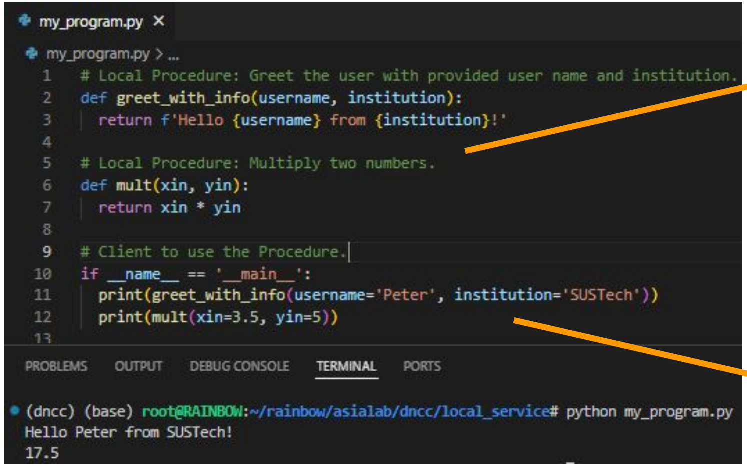
DISTRIBUTED AND CLOUD COMPUTING

LAB 4: SERVICES & API ARCHITECTURES

(Module: RPC & RESTFUL API)

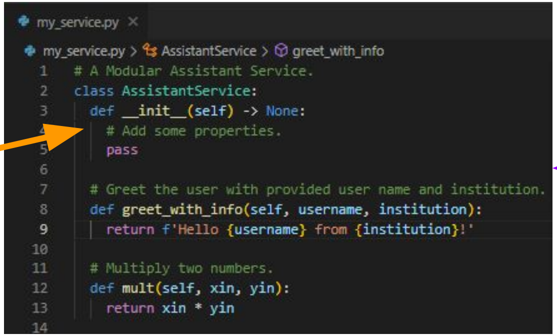
Local Program ➡ Web Service

- Local Program ➡ Modular Service
  a. Task Orientation + Reusability: group functions/procedures as a **service**.
  b. Independent Development: Separate client and service code - **Modularization**.



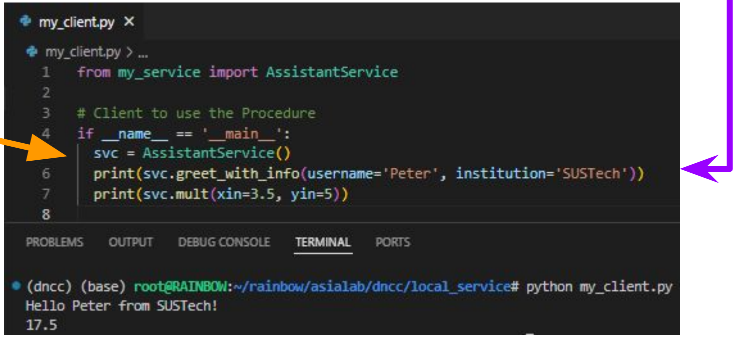
```python
# Local Procedure: Greet the user with provided user name and institution.
def greet_with_info(username, institution):
    return f'Hello {username} from {institution}!'

# Local Procedure: Multiply two numbers.
def mult(xin, yin):
    return xin * yin

# Client to use the Procedure.
if __name__ == '__main__':
    print(greet_with_info(username='Peter', institution='SUSTech'))
    print(mult(xin=3.5, yin=5))
```

```
(dncc) (base) root@RAINBOW:~/rainbow/asialab/dncc/local_service# python my_program.py
Hello Peter from SUSTech!
17.5
```



```python
# A Modular Assistant Service.
class AssistantService:
    def __init__(self) -> None:
        # Add some properties.
        pass

    # Greet the user with provided user name and institution.
    def greet_with_info(self, username, institution):
        return f'Hello {username} from {institution}!'

    # Multiply two numbers.
    def mult(self, xin, yin):
        return xin * yin
```

What if service is no longer local?



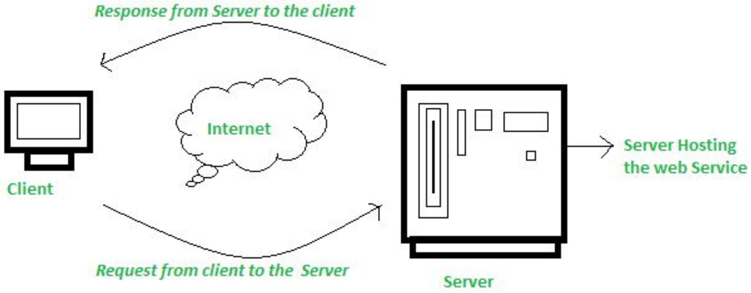
```python
from my_service import AssistantService

# Client to use the Procedure
if __name__ == '__main__':
    svc = AssistantService()
    print(svc.greet_with_info(username='Peter', institution='SUSTech'))
    print(svc.mult(xin=3.5, yin=5))
```

```
(dncc) (base) root@RAINBOW:~/rainbow/asialab/dncc/local_service# python my_client.py
Hello Peter from SUSTech!
17.5
```

Local Program ➡ Web Service

- Local Program ➡ Modular Service
  a. Task Orientation + Reusability: group functions/procedures as a **service**.
  b. Independent Development: Separate client and service code - **Modularization**.
- Local Service ➡ Web Service
  a. **Web Service**: “a service offered by an electronic device to another electronic device, communicating with each other via the Internet”.
  b. Making services web-ready: utilize web standards.
     - Network Protocols (e.g., HTTP, HTTPS)
     - Data Formats (e.g., XML, JSON)

- Web Services specify communication standards and interfaces, making them **APIs**.

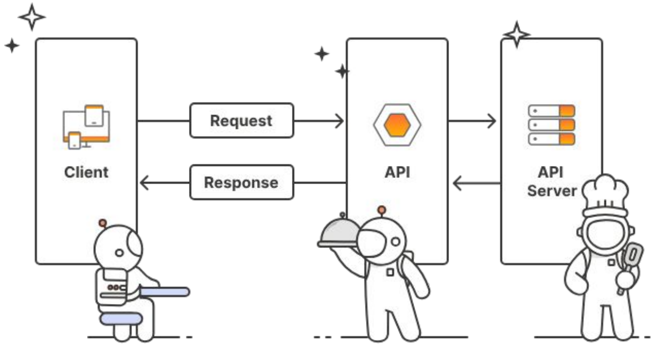

Response from Server to the client
Client
Internet
Server Hosting the web Service
Request from client to the Server
Server

https://en.wikipedia.org/wiki/Web_service
https://www.geeksforgeeks.org/what-are-web-services/

Application Programming Interface (API)

- "APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols."
- Examples:
  a. GitHub REST API - build scripts/apps for automation and extension
  b. OpenAI API (RESTful API) - programmatically interact with LLMs like ChatGPT
  c. Slack Web API (gRPC) - query and manage Slack messages, channels, workspaces
  d. Netflix Backend-to-backend Communication (gRPC)
  e. …

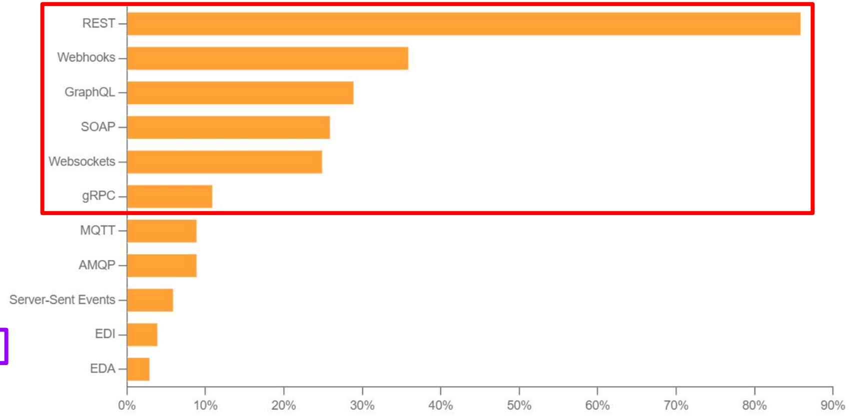
- There are different ways to design and build APIs, each tailored to different purposes and scenarios. These designs are referred to as "**API Architecture Patterns**".

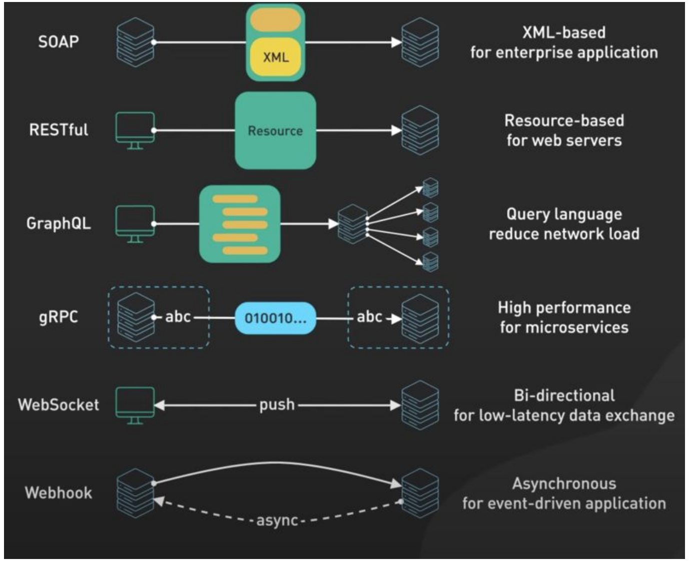


https://aws.amazon.com/what-is/api/
https://www.postman.com/what-is-an-api/

API Architecture Patterns

- **SOAP**:
  a. uses XML messages + a predefined contract
  b. strict, complex, and verbose
- **RESTful API**: **Currently the most popular architecture for Web Services!**
  a. manages resources via common web standards
  b. uses HTTP methods to operate on JSON/XML
  c. simple, fast, and flexible
- **GraphQL**:
  a. queries exactly needed within a single requests
  b. suitable for complex data requirements
- **RPC**:
  a. accesses remote services as if they were local
  b. abstracts the complexity of network communication
- **WebSocket**:
  a. enables fast, bidirectional, and persistent connections
  b. benefits live chat apps and real-time gaming
- **Webhook**:
  a. supports asynchronous & event-driven notifications





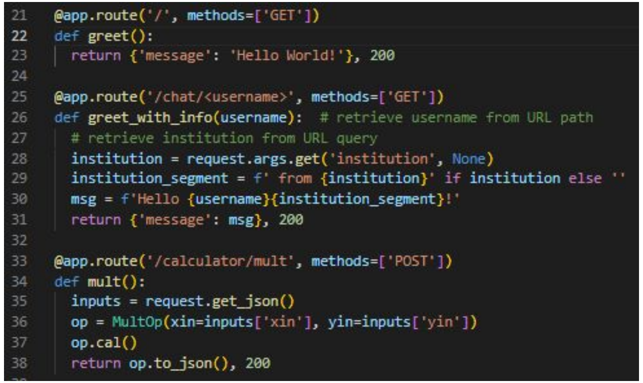
https://www.postman.com/state-of-api/api-technologies/#api-technologies
https://dev.to/kanani_nirav/top-6-most-popular-api-architecture-styles-you-need-to-know-with-pros-cons-and-use-cases-564j

TASK: RESTful API - Hello World with Python Flask

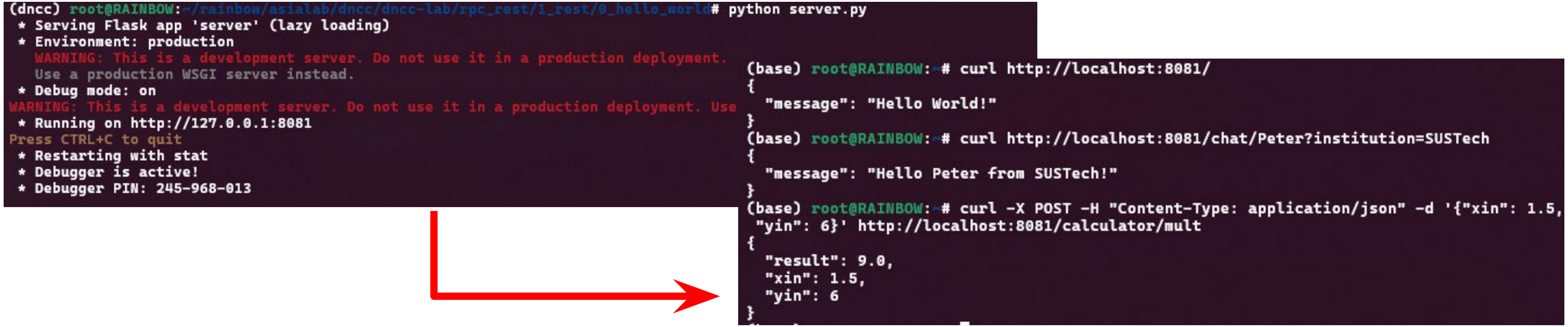
Implement a simple RESTful API server using Python Flask.

> Reference codebase: `rest_hello_world`

1. Set up Python (Miniconda is recommended).
2. Install Python dependencies into a Conda environment via:
   - `python -m pip install -r requirements.txt`
3. Run the API server via:
   - `python server.py`
4. In another terminal, test the API with HTTP requests.



```python
@app.route('/', methods=['GET'])
def greet():
  return {'message': 'Hello World!'}, 200

@app.route('/chat/<username>', methods=['GET'])
def greet_with_info(username):  # retrieve username from URL path
  # retrieve institution from URL query
  institution = request.args.get('institution', None)
  institution_segment = f' from {institution}' if institution else ''
  msg = f'Hello {username}{institution_segment}!'
  return {'message': msg}, 200

@app.route('/calculator/mult', methods=['POST'])
def mult():
  inputs = request.get_json()
  op = MultOp(xin=inputs['xin'], yin=inputs['yin'])
  op.cal()
  return op.to_json(), 200
```



```
(dncc) root@RAINBOW:~/rainbow/asialab/dncc/dncc-lab/rpc_rest/1_rest/0_hello_world# python server.py
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use
 * Running on http://127.0.0.1:8081
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 245-968-013
```

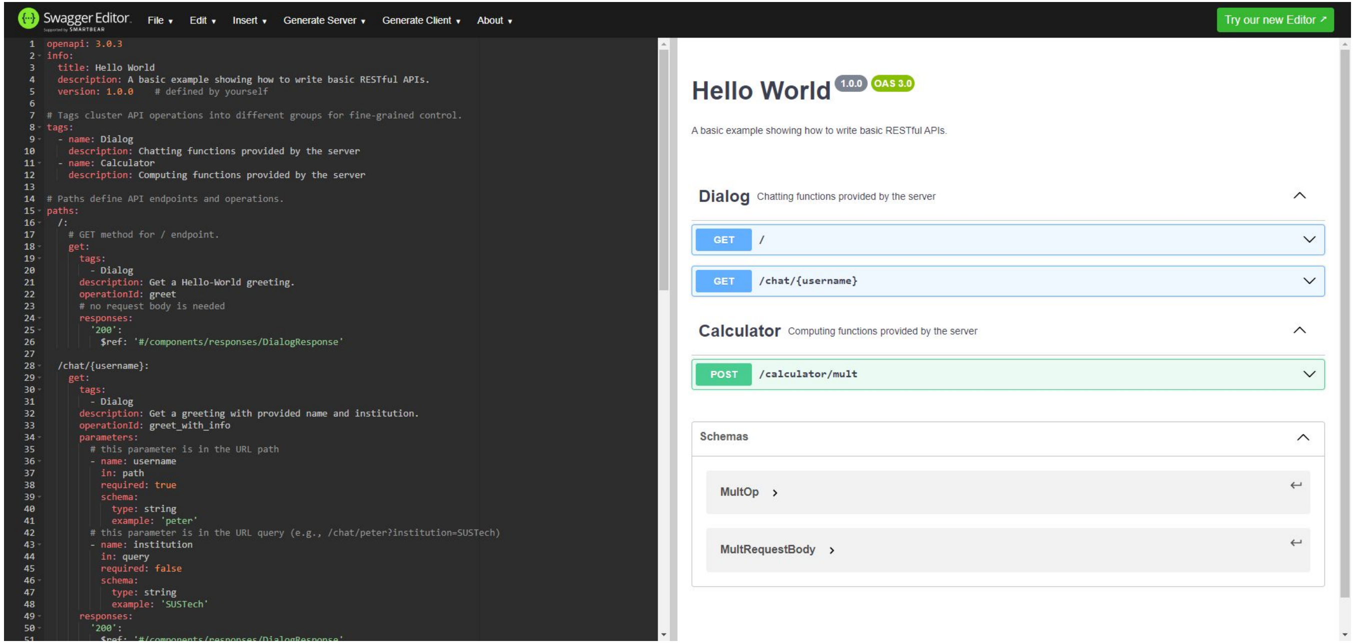
```
(base) root@RAINBOW:~# curl http://localhost:8081/
{
  "message": "Hello World!"
}
(base) root@RAINBOW:~# curl http://localhost:8081/chat/Peter?institution=SUSTech
{
  "message": "Hello Peter from SUSTech!"
}
(base) root@RAINBOW:~# curl -X POST -H "Content-Type: application/json" -d '{"xin": 1.5,
 "yin": 6}' http://localhost:8081/calculator/mult
{
  "result": 9.0,
  "xin": 1.5,
  "yin": 6
}
```

TASK: RESTful API - Hello World with Python Flask

Implement a simple RESTful API server using Python Flask.

> Reference codebase: `rest_hello_world`

5. Check the API specification file `hello_world.yaml`
6. Copy the file content into the online Swagger Editor. Explore the generated Swagger UI page.



```yaml
openapi: 3.0.3
info:
  title: Hello World
  description: A basic example showing how to write basic RESTful APIs.
  version: 1.0.0    # defined by yourself

# Tags cluster API operations into different groups for fine-grained control.
tags:
  - name: Dialog
    description: Chatting functions provided by the server
  - name: Calculator
    description: Computing functions provided by the server

# Paths define API endpoints and operations.
paths:
  /:
    # GET method for / endpoint.
    get:
      tags:
        - Dialog
      description: Get a Hello-World greeting.
      operationId: greet
      # no request body is needed
      responses:
        '200':
          $ref: '#/components/responses/DialogResponse'

  /chat/{username}:
    get:
      tags:
        - Dialog
      description: Get a greeting with provided name and institution.
      operationId: greet_with_info
      parameters:
        # this parameter is in the URL path
        - name: username
          in: path
          required: true
          schema:
            type: string
            example: 'peter'
        # this parameter is in the URL query (e.g., /chat/peter?institution=SUSTech)
        - name: institution
          in: query
          required: false
          schema:
            type: string
            example: 'SUSTech'
      responses:
        '200':
          $ref: '#/components/responses/DialogResponse'
```

Hello World 1.0.0 OAS 3.0

A basic example showing how to write basic RESTful APIs.

Dialog Chatting functions provided by the server

GET `/`

GET `/chat/{username}`

Calculator Computing functions provided by the server

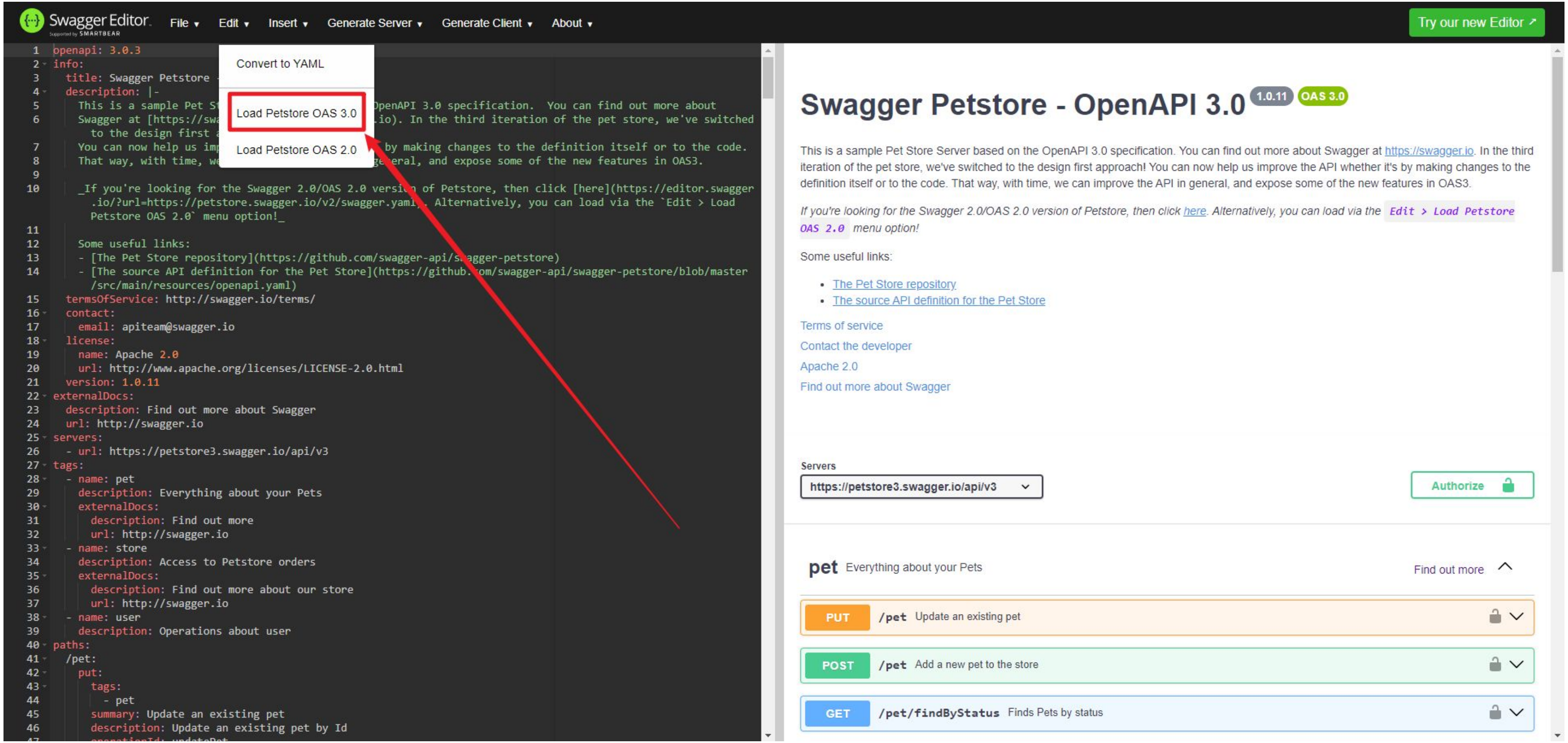
POST `/calculator/mult`

Schemas

MultOp ›

MultRequestBody ›

Another RESTful API Example - Petstore



```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore -
  description: |-
    This is a sample Pet St                        OpenAPI 3.0 specification.  You can find out more about
    Swagger at [https://swa                        .io). In the third iteration of the pet store, we've switched
      to the design first a
    You can now help us imp                        by making changes to the definition itself or to the code.
    That way, with time, we                        general, and expose some of the new features in OAS3.

    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](https://editor.swagger
      .io/?url=https://petstore.swagger.io/v2/swagger.yaml). Alternatively, you can load via the `Edit > Load
      Petstore OAS 2.0` menu option!_

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master
      /src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: http://swagger.io
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: http://swagger.io
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
```

Swagger Petstore - OpenAPI 3.0 1.0.11 OAS 3.0

This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can find out more about Swagger at https://swagger.io. In the third iteration of the pet store, we've switched to the design first approach! You can now help us improve the API whether it's by making changes to the definition itself or to the code. That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click here. Alternatively, you can load via the `Edit > Load Petstore OAS 2.0` menu option!

Some useful links:

- The Pet Store repository
- The source API definition for the Pet Store

Terms of service

Contact the developer

Apache 2.0

Find out more about Swagger

Servers

https://petstore3.swagger.io/api/v3 ⌄

Authorize 🔓

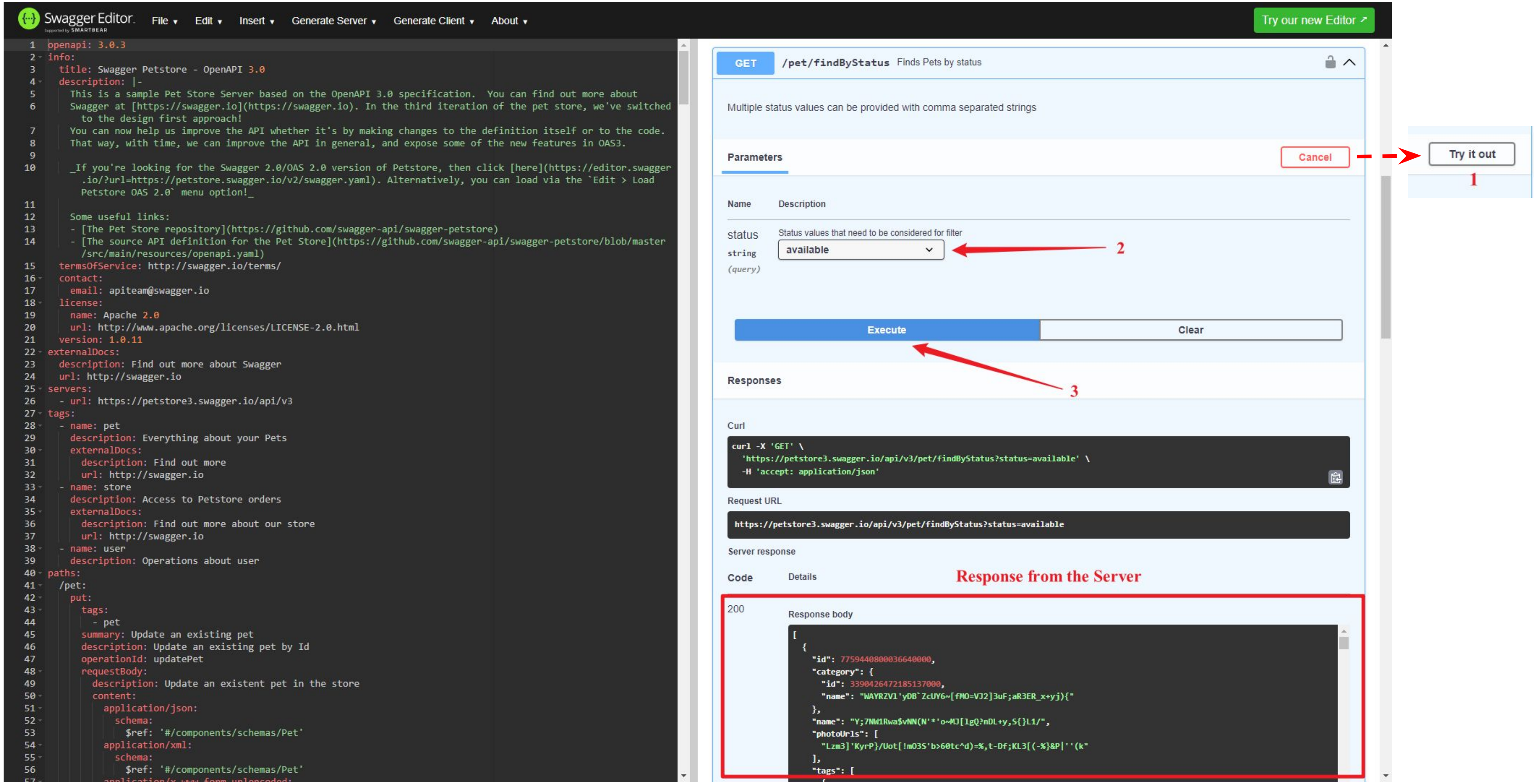
pet Everything about your Pets — Find out more

PUT /pet Update an existing pet

POST /pet Add a new pet to the store

GET /pet/findByStatus Finds Pets by status

Another RESTful API Example - Petstore



```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about
    Swagger at [https://swagger.io](https://swagger.io). In the third iteration of the pet store, we've switched
      to the design first approach!
    You can now help us improve the API whether it's by making changes to the definition itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](https://editor.swagger
      .io/?url=https://petstore.swagger.io/v2/swagger.yaml). Alternatively, you can load via the `Edit > Load
      Petstore OAS 2.0` menu option!_

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master
      /src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: http://swagger.io
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: http://swagger.io
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
```

Multiple status values can be provided with comma separated strings

status string (query): Status values that need to be considered for filter

Curl

```
curl -X 'GET' \
  'https://petstore3.swagger.io/api/v3/pet/findByStatus?status=available' \
  -H 'accept: application/json'
```

Request URL

```
https://petstore3.swagger.io/api/v3/pet/findByStatus?status=available
```

Server response

Response from the Server

200

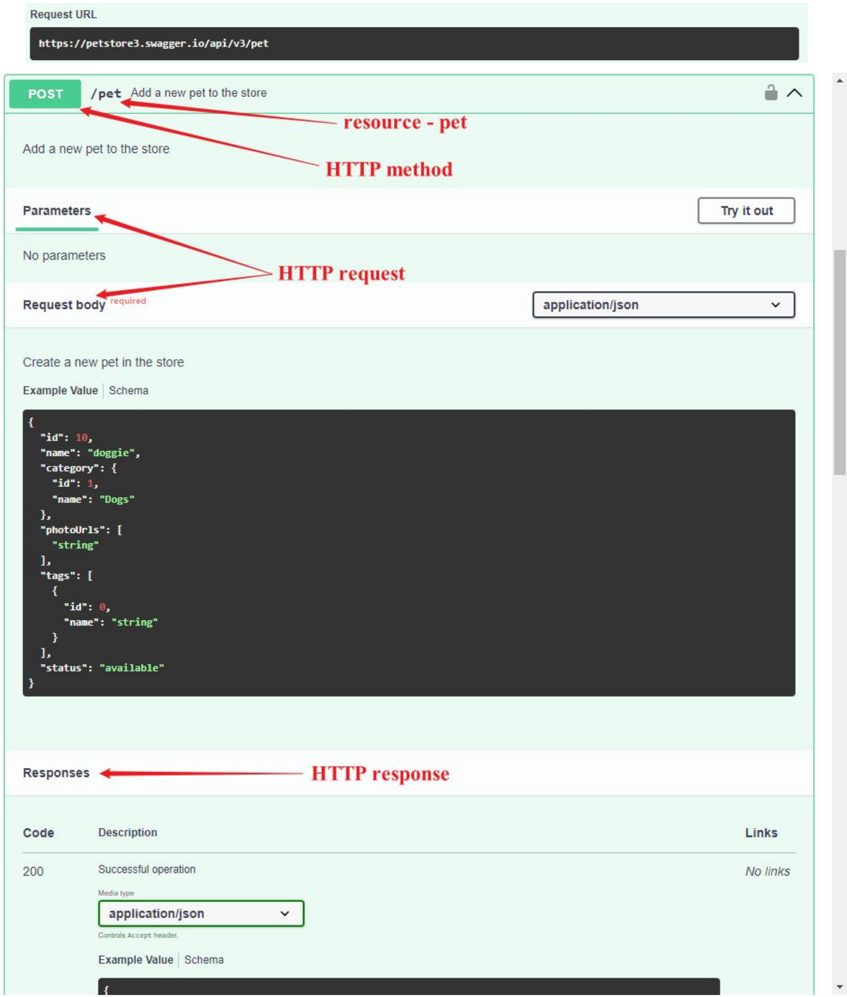
Response body

```json
[
  {
    "id": 7759440800036640000,
    "category": {
      "id": 3390426472185137000,
      "name": "WAYRZV1'yDB`ZcUY6~[fMO=VJ2]3uF;aR3ER_x+yj){"
    },
    "name": "Y;7NW1Rwa$vNN(N'*'o~MJ[1gQ?nDL+y,S{}L1/",
    "photoUrls": [
      "Lzm3]'KyrP}/Uot[!mO3S'b>60tc^d)=%,t-Df;KL3[(-%}&P|''(k"
    ],
    "tags": [
```

RESTful API Components

Take OpenAPI specification as an example:

- Resource: identified by an endpoint/URL.
- HTTP method: specifies an operation on the resource (e.g., GET, POST, etc.).
- HTTP request:
  - Header: metadata
    - `Content-Type = application/json`)
  - Query Parameters
    - `/users?age=25&sort=desc`
  - Request Body
- HTTP response:
  - Status code + message
    - `200 OK`
    - `404 Not Found`
    - ...
  - Header
  - Response Body



Request URL

`https://petstore3.swagger.io/api/v3/pet`

POST /pet Add a new pet to the store

Add a new pet to the store

Parameters

No parameters

Request body required

application/json

Create a new pet in the store

Example Value | Schema

```
{
  "id": 10,
  "name": "doggie",
  "category": {
    "id": 1,
    "name": "Dogs"
  },
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

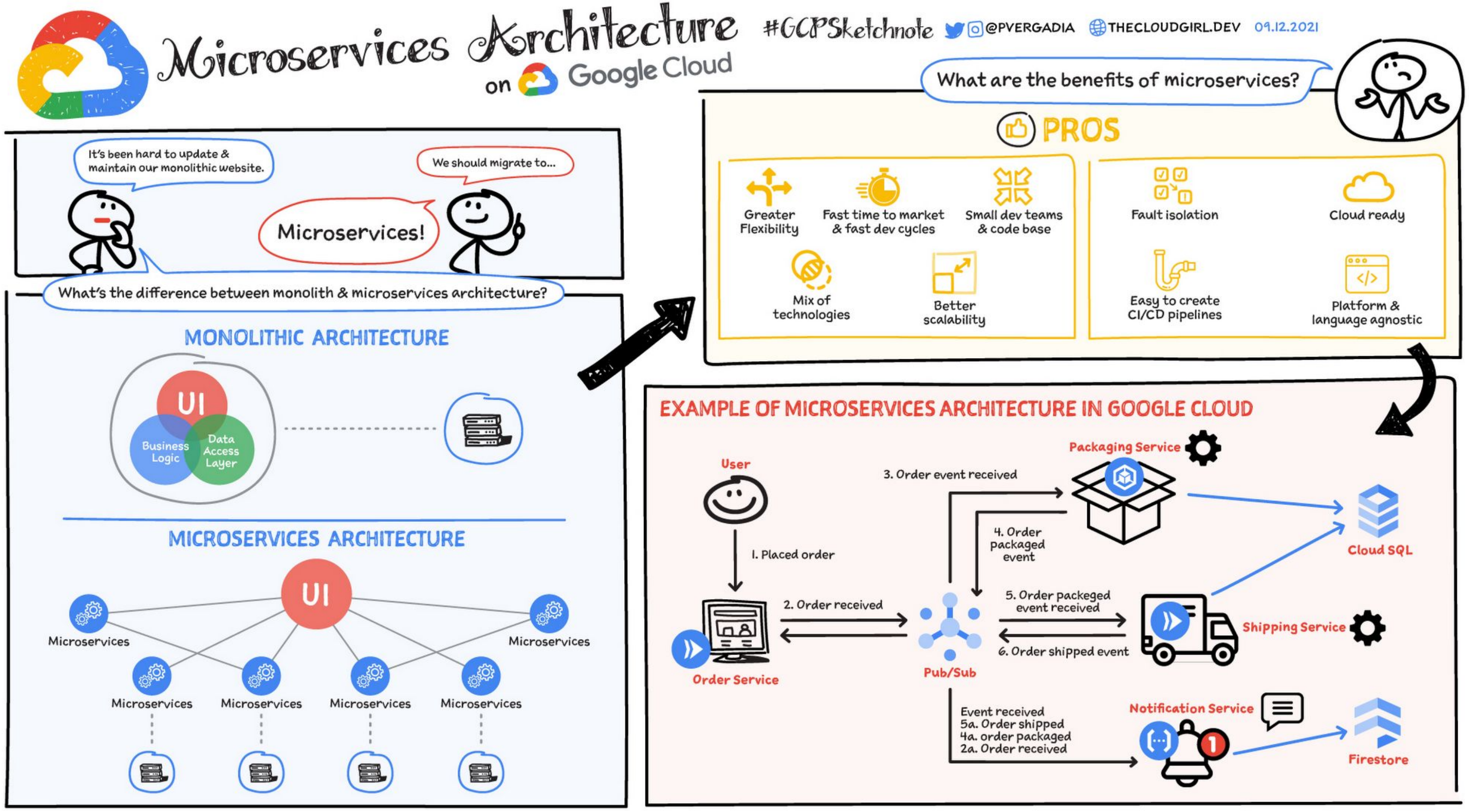
Responses

Code	Description	Links
200	Successful operation Media type application/json Controls Accept header.	*No links*

Example Value | Schema

```
{
```

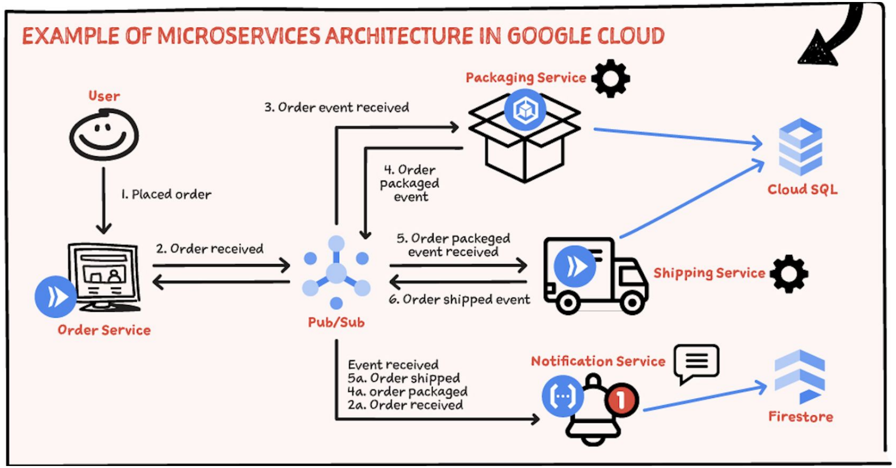
Monolithic Service ➡ Microservices



https://cloud.google.com/blog/topics/developers-practitioners/microservices-architecture-google-cloud

Microservices

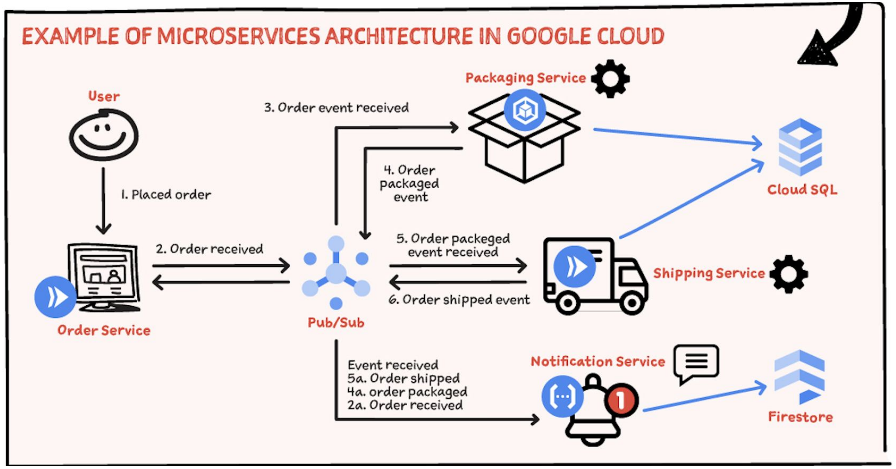
- Definition: “an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols.”
- Features:
  a. Finer granularity: each microservice is small and modular
  b. Improved Flexibility & Scalability: easier migration & load balancing
  c. Loosely coupled:
     - Design phase: refactoring a microservice does not heavily the others.
     - Deployment phase: self-contained microservices can be deployed concurrently.
     - Runtime phase: an unavailable microservice does not severely affect the others.



https://en.wikipedia.org/wiki/Microservices
https://cloud.google.com/blog/topics/developers-practitioners/microservices-architecture-google-cloud

Microservices

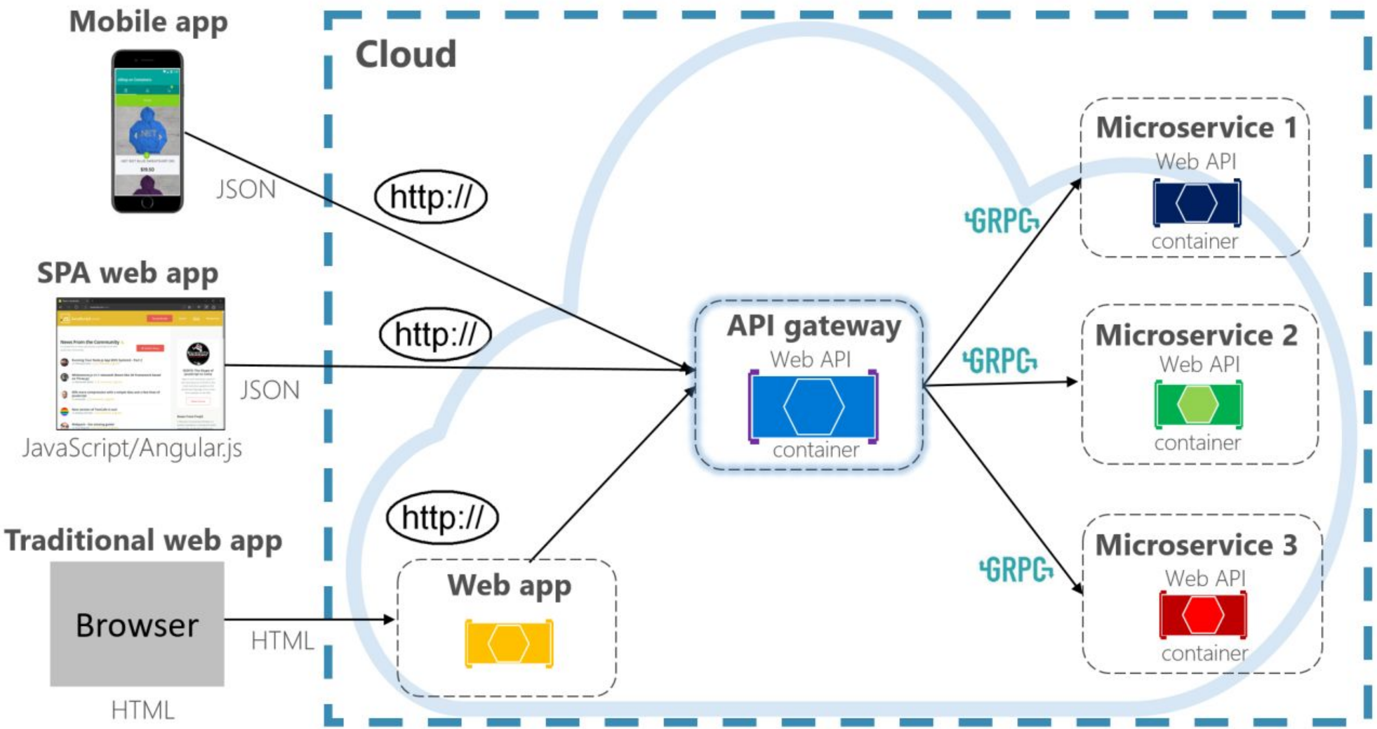
- Definition: “an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols.”
- Features:
  a. Finer granularity: each microservice is small and modular
  b. Improved Flexibility & Scalability: easier migration & load balancing
  c. Loosely coupled
  d. Cross-platform & Cross-Language: each microservice can be written with different programming languages and deployed to different OS platforms.
  e. Cloud Ready: easier packaging into containers, more efficient resource utilization



- **gRPC** is currently a popular architecture for microservices!

https://en.wikipedia.org/wiki/Microservices
https://cloud.google.com/blog/topics/developers-practitioners/microservices-architecture-google-cloud

Microservices with gRPC



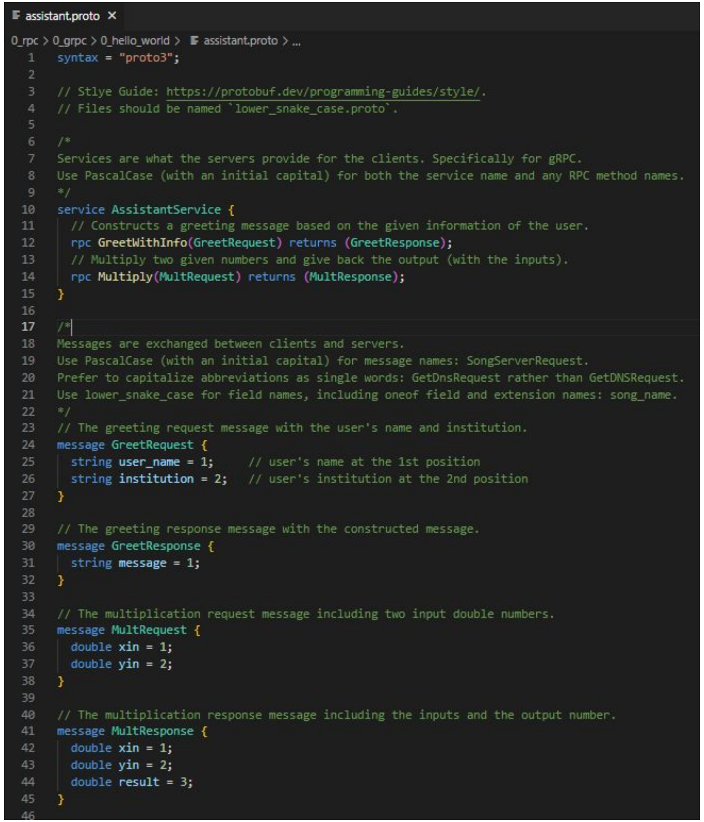
https://www.linkedin.com/pulse/why-rest-when-you-can-grpc-ramit-sharma/

TASK: gRPC - Hello World

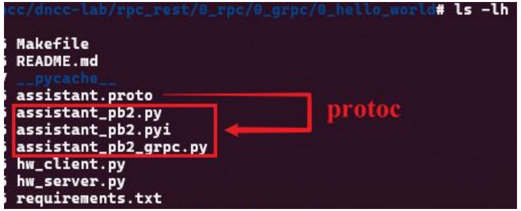
Implement a Hello World gRPC Python example.

> Reference codebase: `rpc_grpc_hello_world`

1. Set up Python (Miniconda is recommended).
2. Install Python dependencies into a Conda environment via:
   - `python -m pip install -r requirements.txt`
3. Check the protocol file `assistant.proto`. Use `protoc` to generate some code:
   - `python -m grpc_tools.protoc -I./ --python_out=. --pyi_out=. --grpc_python_out=. assistant.proto`
   - These generated code will be utilized to implement the gRPC client and the gRPC server.



```
syntax = "proto3";

// Stlye Guide: https://protobuf.dev/programming-guides/style/.
// Files should be named `lower_snake_case.proto`.

/*
Services are what the servers provide for the clients. Specifically for gRPC.
Use PascalCase (with an initial capital) for both the service name and any RPC method names.
*/
service AssistantService {
  // Constructs a greeting message based on the given information of the user.
  rpc GreetWithInfo(GreetRequest) returns (GreetResponse);
  // Multiply two given numbers and give back the output (with the inputs).
  rpc Multiply(MultRequest) returns (MultResponse);
}

/*
Messages are exchanged between clients and servers.
Use PascalCase (with an initial capital) for message names: SongServerRequest.
Prefer to capitalize abbreviations as single words: GetDnsRequest rather than GetDNSRequest.
Use lower_snake_case for field names, including oneof field and extension names: song_name.
*/
// The greeting request message with the user's name and institution.
message GreetRequest {
  string user_name = 1;     // user's name at the 1st position
  string institution = 2;   // user's institution at the 2nd position
}

// The greeting response message with the constructed message.
message GreetResponse {
  string message = 1;
}

// The multiplication request message including two input double numbers.
message MultRequest {
  double xin = 1;
  double yin = 2;
}

// The multiplication response message including the inputs and the output number.
message MultResponse {
  double xin = 1;
  double yin = 2;
  double result = 3;
}

```



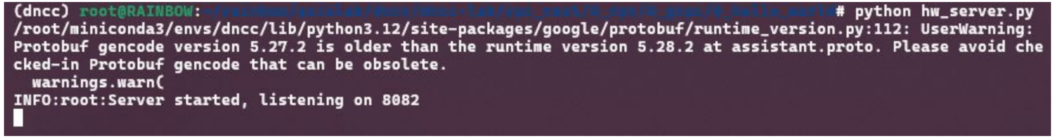
```
cc/dncc-lab/rpc_rest/0_rpc/0_grpc/0_hello_world# ls -lh

Makefile
README.md
__pycache__
assistant.proto
assistant_pb2.py
assistant_pb2.pyi
assistant_pb2_grpc.py
hw_client.py
hw_server.py
requirements.txt
```

TASK: gRPC - Hello World

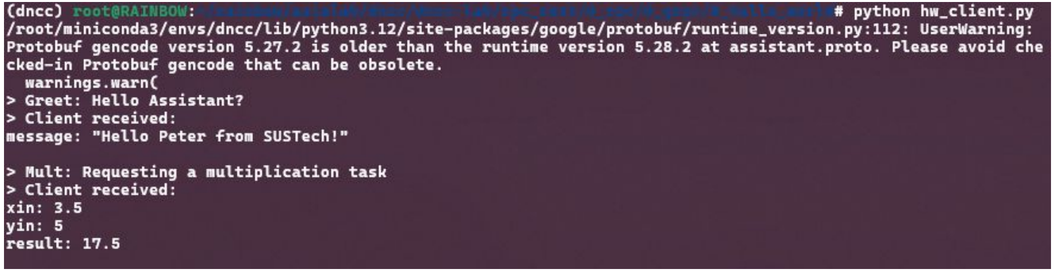
Implement a Hello World gRPC Python example.

> Reference codebase: `rpc_grpc_hello_world`

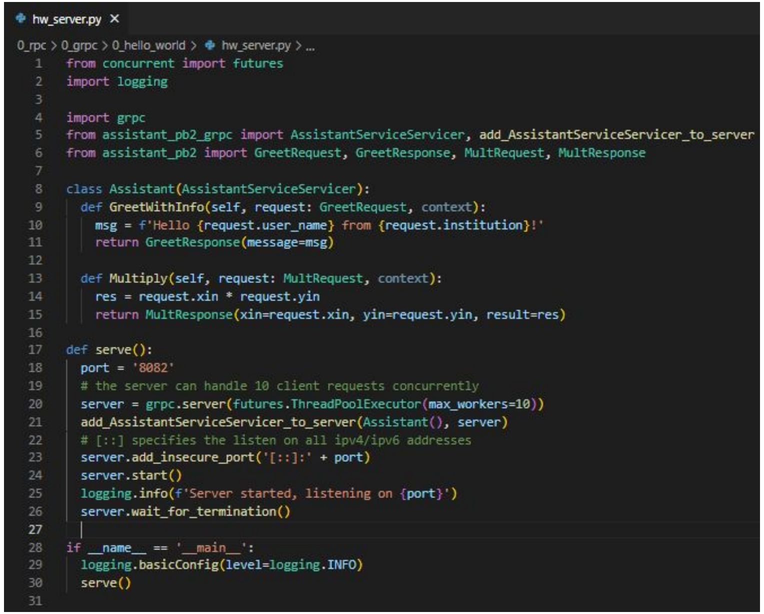
4. Implement the gRPC server in `hw_server.py`. Run the gRPC server via:
   - ○ `python hw_server.py`
5. Implement the gRPC client in `hw_client.py`. In another terminal, run the gRPC client via:
   - ○ `python hw_client.py`



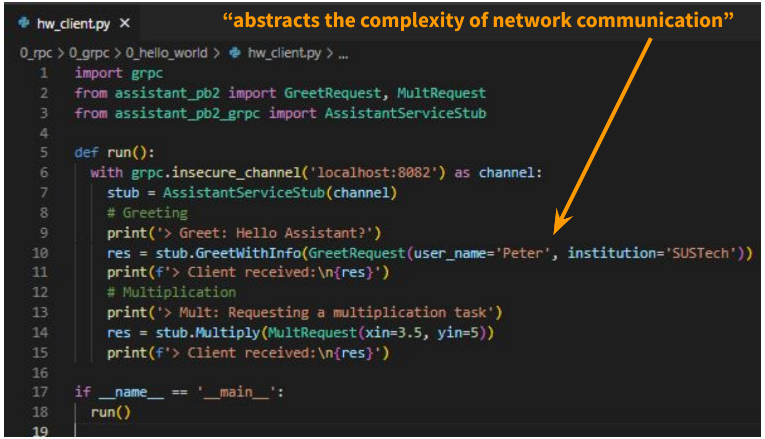
```
(dncc) root@RAINBOW:[redacted]# python hw_server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/runtime_version.py:112: UserWarning:
Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at assistant.proto. Please avoid che
cked-in Protobuf gencode that can be obsolete.
  warnings.warn(
INFO:root:Server started, listening on 8082
█
```



```
(dncc) root@RAINBOW:[redacted]# python hw_client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/runtime_version.py:112: UserWarning:
Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at assistant.proto. Please avoid che
cked-in Protobuf gencode that can be obsolete.
  warnings.warn(
> Greet: Hello Assistant?
> Client received:
message: "Hello Peter from SUSTech!"

> Mult: Requesting a multiplication task
> Client received:
xin: 3.5
yin: 5
result: 17.5
```



```python
from concurrent import futures
import logging

import grpc
from assistant_pb2_grpc import AssistantServiceServicer, add_AssistantServiceServicer_to_server
from assistant_pb2 import GreetRequest, GreetResponse, MultRequest, MultResponse

class Assistant(AssistantServiceServicer):
    def GreetWithInfo(self, request: GreetRequest, context):
        msg = f'Hello {request.user_name} from {request.institution}!'
        return GreetResponse(message=msg)

    def Multiply(self, request: MultRequest, context):
        res = request.xin * request.yin
        return MultResponse(xin=request.xin, yin=request.yin, result=res)

def serve():
    port = '8082'
    # the server can handle 10 client requests concurrently
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    add_AssistantServiceServicer_to_server(Assistant(), server)
    # [::] specifies the listen on all ipv4/ipv6 addresses
    server.add_insecure_port('[::]:' + port)
    server.start()
    logging.info(f'Server started, listening on {port}')
    server.wait_for_termination()

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    serve()

```



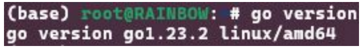
“abstracts the complexity of network communication”

```python
import grpc
from assistant_pb2 import GreetRequest, MultRequest
from assistant_pb2_grpc import AssistantServiceStub

def run():
    with grpc.insecure_channel('localhost:8082') as channel:
        stub = AssistantServiceStub(channel)
        # Greeting
        print('> Greet: Hello Assistant?')
        res = stub.GreetWithInfo(GreetRequest(user_name='Peter', institution='SUSTech'))
        print(f'> Client received:\n{res}')
        # Multiplication
        print('> Mult: Requesting a multiplication task')
        res = stub.Multiply(MultRequest(xin=3.5, yin=5))
        print(f'> Client received:\n{res}')

if __name__ == '__main__':
    run()

```

TASK: gRPC - Quick Start with Go

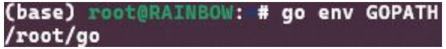
Hello World again, but this time with Go.

We will also use a bit of Go in our future lab sessions to demonstrate the cross-language feature of gRPC. Set up Go and try the official quick start guide.



```
(base) root@RAINBOW:~# go version
go version go1.23.2 linux/amd64
```

1. Install Go according to the official documentation. Verify with `go version`.
2. (optional) Set a proxy for Go to smoothly download the dependencies:
   - `export GOPROXY=https://goproxy.io,direct`
3. Install the Protobuf compiler, `protoc`, version 3 according to the official documentation.
4. Install Go plugins for Protobuf:
   - `go install google.golang.org/protobuf/cmd/protoc-gen-go@latest`
   - `go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest`
   - These plugins will be installed to a `bin/` folder from a preset `GOPATH`. Add this folder to the system path and `protoc` will be able to find these plugins:
     - `export PATH="$PATH:$(go env GOPATH)/bin"`
5. Clone the official gRPC Go GitHub repository and enter the Hello World example folder:
   - `git clone -b v1.67.0 --depth 1 https://github.com/grpc/grpc-go`
   - `cd grpc-go/examples/helloworld`



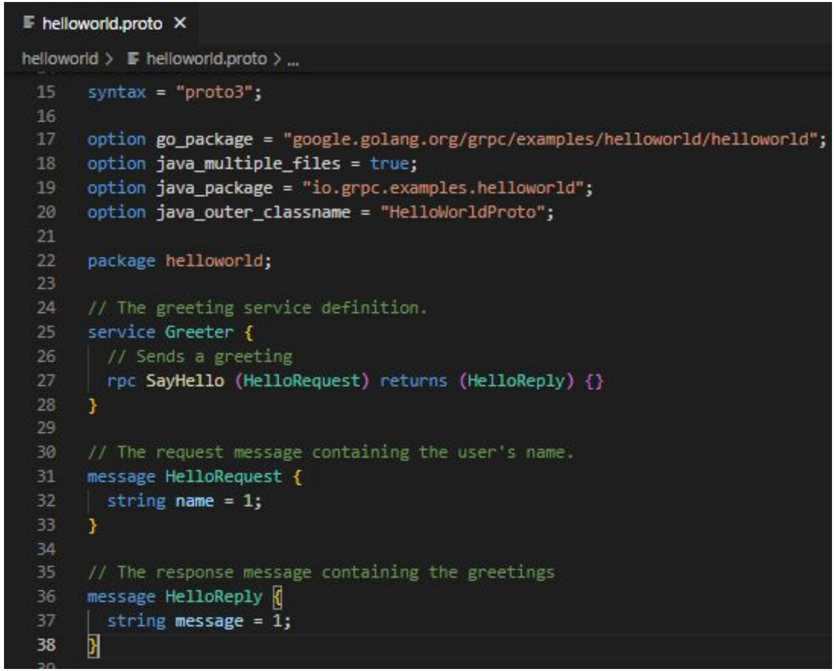
```
(base) root@RAINBOW:~# go env GOPATH
/root/go
```

TASK: gRPC - Quick Start with Go

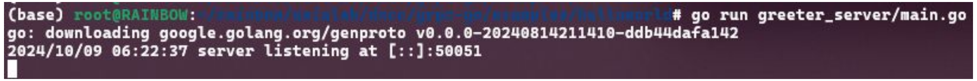
Hello World again, but this time with Go.

We will also use a bit of Go in our future lab sessions to demonstrate the cross-language feature of gRPC. Set up Go and try the official quick start guide.

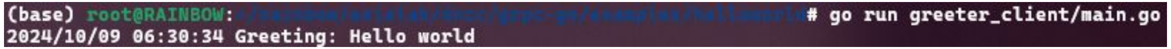
6. Check the proto file: `helloworld/helloworld.proto`. Some Go code has already been generated by protoc within the same directory. To keep it simple, we will not generate them again for this lab session.
7. Run the server code:
    - `go run greeter_server/main.go`
8. In another terminal, run the client code:
    - `go run greeter_client/main.go`



```
syntax = "proto3";

option go_package = "google.golang.org/grpc/examples/helloworld/helloworld";
option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```



```
(base) root@RAINBOW:~/rainbow/ssislab/docs/grpc-go/examples/helloworld# go run greeter_server/main.go
go: downloading google.golang.org/genproto v0.0.0-20240814211410-ddb44dafa142
2024/10/09 06:22:37 server listening at [::]:50051
```



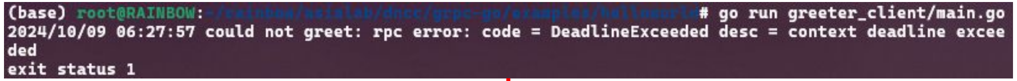
```
(base) root@RAINBOW:~/rainbow/ssislab/docs/grpc-go/examples/helloworld# go run greeter_client/main.go
2024/10/09 06:30:34 Greeting: Hello world
```

TASK: gRPC - Quick Start with Go

Hello World again, but this time with Go.

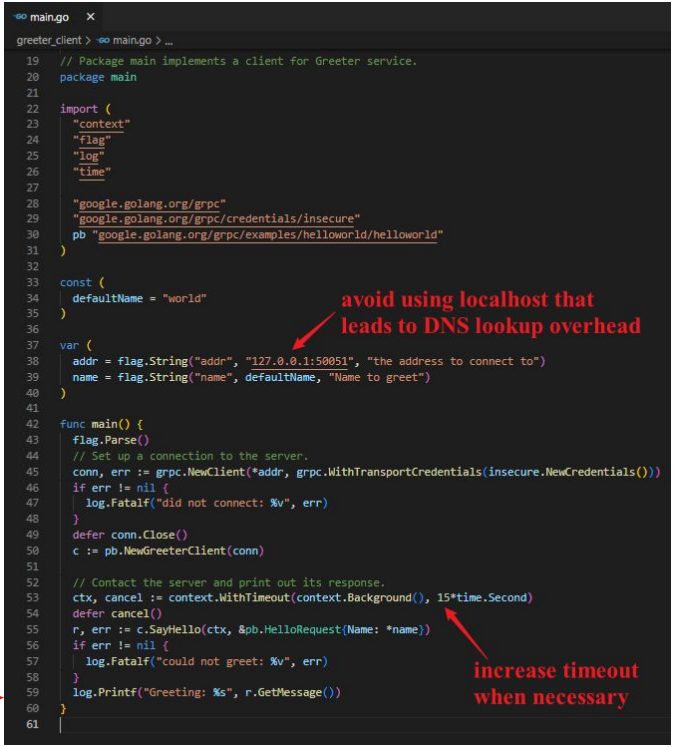
We will also use a bit of Go in our future lab sessions to demonstrate the cross-language feature of gRPC. Set up Go and try the official quick start guide.

6. Check the proto file: `helloworld/helloworld.proto`. Some Go code has already been generated by protoc within the same directory. To keep it simple, we will not generate them again for this lab session.
7. Run the server code:
   - `go run greeter_server/main.go`
8. In another terminal, run the client code:
   - `go run greeter_client/main.go`



```
(base) root@RAINBOW:~/rainbow/asialab/dncc/grpc-go/examples/helloworld# go run greeter_client/main.go
2024/10/09 06:27:57 could not greet: rpc error: code = DeadlineExceeded desc = context deadline excee
ded
exit status 1
```

Q: getting `DeadlineExceeded`?



```go
// Package main implements a client for Greeter service.
package main

import (
	"context"
	"flag"
	"log"
	"time"

	"google.golang.org/grpc"
	"google.golang.org/grpc/credentials/insecure"
	pb "google.golang.org/grpc/examples/helloworld/helloworld"
)

const (
	defaultName = "world"
)

var (
	addr = flag.String("addr", "127.0.0.1:50051", "the address to connect to")
	name = flag.String("name", defaultName, "Name to greet")
)

func main() {
	flag.Parse()
	// Set up a connection to the server.
	conn, err := grpc.NewClient(*addr, grpc.WithTransportCredentials(insecure.NewCredentials()))
	if err != nil {
		log.Fatalf("did not connect: %v", err)
	}
	defer conn.Close()
	c := pb.NewGreeterClient(conn)

	// Contact the server and print out its response.
	ctx, cancel := context.WithTimeout(context.Background(), 15*time.Second)
	defer cancel()
	r, err := c.SayHello(ctx, &pb.HelloRequest{Name: *name})
	if err != nil {
		log.Fatalf("could not greet: %v", err)
	}
	log.Printf("Greeting: %s", r.GetMessage())
}

```

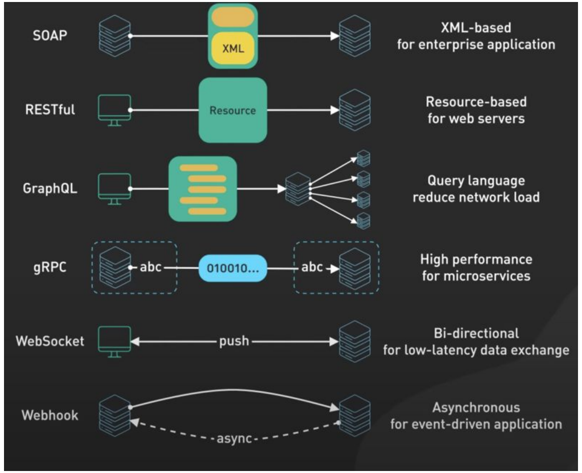
avoid using localhost that leads to DNS lookup overhead

increase timeout when necessary

Summary

- Distributed Architecture Patterns
  a. Web Services
     - Serve clients over the Internet using web standard protocols & data formats.
     - **RESTful API** is currently the most popular architecture for Web Services.
  b. Microservices
     - Break services into loosely coupled, fine-grained services, communicating through lightweight protocols
     - **gRPC** is a popular architecture for microservices.
- 6 Most Popular API Architecture Patterns
  a. SOAP
  b. RESTful API
  c. GraphQL
  d. RPC
  e. WebSocket
  f. Webhook

More details in the following lab sessions…



https://dev.to/kanani_nirav/top-6-most-popular-api-architecture-styles-you-need-to-know-with-pros-cons-and-use-cases-564j