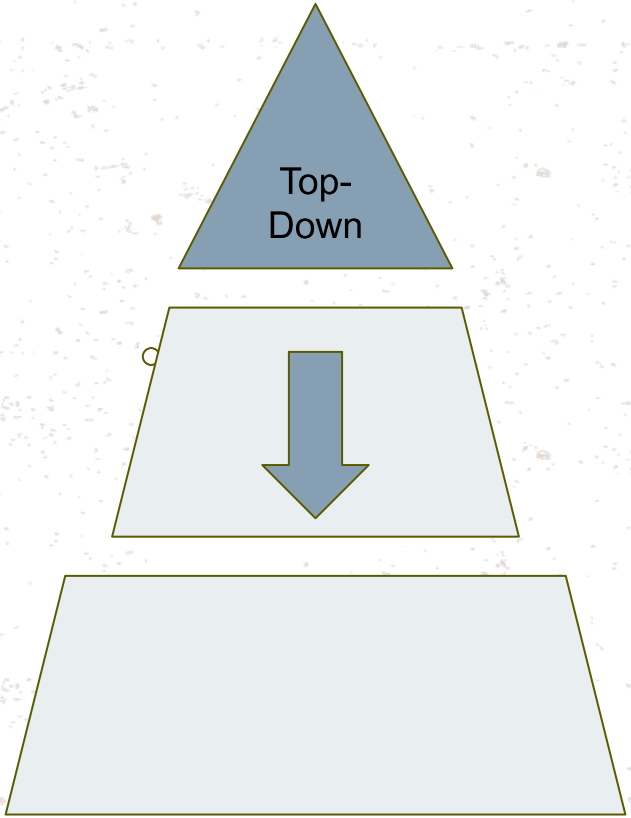
However, **the pros outweigh the cons** because:

- The direct connection between a client and a single server fits our design of the single-player game well
- For multi-player game, client server architecture ensure high integrity of the system

04

Subsystem Design

Decompositional Design



Decompositional Design

Decompose

Break the game into smaller functionalities

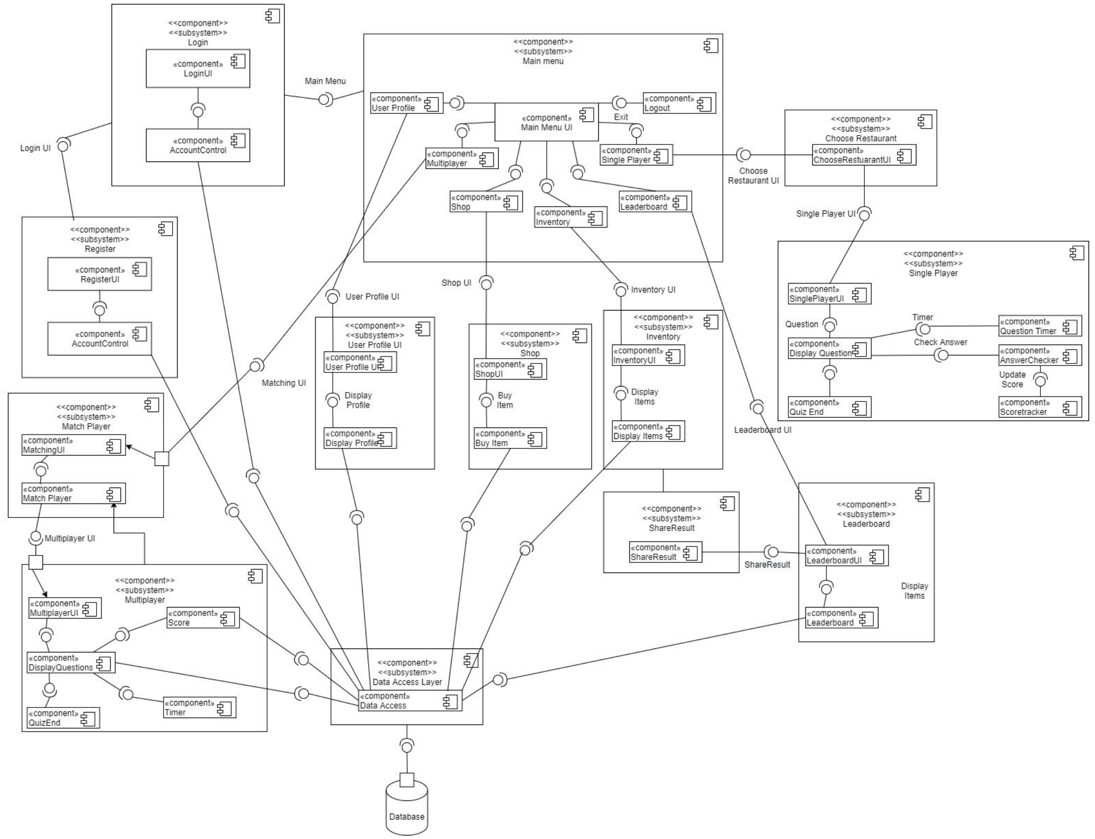
Solve

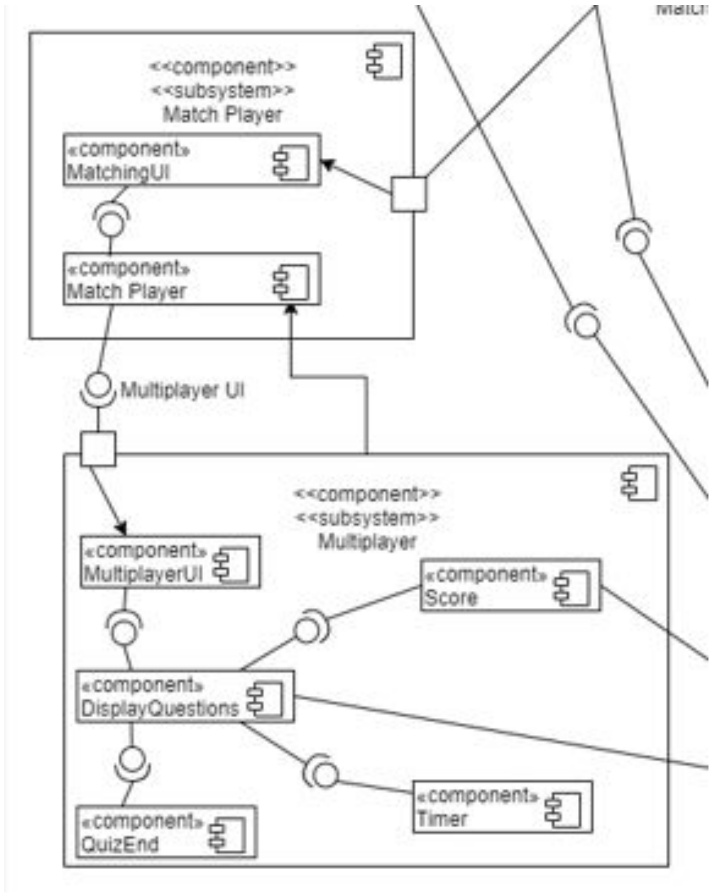
Implement the various functions in their respective components

Assemble

Integrate all the components to form the game

Component Diagram

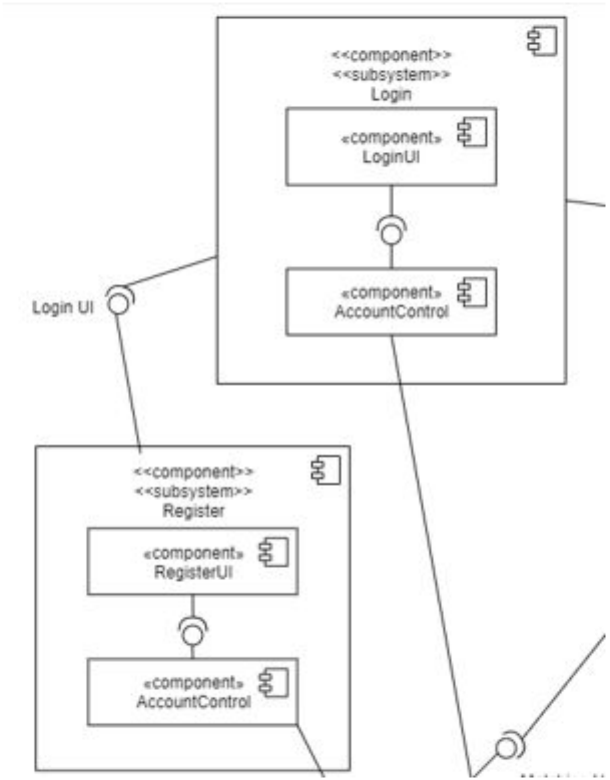




Multiplayer Game Logic Component

Game Logic Components

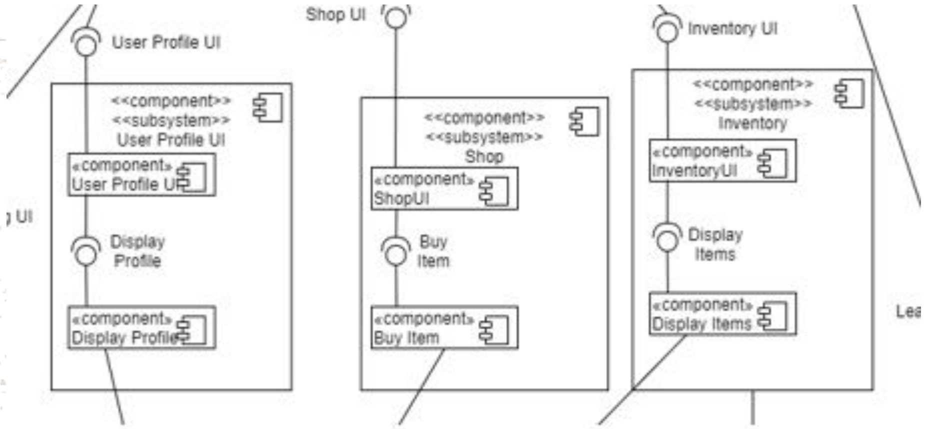
- ➔ Two components for game logic
  - ◆ Single Player Mode
  - ◆ Multiplayer Mode
- ➔ Inherits from a game logic abstract class which enables loose coupling
- ➔ Game modes can be added by inheriting the game logic interface thus increasing scalability



Authentication Component

Authentication Components

- ➔ Two components for authentication
  - ◆ Login
  - ◆ Register
- ➔ Implemented using Firebase
- ➔ Data is encrypted which increases security
- ➔ Additional features such as password requirement and email validation to ensure security



<u>Peripheral Component</u>

Peripheral Components

➔ Components such as:
- ◆ Shop
- ◆ Inventory
- ◆ User Profile
- ◆ Leaderboard
- ◆ Sharing Results

➔ Each component fulfills a specific function which achieves high cohesion

➔ Improved maintainability

Why did we use Design Patterns?

Flexible

Design Patterns are meant to be used in different scenarios

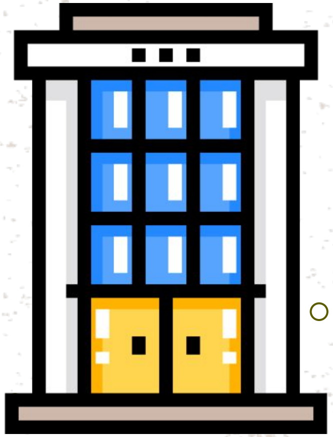
Effective

Design Patterns are effective solutions to a given problem with other's experience

Maintainable

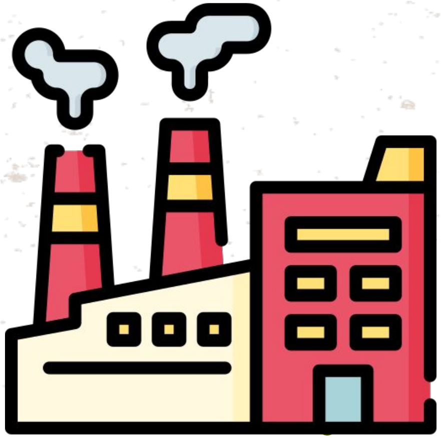
Programmers are likely to know the design patterns and understand the design

Design Patterns

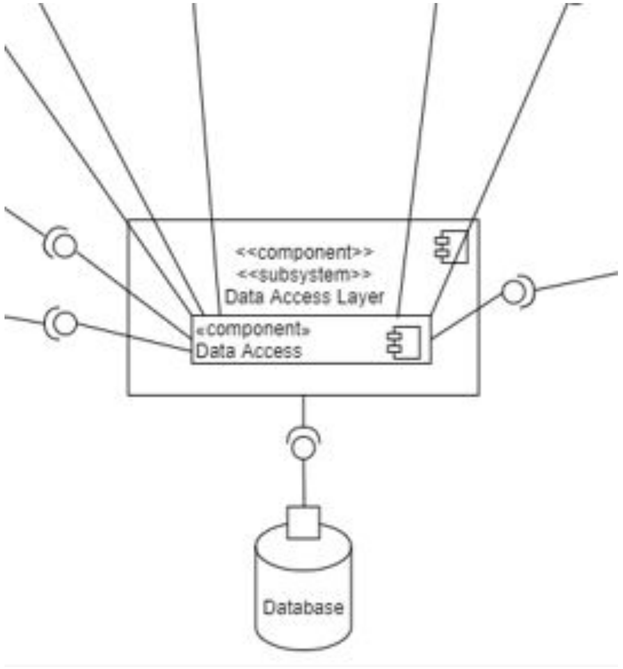


Facade

Singleton



Abstract Factory



Interface for Database

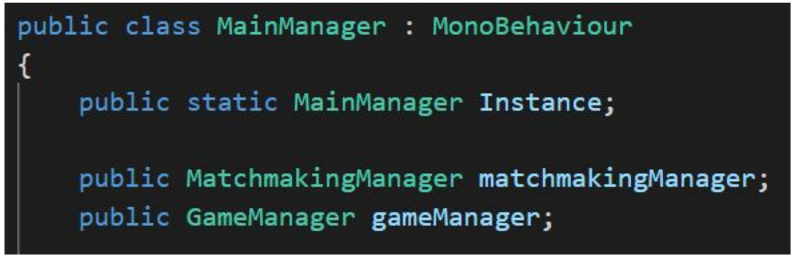
Facade Design Pattern

- ➔ Provide a simple interface to a complex subsystem
- ➔ Improves Readability and Usability by masking interactions with complex components
- ➔ Allows for looser coupling

Example of Data Access Layer

- ➔ Other components access the database through the Data Access Layer
- ➔ Encourages Loose Coupling as changes to the database will not affect other components

Singleton Design Pattern



```
public class MainManager : MonoBehaviour
{
    public static MainManager Instance;

    public MatchmakingManager matchmakingManager;
    public GameManager gameManager;
```

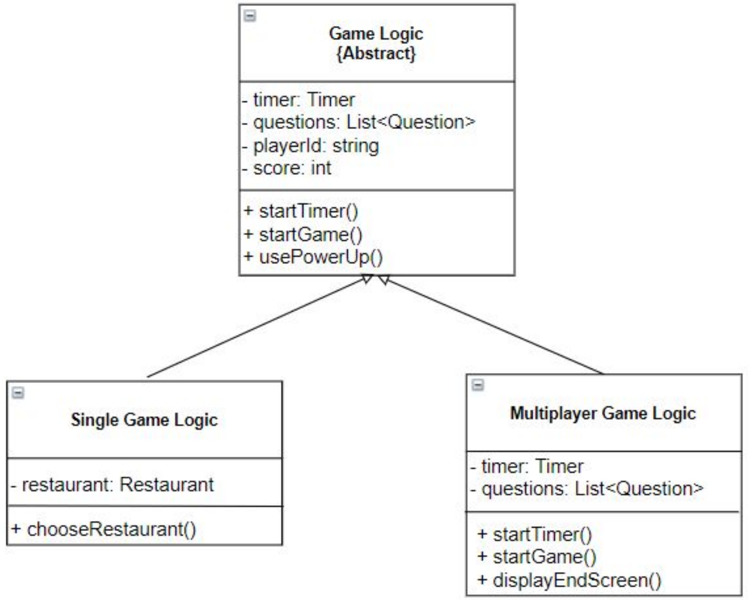
<u>Example of Main Manager</u>

➔ Creational Pattern to ensure that only one instance of a class is instantiated

<u>Example of Main Manager</u>

➔ A static instance of the Main Manager is created

➔ Prevents errors as it in turn ensures that there is only one Matchmaking and Game Manager active at any point in time



Inheritance for Game Logic

Abstract Factory Design Pattern

➔ Declare abstract classes for similar classes and get specific variants through inheriting

Abstract Factory Design Pattern

➔ Examples: Game Logic, Restaurant, Power-Ups, Characters

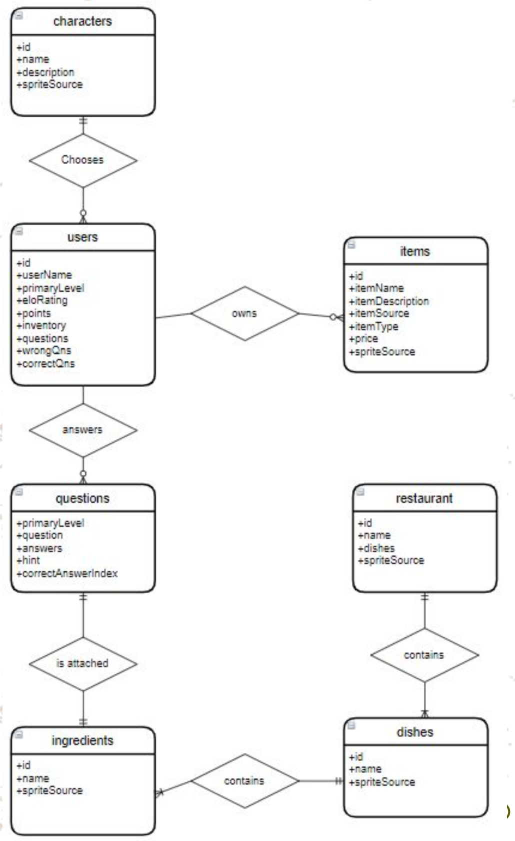
➔ Encourages Loose Coupling as classes can access concrete components using the parent class

Backend Design

Database

Server

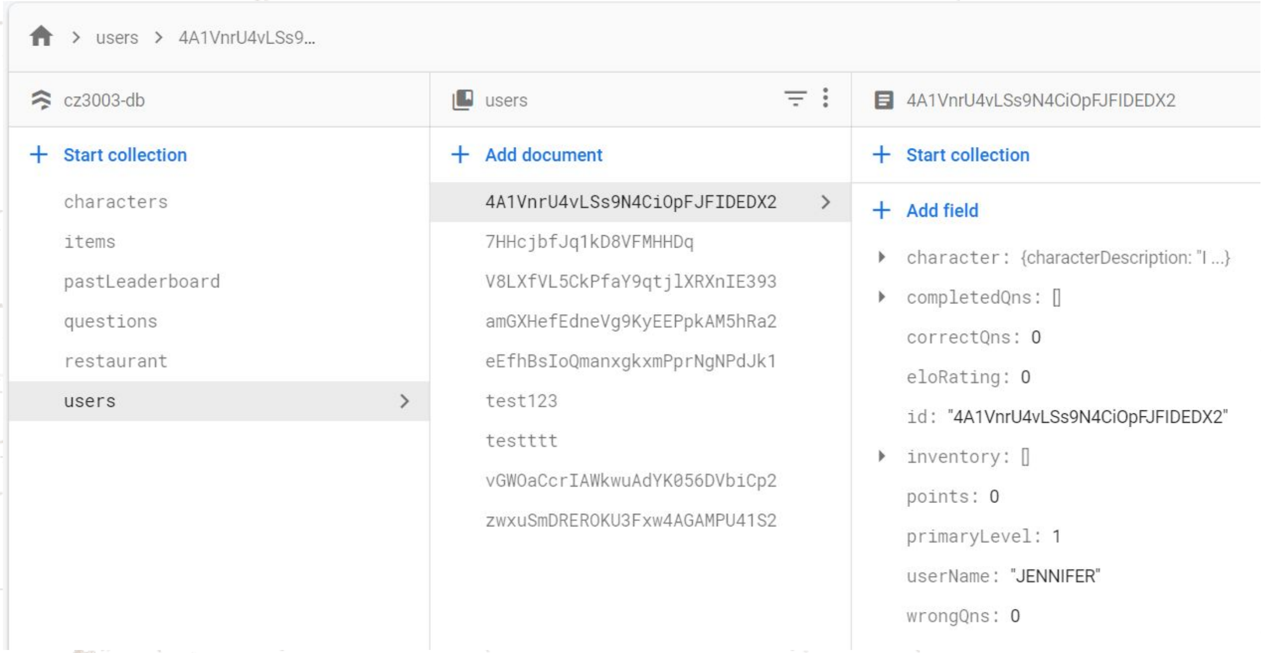
Entity Relationship Diagram



Entities include

➔ Characters
➔ Users
➔ Questions
➔ Ingredients
➔ Dishes
➔ Restaurant
➔ Items

Firebase Database


users > 4A1VnrU4vLSs9...

cz3003-db

+ Start collection

characters
items
pastLeaderboard
questions
restaurant
users

users

+ Add document

4A1VnrU4vLSs9N4CiOpFJFIDEDX2
7HHcjbfJq1kD8VFMHHDq
V8LXfVL5CkPfaY9qtjlXRXnIE393
amGXHefEdneVg9KyEEPpkAM5hRa2
eEfhBsIoQmanxgkxmPprNgNPdJk1
test123
testttt
vGWOaCcrIAWkwuAdYK056DVbiCp2
zwxuSmDREROKU3Fxw4AGAMPU41S2

4A1VnrU4vLSs9N4CiOpFJFIDEDX2

+ Start collection

+ Add field

```
character: {characterDescription: "I ...}
completedQns: []
correctQns: 0
eloRating: 0
id: "4A1VnrU4vLSs9N4CiOpFJFIDEDX2"
inventory: []
points: 0
primaryLevel: 1
userName: "JENNIFER"
wrongQns: 0
```

Backend Framework

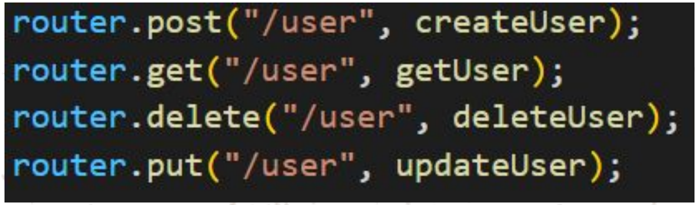
- ➔ Representational State Transfer
- ➔ Application Programming Interface
- ➔ Client-Server Interactions

REST API

- ➔ Based on a request/response design
- ➔ Requests are made up of
  - ◆ Endpoint
  - ◆ Method
  - ◆ Headers
  - ◆ Data

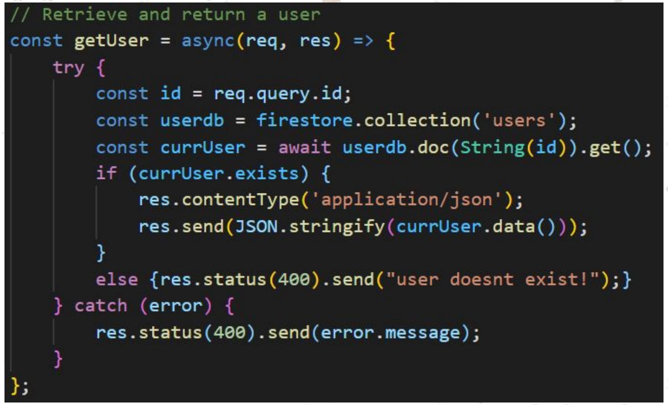
Server Controllers

User Controller Methods



```
router.post("/user", createUser);
router.get("/user", getUser);
router.delete("/user", deleteUser);
router.put("/user", updateUser);
```

Question Controller Methods

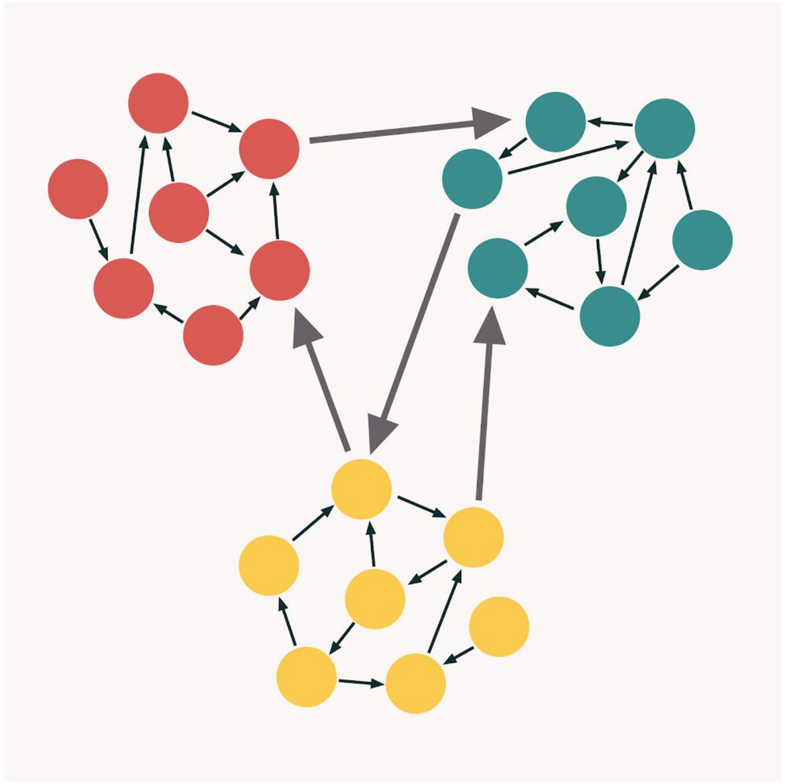


```
// Retrieve and return a user
const getUser = async(req, res) => {
    try {
        const id = req.query.id;
        const userdb = firestore.collection('users');
        const currUser = await userdb.doc(String(id)).get();
        if (currUser.exists) {
            res.contentType('application/json');
            res.send(JSON.stringify(currUser.data()));
        }
        else {res.status(400).send("user doesnt exist!");}
    } catch (error) {
        res.status(400).send(error.message);
    }
};
```

- ➔ Controllers for each collection of data are separated from one another
- ➔ Each controller consists of the various REST methods (GET, PUT, POST, DELETE) as required by client calls from the game
- ➔ Takes in data/headers in requests and returns the appropriate response back to the client and updates the database (if applicable)

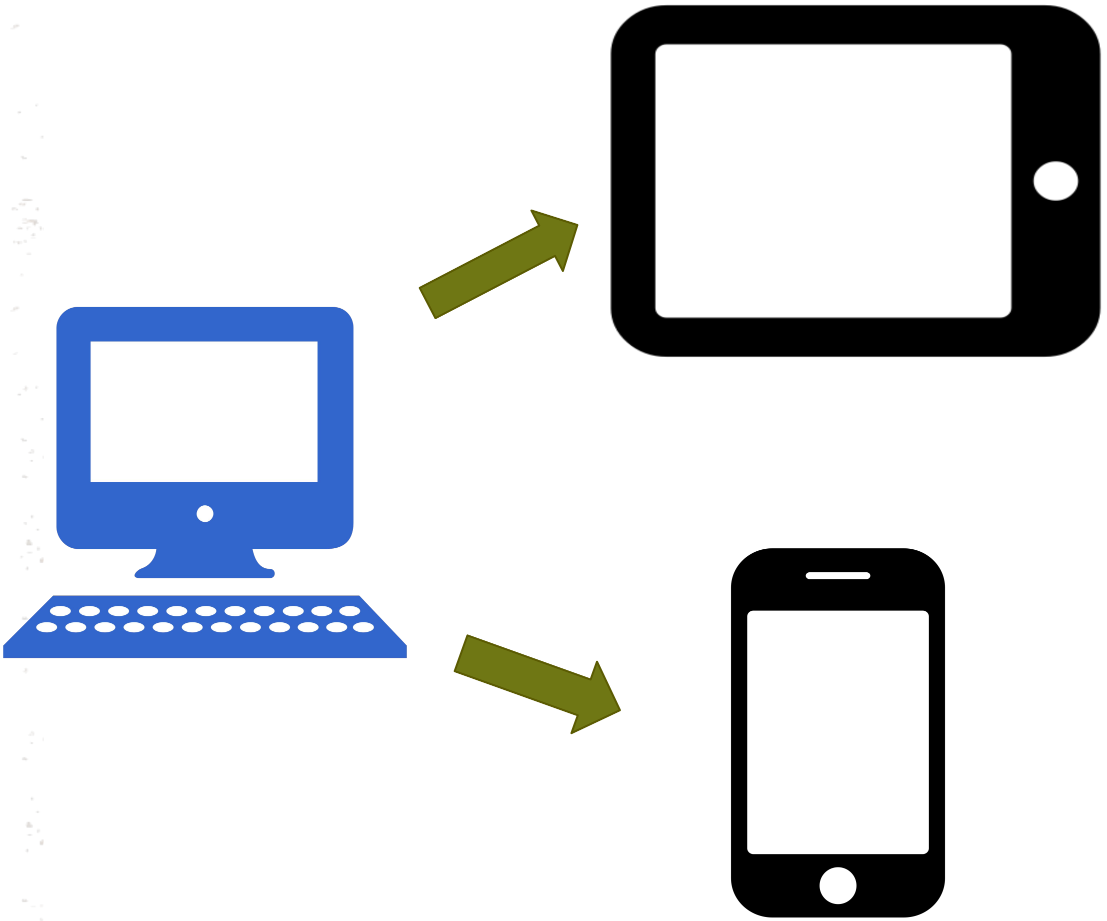
05

Elements of Good Design



Single Responsibility Principle

- ➔ One Manager script per functionality
- ➔ Related attributes / methods are grouped together
- ➔ High cohesion



Portability

- ➔ Originally designed for PC
- ➔ Can be played on other devices and operating systems
- ➔ Easy to port over to other environments

Loose coupling and high cohesion (jordon)

- Use of dao.

Single Responsibility Principle (jordon)

- High Cohesion

Designed for portability (jordon)

- Web game so can run on anything

Open-Closed Principle

- Game modes (use of abstractions)
- DAOs (Implement a interface DAO → add skins then inherit new DAO)
- Main Manager Interface that implements gameManager and matchmakingManager

Reusability

- DAOs can be used in other projects

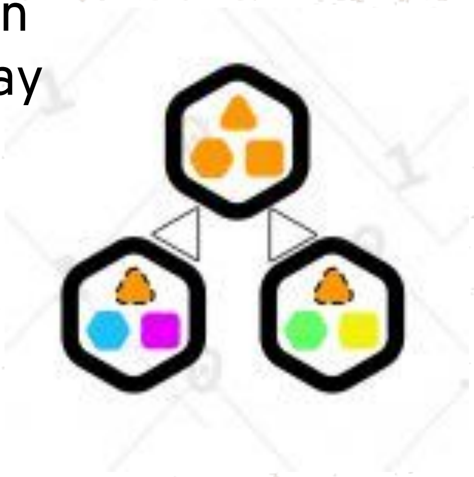
Open-Closed Principle

Opened for extension but closed for modification

➔ Use of an abstraction class for the Single Play and Multiplayer game modes

➔ Implemented a DAO interface from which other DAO classes can inherit from

➔ Main Manager Interface that implements gameManager and matchmakingManager



Open-Closed Principle

Opened for extension but closed for modification

Abstract Quiz Class

Single Player and Multiplayer Qui
modes can be **inherited** from this
Abstract class

Data Access Object Interface

Reusable interface to create other data access classes for different collections in our database

Reusability

The same DAO classes can be called from different scenes in unity

Concisely written class for other developers to utilise

Written class methods are common amongst other web applications

Ease to extending to new collections or database created in the future

DAO classes

06

Testing

Unit Testing

AltUnityTester

unity

Unity Testing Framework

unittest

MOCHA

chai

Mocha & Chai

Se

Selenium

Unit Testing
Unity Testing Framework

Unity Testing Framework

- Offers the flexibility to test in both the Edit & Play Mode
- Comprehensive tutorial & documentations

A total of **33** test cases were conducted

Testing Strategies
Equivalence Partitioning

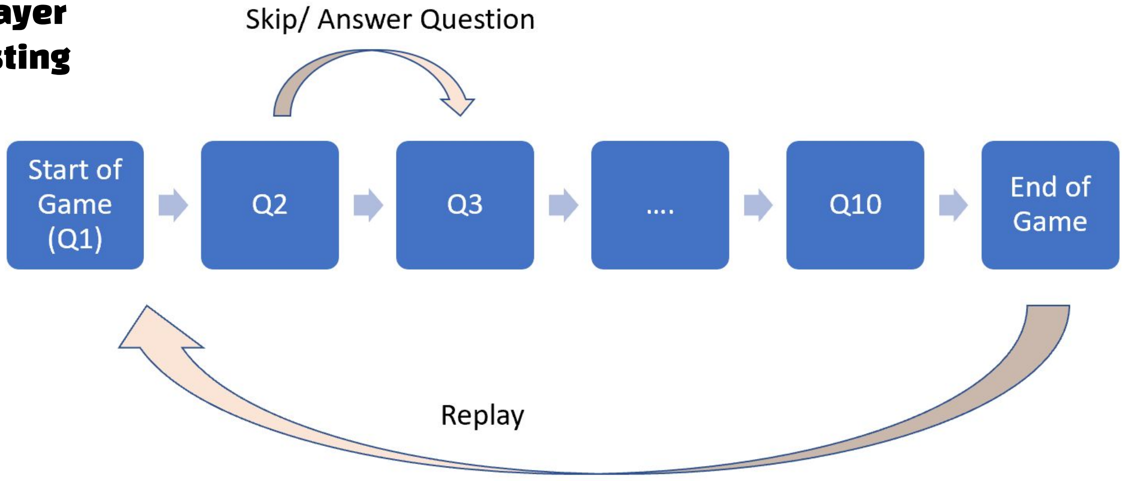
Login Test Cases

Invalid 2 test cases	Invalid 3 test cases	Valid 1 test case
Missing input fields	Wrong Inputs (not registered email/ wrong password etc)	Valid & registered email address & password

Input data is partitioned into partitions of valid & invalid values.
All partitions exhibit the same behaviour

Testing Strategies
Model Based Testing

Single Player Mode Testing



Describes how a system behaves in response to an action & see if the system responds as per expectation

Unit Testing
Mocha & Chai Testing

Mocha & Chai

- Mocha: testing framework that provides functions that are executed in a specific order
- Chai: Assertion Library

Used for database testing

A total of **35** test cases were conducted

Testing Strategies
Equivalence Partitioning

Question API - GET test cases

Invalid 3 test cases	Valid 1 test case
Invalid parameters (pri 0, pri 7) , Missing Parameters	All parameters valid & present.
Error message	Gets all question for the specified primary level

Input data is partitioned into partitions of valid & invalid values.
All partitions exhibit the same behaviour

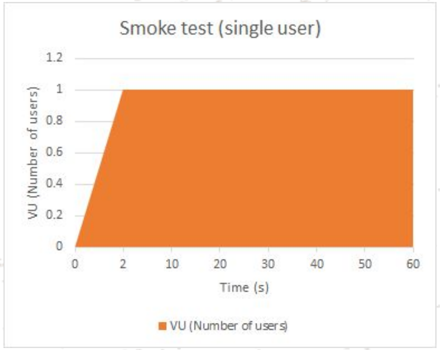
Performance Testing
Testing Process

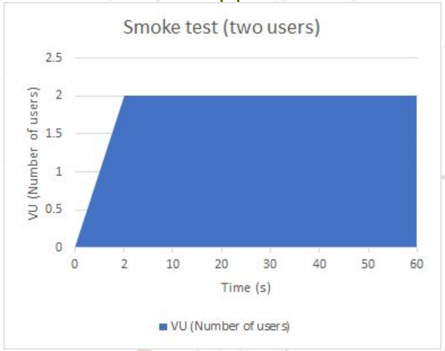
Test Name	Description
Login()	Used to measure time taken for complete login process for a user
Rendering_Scene()	Used to measure time taken for scene to render
MainMenu_to_SingleMode()	Used to measure time taken for complete transition from main menu scene to single player scene
MainMenu_to_MP_Finding()	Used to measure time taken for complete transition from main menu scene to matching player scene
Click_Answer_Button()	Used to measure time taken to receive content update after clicking answer button
Measure_Empty()	Custom measurement used to capture total allocated and reserved memory

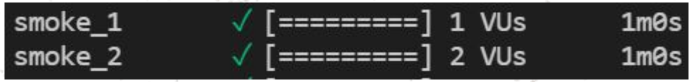
Performance Testing
Testing Results

Parameter tested	Average Performance Result	Implication
Scene load speed	0.11 - 0.81 ms	→ Responsive design → Immersive gameplay → Good user experience uninterrupted by long load screens
Memory usage	129 MB	→ Game does not have prohibitive hardware requirements → Accessible to students → Game does not take up resources from other processes

Load Testing
Smoke Test



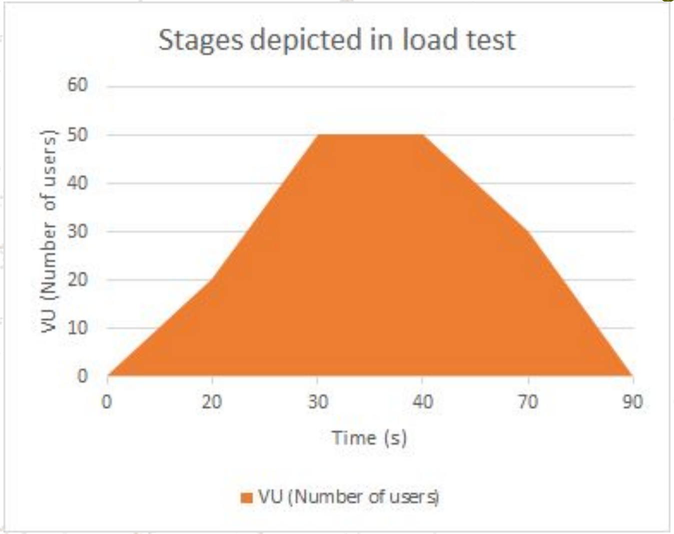


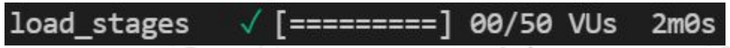


```
smoke_1         ✓ [=========] 1 VUs      1m0s
smoke_2         ✓ [=========] 2 VUs      1m0s
```

- We test both single player (1 user) and multiplayer (2 users) cases as a best-case load scenario
- Performance remains consistent

Load Testing
Load Test

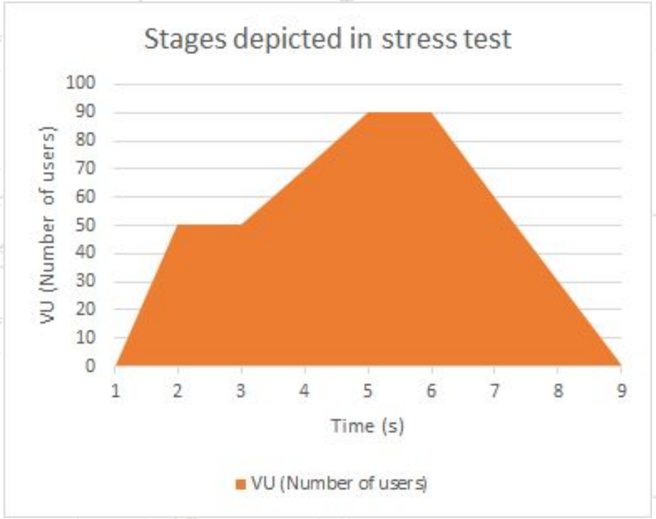




```
load_stages   ✓ [==========] 00/50 VUs  2m0s
```

- Test successfully executed ramping from 0 to 50 users then back down to 0
- Performance remains consistent

Load Testing
Stress Test



```
stress_stages ✓ [==========] 00/90 VUs  4m10s
```

- Test successfully executed ramping from 0 to 90 users then back down to 0 - slight pause at 50 users
- Performance starts slowing down but comfortably within the realm of responsive gameplay

Member	Work Completed
Grace	Login UI / Register UI / Inventory UI / Character UI / Game History UI / Loading Screen
Wanyao	Single player game UI / Multiplayer game UI / Single player game logic / Multiplayer game logic / Game logic testing / Performance testing
Joy	Backend scripts / Creation and population of database / User profile UI / User profile logic / User history logic / Mocha & Chai / Load Testing using k6
Ryan	Backend scripts/ Creation and population of database / Data Access Managers / Integration of Unity Game with database / Mocha & Chai / Load Testing using k6
Chio	Lab Reports / Diagrams / Video Editing
Hoong Jing	Main menu UI / Multiplayer Loading & Transition UI

Member	Work Completed
David	Matchmaking Logic / Loading & Transition Logic / Leaderboard & Register Testing / Multiplayer Game Logic / Synchronization of Realtime Database with Live Games
John	Login logic and authentication / Registration Logic and verification / Leaderboard Logic / Login and Registration Testing / Restaurant Selection Logic / PlayerPrefs Info
Zaphyr	Leaderboard UI / End Season UI / Last Season UI / View Reward UI
Wei Rong	Single player game UI/Choose Restaurant UI/Shop UI/Multiplayer UI
Jordon	Inventory UI, Logic & Testing / Shop UI, Logic & Testing / Character Selection UI & Logic

Thank you!