Testing APIs on Postman

Project Description

This project involves using gRPC Protocol Buffers to create APIs for managing employee data. The data is stored in an SQLite database and simultaneously cached in Redis for improved performance. The database consists of three tables: employee data, designation data, and web data. The project is implemented in Python, using Flask. The Flask application acts as a client for a gRPC server. It provides REST API endpoints for creating, updating, deleting, and retrieving employee and designation data, as well as interacting with Redis and a SignalR hub. The Flask app communicates with the gRPC server using generated stubs from the Protocol Buffers definitions. The APIs created on Flask are then tested on Postman.

Project Structure

assetsclientsAPI/

assetproto/

employee.proto : *proto file for employee*

employee_pb2_grpc.py: *Contains server-side gRPC service definitions and stub classes for implementing gRPC*

employee_pb2.py: *Contains Python classes for messages defined in employee.proto*

designation.proto: *proto file for designation*

designation_pb2.py: *Contains Python classes for messages defined in designation.proto*

designation_pb2_grpc.py: *Contains server-side gRPC service definitions and stub classes for implementing gRPC*

web.proto: *proto file for web*

web_pb2_grpc.py: *Contains server-side gRPC service definitions and stub classes for implementing gRPC*

web_pb2.py: *Contains Python classes for messages defined in web.proto*

assetserver/

controller/

employeecontroller.py: *defines gRPC service controller for EmployeeService class*

designationcontroller.py: *defines gRPC service controller for DesignationService class*

webcontroller.py: *defines gRPC service controller for WebService class*

model/

EmployeeDetailsSchema.py: *Creates Employee table and defines Table Schema*

DesignationDetailsSchema.py: *Creates Designation table and defines Table Schema*

WebDetailsSchema.py: *Creates Web table and defines Table Schema*

packages/

session.py: *Initializes SQLite create engine, Redis-client, SignalR HubConnection and provides methods for them*

base.py: *Initializes objects for methods in session.py by extending sessions()*

service/

EmployeeService.py: *gRPC service implementation for handling operations on Employee Table using SQLAlchemy as well as Redis caching*

DesignationService.py: *gRPC sevice implementation for handling operation On Designation Table using SQLAlchemy as well as Redis for caching*

WebService.py: *gRPC service implementation for handling operations on Web Table using SQLAlchemy and SignalR for http interfacing*

employeedata.db: *Database containing employee, designation and web tables*

server.py: *Starts gRPC server with EmployeeService. DesignationService and WebService which is done by accessing their controllers*

assetclient/

asset_client.py: *python file containing code for client using flask*

employee.csv: *CSV file containing some employee records*

pythonimp/

Webservice/

ChatHub.cs: *Defines SignalR hub called ChatHub*

Program.cs: *Sets up entry point and Configuration for ASP.NET core web application*

Startup.cs: *Configures Services and request processing using ASP.NET core web Application and sets up SignalR and Razor Pages*

pages/

Index.cshtml: *Razor pages view file for creating html home page for ASP.NET core web application*

wwwroot/

js/

signalr/

signalr.js: *SignalR javascript file*

chat.js: *Establishes Connection to SignalR Hub and handles sending and receiving messages to the Hub using webpages.*

main.py: *Establishes WebSocket connection to SignalR Hub enabling a user a interact using simple console interface*

APIs

- Create Employee Data (POST /create): Creates a new employee record.
- Create Designation (POST /createdesignation): Creates a new designation.
- Get Employee Data (GET /get): Retrieves employee details based on employee ID.
- Get Designation Data (GET /getdesignation): Retrieves designation details based on designation ID.
- Update Employee Data (PUT /update): Updates employee details based on employee ID.
- Delete Employee Data (DELETE /delete): Deletes an employee record based on employee ID.
- Select All Employee Data (GET /selectallemployee): Retrieves details of all employees.
- Select Specific Column (POST /selectspecificcolumn): Retrieves values of specific columns for all employees.

- Select Salary by Range (POST /selectsalarybyrange): Retrieves employees whose salary falls within a specified range.
- Get Redis Data (POST /getredis): Retrieves data from Redis based on a key.
- Post Form Data (POST /postform): Inserts employee records from a CSV file uploaded as form data.
- Get Web App Data (POST /getwebapp): Sends a message to a SignalR hub for a web application.

Project Work Flow

1. Start the gRPC Server:
   Run the Python script that starts the gRPC server. This initializes the server, registers the service controllers, and starts the server on port 50051.

2. Start the Flask Application:
   Run the Flask application on port 5001. This script initializes the Flask app, sets up the gRPC channel, and defines the API endpoints for handling HTTP requests.

3. Start the ASP.NET Core Server:
   - Navigate to the directory containing your ASP.NET Core project.
   - Run the ASP.NET Core server using the “dotnet run” command
   - The ASP.NET Core server should start and listen for SignalR connections on port 6001

4. Send HTTP Requests:
   - Use Postman to send HTTP requests to the Flask application's API endpoints.
   - Use different endpoints for different operations.

5. Handle Requests on the Server:
   - The Flask application receives the HTTP requests and converts them into gRPC requests.
   - The gRPC requests are sent to the gRPC server using the client stubs.

6. Process Requests on the Server:
   The gRPC server receives the requests, processes them using the service controllers, interacts with the database and Redis, and prepares responses.

7. Send Responses to Clients:
   - The gRPC server sends responses back to the Flask application, which converts them into HTTP responses.
   - The Flask application returns the HTTP responses to the client.

Commands

1. For generating protobuf files:
   python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=.
   protofilename.proto

2. For creating a C# project:
   dotnet new webapp -o filename