Name: Suguna Naidu Karanam

Assignment: IP2 - Individual Project – due

Submission date: March 3rd 2024

Course: [CPSC 5200 02](https://seattleu.instructure.com/courses/1612931)

Client generated- Grpc tools

API Call (Stub Client)

Proto Messages

Image Processing Method-Server Code

Service/request response

Parameter (prtoto file)

Method execution

HTTP/2 TLS

Client request

Response (Encoding)

(Service Response)

protoc

(Compiler)

Grpc Intra (Server)

Client (Grpc)

protoc

(Compiler)

Request (Encoding)

(API Call + parameter)

Architectural Overview

protobuf

**require**

image data is not null and operation = resize or left – request sent to server using protobuf

**ensure**

image processed and return to client – server infers as per messages defined in proto

High-Level Overview Diagram

Represents the client-side application or user interface that interacts with the image processing service.

Sends requests to the API server to perform image processing operations.

API Gateway(HTTP2/TLS):

Acts as a single entry point for clients to access the image processing service.

Routes requests to the appropriate services based on the requested operation.

Image Processing Service(Grpc-Infra):

A service responsible for handling image processing operations.

Utilizes the Image Processing Library to execute operations such as rotate, flip, etc.

Communicates with the Persistence Layer to store and retrieve processed images, session data, and configuration settings.

Messages are in .proto form that provides abstract properties and Grpc services methods.

Client-Server Interaction(Grpc-Client):

Clients send requests to the API Gateway to perform image processing operations.

The API Gateway routes requests to the appropriate service based on the requested operation.

services process requests, perform image processing operations, and return the processed images to clients.

Stub client will be generated using Grpc tools and these client will handle user request.

Data Flow(Service Rquest/Response):

Image data and operation parameters flow from the client to the API Gateway, which then forwards the request to the Image Processing service.

OpenAPI(Swagger)

while OpenAPI (formerly known as Swagger) is primarily designed for RESTful APIs, it can also be used to document gRPC services. However, it's essential to note that OpenAPI is more tailored towards HTTP-based APIs and may not fully capture the nuances of gRPC services, such as bidirectional streaming or protocol buffers.

It can not defined as RESTAPIs swagger but as protobuf is responsible for communication so proto can be used as OpenAPI swagger, below is the OpenAPI documentation using proto file.

// imageprocessing.proto

syntax = "proto3";

package imageprocessing;

import "google/api/annotations.proto";

service ImageProcessing {

// Process image baed on user selection.

// request: ImageRequest

// response: ImageResponse

rpc ProcessImage(imagerequest) returns (imageresponse) {

option (google.api.http) = {

post: "/v1/processimage"

body: "requestimage"

};

}

}

message imagerequest {

bytes image\_data = 1; // Raw image data as bytes

ImageOperation operation = 2; // Type of image processing operation

int32 rotation\_angle = 3; // Rotation angle in degrees

int32 width = 4; // Target width for resizing

int32 height = 5; // Target height for resizing

}

message imageresponse {

bytes processed\_image\_data = 1; // Processed image data as bytes

}

------------------------------------------------------------------------------------

json

{

"type": "object",

"properties": {

"image\_data": {

"type": "string",

"format": "byte",

"description": "Base64-encoded image data"

},

"operations": {

"type": "array",

"items": {

"type": "string",

"enum": ["flip\_horizontal", "flip\_vertical", "rotate", "convert\_to\_grayscale", "resize", "generate\_thumbnail", "rotate\_left", "rotate\_right"],

"description": "Image processing operation"

},

"description": "List of image processing operations to perform"

}

}

}

Response Schema

<a name="processimageresponse"></a>

json

{

"type": "object",

"properties": {

"processed\_image\_data": {

"type": "string",

"format": "byte",

"description": "Base64-encoded processed image data"

}

}

}

write-up covering non-diagrammable details

Security Measures:

The system implements various security measures to protect sensitive data and ensure secure communication between components. This includes:

Authentication and authorization mechanisms to control access to the API and sensitive resources.

Scalability Strategies:

The system is designed to scale horizontally to handle increasing loads and accommodate growing user demand. This is achieved through:

Distributed architecture using services to enable independent scaling of components.

Error Handling Approaches:

The system implements robust error handling mechanisms to ensure resilience and reliability. This includes:

Proper validation of input data to prevent malformed requests and potential security vulnerabilities.

Data Privacy and Compliance:

The system adheres to data privacy regulations and compliance requirements to protect user privacy and ensure legal compliance. This includes:

Data anonymization and pseudonymization techniques to minimize the risk of data breaches and unauthorized access.

Monitoring and Performance Optimization:

The system is continuously monitored to track performance metrics, identify bottlenecks, and optimize resource utilization. This includes:

Real-time monitoring of key performance indicators (KPIs) such as response time, throughput, and error rates.

Disaster Recovery and Business Continuity:

The system implements disaster recovery and business continuity measures to minimize downtime and ensure continuity of operations in the event of catastrophic failures. This includes:

Regular backups of critical data and infrastructure to ensure data integrity and facilitate recovery in case of data loss or corruption.

Failover and redundancy configurations to enable seamless failover to backup systems or regions in case of infrastructure failures.

Communication Protocols

API Communication:

For communication between clients and the API server, HTTP/HTTP2/HTTPS is commonly used due to its widespread support, simplicity, and compatibility with web applications.

Alternatively, if performance and efficiency are primary concerns, especially for large-scale or real-time applications, consider using a binary protocol like Protocol Buffers or MessagePack over HTTP to reduce payload size and improve serialization/deserialization speed.

Internal service Communication:

Within the system's services architecture, inter-service communication is essential for coordination and collaboration between different components.

gRPC (Google Remote Procedure Call) is a popular choice for internal communication between services. It offers high performance, bi-directional streaming, and support for multiple programming languages.

With gRPC, services define their interface using Protocol Buffers, a language-neutral interface definition language, and communicate over HTTP/2, providing features such as multiplexing, header compression, and flow control.

Messaging Queues/Event Streams:

For asynchronous communication and decoupling between components, messaging queues or event streams are often employed.

Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), or Simple Queue Service (SQS) are common choices for implementing message queues, offering features like message persistence, delivery guarantees, and pub/sub messaging patterns.

Monitoring and Logging:

For monitoring system health and collecting logs, protocols like Simple Network Management Protocol (SNMP) or Telegraf/InfluxDB can be utilized.

SNMP is a standard protocol for monitoring and managing network devices, offering features for querying device status, collecting performance metrics, and receiving notifications about critical events.

Justification for Architecture

Modularity and Separation of Concerns:

The system is designed as a collection of loosely coupled services, each responsible for specific tasks such as image processing, API gateway, persistence, and monitoring/logging.

Scalability and Performance:

The service architecture enables horizontal scalability, allowing the system to handle increased loads by adding more instances of individual services.

Components like the API gateway and image processing service can scale independently based on demand, ensuring optimal resource utilization and improved performance during peak usage periods.

Fault Tolerance and Resilience:

The distributed nature of services architecture enhances fault tolerance and resilience by isolating failures to specific components rather than affecting the entire system.

Redundancy, fault detection, and recovery mechanisms are implemented at the service level, ensuring that the system remains operational even in the face of failures or disruptions.

Flexibility and Agility:

The use of gRPC APIs facilitates interoperability and integration with other systems and clients, allowing for flexibility in client applications and service composition.

Changes and updates to individual components can be implemented independently without affecting the overall system, enabling agility in development and deployment processes.

Security and Compliance:

Security measures such as encryption, authentication, and access control are implemented at various layers of the architecture to protect sensitive data and ensure compliance with regulatory requirements.

Operational Efficiency and Monitoring:

The architecture includes components for monitoring, logging, and performance optimization, enabling proactive detection of issues, efficient troubleshooting, and continuous improvement.

Metrics and logs collected from different components provide insights into system behavior, resource utilization, and user interactions, facilitating informed decision-making and optimization efforts.

API Design Document

Introduction

The Image Processing API provides a set of endpoints for performing various image processing operations, including flipping, rotation, resizing, conversion to grayscale, and thumbnail generation.

API Overview

Base URL: http://api.example.com/v1

API Version: 1.0

Protocol: gRPC with HTTP/2

Authentication: None (proto file should be reference at client side)

API Endpoints

The API exposes the following gRPC methods, mapped to HTTP endpoints:

1. ProcessImage

Description: Process an image with the specified operations.

HTTP Method: POST

HTTP Path: /images/process

Request Body: ProcessImageRequest

Response Body: ProcessImageResponse

Request Schema

<a name="processimagerequest"></a>

json

{

"type": "object",

"properties": {

"image\_data": {

"type": "string",

"format": "byte",

"description": "Base64-encoded image data"

},

"operations": {

"type": "array",

"items": {

"type": "string",

"enum": ["flip\_horizontal", "flip\_vertical", "rotate", "convert\_to\_grayscale", "resize", "generate\_thumbnail", "rotate\_left", "rotate\_right"],

"description": "Image processing operation"

},

"description": "List of image processing operations to perform"

}

}

}

Response Schema

<a name="processimageresponse"></a>

json

{

"type": "object",

"properties": {

"processed\_image\_data": {

"type": "string",

"format": "byte",

"description": "Base64-encoded processed image data"

}

}

}

Error Responses

The API may return the following error responses:

400 Bad Request: Invalid request parameters.

404 Not Found: Resource not found.

500 Internal Server Error: Server error occurred.

Example Requests

1. ProcessImage

Request

http(protobuf)

POST /v1/images/process HTTP/1.1

Host: api.example.com

Content-Type: application/json

{

"image\_data": "base64\_encoded\_image\_data",

"operations": ["flip\_horizontal", "convert\_to\_grayscale"]

}

Response

http(protobuf)

HTTP/1.1 200 OK

Content-Type: application/json

{

"processed\_image\_data": "base64\_encoded\_processed\_image\_data"

}

API Specification from Client Perspective

From the client perspective, interacting with the image processing API involves making HTTP requests to the API endpoints with the appropriate parameters. Here's the API specification outlining the endpoints, parameters, and expected behavior:

Endpoint: /image/processing

Method: POST

Description: Performs image processing operations on the provided image data.

Request Parameters:

image\_data: Binary image data representing the image to be processed.

operations: List of strings representing the image processing operations to be applied. Possible values include "rotate", "flip", "resize", etc.

Authentication and Authorization:

The API may require authentication and authorization for certain endpoints to ensure secure access and protect sensitive operations.

Error Handling:

The API should provide informative error messages and appropriate HTTP status codes to client.

Internal Implementation Details

Services Architecture:

The image processing system is designed as a collection of services, each responsible for specific functionalities such as image processing, API gateway, persistence, monitoring, and logging.

Services are independently deployable and scalable, allowing for better isolation of concerns and easier maintenance.

Service Orchestration:

Service orchestration is implemented to coordinate the interaction between different services and manage the flow of data within the system.

Image Processing Logic:

The image processing service contains the core logic for performing image processing operations such as rotation, flipping, resizing, filtering, etc.

Image processing algorithms are implemented using libraries such as OpenCV or ImageSharp, depending on the programming language and platform used.

Error handling and validation mechanisms are incorporated to handle edge cases and ensure the reliability of image processing operations.

Persistence Layer:

The persistence layer is responsible for storing and retrieving processed images, session data, configuration settings, and logging information.

API Gateway:

The API gateway serves as a single entry point for clients to access the image processing service.

It handles incoming requests, routes them to the appropriate services, and performs tasks such as authentication, rate limiting, and request transformation.

Monitoring and Logging:

Monitoring and logging components are integrated into the system to track system health, performance metrics, and user interactions.

Security Measures:

Security measures such as authentication, authorization, encryption, and access control are implemented to protect sensitive data and ensure compliance with security standards.

API Method Implementation

The /image/processing endpoint accepts POST requests with image data (sent as a file) and operations.

The process\_image function extracts the image data and operations from the request, processes the image using the apply\_operations function, and returns the processed image.

The apply\_operations function applies each operation (e.g., rotate, flip) to the image.

The send\_image function converts the processed image back to binary data and returns it as a response with the appropriate content type.

Apply Operations Considerations (Problem Specifications)

Flip:

Allows flipping the image horizontally or vertically.

Use horizontal or vertical as parameters to indicate the direction of flipping.

Example: operations=flip,horizontal or operations=flip,vertical

Rotate:

Supports rotating the image by custom degrees, both clockwise and counter-clockwise.

Use positive values for clockwise rotation (+) and negative values for counter-clockwise rotation (-).

Example: operations=rotate,45 (Rotate by 45 degrees clockwise) or operations=rotate,-90 (Rotate by 90 degrees counter-clockwise)

Grayscale:

Provides an option to convert the image to grayscale, removing color information.

Example: operations=grayscale

Resize:

Enables resizing the image to custom dimensions, preserving aspect ratio.

Specify width and height as custom dimensions.

Example: operations=resize,300,200 (Resize to width: 300 pixels, height: 200 pixels)

Generate Thumbnail:

Allows generating a thumbnail of the image with specified maximum dimensions.

Specify the maximum width and height of the thumbnail.

Example: operations=generatethumbnail,100,100 (Generate a thumbnail with a maximum width and height of 100 pixels)

Rotate Left and Right:

Offers shortcuts for rotating the image by 90 degrees counter-clockwise (rotateleft) and clockwise (rotateright).

Example: operations=rotateleft or operations=rotateright

You can extend this implementation by adding more image processing operations and error handling as needed.

System Validation Approach

Unit Testing:

Write unit tests for individual components, including image processing algorithms, API endpoints, and persistence layer functionality.

Integration Testing:

Conduct integration tests to verify the interactions between different components of the system.

Load Testing:

Perform load testing to evaluate the system's performance under various levels of load and stress.

Security Testing:

Conduct security testing to identify and mitigate vulnerabilities in the system.

Fault Tolerance Testing:

Test the system's resilience to failures and disruptions by introducing faults and failures into the environment.

Scalability Testing:

Evaluate the system's scalability by gradually increasing the load and measuring its ability to handle additional requests.

Regression Testing:

Perform regression testing to ensure that new changes or updates do not introduce regressions or break existing functionality.

Monitoring and Observability:

Implement monitoring and observability tools to continuously monitor the system in production.

Common Bad Request Input Validation

invalid argument, outOfRange, FailedPreCondition

Validate the incoming request parameters to ensure they meet the expected criteria.

Check for null values, empty strings, out-of-range values, or any other invalid inputs.

Use data annotations or custom validation logic to validate message fields.

Handling Bad Request Methods and Response Logic

Interceptor:

Implement a server-side interceptor to intercept incoming requests before they reach the actual service methods.

Perform input validation and error handling logic in the interceptor.

Return error responses directly from the interceptor if the request is invalid.

Validation Middleware:

Implement validation middleware to handle input validation separately from the service methods.

Pass the request through the validation middleware before invoking the actual service methods.

Return error responses from the middleware if the request is invalid.

Middleware can check response like multiple images, response data contents(processed image and zero or more).

Pseudo Code

// Inside your service method

if (request.SomeField == null || request.SomeField.Length == 0)

{

throw new RpcException(new Status(StatusCode.InvalidArgument, "SomeField cannot be null or empty."));

}

Sample Client Code:

class Program

{

private const string ServerAddress = "localhost"; // Replace with your server address

private const int ServerPort = 50051; // Replace with your server port

static async Task Main(string[] args)

{

// Initialize a channel to the gRPC server

Channel channel = new Channel($"{ServerAddress}:{ServerPort}", ChannelCredentials.Insecure);

try

{

// Create a client for the image processing service

var client = new YourServiceClient(channel);

// Example usage: call a gRPC method to perform an image processing operation

// Replace with your actual method calls and parameters

var request = new ImageProcessingRequest

{

ImageData = ByteString.CopyFrom(File.ReadAllBytes("input.jpg")), // Read image data from file

Operation = ImageOperation.Rotate, // Example: specify the operation

RotationAngle = 90 // Example: specify rotation angle

};

// Call the gRPC method asynchronously

var response = await client.ProcessImageAsync(request);

// Handle the response

if (response != null)

{

Console.WriteLine("Image processed successfully.");

// Example: Save the processed image data to a file

File.WriteAllBytes("output.jpg", response.ProcessedImageData.ToByteArray());

}

else

{

Console.WriteLine("Failed to process image.");

}

}

catch (RpcException ex)

{

Console.WriteLine($"RPC Error: {ex.Status.Detail}");

}

finally

{

// Shutdown the channel when done

await channel.ShutdownAsync();

}

}

}

API Method Implementation

Protobuf Messages

syntax = "proto3";

message ImageData {

bytes data = 1;

string format = 2;

}

message OperationRequest {

ImageData image = 1;

repeated OperationType operations = 2;

}

enum OperationType {

FLIP\_HORIZONTAL = 0;

FLIP\_VERTICAL = 1;

ROTATE\_CLOCKWISE = 2;

ROTATE\_COUNTERCLOCKWISE = 3;

CONVERT\_TO\_GRAYSCALE = 4;

RESIZE = 5;

GENERATE\_THUMBNAIL = 6;

}

Grpc Service pseudo code

public class ImageProcessingService : ImageProcessor.ImageProcessorBase

{

public override Task<OperationResponse> ProcessImage(OperationRequest request, ServerCallContext context)

{

// Implement image processing logic based on the requested operations

// This includes flipping, rotating, converting to grayscale, resizing, etc.

// Use third-party libraries like ImageSharp or OpenCV for image processing.

public byte[] ProcessImage(byte[] imageData, ImageOperation operation, int rotationAngle = 0, int width = 0, int height = 0)

{

using (MemoryStream inStream = new MemoryStream(imageData))

using (Image image = Image.Load(inStream))

{

switch (operation)

{

case ImageOperation.FlipHorizontal:

image.Mutate(x => x.Flip(FlipMode.Horizontal));

break;

case ImageOperation.FlipVertical:

image.Mutate(x => x.Flip(FlipMode.Vertical));

break;

case ImageOperation.Rotate:

image.Mutate(x => x.Rotate(rotationAngle));

break;

case ImageOperation.ConvertToGrayscale:

image.Mutate(x => x.Grayscale());

break;

case ImageOperation.Resize:

image.Mutate(x => x.Resize(width, height));

break;

case ImageOperation.GenerateThumbnail:

// Resize to thumbnail dimensions (e.g., 100x100 pixels)

image.Mutate(x => x.Resize(100, 100));

break;

case ImageOperation.RotateLeft:

image.Mutate(x => x.Rotate(-90));

break;

case ImageOperation.RotateRight:

image.Mutate(x => x.Rotate(90));

break;

default:

throw new ArgumentException("Invalid image operation specified.");

}

using (MemoryStream outStream = new MemoryStream())

{

image.Save(outStream, new SixLabors.ImageSharp.Formats.Jpeg.JpegEncoder());

return outStream.ToArray();

}

}

}

// Return the processed image data in the response

}

}

We are using a library like ImageSharp for image manipulation. Make sure to install the ImageSharp package via NuGet if you haven't already.

The ImageProcessor class contains a method ProcessImage which takes the image data as a byte array (imageData) and performs the specified image processing operation (operation).

Depending on the value of operation, it applies the corresponding transformation using the ImageSharp library's Mutate method.

The processed image is then saved to a new MemoryStream, and its byte array is returned as the result.

Little Languages

Image Processing Server

ProcessImageResponse ProcessImage(ProcessImageRequest)

{

New MemoryStream(imageData.ByteArray())

Image.Load(MemoryStream)

Switch(operation)

{

Image.Mutate(operation)

}

Image.Save(OutStream,JpegEncoder)

ProcessImageData(OutStream)

}

Below are the little description of the ProcessImage method in the context of image processing using the ImageSharp library:

Method Signature:

public byte[] ProcessImage(byte[] imageData, ImageOperation operation, int rotationAngle = 0, int width = 0, int height = 0)

Parameters:

imageData: A byte array representing the raw image data. This is the input image that will undergo processing.

operation: An enum value representing the type of image processing operation to perform. It specifies the transformation to be applied to the input image.

rotationAngle: An optional parameter representing the rotation angle in degrees. It is used for operations such as rotation and defaults to 0 if not provided.

width and height: Optional parameters representing the target width and height for resizing. They are used when resizing the image and default to 0, which means maintaining the original aspect ratio.

Return Type:

byte[]: The method returns a byte array representing the processed image data. This allows for easy transfer and manipulation of image data in binary format.

Method Description:

The ProcessImage method is the core functionality responsible for performing various image processing operations.

It accepts input image data along with parameters specifying the desired operation.

The method uses the ImageSharp library for image manipulation, making use of its versatile features for handling image processing tasks.

Depending on the specified operation, the method applies the corresponding transformation to the input image using the Mutate method provided by ImageSharp.

After processing, the method converts the processed image to a byte array and returns it to the caller for further use or storage.

Error Handling:

The method does not perform extensive error handling in this implementation, assuming valid input data and supported operations.

However, in a production environment, you would typically add error handling to handle exceptions that may occur during image processing, such as invalid input data, unsupported image formats, or library-specific errors.

Error handling would involve catching and appropriately handling exceptions, possibly providing informative error messages to aid in debugging or user feedback.

Memory Management:

The method uses using statements to ensure proper management of resources such as memory streams and image objects.

It loads the input image into a memory stream and disposes of it when no longer needed to avoid memory leaks.

Similarly, it creates a new memory stream to store the processed image and ensures proper disposal after conversion to a byte array.

Host Grpc Server

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddGrpc();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapGrpcService<ImageProcessingService>();

});

}

}

Host Grpc Server

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddGrpc();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapGrpcService<ImageProcessingService>();

});

}

}

Finally, generate client stub using grpc tools and call required methods.