Building Microservices with gRPC
(with Python)

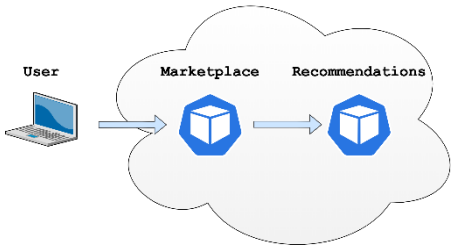
---

1 Introduction

Considering an example scenario of an online Book Store. We will define an API and use Python for the coding of the microservices.

To keep things manageable, you'll define only two microservices:

1. **Marketplace** will be a very minimal web app that displays a list of books to the user.
2. **Recommendations** will be a microservice that provides a list of books in which the user may be interested.
3. Here's a diagram that shows how your user interacts with the microservices:



You can see that the user will interact with the Marketplace microservice via their browser, and the Marketplace microservice will interact with the Recommendations microservice.

A recommendation request will have 3 parameters:

- **User ID:** You could use this to personalize the recommendations. However, for simplicity, all recommendations in this tutorial will be random.
- **Book category:** To make the API a little more interesting, you'll add book categories, such as mystery, self-help, and so on.
- **Max results:** You don't want to return every book in stock, so you'll add a limit to the request.

The response will be a list of books. Each book will have the following data:

- **Book ID:** A unique numeric ID for the book.
- **Book title:** The title you can display to the user.

2 API

The API is defined using Protocol Buffers:

```
syntax = "proto3";

enum BookCategory {
    MYSTERY = 0;
    SCIENCE_FICTION = 1;
    SELF_HELP = 2;
}

message RecommendationRequest {
    int32 user_id = 1;
    BookCategory category = 2;
    int32 max_results = 3;
}

message BookRecommendation {
    int32 id = 1;
    string title = 2;
}

message RecommendationResponse {
    repeated BookRecommendation
recommendations = 1;
}

service Recommendations {
    rpc Recommend (RecommendationRequest)
returns (RecommendationResponse);
}
```

It declares a service that has a `RecommendationRequest` as input and as output a `RecommendationResponse`.

The repository is in:
https://github.com/realpython/materials/tree/master/python-microservices-with-grpc

3 Requirements and dependencies

To create virtual environments for Python you need to have the package python3-venv

```
$ sudo apt-get install python3-venv python3.8-venv
```

The gRPC Python tools package is required. It can be installed with:

```
$ pip3 install --upgrade pip
$ pip3 install --upgrade testresources
$ pip3 install --upgrade setuptools
$ pip3 install --no-cache-dir  --force-reinstall -Iv grpcio
$ pip3 install grpcio-tools
$ pip3 install grpc-interceptor
$ pip3 install --upgrade protobuf
```

To build Wheel archives for your requirements and dependencies install the following:

```
$ pip3 install wheel
```

The `protobufs/` directory will contain a file called `recommendations.proto`. The content of this file is the protobuf code above. This description will be used to generate Python code.
Check the required versions of each tool in the file `recommendations/requirements.txt`
To run the code locally, you'll need to install the dependencies into a virtual environment.

On Linux, use the following commands to create a virtual environment and install the dependencies:

```
$ python3 -m venv venv
$ source venv/bin/activate  # Linux/macOS only

$ cd recommendations
(venv) $ python3 -m pip install -r requirements.txt
```

Now, to generate Python code from the protobufs, run the following:

```
$ python3 -m grpc_tools.protoc -I ../protobufs --python_out=. \
         --grpc_python_out=. ../protobufs/recommendations.proto
```

This generates several Python files from the .proto file. Here's a breakdown:

- **python -m grpc_tools.protoc** runs the protobuf compiler, which will generate Python code from the protobuf code.
- **-I ../protobufs** tells the compiler where to find files that your protobuf code imports. You don't actually use the import feature, but the -I flag is required nonetheless.
- **--python_out=. --grpc_python_out=.** tells the compiler where to output the Python files. As you'll see shortly, it will generate two files, and you could put each in a separate directory with these options if you wanted to.
- **../protobufs/recommendations.proto** is the path to the protobuf file, which will be used to generate the Python code.

If you look at what's generated, you'll see two files:

```
$ ls
recommendations_pb2.py recommendations_pb2_grpc.py
```

These files include Python types and functions to interact with your API. The compiler will generate client code to call an RPC and server code to implement the RPC. You'll look at the client side first.

4 The RPC server

Study the server file `recommendations.py`

Check that a `books_by_category` dictionary is created, in which the keys are book categories and the values are lists of books in that category. In a real Recommendations microservice, the books would be stored in a database.

After this dictionary you'll find the class `RecommendationService` that implements the microservice functions.

The method `Recommend` implements the RPC. This must have the same name as the RPC you define in your protobuf file. It also takes a `RecommendationRequest` and returns a `RecommendationResponse` just like in the protobuf definition. It also takes a `context` parameter. The context allows you to set the status code for the response. If an unexpected category is sent `abort()` ends the request and set the status code to `NOT_FOUND` if you get an unexpected category. Since gRPC is built on top of HTTP/2, the status code is similar to the standard HTTP status code. Setting it allows the client to take different actions based on the code it receives. It also allows middleware, like monitoring systems, to log how many requests have errors. If the category is valid, the program just recommends some random books from the category. You make sure to limit the number of recommendations to `max_results`. You use `min()` to ensure you don't ask for more books than there are, or else `random.sample` will error out. The method returns a `RecommendationResponse` object with your list of book recommendations.

The `RecommendationService` class defines your microservice implementation, but you still need to run it. That's what `serve()` does. It starts a network server and uses the microservice class to handle requests. It creates a thread pool with, up to, 10 threads to serve requests. After this the program adds the class to the server and starts the service.

Check that the server setups authentication before starting the service. For now, we'll skip on this requirement and change the connection to unencrypted. To do this comment the following lines:

```
"""
    with open("server.key", "rb") as fp:
        server_key = fp.read()
    with open("server.pem", "rb") as fp:
        server_cert = fp.read()
    with open("ca.pem", "rb") as fp:
        ca_cert = fp.read()

    creds = grpc.ssl_server_credentials(
        [(server_key, server_cert)],
        root_certificates=ca_cert,
        require_client_auth=True,
    )
"""
```

Also change the following line:

```
    server.add_secure_port("[::]:443", creds)
```

To:

```
    server.add_insecure_port("[::]:50051")
```

Start the server in the terminal with:

```
$ python recommendations.py
```

5 The RPC client

The `recommendations_pb2.py` file that was generated for you contains the type definitions. The `recommendations_pb2_grpc.py` file contains the framework for a client and a server.

The server can already be tested simply by starting Python REPL at the same `recommendations` folder with:

```
$ python
>>> from recommendations_pb2 import BookCategory, RecommendationRequest
>>> import grpc
>>> from recommendations_pb2_grpc import RecommendationsStub
>>> channel = grpc.insecure_channel("localhost:50051")
>>> client = RecommendationsStub(channel)
>>>            request            =            RecommendationRequest(user_id=1,
category=BookCategory.SCIENCE_FICTION, max_results=3)
>>> client.Recommend(request)
```

You should see 3 recommendations as result of the last command.

Now that you have the server implemented, you can implement the Marketplace microservice and have it call the Recommendations microservice. Check the contents of `marketplace/`
The Marketplace microservice is a Flask app that displays a webpage to the user. It calls the Recommendations microservice to get book recommendations to display on the page.
Take care of the requirements with:

```
$ python3 -m pip install -r marketplace/requirements.txt
```

Now that you've installed the dependencies, you need to generate code for your protobufs in the `marketplace/` directory as well. To do that, run the following in a console:

```
$ python3 -m grpc_tools.protoc -I ../protobufs --python_out=. \
         --grpc_python_out=. ../protobufs/recommendations.proto
```

Check that the client setups authentication before starting the service. For now, we'll skip on this requirement and change the connection to unencrypted. To do this comment the following lines:

```
"""
with open("client.key", "rb") as fp:
    client_key = fp.read()
with open("client.pem", "rb") as fp:
    client_cert = fp.read()
with open("ca.pem", "rb") as fp:
    ca_cert = fp.read()
creds = grpc.ssl_channel_credentials(ca_cert, client_key, client_cert)
"""
```

Also change the following line:

```
recommendations_channel = grpc.secure_channel(f"{recommendations_host}:443",
creds)
```

To:

```
recommendations_channel = grpc.insecure_channel(f"{recommendations_host}:50051")
```

Run the client with:

```
$ FLASK_APP=marketplace.py flask run
```

Open the browser and type the following URL:

```
http://localhost:5000
```

6 Docker containers

6.1 Play with Docker

If you have administrative rights for your Linux machine you can install Docker and continue to the next section.

If you just want to use Docker for up to a few hours or you don't have administrative rights to the local Linux (like in the lab), but you have an internet connection, you can use an **online Docker playground** like: https://labs.play-with-docker.com/

You need a free DockerHub account (https://hub.docker.com/) to login to the playground.

If after starting the environment, the screen goes black just refresh the web page.

Now you have a temporary "VM" session with Alpine Linux that last up to 4h and is mapped to a public IP. This instance already has Docker installed. You can add more instances, each with its own running shell. You can also create a "cluster" with a master and a chosen number of slaves by using the wrench tool.
You can use a basic GUI editor that launches as a separate browser window.

You can **transfer files** between your machine and that instance by using a service like `https://transfer.sh/`

6.2 Containerizing the services

You'll create two Docker **images**, one for the Marketplace microservice and one for the Recommendations microservice. An image is basically a file system plus some metadata. In essence, each of your microservices will have a mini Linux environment to itself. It can write files without affecting the actual file system and open ports without conflicting with other processes.

The images are created from each `Dockerfile`. Check their contents.

To proceed you have to check that Docker is installed:

```
$ docker --version
```

Comment the following lines with a hash '#' from `recommendations/Dockerfile`:

```
COPY ca.pem /service/recommendations/
RUN openssl req -nodes -newkey rsa:4096 -subj /CN=recommendations \
                -keyout server.key -out server.csr
RUN --mount=type=secret,id=ca.key \
    openssl x509 -req -in server.csr -CA ca.pem -CAkey /run/secrets/ca.key \
                 -set_serial 1 -out server.pem
```

Now you can generate a Docker image from your `Dockerfile`. Run the following command from the directory containing all your code—not inside the `recommendations/` directory, but one level up from that:

```
$ docker build . -f recommendations/Dockerfile -t recommendations
```

Note: *To run some Docker commands, it may be necessary to append `sudo` at the beginning*.

This will build the Docker image for the Recommendations microservice. You should see some output as Docker builds the image. Now you can run it:

```
$ docker run -p 127.0.0.1:50051:50051/tcp recommendations
```

You won't see any output, but your Recommendations microservice is now running inside a Docker container. When you run an image, you get a container. You could run the image multiple times to get multiple containers, but there's still only one image.

Comment the following lines with a hash '#' from `marketplace/Dockerfile`:

```
COPY ca.pem /service/marketplace/
RUN openssl req -nodes -newkey rsa:4096 -subj /CN=marketplace \
                -keyout client.key -out client.csr
RUN --mount=type=secret,id=ca.key \
    openssl x509 -req -in client.csr -CA ca.pem -CAkey /run/secrets/ca.key \
                 -set_serial 1 -out client.pem
```

Next, you'll build your Marketplace image from the `marketplace/Dockerfile`

```
$ docker build . -f marketplace/Dockerfile -t marketplace
```

That creates the Marketplace image. You can now run it in a container with this command:

```
$ docker run -p 127.0.0.1:5000:5000/tcp marketplace
```

You won't see any output, but your Marketplace microservice is now running.

6.3 Networking

Unfortunately, even though both your Recommendations and Marketplace containers are running, if you now go to `http://localhost:5000` in your browser, you'll get an error. You can connect to your Marketplace microservice, but it can't connect to the Recommendations microservice anymore. The containers are isolated.

With Docker we can create a virtual network and add both your containers to it. We can also give them DNS names so they can find each other.

Below, you'll create a network called `microservices` and run the Recommendations microservice on it. You'll also give it the DNS name `recommendations`. First, stop the currently running containers with `Ctrl+C`. Then run the following:

```
$ docker network create microservices
$ docker run -p 127.0.0.1:50051:50051/tcp --network microservices \
             --name recommendations recommendations
```

The `docker network create` command creates the network. You only need to do this once and then you can connect multiple containers to it. You then add `--network microservices` to the `docker run` command to start the container on this network. The `--name recommendations` option gives it the DNS name `recommendations`.

Before you restart the marketplace container, you need to change the code. This is because you hard-coded `localhost:50051` in this line from `marketplace.py`:

```
recommendations_channel = grpc.insecure_channel("localhost:50051")
```

Now you want to connect to `recommendations:50051` instead. But rather than hardcode it again, you can load it from an environment variable. Replace the line above with the following two:

```
recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
recommendations_channel = grpc.insecure_channel(
    f"{recommendations_host}:50051"
)
```

This loads the hostname of the Recommendations microservice in the environment variable `RECOMMENDATIONS_HOST`. If it's not set, then you can default it to `localhost`. This allows you to run the same code both directly on your machine or inside a container.

You'll need to rebuild the marketplace image since you changed the code. Then try running it on your network:

```
$ docker build . -f marketplace/Dockerfile -t marketplace
$ docker run -p 127.0.0.1:5000:5000/tcp --network microservices \
             -e RECOMMENDATIONS_HOST=recommendations marketplace
```

This is similar to how you ran it before, but with two differences:

1. You added the `--network microservices` option to run it on the same network as your Recommendations microservice. You didn't add a `--name` option because, unlike the Recommendations microservice, nothing needs to look up the IP address of the Marketplace microservice. The port forwarding provided by `-p 127.0.0.1:5000:5000/tcp` is enough, and it doesn't need a DNS name.
2. You added `-e RECOMMENDATIONS_HOST=recommendations`, which sets the environment variable inside the container. This is how you pass the hostname of the Recommendations microservice to your code.

At this point, you can try `localhost:5000` in your browser once again, and it should load correctly.

6.4 Docker compose

Rather than running a bunch of commands to build images, create networks, and run containers, you can declare your microservices in a YAML file and start all containers with:

```
$ docker-compose up
```

Once this is running, you should again be able to open `localhost:5000` in your browser, and all should work perfectly.
Note that you don't need to expose `50051` in the `recommendations` container when it's in the same network as the Marketplace microservice, so you can drop that part.

If you'd like to stop `docker-compose` to make some edits before moving up, press `Ctrl`+`C`.