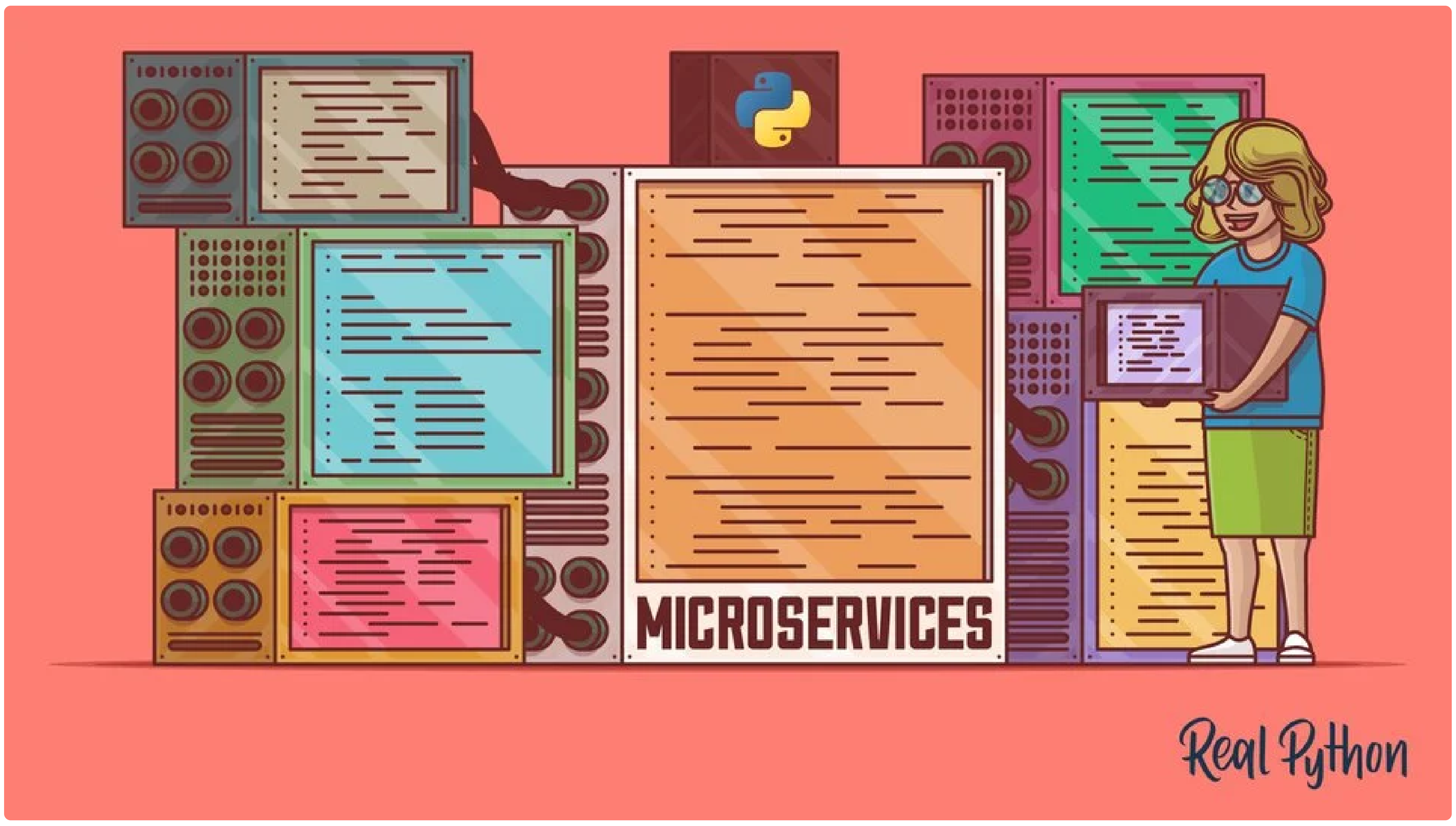


Python Microservices With gRPC

Table of Contents

Microservices are a way to organize complex software systems. Instead of putting all your code into one app, you break your app into microservices that are deployed independently and communicate with each other. This tutorial teaches you how to get up and running with Python microservices using gRPC, one of the most popular frameworks.

Implementing a microservices framework well is important. When you're building a framework to support critical applications, you must ensure it's robust and developer-friendly. In this tutorial, you'll learn how to do just that. This knowledge will make you more valuable to growing companies.

In order to benefit most from this tutorial, you should understand the basics of Python and web apps. If you'd like a refresher on those, read through the links provided first.

By the end of this tutorial, you'll be able to:

- Implement **microservices** in Python that communicate with one another over gRPC
- Implement **middleware** to monitor microservices
- **Unit test** and **integration test** your microservices and middleware
- Deploy microservices to a Python production environment with **Kubernetes**

You can download all the source code used in this tutorial by clicking the link below:

Get the Source Code: Click here to get the source code you'll use to learn about creating Python microservices with gRPC in this tutorial.

Why Microservices?

Imagine you work at Online Books For You, a popular e-commerce site that sells books online. The company has several hundred developers. Each developer is writing code for some product or back-end feature, such as managing the user's cart, generating recommendations, handling payment transactions, or dealing with warehouse inventory.

Now ask yourself, would you want all that code in one giant application? How hard would that be to understand? How long would it take to test? How would you keep the code and database schemas sane? It definitely would be hard, especially as the business tries to move quickly.

Wouldn't you rather have code corresponding to modular product features be, well, modular? A cart microservice to manage carts. An inventory microservice to manage inventory.

In the sections below, you'll dig a bit deeper into some reasons to separate Python code into microservices.

Modularity

Code changes often take the path of least resistance. Your beloved Online Books For You CEO wants to add a new buy-two-books-get-one-free feature. You're part of the team that's been asked to launch it as quickly as possible. Take a look at what happens when all your code is in a single application.

Being the smartest engineer on your team, you mention that you can add some code to the cart logic to check if there are more than two books in the cart. If so, you can simply subtract the cost of the cheapest book from the cart total. No sweat—you make a pull request.

Then your product manager says you need to track this campaign's impact on books sales. This is pretty straightforward, too. Since the logic that implements the buy-two-get-one feature is in the cart code, you'll add a line in the checkout flow that updates a new column on the transactions database to indicate the sale was part of the promotion: `buy_two_get_one_free_promo = true`. Done.

Next, your product manager reminds you that the deal is valid for only one use per customer. You need to add some logic to check whether any previous transactions had that `buy_two_get_one_free_promo` flag set. Oh, and you need to hide the promotion banner on the home page, so you add that check, too. Oh, and you need to send emails to people who haven't used the promo. Add that, too.

Several years later, the transactions database has grown too large and needs to be replaced with a new shared database. All those references need to be changed. Unfortunately, the database is referenced all over the codebase at this point. You consider that it was actually a little *too* easy to add all those references.

That's why having all your code in a single application can be dangerous in the long run. Sometimes it's good to have boundaries.

The transactions database should be accessible only to a transactions microservice. Then, if you need to scale it, it's not so bad. Other parts of the code can interact with transactions through an abstracted API that hides the implementation details. You *could* do this in a single application—it's just less likely that you would. Code changes often take the path of least resistance.

Flexibility

Splitting your Python code into microservices gives you more flexibility. For one thing, you can write your microservices in different languages. Oftentimes, a company's first web app will be written in Ruby or PHP. That doesn't mean everything else has to be, too!

You can also scale each microservice independently. In this tutorial, you'll be using a web app and a Recommendations microservice as a running example.

Your web app will likely be I/O bound, fetching data from a database and maybe loading templates or other files from disk. A Recommendations microservice may be doing a lot of number crunching, making it CPU bound. It makes sense to run these two Python microservices on different hardware.

Robustness

If all your code is in one application, then you have to deploy it all at once. This is a big risk! It means a change to one small part of the code can take down the entire site.

Ownership

When a single codebase is shared by a large number of people, there's often no clear vision for the architecture of the code. This is especially true at large companies where employees come and go. There may be people who have a vision for how the code should look, but it's hard to enforce when anyone can modify it and everyone is moving quickly.

One benefit of microservices is that teams can have clear ownership of their code. This makes it more likely that there will be a clear vision for the code and that the code will remain clean and organized. It also makes it clear who's responsible for adding features to the code or making changes when something goes wrong.

How Small Is "Micro"?

How small microservices should be is one of those topics that can spark a heated debate among engineers. Here’s my two cents: *micro* is a misnomer. We should just say *services*. However, in this tutorial you’ll see *microservices* used for consistency.

Making microservices too small can lead to problems. First of all, it actually defeats the purpose of making code modular. The code in a microservice should make sense together, just like the data and methods in a class make sense together.

To use classes as an analogy, consider `file` objects in Python. The `file` object has all the methods you need. You can `.read()` and `.write()` to it, or you can `.readlines()` if you want. You shouldn’t need a `FileReader` and a `FileWriter` class. Maybe you’re familiar with languages that do this, and maybe you always thought it was a bit cumbersome and confusing.

Microservices are the same. The scope of the code should feel right. Not too large, not too small.

Second, microservices are harder to test than monolithic code. If a developer wants to test a feature that spans across many microservices, then they need to get those all up and running in their development environment. That adds friction. It’s not so bad with a few microservices, but if it’s dozens, then it’ll be a significant issue.

Getting microservice size right is an art. One thing to watch for is that each team should own a reasonable number of microservices. If your team has five people but twenty microservices, then this is a red flag. On the other hand, if your team works on just one microservice that’s also shared by five other teams, then this could be a problem, too.

Don’t make microservices as small as possible just for the sake of it. Some microservices may be large. But watch out for when a single microservice is doing two or more totally unrelated things. This usually happens because adding unrelated functionality to an existing microservice is the path of least resistance, not because it belongs there.

Here are some ways you could break up your hypothetical online bookstore into microservices:

- **Marketplace** serves the logic for the user to navigate around the site.
- **Cart** keeps track of what the user has put in their cart and the checkout flow.
- **Transactions** handles payment processing and sending receipts.
- **Inventory** provides data about which books are in stock.
- **User Account** manages user signup and account details, such as changing their password.
- **Reviews** stores book ratings and reviews entered by users.

These are just a few examples, not an exhaustive list. However, you can see how each of these would probably be owned by its own team, and the logic of each is relatively independent. Also, if the Reviews microservice was deployed with a bug that caused it to crash, then the user could still use the site and make purchases despite reviews failing to load.

The Microservice-Monolith Trade-Off

Microservices aren’t always better than monoliths that keep all your code in one app. Generally, and especially at the beginning of a software development lifecycle, monoliths will let you move faster. They make it less complicated to share code and add functionality, and having to deploy only one service allows you to get your app to users quickly.

The trade-off is that, as complexity grows, all these things can gradually make the monolith harder to develop, slower to deploy, and more fragile. Implementing a monolith will likely save you time and effort up front, but it may come back later to haunt you.

Implementing microservices in Python will likely cost you time and effort in the short term, but if done well, it can set you up to scale better in the long run. Of course, implementing microservices too soon could slow you down when speed is most valuable.

The typical Silicon Valley startup cycle is to begin with a monolith to enable quick iteration as the business finds a product fit with customers. After the company has a successful product and hires more engineers, it’s time to start thinking about microservices. Don’t implement them too soon, but don’t wait too long.

For more on the microservice-monolith trade-off, watch Sam Newman and Martin Fowler's excellent discussion, When To Use Microservices (And When Not To!).

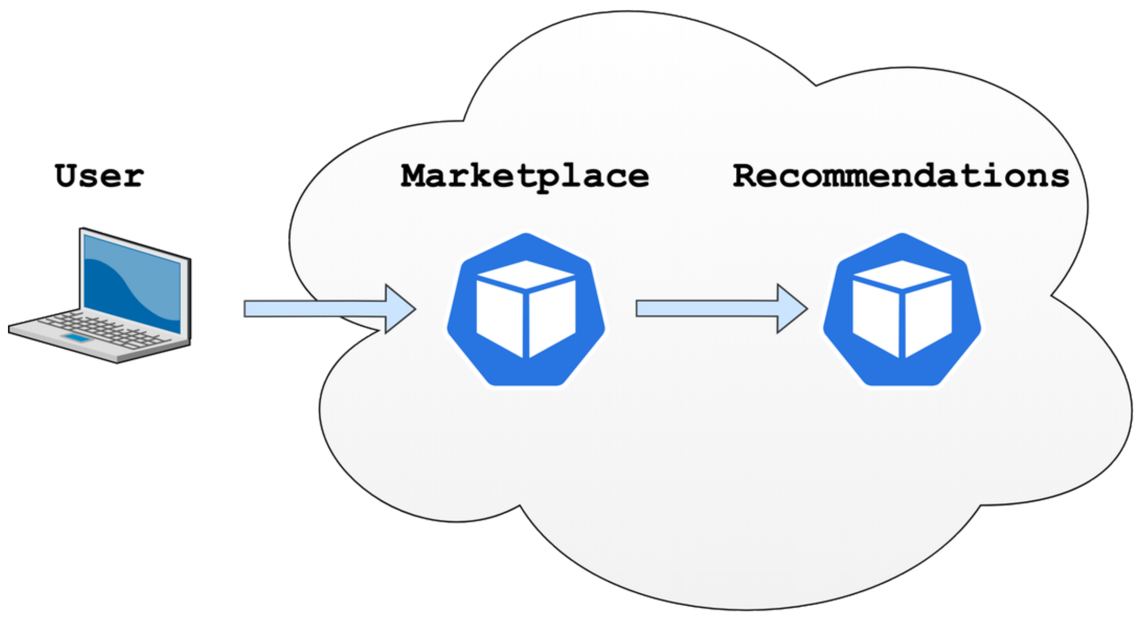
Example Microservices

In this section, you'll define some microservices for your Online Books For You website. You'll define an API for them and write the Python code that implements them as microservices as you go through this tutorial.

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.

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.

Think for a moment about the Recommendations API. You want the recommendations request to have a few features:

- **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.

A real website would have more data, but you'll keep the number of features limited for the sake of this example.

Now you can define this API more formally, in the syntax of **protocol buffers**:

Protocol Buffer

```
 1  syntax = "proto3";
 2
 3  enum BookCategory {
 4      MYSTERY = 0;
 5      SCIENCE_FICTION = 1;
 6      SELF_HELP = 2;
 7  }
 8
 9  message RecommendationRequest {
10      int32 user_id = 1;
11      BookCategory category = 2;
12      int32 max_results = 3;
13  }
14
15  message BookRecommendation {
16      int32 id = 1;
17      string title = 2;
18  }
19
20  message RecommendationResponse {
21      repeated BookRecommendation recommendations = 1;
22  }
23
24  service Recommendations {
25      rpc Recommend (RecommendationRequest) returns (RecommendationResponse);
26  }
```

This protocol buffer file declares your API. Protocol buffers were developed at Google and provide a way to formally specify an API. This might look a bit cryptic at first, so here's a line-by-line breakdown:

- **Line 1** specifies that the file uses the `proto3` syntax instead of the older `proto2` version.
- **Lines 3 to 7** define your book categories, and each category is also assigned a numeric ID.
- **Lines 9 to 13** define your API request. A `message` contains fields, each of a specific type. You're using `int32`, which is a 32-bit integer, for the `user_ID` and `max_results` fields. You're also using the `BookCategory` enum you defined above as the `category` type. In addition to each field having a name, it's also assigned a numeric field ID. You can ignore this for now.
- **Lines 15 to 18** define a new type that you can use for a book recommendation. It has a 32-bit integer ID and a string-based title.
- **Lines 20 to 22** define your Recommendations microservice response. Note the `repeated` keyword, which indicates that the response actually has a list of `BookRecommendation` objects.
- **Lines 24 to 26** define the **method** of the API. You can think of this like a function or a method on a class. It takes a `RecommendationRequest` and returns a `RecommendationResponse`.

`rpc` stands for **remote procedure call**. As you'll see shortly, you can call an RPC just like a normal function in Python. But the implementation of the RPC executes on another server, which is what makes it a *remote* procedure call.

Why RPC and Protocol Buffers?

Okay, so why should you use this formal syntax to define your API? If you want to make a request from one microservice to another, can't you just make an HTTP request and get a JSON response? Well, you can do that, but there are benefits to using protocol buffers.

Documentation

The first benefit of using protocol buffers is that they give your API a well-defined and self-documented schema. If you use JSON, then you must document the fields it contains and their types. As with any documentation, you run the risk of the documentation being inaccurate or incomplete or going out of date.

When you write your API in the protocol buffer language, you can generate Python code from it. Your code will never be out of sync with your documentation. Documentation is good, but self-documented code is better.

Validation

The second benefit is that, when you generate Python code from protocol buffers, you get some basic validation for free. For instance, the generated code won't accept fields of the wrong type. The generated code also has all the RPC boilerplate built in.

If you use HTTP and JSON for your API, then you need to write a little code that constructs the request, sends it, waits for the response, checks the status code, and parses and validates the response. With protocol buffers, you can generate code that looks just like a regular function call but does a network request under the hood.

You can get these same benefits using HTTP and JSON frameworks such as Swagger and RAML. For an example of Swagger in action, check out Python REST APIs With Flask, Connexion, and SQLAlchemy.

So are there reasons to use gRPC rather than one of those alternatives? The answer is still yes.

Performance

The gRPC framework is generally more efficient than using typical HTTP requests. gRPC is built on top of HTTP/2, which can make multiple requests in parallel on a long-lived connection in a thread-safe way. Connection setup is relatively slow, so doing it once and sharing the connection across multiple requests saves time. gRPC messages are also binary and smaller than JSON. Further, HTTP/2 has built-in header compression.

gRPC has built-in support for streaming requests and responses. It will manage network issues more gracefully than a basic HTTP connection, reconnecting automatically even after long disconnects. It also has **interceptors**, which you'll learn about later in this tutorial. You can even implement plugins to the generated code, which people have done to output Python type hints. Basically, you get a lot of great infrastructure for free!

Developer-Friendliness

Probably the most interesting reason why many people prefer gRPC over REST is that you can define your API in terms of functions, not HTTP verbs and resources. As an engineer, you're used to thinking in terms of function calls, and this is exactly how gRPC APIs look.

Mapping functionality onto a REST API is often awkward. You have to decide what your resources are, how to construct paths, and which verbs to use. Often there are multiple choices, such as how to nest resources or whether to use POST or some other verb. REST vs gRPC can turn into a debate over preferences. One is not always better than the other, so use what suits your use case best.

Strictly speaking, *protocol buffers* refers to the serialization format of data sent between two microservices. So protocol buffers are akin to JSON or XML in that they're ways to format data. Unlike JSON, protocol buffers have a strict schema and are more compact when sent over the network.

On the other hand, the RPC infrastructure is actually called **gRPC**, or Google RPC. This is more akin to HTTP. In fact, as mentioned above, gRPC is built on top of HTTP/2.

Example Implementation

After all this talk about protocol buffers, it's time to see what they can do. The term *protocol buffers* is a mouthful, so you'll see the common shorthand **protobufs** used in this tutorial going forward.

As mentioned a few times, you can generate Python code from protobufs. The tool is installed as part of the `grpcio-tools` package.

First, define your initial directory structure:

```
.
├── protobufs/
│   └── recommendations.proto
|
└── recommendations/
```

The `protobufs/` directory will contain a file called `recommendations.proto`. The content of this file is the protobuf code above. For convenience, you can view the code by expanding the collapsible section below:

Complete `recommendations.proto` Code Show/Hide

You're going to generate Python code to interact with this inside the `recommendations/` directory. First, you must install `grpcio-tools`. Create the file `recommendations/requirements.txt` and add the following:

Text

```
grpcio-tools ~= 1.30
```

To run the code locally, you'll need to install the dependencies into a virtual environment. The following commands will install the dependencies on Windows:

Windows Console

```
C:\ python -m venv venv
C:\ venv\Scripts\activate.bat
(venv) C:\ python -m pip install -r requirements.txt
```

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

Shell

```
$ python3 -m venv venv
$ source venv/bin/activate  # Linux/macOS only
(venv) $ python -m pip install -r requirements.txt
```

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

Shell

```
$ cd recommendations
$ python -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:

Shell

```
$ 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.

The RPC Client

The code that's generated is something only a motherboard could love. That is to say, it's not very pretty Python. This is because it's not really meant to be read by humans. Open a Python shell to see how to interact with it:

Python >>>

```python
>>> from recommendations_pb2 import BookCategory, RecommendationRequest
>>> request = RecommendationRequest(
...     user_id=1, category=BookCategory.SCIENCE_FICTION, max_results=3
... )
>>> request.category
1
```

You can see that the protobuf compiler generated Python types corresponding to your protobuf types. So far, so good. You can also see that there's some type checking on the fields:

Python >>>

```python
>>> request = RecommendationRequest(
...     user_id="oops", category=BookCategory.SCIENCE_FICTION, max_results=3
... )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'oops' has type str, but expected one of: int, long
```

This shows that you get a TypeError if you pass the wrong type to one of your protobuf fields.

One important note is that all fields in `proto3` are optional, so you'll need to validate that they're all set. If you leave one unset, then it'll default to zero for numeric types or to an empty string for strings:

Python >>>

```python
>>> request = RecommendationRequest(
...     user_id=1, category=BookCategory.SCIENCE_FICTION
... )
>>> request.max_results
0
```

Here you get `0` because that's the default value for unset `int` fields.

While protobufs do type checking for you, you still need to validate the actual values. So when you implement your Recommendations microservice, you should validate that all the fields have good data. This is always true for any server regardless of whether you use protobufs, JSON, or anything else. Always validate input.

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. Take a look at the imports needed to create a client:

Python >>>

```python
>>> import grpc
>>> from recommendations_pb2_grpc import RecommendationsStub
```

You import the `grpc` module, which provides some functions for setting up connections to remote servers. Then you import the RPC client stub. It's called a **stub** because the client itself doesn't have any functionality. It calls out to a remote server and passes the result back.

If you look back at your protobuf definition, then you'll see the `service Recommendations {...}` part at the end. The protobuf compiler takes this microservice name, `Recommendations`, and appends `Stub` to it to form the client name, RecommendationsStub.

Now you can make an RPC request:

```
Python                                                                  >>>
>>> channel = grpc.insecure_channel("localhost:50051")
>>> client = RecommendationsStub(channel)
>>> request = RecommendationRequest(
...     user_id=1, category=BookCategory.SCIENCE_FICTION, max_results=3
... )
>>> client.Recommend(request)
Traceback (most recent call last):
  ...
grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that terminated with:
    status = StatusCode.UNAVAILABLE
    details = "failed to connect to all addresses"
    ...
```

You create a connection to `localhost`, your own machine, on port `50051`. This port is the standard port for gRPC, but you could change it if you like. You'll use an insecure channel for now, which is unauthenticated and unencrypted, but you'll learn how to use secure channels later in this tutorial. You then pass this channel to your stub to instantiate your client.

You can now call the `Recommend` method you defined on your `Recommendations` microservice. Think back to line 25 in your protobuf definition: `rpc Recommend (...) returns (...)`. That's where the `Recommend` method comes from. You'll get an exception because there's no microservice actually running on `localhost:50051`, so you'll implement that next!

Now that you have the client sorted out, you'll look at the server side.

The RPC Server

Testing the client in the console is one thing, but implementing the server there is a little much. You can leave your console open, but you'll implement the microservice in a file.

Start with the imports and some data:

Python

```python
 1  # recommendations/recommendations.py
 2  from concurrent import futures
 3  import random
 4
 5  import grpc
 6
 7  from recommendations_pb2 import (
 8      BookCategory,
 9      BookRecommendation,
10      RecommendationResponse,
11  )
12  import recommendations_pb2_grpc
13
14  books_by_category = {
15      BookCategory.MYSTERY: [
16          BookRecommendation(id=1, title="The Maltese Falcon"),
17          BookRecommendation(id=2, title="Murder on the Orient Express"),
18          BookRecommendation(id=3, title="The Hound of the Baskervilles"),
19      ],
20      BookCategory.SCIENCE_FICTION: [
21          BookRecommendation(
22              id=4, title="The Hitchhiker's Guide to the Galaxy"
23          ),
24          BookRecommendation(id=5, title="Ender's Game"),
25          BookRecommendation(id=6, title="The Dune Chronicles"),
26      ],
27      BookCategory.SELF_HELP: [
28          BookRecommendation(
29              id=7, title="The 7 Habits of Highly Effective People"
30          ),
31          BookRecommendation(
32              id=8, title="How to Win Friends and Influence People"
33          ),
34          BookRecommendation(id=9, title="Man's Search for Meaning"),
35      ],
36  }
```

This code imports your dependencies and creates some sample data. Here's a breakdown:

- **Line 2** imports `futures` because gRPC needs a thread pool. You'll get to that later.
- **Line 3** imports `random` because you're going to randomly select books for recommendations.
- **Line 14** creates the `books_by_category` dictionary, 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.

Next, you'll create a class that implements the microservice functions:

Python

```python
29  class RecommendationService(
30      recommendations_pb2_grpc.RecommendationsServicer
31  ):
32      def Recommend(self, request, context):
33          if request.category not in books_by_category:
34              context.abort(grpc.StatusCode.NOT_FOUND, "Category not found")
35
36          books_for_category = books_by_category[request.category]
37          num_results = min(request.max_results, len(books_for_category))
38          books_to_recommend = random.sample(
39              books_for_category, num_results
40          )
41
42          return RecommendationResponse(recommendations=books_to_recommend)
```

You've created a class with a method to implement the `Recommend` RPC. Here are the details:

- **Line 29** defines the `RecommendationService` class. This is the implementation of your microservice. Note that you subclass `RecommendationsServicer`. This is part of the integration with gRPC that you need to do.

- **Line 32** defines a `Recommend()` method on your class. 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.

- **Lines 33 and 34** use `abort()` to end 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.

- **Lines 36 to 40** randomly pick some books from the given category to recommend. 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.

- **Line 38** returns a `RecommendationResponse` object with your list of book recommendations.

Note that it would be nicer to raise an exception on error conditions rather than use `abort()` like you do in this example, but then the response wouldn't set the status code correctly. There's a way around this, which you'll get to later in the tutorial when you look at interceptors.

The `RecommendationService` class defines your microservice implementation, but you still need to run it. That's what `serve()` does:

Python

```python
41 def serve():
42     server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
43     recommendations_pb2_grpc.add_RecommendationsServicer_to_server(
44         RecommendationService(), server
45     )
46     server.add_insecure_port("[::]:50051")
47     server.start()
48     server.wait_for_termination()
49
50
51 if __name__ == "__main__":
52     serve()
```

`serve()` starts a network server and uses your microservice class to handle requests:

- **Line 42** creates a gRPC server. You tell it to use `10` threads to serve requests, which is total overkill for this demo but a good default for an actual Python microservice.
- **Line 43** associates your class with the server. This is like adding a handler for requests.
- **Line 46** tells the server to run on port `50051`. As mentioned before, this is the standard port for gRPC, but you could use anything you like instead.
- **Lines 47 and 48** call `server.start()` and `server.wait_for_termination()` to start the microservice and wait until it's stopped. The only way to stop it in this case is to type ^ Ctrl + C in the terminal. In a production environment, there are better ways to shut down, which you'll get to later.

Without closing the terminal you were using to test the client, open a new terminal and run the following command:

Shell

```shell
$ python recommendations.py
```

This runs the Recommendations microservice so that you can test the client on some actual data. Now return to the terminal you were using to test the client so you can create the channel stub. If you left your console open, then you can skip the imports, but they're repeated here as a refresher:

```python
>>> import grpc
>>> from recommendations_pb2_grpc import RecommendationsStub
>>> channel = grpc.insecure_channel("localhost:50051")
>>> client = RecommendationsStub(channel)
```

Now that you have a client object, you can make a request:

```python
>>> request = RecommendationRequest(
...     user_id=1, category=BookCategory.SCIENCE_FICTION, max_results=3)
>>> client.Recommend(request)
recommendations {
  id: 6
  title: "The Dune Chronicles"
}
recommendations {
  id: 4
  title: "The Hitchhiker\'s Guide To The Galaxy"
}
recommendations {
  id: 5
  title: "Ender\'s Game"
}
```

It works! You made an RPC request to your microservice and got a response! Note that the output you see may be different because recommendations are chosen at random.

Now that you have the server implemented, you can implement the Marketplace microservice and have it call the Recommendations microservice. You can close your Python console now if you'd like, but leave the Recommendations microservice running.

Tying It Together

Make a new `marketplace/` directory and put a `marketplace.py` file in it for your Marketplace microservice. Your directory tree should now look like this:

```
.
├── marketplace/
│   ├── marketplace.py
│   ├── requirements.txt
│   └── templates/
│       └── homepage.html
|
├── protobufs/
│   └── recommendations.proto
|
└── recommendations/
    ├── recommendations.py
    ├── recommendations_pb2.py
    ├── recommendations_pb2_grpc.py
    └── requirements.txt
```

Note the new `marketplace/` directory for your microservice code, `requirements.txt`, and a home page. All will be described below. You can create empty files for them for now and fill them in later.

You can start with the microservice code. The Marketplace microservice will be a Flask app to display a webpage to the user. It'll call the Recommendations microservice to get book recommendations to display on the page.

Open the `marketplace/marketplace.py` file and add the following:

Python

```python
 1  # marketplace/marketplace.py
 2  import os
 3  
 4  from flask import Flask, render_template
 5  import grpc
 6  
 7  from recommendations_pb2 import BookCategory, RecommendationRequest
 8  from recommendations_pb2_grpc import RecommendationsStub
 9  
10  app = Flask(__name__)
11  
12  recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
13  recommendations_channel = grpc.insecure_channel(
14      f"{recommendations_host}:50051"
15  )
16  recommendations_client = RecommendationsStub(recommendations_channel)
17  
18  
19  @app.route("/")
20  def render_homepage():
21      recommendations_request = RecommendationRequest(
22          user_id=1, category=BookCategory.MYSTERY, max_results=3
23      )
24      recommendations_response = recommendations_client.Recommend(
25          recommendations_request
26      )
27      return render_template(
28          "homepage.html",
29          recommendations=recommendations_response.recommendations,
30      )
```

You set up Flask, create a gRPC client, and add a function to render the homepage. Here's a breakdown:

- **Line 10** creates a Flask app to render a web page for the user.
- **Lines 12 to 16** create your gRPC channel and stub.
- **Lines 20 to 30** create `render_homepage()` to be called when the user visits the home page of your app. It returns an HTML page loaded from a template, with three science fiction book recommendations.

Note: In this example, you create the gRPC channel and stub as globals. Usually globals are a no-no, but in this case an exception is warranted.

The gRPC channel keeps a persistent connection to the server to avoid the overhead of having to repeatedly connect. It can handle many simultaneous requests and will reestablish dropped connections. However, if you create a new channel before each request, then Python will garbage collect it, and you won't get most of the benefits of a persistent connection.

You want the channel to stay open so you don't need to reconnect to the recommendations microservice for every request. You could hide the channel inside another module, but since you only have one file in this case, you can keep things simpler by using globals.

Open the `homepage.html` file in your `marketplace/templates/` directory and add the following HTML:

HTML

```html
<!-- homepage.html -->
<!doctype html>
<html lang="en">
<head>
    <title>Online Books For You</title>
</head>
<body>
    <h1>Mystery books you may like</h1>
    <ul>
    {% for book in recommendations %}
        <li>{{ book.title }}</li>
    {% endfor %}
    </ul>
</body>
```

This is only a demo home page. It should display a list of book recommendations when you're done.

To run this code, you'll need the following dependencies, which you can add to `marketplace/requirements.txt`:

Text

```
flask ~= 1.1
grpcio-tools ~= 1.30
Jinja2 ~= 2.11
pytest ~= 5.4
```

The Recommendations and Marketplace microservices will each have their own `requirements.txt`, but for convenience in this tutorial, you can use the same virtual environment for both. Run the following to update your virtual environment:

Shell

```shell
$ python -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:

Shell

```shell
$ cd marketplace
$ python -m grpc_tools.protoc -I ../protobufs --python_out=. \
         --grpc_python_out=. ../protobufs/recommendations.proto
```

This is the same command that you ran before, so there's nothing new here. It might feel strange to have the same files in both the `marketplace/` and `recommendations/` directories, but later you'll see how to automatically generate these as part of a deployment. You typically wouldn't store them in a version control system like Git.

To run your Marketplace microservice, enter the following in your console:

Shell

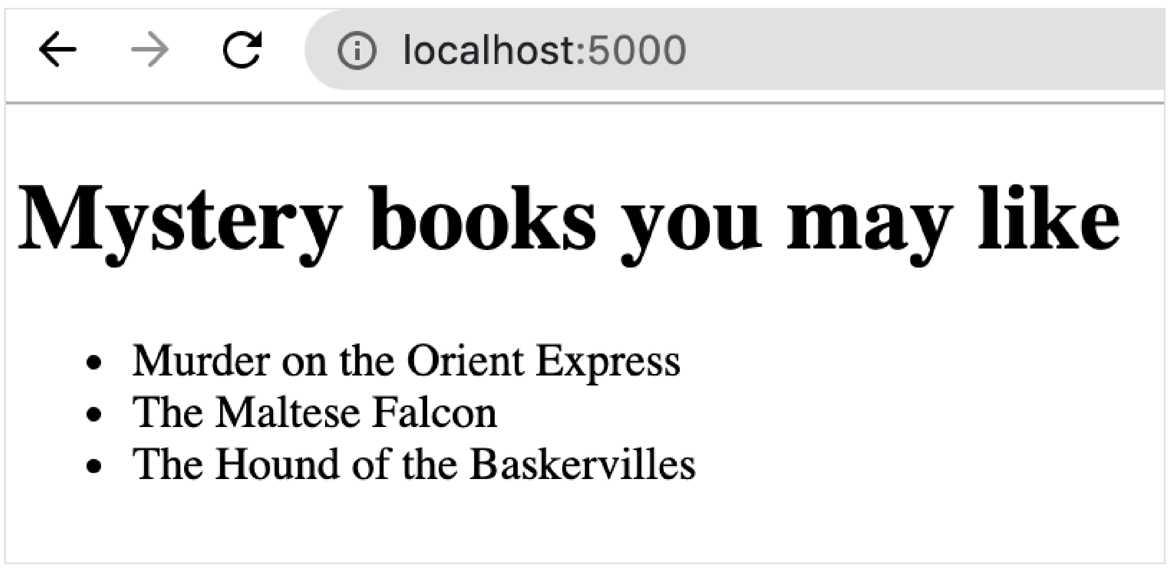
```shell
$ FLASK_APP=marketplace.py flask run
```

You should now have the Recommendations and Marketplace microservices running in two separate consoles. If you shut down the Recommendations microservice, restart it in another console with the following:

Shell

```shell
$ cd recommendations
$ python recommendations.py
```

This runs your Flask app, which runs by default on port `5000`. Go ahead and open that up in your browser and check it out:


localhost:5000

Mystery books you may like

- Murder on the Orient Express
- The Maltese Falcon
- The Hound of the Baskervilles

You now have two microservices talking to each other! But they're still just on your development machine. Next, you'll learn how to get these into a production environment.

You can stop your Python microservices by typing `^ Ctrl` + `C` in the terminal where they're running. You'll be running these in Docker next, which is how they'll run in a production environment.

Production-Ready Python Microservices

At this point, you have a Python microservice architecture running on your development machine, which is great for testing. In this section, you'll get it running in the cloud.

Docker

Docker is an amazing technology that lets you isolate a group of processes from other processes on the same machine. You can have two or more groups of processes with their own file systems, network ports, and so on. You can think of it as a Python virtual environment, but for the whole system and more secure.

Docker is perfect for deploying a Python microservice because you can package all the dependencies and run the microservice in an isolated environment. When you deploy your microservice to the cloud, it can run on the same machine as other microservices without them stepping on one another's toes. This allows for better resource utilization.

This tutorial won't dive deeply into Docker because it would take an entire book to cover. Instead, you'll just get set up with the basics you need to deploy your Python microservices to the cloud. For more information on Docker, you can check out Python Docker Tutorials.

Before you get started, if you'd like to follow along on your machine, then make sure you have Docker installed. You can download it from the official site.

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.

To create your images, you need to define a `Dockerfile`. You always start with a **base image** that has some basic things in it. In this case, your base image will include a Python interpreter. You'll then copy files from your development machine into your Docker image. You can also run commands inside the Docker image. This is useful for installing dependencies.

Recommendations `Dockerfile`

You'll start by creating the Recommendations microservice Docker image. Create `recommendations/Dockerfile` and add the following:

Dockerfile

```
 1  FROM python
 2
 3  RUN mkdir /service
 4  COPY protobufs/ /service/protobufs/
 5  COPY recommendations/ /service/recommendations/
 6  WORKDIR /service/recommendations
 7  RUN python -m pip install --upgrade pip
 8  RUN python -m pip install -r requirements.txt
 9  RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
10             --grpc_python_out=. ../protobufs/recommendations.proto
11
12  EXPOSE 50051
13  ENTRYPOINT [ "python", "recommendations.py" ]
```

Here's a line-by-line walkthrough:

- **Line 1** initializes your image with a basic Linux environment plus the latest version of Python. At this point, your image has a typical Linux file system layout. If you were to look inside, it would have `/bin`, `/home`, and all the basic files you would expect.

- **Line 3** creates a new directory at `/service` to contain your microservice code.

- **Lines 4 and 5** copy the `protobufs/` and `recommendations/` directories into `/service`.

- **Line 6** gives Docker a `WORKDIR /service/recommendations` instruction, which is kind of like doing a `cd` inside the image. Any paths you give to Docker will be relative to this location, and when you run a command, it will be run in this directory.

- **Line 7** updates `pip` to avoid warnings about older versions.

- **Line 8** tells Docker to run `pip install -r requirements.txt` inside the image. This will add all the `grpcio-tools` files, and any other packages you might add, into the image. Note that you're not using a virtual environment because it's unnecessary. The only thing running in this image will be your microservice, so you don't need to isolate its environment further.

- **Line 9** runs the `python -m grpc_tools.protoc` command to generate the Python files from the protobuf file. Your `/service` directory inside the image now looks like this:

  ```
  /service/
  |
  ├── protobufs/
  │   └── recommendations.proto
  |
  └── recommendations/
      ├── recommendations.py
      ├── recommendations_pb2.py
      ├── recommendations_pb2_grpc.py
      └── requirements.txt
  ```

- **Line 12** tells Docker that you're going to run a microservice on port `50051`, and you want to expose this outside the image.

- **Line 13** tells Docker how to run your microservice.

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:

Shell

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

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:

Shell

```
$ 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.

The `-p 127.0.0.1:50051:50051/tcp` option tells Docker to forward TCP connections on port `50051` on your machine to port `50051` inside the container. This gives you the flexibility to forward different ports on your machine.

For example, if you were running two containers that both ran Python microservices on port `50051`, then you would need to use two different ports on your host machine. This is because two processes can't open the same port at the same time unless they're in separate containers.

Marketplace `Dockerfile`

Next, you'll build your Marketplace image. Create `marketplace/Dockerfile` and add the following:

Dockerfile

```
 1 FROM python
 2 
 3 RUN mkdir /service
 4 COPY protobufs/ /service/protobufs/
 5 COPY marketplace/ /service/marketplace/
 6 WORKDIR /service/marketplace
 7 RUN python -m pip install --upgrade pip
 8 RUN python -m pip install -r requirements.txt
 9 RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
10            --grpc_python_out=. ../protobufs/recommendations.proto
11 
12 EXPOSE 5000
13 ENV FLASK_APP=marketplace.py
14 ENTRYPOINT [ "flask", "run", "--host=0.0.0.0"]
```

This is very similar to the Recommendations `Dockerfile`, with a few differences:

- **Line 13** uses `ENV FLASK_APP=marketplace.py` to set the environment variable `FLASK_APP` inside the image. Flask needs this to run.
- **Line 14** adds `--host=0.0.0.0` to the `flask run` command. If you don't add this, then Flask will only accept connections from localhost.

But wait, aren't you still running everything on `localhost`? Well, not really. When you run a Docker container, it's isolated from your host machine by default. `localhost` inside the container is different from `localhost` outside, even on the same machine. That's why you need to tell Flask to accept connections from anywhere.

Go ahead and open a new terminal. You can build your Marketplace image with this command:

Shell

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

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

Shell

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

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

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.

Luckily, Docker provides a solution to this. You can create a virtual network and add both your containers to it. You 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:

Shell

```
$ 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`:

Python

```python
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:

Python

```python
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:

Shell

```
$ 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. Huzzah!

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

Docker Compose

It’s amazing that you can do all this with Docker, but it’s a little tedious. It would be nice if there were a single command that you could run to start all your containers. Luckily there is! It’s called `docker-compose`, and it’s part of the Docker project.

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

YAML

```yaml
version: "3.8"
services:

    marketplace:
        build:
            context: .
            dockerfile: marketplace/Dockerfile
        environment:
            RECOMMENDATIONS_HOST: recommendations
        image: marketplace
        networks:
            - microservices
        ports:
            - 5000:5000

    recommendations:
        build:
            context: .
            dockerfile: recommendations/Dockerfile
        image: recommendations
        networks:
            - microservices

networks:
    microservices:
```

Typically, you put this into a file called `docker-compose.yaml`. Place this in the root of your project:

```
.
├── marketplace/
│   ├── marketplace.py
│   ├── requirements.txt
│   └── templates/
│       └── homepage.html
│
├── protobufs/
│   └── recommendations.proto
│
├── recommendations/
│   ├── recommendations.py
│   ├── recommendations_pb2.py
│   ├── recommendations_pb2_grpc.py
│   └── requirements.txt
│
└── docker-compose.yaml
```

This tutorial won’t go into much detail on syntax since it’s well documented elsewhere. It really just does the same thing you’ve done manually already. However, now you only need to run a single command to bring up your network and containers:

Shell

```shell
$ 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.

Note: When developing with `docker-compose`, if you change any of the files, then run `docker-compose build` to rebuild the images. If you run `docker-compose up`, it will use the old images, which can be confusing.

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

Testing

To unit test your Python microservice, you can instantiate your microservice class and call its methods. Here's a basic example test for your `RecommendationService` implementation:

Python

```python
1  # recommendations/recommendations_test.py
2  from recommendations import RecommendationService
3
4  from recommendations_pb2 import BookCategory, RecommendationRequest
5
6  def test_recommendations():
7      service = RecommendationService()
8      request = RecommendationRequest(
9          user_id=1, category=BookCategory.MYSTERY, max_results=1
10     )
11     response = service.Recommend(request, None)
12     assert len(response.recommendations) == 1
```

Here's a breakdown:

- **Line 6** instantiates the class like any other and calls methods on it.
- **Line 11** passes `None` for the context, which works as long as you don't use it. If you want to test code paths that use the context, then you can mock it.

Integration testing involves running automated tests with multiple microservices not mocked out. So it's a bit more involved, but it's not overly difficult. Add a `marketplace/marketplace_integration_test.py` file:

Python

```python
from urllib.request import urlopen

def test_render_homepage():
    homepage_html = urlopen("http://localhost:5000").read().decode("utf-8")
    assert "<title>Online Books For You</title>" in homepage_html
    assert homepage_html.count("<li>") == 3
```

This makes an HTTP request to the home page URL and checks that it returns some HTML with a title and three `<li>` bullet point elements in it. This isn't the greatest test since it wouldn't be very maintainable if the page had more on it, but it demonstrates a point. This test will pass only if the Recommendations microservice is up and running. You could even test the Marketplace microservice as well by making an HTTP request to it.

So how do you run this type of test? Fortunately, the good people at Docker have also provided a way to do this. Once you're running your Python microservices with `docker-compose`, you can run commands inside them with `docker-compose exec`. So if you wanted to run your integration test inside the `marketplace` container, you could run the following command:

Shell

```shell
$ docker-compose build
$ docker-compose up
$ docker-compose exec marketplace pytest marketplace_integration_test.py
```

This runs the `pytest` command inside the `marketplace` container. Because your integration test connects to `localhost`, you need to run it in the same container as the microservice.

Deploying to Kubernetes

Great! You now have a couple of microservices running on your computer. You can quickly bring them up and run integration tests on both of them. But you need to get them into a production environment. For this, you'll use Kubernetes.

This tutorial won't go into depth on Kubernetes because it's a large topic, and comprehensive documentation and tutorials are available elsewhere. However, in this section you'll find the basics to get your Python microservices to a Kubernetes cluster in the cloud.

> **Note:** To deploy Docker images to a cloud provider, you need to push your Docker images to an **image registry** like Docker Hub.
>
> The following examples use the images in this tutorial, which have already been pushed to Docker Hub. If you'd like to change them, or if you want to create your own microservices, then you'll need to create an account on Docker Hub so you can push images. You can create also create a private registry if you'd like, or use another registry, such as Amazon's ECR.

Kubernetes Configs

You can start with a minimal Kubernetes configuration in `kubernetes.yaml`. The complete file is a little long, but it consists of four distinct sections, so you'll look at them one by one:

YAML

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: marketplace
    labels:
        app: marketplace
spec:
    replicas: 3
    selector:
        matchLabels:
            app: marketplace
    template:
        metadata:
            labels:
                app: marketplace
        spec:
            containers:
                - name: marketplace
                  image: hidan/python-microservices-article-marketplace:0.1
                  env:
                      - name: RECOMMENDATIONS_HOST
                        value: recommendations
```

This defines a **Deployment** for the Marketplace microservice. A Deployment tells Kubernetes how to deploy your code. Kubernetes needs four main pieces of information:

1. What Docker image to deploy
2. How many instances to deploy
3. What environment variables the microservices need
4. How to identify your microservice

You can tell Kubernetes how to identify your microservice by using **labels**. Although not shown here, you can also tell Kubernetes what memory and CPU resources your microservice needs. You can find many other options in the Kubernetes documentation.

Here's what's happening in the code:

- **Line 9** tells Kubernetes how many pods to create for your microservice. A **pod** is basically an isolated execution environment, like a lightweight virtual machine implemented as a set of containers. Setting `replicas: 3` gives you three pods for each microservice. Having more than one allows for redundancy, enabling rolling updates without downtime, scaling as you need more machines, and having failovers in case one goes down.
- **Line 20** is the Docker image to deploy. You must use a Docker image on an image registry. To get your image there, you must push it to the image registry. There are instructions on how to do this when you log in to your account on Docker Hub.

The Deployment for the Recommendations microservice is very similar:

YAML

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: recommendations
    labels:
        app: recommendations
spec:
    replicas: 3
    selector:
        matchLabels:
            app: recommendations
    template:
        metadata:
            labels:
                app: recommendations
        spec:
            containers:
                - name: recommendations
                  image: hidan/python-microservices-article-recommendations:0.1
```

The main difference is that one uses the name `marketplace` and the other uses `recommendations`. You also set the `RECOMMENDATIONS_HOST` environment variable on the `marketplace` Deployment but not on the `recommendations` Deployment.

Next, you define a **Service** for the Recommendations microservice. Whereas a Deployment tells Kubernetes how to deploy your code, a Service tells it how to route requests to it. To avoid confusion with the term *service* that is commonly used to talk about microservices, you'll see the word capitalized when used in reference to a Kubernetes Service.

Here's the Service definition for `recommendations`:

YAML

```yaml
---
apiVersion: v1
kind: Service
metadata:
    name: recommendations
spec:
    selector:
        app: recommendations
    ports:
        - protocol: TCP
          port: 50051
          targetPort: 50051
```

Here's what's happening in the definition:

- **Line 48:** When you create a Service, Kubernetes essentially creates a DNS hostname with the same `name` within the cluster. So any microservice in your cluster can send a request to `recommendations`. Kubernetes will forward this request to one of the pods in your Deployment.

- **Line 51:** This line connects the Service to the Deployment. It tells Kubernetes to forward requests to `recommendations` to one of the pods in the `recommendations` Deployment. This must match one of the key-value pairs in the `labels` of the Deployment.

The `marketplace` Service is similar:

YAML

```yaml
---
apiVersion: v1
kind: Service
metadata:
    name: marketplace
spec:
    type: LoadBalancer
    selector:
        app: marketplace
    ports:
        - protocol: TCP
          port: 5000
          targetPort: 5000
```

Aside from the names and ports, there's only one difference. You'll notice that `type: LoadBalancer` appears only in the `marketplace` Service. This is because `marketplace` needs to be accessible from outside the Kubernetes cluster, whereas `recommendations` only needs to be accessible inside the cluster.

Note: It's more common to use an **`Ingress`** Service than a **`LoadBalancer`** Service in a large cluster with many microservices. If you're developing microservices in a corporate environment, then that's probably the way to go.

Check out Sandeep Dinesh's article Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what? for more information.

You can see the complete file by expanding the box below:

Complete `kubernetes.yaml` Code Show/Hide

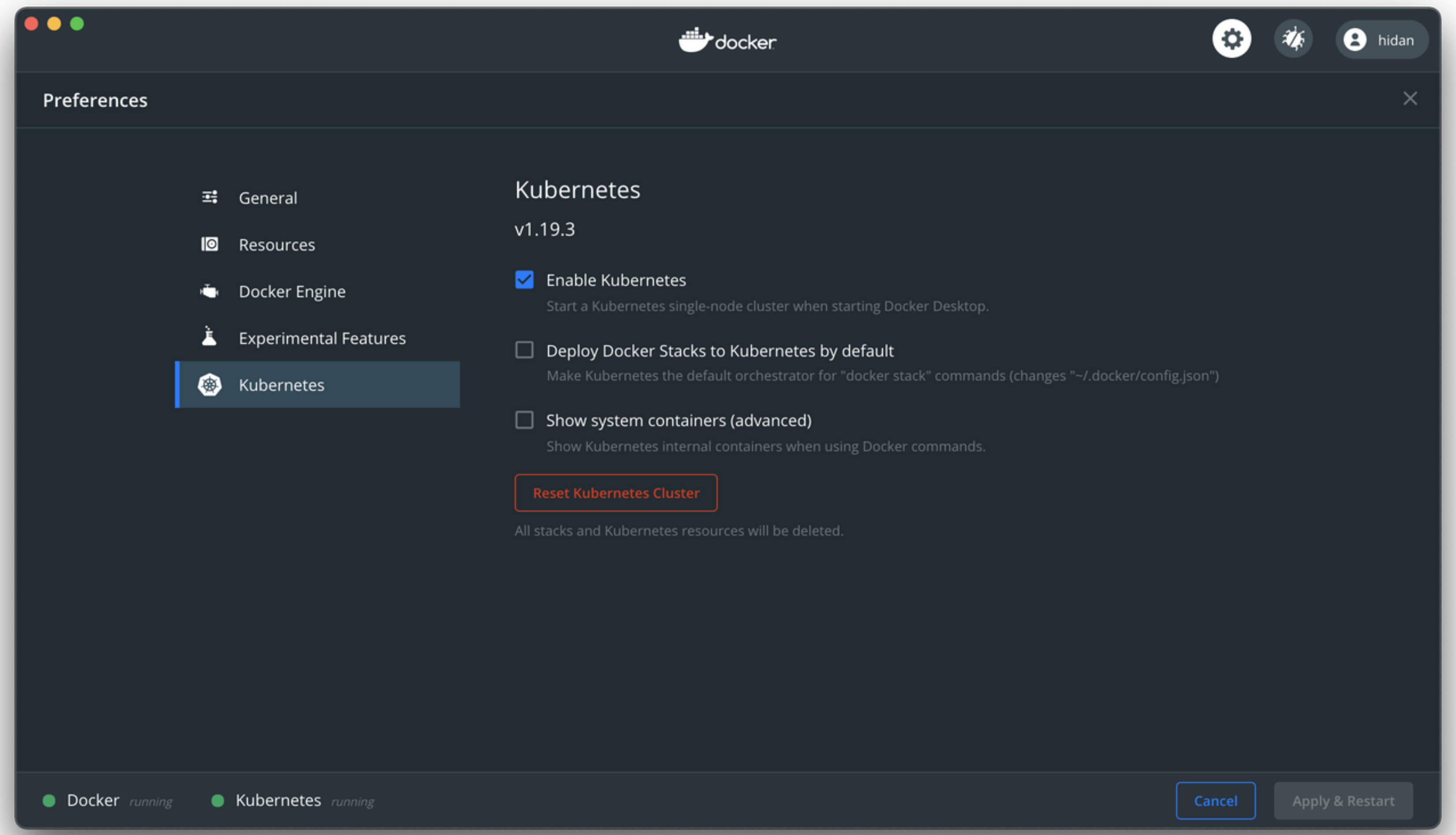
Now that you have a Kubernetes configuration, your next step is to deploy it!

Deploying Kubernetes

You typically deploy Kubernetes using a cloud provider. There are many cloud providers you can choose from, including Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), and DigitalOcean.

If you're deploying microservices at your company, then the cloud provider you use will likely be dictated by your infrastructure. For this demo, you'll run Kubernetes locally. Almost everything will be the same as using a cloud provider.

If you're running Docker Desktop on Mac or Windows, then it comes with a local Kubernetes cluster that you can enable in the Preferences menu. Open Preferences by clicking the Docker icon in the system tray, then find the Kubernetes section and enable it:



Kubernetes

v1.19.3

- [✓] Enable Kubernetes
  Start a Kubernetes single-node cluster when starting Docker Desktop.
- [ ] Deploy Docker Stacks to Kubernetes by default
  Make Kubernetes the default orchestrator for "docker stack" commands (changes "~/.docker/config.json")
- [ ] Show system containers (advanced)
  Show Kubernetes internal containers when using Docker commands.

Reset Kubernetes Cluster

All stacks and Kubernetes resources will be deleted.

If you're running on Linux, then you can install minikube. Follow the instructions on the start page to get set up.

Once you've created your cluster, you can deploy your microservices with the following command:

Shell
```
$ kubectl apply -f kubernetes.yaml
```

If you'd like to try deploying to Kubernetes in the cloud, DigitalOcean is the least complicated to set up and has a simple pricing model. You can sign up for an account and then create a Kubernetes cluster in a few clicks. If you change the defaults to use only one node and the cheapest options, then at the time of this writing the cost was only $0.015 per hour.

Follow the instructions DigitalOcean provides to download a config file for `kubectl` and run the command above. You can then click the *Kubernetes* button in DigitalOcean to see your Services running there. DigitalOcean will assign an IP address to your `LoadBalancer` Service, so you can visit your Marketplace app by copying that IP address into your browser.

Important: When you're done, destroy your cluster so you don't continue being charged for it. You should also go to the Networking tab and destroy the Load Balancer, which is separate from the cluster but also accrues a charge.

That wraps up deploying to Kubernetes. Next, you'll learn how to monitor you Python microservices.

Python Microservice Monitoring With Interceptors

Once you have some microservices in the cloud, you want to have visibility into how they're doing. Some things you want to monitor include:

- How many requests each microservice is getting
- How many requests result in an error, and what type of error they raise
- The latency on each request
- Exception logs so you can debug later

You'll learn about a few ways of doing this in the sections below.

Why Not Decorators

One way you could do this, and the most natural to Python developers, is to add a decorator to each microservice endpoint. However, in this case, there are several downsides to using decorators:

- Developers of new microservices have to remember to add them to each method.
- If you have a lot of monitoring, then you might end up with a stack of decorators.
- If you have a stack of decorators, then developers may stack them in the wrong order.
- You could consolidate all your monitoring into a single decorator, but then it could get messy.

This stack of decorators is what you want to avoid:

Python

```python
class RecommendationService(recommendations_pb2_grpc.RecommendationsServicer):
    @catch_and_log_exceptions
    @log_request_counts
    @log_latency
    def Recommend(self, request, context):
        ...
```

Having this stack of decorators on every method is ugly and repetitive, and it violates the DRY programming principle: don't repeat yourself. Decorators are also a challenge to write, especially if they accept arguments.

Interceptors

There an alternative approach to using decorators that you'll pursue in this tutorial: gRPC has an **interceptor** concept that provides functionality similar to a decorator but in a cleaner way.

Implementing Interceptors

Unfortunately, the Python implementation of gRPC has a fairly complex API for interceptors. This is because it's incredibly flexible. However, there's a `grpc-interceptor` package to simplify them. For full disclosure, I'm the author.

Add it to your `recommendations/requirements.txt` along with `pytest`, which you'll use shortly:

Text

```
grpc-interceptor ~= 0.12.0
grpcio-tools ~= 1.30
pytest ~= 5.4
```

Then update your virtual environment:

Shell

```shell
$ python -m pip install recommendations/requirements.txt
```

You can now create an interceptor with the following code. You don't need to add this to your project as it's just an example:

Python

```python
from grpc_interceptor import ServerInterceptor

class ErrorLogger(ServerInterceptor):
    def intercept(self, method, request, context, method_name):
        try:
            return method(request, context)
        except Exception as e:
            self.log_error(e)
            raise

    def log_error(self, e: Exception) -> None:
        # ...
```

This will call `log_error()` whenever an unhandled exception in your microservice is called. You could implement this by, for example, logging exceptions to Sentry so you get alerts and debugging info when they happen.

To use this interceptor, you would pass it to `grpc.server()` like this:

Python

```python
interceptors = [ErrorLogger()]
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10),
                     interceptors=interceptors)
```

With this code, every request to and response from your Python microservice will go through your interceptor, so you can count how many requests and errors it gets.

`grpc-interceptor` also provides an exception for each gRPC status code and an interceptor called `ExceptionToStatusInterceptor`. If one of the exceptions is raised by the microservice, then `ExceptionToStatusInterceptor` will set the gRPC status code. This allows you to simplify your microservice by making the changes highlighted below to `recommendations/recommendations.py`:

Python

```python
from grpc_interceptor import ExceptionToStatusInterceptor
from grpc_interceptor.exceptions import NotFound

# ...

class RecommendationService(recommendations_pb2_grpc.RecommendationsServicer):
    def Recommend(self, request, context):
        if request.category not in books_by_category:
            raise NotFound("Category not found")

        books_for_category = books_by_category[request.category]
        num_results = min(request.max_results, len(books_for_category))
        books_to_recommend = random.sample(books_for_category, num_results)

        return RecommendationResponse(recommendations=books_to_recommend)

def serve():
    interceptors = [ExceptionToStatusInterceptor()]
    server = grpc.server(
        futures.ThreadPoolExecutor(max_workers=10),
        interceptors=interceptors
    )
    # ...
```

This is more readable. You can also raise the exception from many functions down the call stack rather than having to pass `context` so you can call `context.abort()`. You also don’t have to catch the exception yourself in your microservice—the interceptor will catch it for you.

Testing Interceptors

If you want to write your own interceptors, then you should test them. But it's dangerous to mock too much out when testing something like interceptors. For example, you could call `.intercept()` in the test and make sure it returns what you want, but this wouldn't test realistic inputs or that they even get called at all.

To improve testing, you can run a gRPC microservice with interceptors. The `grpc-interceptor` package provides a framework to do that. Below, you'll write a test for the `ErrorLogger` interceptor. This is only an example, so you don't need to add it to your project. If you were to add it, then you would add it to a test file.

Here's how you could write a test for an interceptor:

Python

```python
 1  from grpc_interceptor.testing import dummy_client, DummyRequest, raises
 2
 3  class MockErrorLogger(ErrorLogger):
 4      def __init__(self):
 5          self.logged_exception = None
 6
 7      def log_error(self, e: Exception) -> None:
 8          self.logged_exception = e
 9
10  def test_log_error():
11      mock = MockErrorLogger()
12      ex = Exception()
13      special_cases = {"error": raises(ex)}
14
15      with dummy_client(special_cases=special_cases, interceptors=[mock]) as client:
16          # Test no exception
17          assert client.Execute(DummyRequest(input="foo")).output == "foo"
18          assert mock.logged_exception is None
19
20          # Test exception
21          with pytest.raises(grpc.RpcError) as e:
22              client.Execute(DummyRequest(input="error"))
23          assert mock.logged_exception is ex
```

Here's a walk-through:

- **Lines 3 to 8** subclass `ErrorLogger` to mock out `log_error()`. You don't actually want the logging side effect to happen. You just want to make sure it's called.
- **Lines 15 to 18** use the `dummy_client()` context manager to create a client that's connected to a real gRPC microservice. You send `DummyRequest` to the microservice, and it replies with `DummyResponse`. By default, the `input` of `DummyRequest` is echoed to the `output` of `DummyResponse`. However, you can pass `dummy_client()` a dictionary of special cases, and if `input` matches one of them, then it will call a function you provide and return the result.
- **Lines 21 to 23:** You test that `log_error()` is called with the expected exception. `raises()` returns another function that raises the provided exception. You set `input` to `error` so that the microservice will raise an exception.

For more information about testing, you can read Effective Python Testing With Pytest and Understanding the Python Mock Object Library.

An alternative to interceptors in some cases is to use a **service mesh**. It will send all microservice requests and responses through a proxy, so the proxy can automatically log things like request volume and error counts. To get accurate error logging, your microservice still needs to set status codes correctly. So in some cases, your interceptors can complement a service mesh. One popular service mesh is Istio.

Best Practices

Now you have a working Python microservice setup. You can create microservices, test them together, deploy them to Kubernetes, and monitor them with interceptors. You can get started creating microservices at this point. You should keep some best practices in mind, however, so you'll learn a few in this section.

Protobuf Organization

Generally, you should keep your protobuf definitions separate from your microservice implementation. Clients can be written in almost any language, and if you bundle your protobuf files into a Python wheel or something similar, then if someone wants a Ruby or Go client, it's going to be hard for them to get the protobuf files.

Even if all your code is Python, why should someone need to install the package for the microservice just to write a client for it?

A solution is to put your protobuf files in a separate Git repo from the microservice code. Many companies put *all* the protobuf files for *all* microservices in a single repo. This makes it easier to find all microservices, share common protobuf structures among them, and create useful tooling.

If you do choose to store your protobuf files in a single repo, you need to be careful that the repo stays organized, and you should definitely avoid cyclical dependencies between Python microservices.

Protobuf Versioning

API versioning can be hard. The main reason is that if you change an API and update the microservice, then there may still be clients using the old API. This is especially true when the clients live on customers' machines, such as mobile clients or desktop software.

You can't easily force people to update. Even if you could, network latency causes race conditions, and your microservice is likely to get requests using the old API. Good APIs should be either backward compatible or versioned.

To achieve **backward compatibility**, Python microservices using protobufs version 3 will accept requests with missing fields. If you want to add a new field, then that's okay. You can deploy the microservice first, and it will still accept requests from the old API without the new field. The microservice just needs to handle that gracefully.

If you want to make more drastic changes, then you'll need to **version** your API. Protobufs allow you to put your API into a package namespace, which can include a version number. If you need to drastically change the API, then you can create a new version of it. The microservice can continue to accept the old version as well. This allows you to roll out a new API version while phasing out an older version over time.

By following these conventions, you can avoid making breaking changes. Inside a company, people sometimes feel that making breaking changes to an API is acceptable because they control all the clients. This is up to you to decide, but be aware that making breaking changes requires coordinated client and microservice deploys, and it complicates rollbacks.

This can be okay very early in a microservice's lifecycle, when there are no production clients. However, it's good to get into the habit of making only nonbreaking changes once your microservice is critical to the health of your company.

Protobuf Linting

One way to ensure you don't make breaking changes to your protobufs is to use a **linter**. A popular one is `buf`. You can set this up as part of your CI system so you can check for breaking changes in pull requests.

Type Checking Protobuf-Generated Code

Mypy is a project for statically type checking Python code. If you're new to static type checking in Python, then you can read Python Type Checking to learn all about it.

The code generated by `protoc` is a little gnarly, and it doesn't have type annotations. If you try to type check it with Mypy, then you'll get lots of errors and it won't catch real bugs like misspelled field names. Luckily, the nice people at Dropbox wrote a plugin for the `protoc` compiler to generate type stubs. These should not be confused with gRPC stubs.

In order to use it, you can install the `mypy-protobuf` package and then update the command to generate protobuf output. Note the new `--mypy_out` option:

Shell

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

Most of your Mypy errors should go away. You may still get an error about the `grpc` package not having type info. You can either install unofficial gRPC type stubs or add the following to your Mypy config:

Config File

```ini
[mypy-grpc.*]
ignore_missing_imports = True
```

You'll still get most of the benefits of type checking, such as catching misspelled fields. This is really helpful for catching bugs before they make it to production.

Shutting Down Gracefully

When running your microservice on your development machine, you can press `^Ctrl` + `C` to stop it. This will cause the Python interpreter to raise a `KeyboardInterrupt` exception.

When Kubernetes is running your microservice and needs to stop it to roll out an update, it will send a signal to your microservice. Specifically, it will send a `SIGTERM` signal and wait thirty seconds. If your microservice hasn't exited by then, it will send a `SIGKILL` signal.

You can, and should, catch and handle the `SIGTERM` so you can finish processing current requests but refuse new ones. You can do so by putting the following code in `serve()`:

Python

```python
 1  from signal import signal, SIGTERM
 2
 3  ...
 4
 5  def serve():
 6      server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
 7      ...
 8      server.add_insecure_port("[::]:50051")
 9      server.start()
10
11      def handle_sigterm(*_):
12          print("Received shutdown signal")
13          all_rpcs_done_event = server.stop(30)
14          all_rpcs_done_event.wait(30)
15          print("Shut down gracefully")
16
17      signal(SIGTERM, handle_sigterm)
18      server.wait_for_termination()
```

Here's a breakdown:

- **Line 1** imports `signal`, which allows you to catch and handle signals from Kubernetes or almost any other process.
- **Line 11** defines a function to handle `SIGTERM`. The function will be called when Python receives the `SIGTERM` signal, and Python will pass it two arguments. You don't need the arguments, however, so use `*_` to ignore them both.
- **Line 13** calls `server.stop(30)` to shut down the server gracefully. It will refuse new requests and wait `30` seconds for current requests to complete. It returns immediately, but it returns a `threading.Event` object on which you can wait.
- **Line 14** waits on the `Event` object so Python doesn't exit prematurely.
- **Line 17** registers your handler.

When you deploy a new version of your microservice, Kubernetes will send signals to shut down the existing microservice. Handling these to shut down gracefully will ensure a request isn't dropped.

Securing Channels

So far you've been using insecure gRPC channels. This means a few things:

1. The client can't confirm that it's sending requests to the intended server. Someone could create an imposter microservice and inject it somewhere that the client might send a request to. For instance, they might be able to inject the microservice in a pod to which the load balancer would send requests.

2. The server can't confirm the client sending requests to it. As long as someone can connect to the server, they can send it arbitrary gRPC requests.

3. The traffic is unencrypted, so any nodes routing traffic can also view it.

This section will describe how to add TLS authentication and encryption.

Note: This does not address authenticating users, only microservice processes.

You'll learn two ways to set up TLS:

1. The straightforward way, in which the client can validate the server, but the server doesn't validate the client.
2. The more complex way, with mutual TLS, in which the client and the server validate each other.

In both cases, traffic is encrypted.

TLS Basics

Before diving in, here's a brief overview of TLS: Typically, a client validates a server. For example, when you visit Amazon.com, your browser validates that it's really Amazon.com and not an imposter. To do this, the client must receive some sort of assurance from a trustworthy third party, sort of like how you might trust a new person only if you have a mutual friend who vouches for them.

With TLS, the client must trust a **certificate authority (CA)**. The CA will sign something held by the server so the client can verify it. This is a bit like your mutual friend signing a note and you recognizing their handwriting. For more information, see How internet security works: TLS, SSL, and CA.

Your browser implicitly trusts some CAs, which are typically companies like GoDaddy, DigiCert, or Verisign. Other companies, like Amazon, pay a CA to sign a digital certificate for them so your browser trusts them. Typically, the CA would verify that Amazon owns Amazon.com before signing their certificate. That way, an imposter wouldn't have a signature on a certificate for Amazon.com, and your browser would block the site.

With microservices, you can't really ask a CA to sign a certificate because your microservices run on internal machines. The CA would probably be happy to sign a certificate and charge you for it, but the point is that it's not practical. In this case, your company can act as its own CA. The gRPC client will trust the server if it has a certificate signed by your company or by you if you're doing a personal project.

Server Authentication

The following command will create a CA certificate that can be used to sign a server's certificate:

Shell

```shell
$ openssl req -x509 -nodes -newkey rsa:4096 -keyout ca.key -out ca.pem \
              -subj /O=me
```

This will output two files:

1. `ca.key` is a private key.
2. `ca.pem` is a public certificate.

You can then create a certificate for your server and sign it with your CA certificate:

Shell

```
$ openssl req -nodes -newkey rsa:4096 -keyout server.key -out server.csr \
              -subj /CN=recommendations
$ openssl x509 -req -in server.csr -CA ca.pem -CAkey ca.key -set_serial 1 \
              -out server.pem
```

This will produce three new files:

1. `server.key` is the server's private key.
2. `server.csr` is an intermediate file.
3. `server.pem` is the server's public certificate.

Note: These commands are for example purposes only. The private keys are *not* encrypted. If you want to generate certificates for your company, then consult your security team. They will likely have policies to create, store, and revoke certificates that you should follow.

You can add this to the Recommendations microservice `Dockerfile`. It's very hard to securely add secrets to a Docker image, but there's a way to do it with the latest versions of Docker, shown highlighted below:

Dockerfile

```
 1  # syntax = docker/dockerfile:1.0-experimental
 2  # DOCKER_BUILDKIT=1 docker build . -f recommendations/Dockerfile \
 3  #                      -t recommendations --secret id=ca.key,src=ca.key
 4
 5  FROM python
 6
 7  RUN mkdir /service
 8  COPY infra/ /service/infra/
 9  COPY protobufs/ /service/protobufs/
10  COPY recommendations/ /service/recommendations/
11  COPY ca.pem /service/recommendations/
12
13  WORKDIR /service/recommendations
14  RUN python -m pip install --upgrade pip
15  RUN python -m pip install -r requirements.txt
16  RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
17             --grpc_python_out=. ../protobufs/recommendations.proto
18  RUN openssl req -nodes -newkey rsa:4096 -subj /CN=recommendations \
19                  -keyout server.key -out server.csr
20  RUN --mount=type=secret,id=ca.key \
21      openssl x509 -req -in server.csr -CA ca.pem -CAkey /run/secrets/ca.key \
22                   -set_serial 1 -out server.pem
23
24  EXPOSE 50051
25  ENTRYPOINT [ "python", "recommendations.py" ]
```

The new lines are highlighted. Here's an explanation:

- **Line 1** is needed to enable secrets.
- **Lines 2 and 3** show the command for how to build the Docker image.
- **Line 11** copies the CA public certificate into the image.
- **Lines 18 and 19** generate a new server private key and certificate.
- **Lines 20 to 22** temporarily load the CA private key so you can sign the server's certificate with it. However, it won't be kept in the image.

Note: For more information on `--mount=type=secret`, see the Docker documentation. In the future, this feature may be promoted to the stable release, at which time you won't need to include the `syntax = docker/dockerfile:1.0-experimental` comment in the `Dockerfile`.

According to the versioning policy, the experimental syntax won't go away, so you can continue to use it indefinitely.

Your image will now have the following files:

- `ca.pem`
- `server.csr`
- `server.key`
- `server.pem`

You can now update `serve()` in `recommendations.py` as highlighted:

Python

```python
 1  def serve():
 2      server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
 3      recommendations_pb2_grpc.add_RecommendationsServicer_to_server(
 4          RecommendationService(), server
 5      )
 6
 7      with open("server.key", "rb") as fp:
 8          server_key = fp.read()
 9      with open("server.pem", "rb") as fp:
10          server_cert = fp.read()
11
12      creds = grpc.ssl_server_credentials([(server_key, server_cert)])
13      server.add_secure_port("[::]:443", creds)
14      server.start()
15      server.wait_for_termination()
```

Here are the changes:

- **Lines 7 to 10** load the server's private key and certificate.
- **Lines 12 and 13** run the server using TLS. It will accept only TLS-encrypted connections now.

You'll need to update `marketplace.py` to load the CA cert. You only need the public cert in the client for now, as highlighted:

Python

```python
recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
with open("ca.pem", "rb") as fp:
    ca_cert = fp.read()
creds = grpc.ssl_channel_credentials(ca_cert)
recommendations_channel = grpc.secure_channel(
    f"{recommendations_host}:443", creds
)
recommendations_client = RecommendationsStub(recommendations_channel)
```

You'll also need to add `COPY ca.pem /service/marketplace/` to the Marketplace `Dockerfile`.

You can now run the client and server with encryption, and the client will validate the server. To make running everything straightforward, you can use `docker-compose`. However, at the time of this writing, `docker-compose` didn't support build secrets. You will have to build the Docker images manually instead of with `docker-compose build`.

You can still run `docker-compose up`, however. Update the `docker-compose.yaml` file to remove the `build` sections:

YAML

```yaml
version: "3.8"
services:

    marketplace:
        environment:
            RECOMMENDATIONS_HOST: recommendations
        # DOCKER_BUILDKIT=1 docker build . -f marketplace/Dockerfile \
        #                   -t marketplace --secret id=ca.key,src=ca.key
        image: marketplace
        networks:
            - microservices
        ports:
            - 5000:5000

    recommendations:
        # DOCKER_BUILDKIT=1 docker build . -f recommendations/Dockerfile \
        #                   -t recommendations --secret id=ca.key,src=ca.key
        image: recommendations
        networks:
            - microservices

networks:
    microservices:
```

You're now encrypting traffic and verifying that you're connecting to the correct server.

Mutual Authentication

The server now proves that it can be trusted, but the client does not. Luckily, TLS allows verification of both sides. Update the Marketplace `Dockerfile` as highlighted:

Dockerfile

```dockerfile
# syntax = docker/dockerfile:1.0-experimental
# DOCKER_BUILDKIT=1 docker build . -f marketplace/Dockerfile \
#                     -t marketplace --secret id=ca.key,src=ca.key

FROM python

RUN mkdir /service
COPY protobufs/ /service/protobufs/
COPY marketplace/ /service/marketplace/
COPY ca.pem /service/marketplace/

WORKDIR /service/marketplace
RUN python -m pip install -r requirements.txt
RUN python -m grpc_tools.protoc -I ../protobufs --python_out=. \
           --grpc_python_out=. ../protobufs/recommendations.proto
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

EXPOSE 5000
ENV FLASK_APP=marketplace.py
ENTRYPOINT [ "flask", "run", "--host=0.0.0.0"]
```

These changes are similar to the ones you made for the Recommendations microservice in the preceding section.

Note: If you're putting private keys in your Dockerfiles, then don't host them on a public repository. It would be better to load the private keys at run time, over the network, from a server that's accessible only via VPN.

Update `serve()` in `recommendations.py` to authenticate the client as highlighted:

Python

```python
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    recommendations_pb2_grpc.add_RecommendationsServicer_to_server(
        RecommendationService(), server
    )

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

This loads the CA certificate and requires client authentication.

Finally, update `marketplace.py` to send its certificate to the server as highlighted:

Python

```python
recommendations_host = os.getenv("RECOMMENDATIONS_HOST", "localhost")
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)
recommendations_channel = grpc.secure_channel(
    f"{recommendations_host}:443", creds
)
recommendations_client = RecommendationsStub(recommendations_channel)
```

This loads certificates and sends them to the server for verification.

Now if you try to connect to the server with another client, even one using TLS but with an unknown certificate, then the server will reject it with the error `PEER_DID_NOT_RETURN_A_CERTIFICATE`.

Important: While it's possible to manage mutual TLS this way, it isn't easy to do so. It gets especially difficult if you want to enable only certain microservices to make requests to others.

If you have a need for heightened security like this, it's probably better to use a **service mesh** and let it manage certificates and authorization for you. In addition to the traffic monitoring mentioned in the interceptor section, Istio can manage mutual TLS and per-service authorization, too. It's also more secure because it will manage secrets for you and reissue certificates more frequently.

That wraps up securing communication between microservices. Next, you'll learn about using AsyncIO with microservices.

AsyncIO and gRPC

AsyncIO support in the official gRPC package was lacking for a long time, but has recently been added. It's still experimental and under active development, but if you really want to try AsyncIO in your microservices, then it could be a good option. You can check out the gRPC AsyncIO documentation for more details.

There’s also a third-party package called `grpclib` that implements AsyncIO support for gRPC and has been around longer.

Be extremely careful with AsyncIO on the server side. It’s easy to accidentally write **blocking code**, which will bring your microservice to its knees. As a demonstration, here’s how you might write the Recommendations microservice using AsyncIO with all logic stripped out:

Python

```python
 1  import time
 2  
 3  import asyncio
 4  import grpc
 5  import grpc.experimental.aio
 6  
 7  from recommendations_pb2 import (
 8      BookCategory,
 9      BookRecommendation,
10      RecommendationResponse,
11  )
12  import recommendations_pb2_grpc
13  
14  class AsyncRecommendations(recommendations_pb2_grpc.RecommendationsServicer):
15      async def Recommend(self, request, context):
16          print("Handling request")
17          time.sleep(5)  # Oops, blocking!
18          print("Done")
19          return RecommendationResponse(recommendations=[])
20  
21  async def main():
22      grpc.experimental.aio.init_grpc_aio()
23      server = grpc.experimental.aio.server()
24      server.add_insecure_port("[::]:50051")
25      recommendations_pb2_grpc.add_RecommendationsServicer_to_server(
26          AsyncRecommendations(), server
27      )
28      await server.start()
29      await server.wait_for_termination()
30  
31  asyncio.run(main())
```

There’s a mistake in this code. On line 17, you’ve accidentally made a blocking call inside an `async` function, which is a big no-no. Because AsyncIO servers are single-threaded, this blocks the whole server so it can only process one request at a time. This is much worse than a threaded server.

You can demonstrate this by making multiple concurrent requests:

Python

```python
from concurrent.futures import ThreadPoolExecutor

import grpc

from recommendations_pb2 import BookCategory, RecommendationRequest
from recommendations_pb2_grpc import RecommendationsStub

request = RecommendationRequest(user_id=1, category=BookCategory.MYSTERY)
channel = grpc.insecure_channel("localhost:50051")
client = RecommendationsStub(channel)

executor = ThreadPoolExecutor(max_workers=5)
a = executor.submit(client.Recommend, request)
b = executor.submit(client.Recommend, request)
c = executor.submit(client.Recommend, request)
d = executor.submit(client.Recommend, request)
e = executor.submit(client.Recommend, request)
```

This will make five concurrent requests, but on the server side you'll see this:

Shell

```
Handling request
Done
Handling request
Done
Handling request
Done
Handling request
Done
Handling request
Done
```

The requests are being handled sequentially, which is not what you want!

There are use cases for AsyncIO on the server side, but you must be very careful not to block. This means that you can't use standard packages like `requests` or even make RPCs to other microservices unless you run them in another thread using `run_in_executor`.

You also have to be careful with database queries. Many of the great Python packages you've come to use may not support AsyncIO yet, so be careful to check whether they do. Unless you have a very strong need for AsyncIO on the server side, it might be safer to wait until there's more package support. Blocking bugs can be hard to find.

If you'd like to learn more about AsyncIO, then you can check out Getting Started With Async Features in Python and Async IO in Python: A Complete Walkthrough.

Conclusion

Microservices are a way to manage complex systems. They become a natural way to organize code as an organization grows. Understanding how to effectively implement microservices in Python can make you more valuable to your company as it grows.

In this tutorial, you've learned:

- How to implement Python microservices effectively with **gRPC**
- How to deploy microservices to **Kubernetes**
- How to incorporate features such as **integration testing**, **interceptors**, **TLS**, and **AsyncIO** in your microservices
- What **best practices** to follow when creating Python microservices

You're now equipped to start breaking your larger Python applications into smaller microservices, making your code more organized and maintainable. To review everything you've learned in this tutorial, you can download the source code from the examples by clicking the link below:

Get the Source Code: Click here to get the source code you'll use to learn about creating Python microservices with gRPC in this tutorial.