**Create a architecture for ola types software**

**1. User Interfaces**

* **Mobile Apps**: Separate apps for riders and drivers, built for iOS and Android.
* **Web Interface**: Optional, but useful for bookings, admin, and support tasks.

**2. Backend Services**

* **User Service**: Manages user profiles, authentication, and authorization.
* **Ride Service**: Handles ride requests, matching, pricing, and trip tracking.
* **Payment Service**: Manages fare calculation, payment processing, and invoicing.
* **Notification Service**: Sends ride status updates and promotional messages.
* **Geo-Location Service**: Provides real-time location tracking for rides and displays the driver’s location on the map.
* **Driver Service**: Manages driver profiles, ratings, and availability.

**3. Database Layer**

* **User Database**: Stores user profiles, ride history, and payment information.
* **Driver Database**: Stores driver information, vehicle details, and ride statistics.
* **Ride Database**: Stores ride details, routes, pricing, and feedback.

**4. Third-Party Integrations**

* **Map Services**: Integration with a mapping service like Google Maps or Mapbox for route planning and ETA calculations.
* **Payment Gateways**: Integration with payment providers like Stripe, PayPal, or local payment methods.
* **SMS/Email Services**: Integration with services like Twilio or SendGrid for sending notifications and alerts.

**5. Microservices Architecture**

* Use a microservices architecture to ensure each service is independently deployable and scalable. This helps manage complex operations and reduces the impact of changes or failures in one service.

**6. Scalability and Performance**

* **Load Balancers**: Distribute incoming requests across multiple servers to ensure high availability.
* **Caching**: Use caching mechanisms like Redis to improve the performance of frequently accessed data.
* **Auto-scaling**: Implement auto-scaling policies to handle varying levels of traffic and optimize resource usage.

**7. Security and Compliance**

* **Data Encryption**: Encrypt sensitive data in transit and at rest.
* **Authentication**: Use secure authentication methods like OAuth 2.0.
* **Compliance**: Ensure compliance with local regulations, such as GDPR or PCI DSS for payment processing.

**8. DevOps and CI/CD**

* **Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines for automated testing and deployment.
* **Monitoring and Logging**: Use tools like Prometheus and ELK Stack (Elasticsearch, Logstash, and Kibana) for monitoring and logging.
* **Containerization**: Use Docker and Kubernetes for deploying and managing microservices.

**9. Data Analytics and Machine Learning**

* **Data Warehouse**: Collect and store data for analytics and reporting.
* **Machine Learning Models**: Implement models for demand forecasting, pricing optimization, and fraud detection.

**10. Customer Support and Feedback**

* **Support Tools**: Implement a ticketing system for handling customer support issues.
* **Feedback Mechanism**: Collect and analyze feedback from users and drivers to improve service quality.

**Diagram:**



**Mode View Cotroller Of Youtube**

Model:

Data Handling: YouTube's model manages data related to videos, user accounts, playlists, comments, likes, subscriptions, and recommendations.

Business Logic: It handles operations like video encoding, streaming, user engagement metrics (e.g., views, likes), and personalized content recommendations.

Database Interactions: The model interacts with databases to retrieve video metadata, user preferences, and comments, as well as to store upload data and user interactions.

View:

User Interface: The view is responsible for rendering the video player, video recommendations, search results, and user comments. It includes the HTML/CSS for layout and JavaScript for interactive elements like play/pause, volume control, and video suggestions.

Dynamic Content: YouTube’s view adapts based on user interactions, like updating the list of recommended videos as a user watches more content or shows live updates in the comments section during a live stream.

Controller:

User Interaction: The controller manages actions like searching for a video, subscribing to a channel, liking a video, or posting a comment. It updates the model with these interactions and ensures the view displays the appropriate response.

Routing: When a user clicks on a video, the controller fetches the video data from the model and renders the video player along with related videos and comments in the view.

Video Upload: During video upload, the controller processes the video file, updates the model with the video metadata, and provides feedback to the user via the view.

**Practise Question:**

Implement a custom dynamic array class that supports basic operations like insertion, deletion, resizing, and clearing.

Ans:

Code:

#include <iostream>

#include <stdexcept>

#include <algorithm> // for std::copy

template<typename T>

class DynamicArray {

private:

T\* data;

size\_t capacity;

size\_t size;

void resize(size\_t new\_capacity) {

T\* new\_data = new T[new\_capacity];

std::copy(data, data + size, new\_data);

delete[] data;

data = new\_data;

capacity = new\_capacity;

}

public:

DynamicArray()

: data(nullptr), capacity(0), size(0) {}

~DynamicArray() {

delete[] data;

}

void insert(const T& value) {

if (size == capacity) {

resize(capacity == 0 ? 1 : capacity \* 2);

}

data[size++] = value;

}

void remove(size\_t index) {

if (index >= size) {

throw std::out\_of\_range("Index out of range");

}

std::copy(data + index + 1, data + size, data + index);

size--;

}

void clear() {

delete[] data;

data = nullptr;

size = 0;

capacity = 0;

}

void setSize(size\_t new\_size) {

if (new\_size > capacity) {

resize(new\_size);

}

size = new\_size;

}

T& operator[](size\_t index) {

if (index >= size) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

const T& operator[](size\_t index) const {

if (index >= size) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

size\_t getSize() const {

return size;

}

size\_t getCapacity() const {

return capacity;

}

};

int main() {

DynamicArray<int> arr;

arr.insert(1);

arr.insert(2);

arr.insert(3);

std::cout << "Array size: " << arr.getSize() << std::endl;

for (size\_t i = 0; i < arr.getSize(); ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

arr.remove(1);

std::cout << "After removing element at index 1:" << std::endl;

for (size\_t i = 0; i < arr.getSize(); ++i) {

std::cout << arr[i] << " ";

}

std::cout << std::endl;

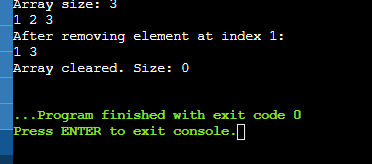
arr.clear();

std::cout << "Array cleared. Size: " << arr.getSize() << std::endl;

return 0;

}

Output:



Create a template-based stack class supporting push, pop, and peek operations. Implement it for different data types like int, float, and std::string.

Code:

#include <iostream>

#include <stdexcept>

#include <string>

template<typename T>

class Stack {

private:

T\* data;

size\_t capacity;

size\_t top;

void resize(size\_t new\_capacity) {

T\* new\_data = new T[new\_capacity];

for (size\_t i = 0; i < top; ++i) {

new\_data[i] = data[i];

}

delete[] data;

data = new\_data;

capacity = new\_capacity;

}

public:

Stack()

: data(new T[1]), capacity(1), top(0) {}

~Stack() {

delete[] data;

}

void push(const T& value) {

if (top == capacity) {

resize(capacity \* 2);

}

data[top++] = value;

}

void pop() {

if (top == 0) {

throw std::out\_of\_range("Stack underflow");

}

top--;

}

T& peek() {

if (top == 0) {

throw std::out\_of\_range("Stack is empty");

}

return data[top - 1];

}

bool isEmpty() const {

return top == 0;

}

size\_t size() const {

return top;

}

};

int main() {

Stack<int> intStack;

intStack.push(1);

intStack.push(2);

intStack.push(3);

std::cout << "Integer Stack top element: " << intStack.peek() << std::endl;

intStack.pop();

std::cout << "Integer Stack top element after pop: " << intStack.peek() << std::endl;

Stack<float> floatStack;

floatStack.push(1.1f);

floatStack.push(2.2f);

floatStack.push(3.3f);

std::cout << "Float Stack top element: " << floatStack.peek() << std::endl;

floatStack.pop();

std::cout << "Float Stack top element after pop: " << floatStack.peek() << std::endl;

Stack<std::string> stringStack;

stringStack.push("Hello");

stringStack.push("World");

stringStack.push("!");

std::cout << "String Stack top element: " << stringStack.peek() << std::endl;

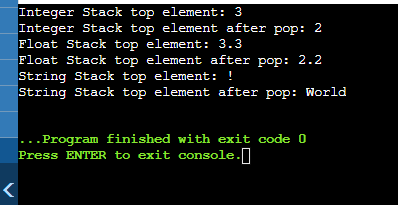
stringStack.pop();

std::cout << "String Stack top element after pop: " << stringStack.peek() << std::endl;

return 0;

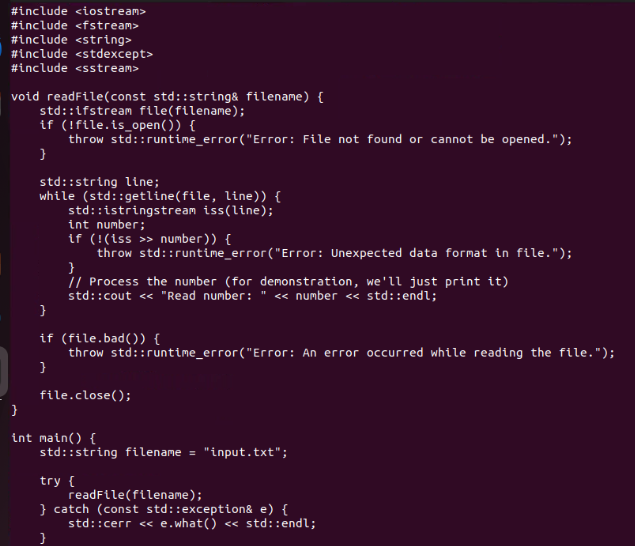
}

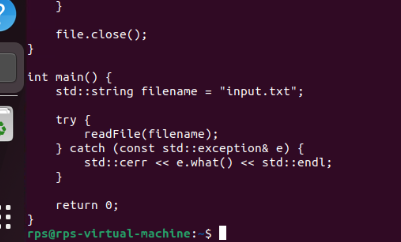
Output:



Write a program that reads from a file and handles various exceptions such as file not found, read errors, and unexpected data formats.

Code:





Output:

