To create a project structure for your microservice app, here is an example project structure for your microservice-based application:

**Suggested Microservice Project Structure**

/car-game-new2 # Root project directory

├── backend # Backend microservice (API and logic)

│ ├── app.py # Main backend logic

│ ├── game\_db.py # Database-related code

│ ├── Dockerfile # Docker configuration for backend service

│ ├── requirements.txt # Python dependencies for backend

│ └── init.sql # Initial SQL schema for database setup

├── db # Database configuration and setup

│ ├── game\_db.py # Database logic for game-related data

│ └── init.sql # Initial database schema (if applicable)

├── frontend # Frontend microservice (UI and display)

│ ├── index.html # Main HTML page for game

│ ├── game.js # JavaScript logic for the frontend

│ ├── style.css # Styling for the frontend

│ ├── Dockerfile # Docker configuration for frontend service

│ └── .dockerignore # Files/folders to exclude from Docker build

├── .dockerignore # Files/folders to exclude globally from Docker

├── docker-compose.yaml # Docker Compose file to manage multiple services

├── .gitignore # Git ignore file for version control

└── README.md # Project description and setup instructions

**Explanation:**

1. **Backend (app.py, game\_db.py)**:
   * **app.py**: This is the main file containing your API logic, routes, and backend business logic for the game.
   * **game\_db.py**: Contains logic to interact with the database (e.g., querying or updating game data).
   * **Dockerfile**: Configuration for building a Docker image for the backend service.
2. **Frontend (index.html, game.js, style.css)**:
   * **index.html**: The main page structure for the frontend, including the canvas and any other UI components.
   * **game.js**: JavaScript code to control game logic on the frontend, such as handling events, user input, and drawing to the canvas.
   * **style.css**: Styles to format the UI, including game and score display styling.
3. **Database (game\_db.py, init.sql)**:
   * **game\_db.py**: Logic for managing and interacting with the database, such as saving scores, player stats, etc.
   * **init.sql**: The schema file to initialize the database tables.
4. **docker-compose.yaml**: This file is used to configure and run your multi-container Docker setup, managing both the frontend and backend services.

### 2. ****Installation and Dependencies****

#### **Backend Dependencies (app.py)**:

The backend service is built using Flask. It has dependencies on the following libraries:

* **Flask**: A micro web framework used to build the API.
* **mysql-connector-python**: A connector to interact with MySQL database.
* **redis**: Redis client to interact with Redis.

Install dependencies using pip:

pip install -r requirements.txt

**requirements.txt**:

txt

flask

mysql-connector-python

redis

#### **Frontend Dependencies (game.js, style.css, index.html)**:

The frontend is a simple HTML/JS game with no external dependencies, except for the browser’s native capabilities for rendering and handling events.

### 3. ****Database and Caching Setup****

#### **MySQL Database**:

MySQL will be used to store game scores in a table called scores. The **init.sql** file contains the SQL schema to initialize the database.

**init.sql** (located in the db/ folder):

sql

CREATE TABLE IF NOT EXISTS scores (

id INT AUTO\_INCREMENT PRIMARY KEY,

score INT,

date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS data (

id INT AUTO\_INCREMENT PRIMARY KEY,

message VARCHAR(255)

);

#### **Redis Setup**:

Redis is used to store the most recent player score in-memory. It is faster for frequent access compared to MySQL.

### 4. ****Interservice Communication****

The project involves four main services: **Frontend**, **Backend**, **MySQL**, and **Redis**. These services communicate via HTTP requests and environment variables.

#### **Frontend to Backend Communication (game.js)**:

* The frontend sends the player's final score to the backend via a POST request when the game ends.
* The backend saves this score in the **MySQL** database and **Redis**.

**Example of game.js communicating with backend**:

javascript

function sendScoreToBackend(finalScore) {

fetch('http://localhost:5000/save\_score', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({

score: finalScore

})

})

.then(response => response.json())

.then(data => {

console.log("Score successfully saved:", data);

})

.catch(error => {

console.error("Error saving score:", error);

});

}

#### **Backend to MySQL and Redis**:

* **MySQL**: The backend stores scores in MySQL using SQL commands (INSERT INTO scores).
* **Redis**: The backend stores the latest score in Redis using Redis commands (SET player\_score).
* **You can access redis cli by this command in terminal**
* docker-compose exec redis redis-cli

#### **Backend to MySQL (app.py)**:

python

# Save score to MySQL

def save\_score(score):

conn = get\_db\_connection()

cursor = conn.cursor()

cursor.execute("""

INSERT INTO scores (score)

VALUES (%s);

""", (score,))

conn.commit()

cursor.close()

conn.close()

#### **Backend to Redis (app.py)**:

python

# Save score to Redis

redis\_client.set('player\_score', score)

### 5. ****Deployment Process****

#### **Step 1: Build Docker Containers**

Ensure that you have Docker installed on your system. Then, follow these steps to build the Docker containers for the frontend, backend, MySQL, and Redis.

docker-compose build

#### **Step 2: Start Services Using Docker Compose**

Once the containers are built, start the services using Docker Compose:

docker-compose up

This command will bring up the following services:

* **frontend**: Accessible on port 80
* **backend**: Accessible on port 5000
* **MySQL**: Running on port 3306
* **Redis**: Running on port 6379

#### **Step 3: Accessing the Application**

* **Frontend**: Open your browser and go to http://localhost to play the game.
* **Backend**: The backend will be available on http://localhost:5000. You can test the API directly using tools like Postman or via the frontend.

### 6. ****Scaling and Enhancements****

This application can be easily scaled by adding additional microservices or improving existing functionality:

* **Adding New Features**: For example, you can add user authentication, store multiple player scores, or create a leaderboard.
* **Redis Caching**: Improve caching mechanisms, like storing high scores and reducing the number of requests to the database.
* **Frontend Enhancements**: Include more game features such as power-ups, levels, or improved UI.

### 7. ****Debugging and Logs****

To debug issues or inspect logs, use the following command to view the logs for any of the services:

docker-compose logs frontend

docker-compose logs backend

docker-compose logs db

docker-compose logs redis

This will display logs for the respective services.

### 8. ****Conclusion****

You now have a fully set up microservice-based Car Obstacle Game using Docker for containerization and deployment. The frontend communicates with the backend via HTTP, and the backend interacts with MySQL and Redis for storing and caching player scores. Docker Compose makes it easy to manage and run all services together.

1. ockerfile to handle containerization and deployment.
2. **.gitignore**: Defines which files or folders to exclude from version control, like local environment setups, node\_modules, or Python cache files.

This structure keeps your application modular and scalable, making it easy to maintain and deploy each part independently using Docker and Docker Compose. Let me know if you need further assistance or modifications!