**6 Building REST APIs with Python**

This chapter covers

* Adding URL query parameters to an endpoint using FastAPI
* Disallowing the presence of unknown properties in a payload using pydantic and marshmallow
* Implementing a REST API using flask-smorest
* Defining validation schemas and URL query parameters using marshmallow

In previous chapters, you learned to design and document REST APIs. In this chapter, you’ll learn to implement REST APIs by working on two examples from the CoffeeMesh platform, the on-demand coffee delivery application that we introduced in chapter 1. We’ll build the APIs for the orders service and for the kitchen service. The orders service is the main gateway to CoffeeMesh for customers of the platform. Through it they can place orders, pay for those orders, update them, and keep track of them. The kitchen service takes care of scheduling orders for production in the CoffeeMesh factories and keeps track of their progress. We’ll learn best practices for implementing REST APIs as we work through these examples.

In chapter 2, we implemented part of the orders API. In the first sections of this chapter, we pick up the orders API where we left it in chapter 2 and implement its remaining features using FastAPI, a highly performant API framework for Python and a popular choice for building REST APIs. We’ll learn how to add URL query parameters to our endpoints using FastAPI. As we saw in chapter 2, FastAPI uses pydantic for data validation, and in this chapter we’ll use pydantic to forbid unknown fields in a payload. We’ll learn about the tolerant reader pattern and balance its benefits against the risk of API integration failures due to errors such as typos.

After completing the implementation of the orders API, we’ll implement the API for the kitchen service. The kitchen service schedules orders for production in the factory and keeps track of their progress. We’ll implement the kitchen API using flask-smorest, a popular API framework built on top of Flask and marshmallow. We’ll learn to implement our APIs following Flask application patterns, and we’ll define validation schemas using marshmallow.

By the end of this chapter, you’ll know how to implement REST APIs using FastAPI and Flask, two of the most popular libraries in the Python ecosystem. You’ll see how the principles for implementing REST APIs transcend the implementation details of each framework and can be applied regardless of the technology that you use. The code for this chapter is available under folder ch06 in the repository provided with this book. Folder ch06 contains two subfolders: one for the orders API (ch06/orders) and one for the kitchen API (ch06/kitchen). With that said, and without further ado, let’s get cracking!

**6.1 Overview of the orders API**

In this section, we recap the minimal implementation of the orders API that we undertook in chapter 2. You can find the full specification of the orders API under ch06/ orders/oas.yaml in the GitHub repository for this book. Before we jump directly into the implementation, let’s briefly analyze the specification and see what’s left to implement.

In chapter 2, we implemented the API endpoints of the orders API, and we created pydantic schemas to validate request and response payloads. We intentionally skipped implementing the business layer of the application, as that’s a complex task that we’ll tackle in chapter 7.

As a reminder, the endpoints exposed by the orders API are the following:

* /orders—Allows us to retrieve lists (GET) of orders and to place orders (POST)
* /orders/{order\_id}—Allows us to retrieve the details of a specific order (GET), to update an order (PUT), and to delete an order (DELETE)
* /orders/{order\_id}/cancel—Allows us to cancel an order (POST)
* /orders/{order\_id}/pay—Allows us to pay for an order (POST)

POST /orders and PUT /orders/{order\_id} require request payloads that define the properties of an order, and in chapter 2 we implemented schemas for those payloads. What’s missing from the implementation is the URL query parameters for the GET /orders endpoint. Also, the pydantic schemas we implemented in chapter 2 don’t invalidate payloads with illegal properties in the payloads. As we’ll see in section 6.3, this is fine in some situations, but it may lead to integration issues in other cases, and you’ll learn to configure the schemas to invalidate payloads with illegal properties.

If you want to follow along with the examples in this chapter, create a folder called ch06 and copy into it the code from ch02 as ch06/orders. Remember to install the dependencies and activate the virtual environment:

$ mkdir ch06

$ cp -r ch02 ch06/orders

$ cd ch06/orders

$ pipenv install --dev && pipenv shell

You can start the web server by running the following command:

$ uvicorn orders.app:app --reload

**FASTAPI + UVICORN REFRESHER** We implement the orders API using the FastAPI framework, a popular Python framework for building REST APIs. FastAPI is built on top of Starlette, an asynchronous web server implementation. To execute our FastAPI application, we use Uvicorn, another asynchronous server implementation that efficiently handles incoming requests.

The --reload flag makes Uvicorn watch for changes on your files so that any time you make an update, the application is reloaded. This saves you the time of having to restart the server every time you make changes to the code. With this covered, let’s complete the implementation of the orders API!

**6.2 URL query parameters for the orders API**

In this section, we enhance the GET /orders endpoint of the orders API by adding URL query parameters. We also implement validation schemas for the parameters. In chapter 4, we learned that URL query parameters allow us to filter the results of a GET endpoint. In chapter 5, we established that the GET /orders endpoint accepts URL query parameters to filter orders by cancellation and also to limit the list of orders returned by the endpoint.

Listing 6.1 Specification for the GET /orders URL query parameters

# file: orders/oas.yaml

paths:

/orders:

get:

parameters:

- name: cancelled

in: query

required: false

schema:

type: boolean

- name: limit

in: query

required: false

schema:

type: integer

We need to implement two URL query parameters: cancelled (Boolean) and limit (integer). Neither are required, so users must be able to call the GET /orders endpoint without specifying them. Let’s see how we do that.

Implementing URL query parameters for an endpoint is easy with FastAPI. All we need to do is include them in the endpoint’s function signature and use type hints to add validation rules for them. Since the query parameters are optional, we’ll mark them as such using the Optional type, and we’ll set their default values to None.

Listing 6.2 Implementation of URL query parameters for GET /orders

# file: orders/orders/api/api.py

import uuid

from datetime import datetime

**from typing import Optional**

from uuid import UUID

...

@app.get('/orders', response\_model=GetOrdersSchema)

def get\_orders(**cancelled: Optional[bool] = None, limit: Optional[int] = None**):①

...

① We include URL query parameters in the function signature.

Now that we have query parameters available in the GET /orders endpoint, how should we handle them within the function? Since the query parameters are optional, we’ll first check whether they’ve been set. We can do that by checking whether their values are something other than None. Listing 6.3 shows how we can handle URL query parameters within the function body of the GET /orders endpoint. Study figure 6.1 to understand the decision flow for filtering the list of orders based on the query parameters.

Listing 6.3 Implementation of URL query parameters for GET /orders

# file: orders/orders/api/api.py

@app.get('/orders', response\_model=GetOrdersSchema)

def get\_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):

**if cancelled is None and limit is None:**  ①

return {'orders': orders}

**query\_set = [order for order in orders]**  ②

**if cancelled is not None:**  ③

**if cancelled:**

**query\_set = [**

**order**

**for order in query\_set**

**if order['status'] == 'cancelled'**

**]**

**else:**

**query\_set = [**

**order**

**for order in query\_set**

**if order['status'] != 'cancelled'**

**]**

**if limit is not None and len(query\_set) > limit:**  ④

**return {'orders': query\_set[:limit]}**

**return {'orders': query\_set}**

① If the parameters haven’t been set, we return immediately.

② If any of the parameters has been set, we filter list into a query\_set.

③ We check whether cancelled is set.

④ If limit is set and its value is lower than the length of query\_set, we return a subset of query\_set.

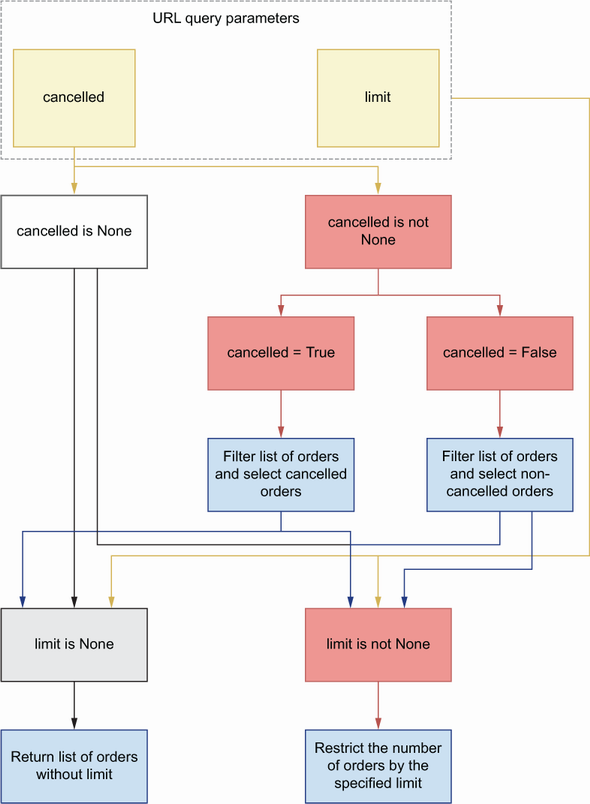


Figure 6.1 Decision flow for filtering orders based on query parameters. If the cancelled parameter is set to True or False, we use it to filter the list of orders. After this step, we check whether the limit parameter is set. If limit is set, we only return the corresponding number of orders from the list.

Now that we know how to add URL query parameters to our endpoints, let’s see how we enhance our validation schemas.

**6.3 Validating payloads with unknown fields**

Until now, our pydantic models have been tolerant with the request payloads. If an API client sends a payload with fields that haven’t been declared in our schemas, the payload will be accepted. As you’ll see in this section, this may be convenient in some cases but misleading or dangerous in other contexts. To avoid integration errors, in this section, we learn how to configure pydantic to forbid the presence of unknown fields. Unknown fields are fields that haven’t been defined in a schema.

**PYDANTIC REFRESHER** As we saw in chapter 2, FastAPI uses pydantic to define validation models for our APIs. Pydantic is a popular data validation library for Python with a modern interface that allows you to define data validation rules using type hints.

In chapter 2, we implemented the schema definitions of the orders API following the tolerant reader pattern (<https://martinfowler.com/bliki/TolerantReader.html>), which follows Postel’s law that recommends to be conservative in what you do and be liberal in what you accept from others.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/06.htm#pgfId-1080830)

In the field of web APIs, this means that we must strictly validate the payloads we send to the client, while allowing for unknown fields in the payloads we receive from API clients. JSON Schema follows this pattern by default, and unless explicitly declared, a JSON Schema object accepts any kind of property. To disallow undeclared properties using JSON Schema, we set additionalProperties to false. If we use model composition, a better strategy is setting unevaluatedProperties to false, since additionalProperties causes conflicts between different models.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/06.htm#pgfId-1080834) OpenAPI 3.1 allows us to use both additionalProperties and unevaluatedProperties, but OpenAPI 3.0 only accepts additionalProperties. Since we’re documenting our APIs using OpenAPI 3.0.3, we’ll ban undeclared properties using additionalProperties:

# file: orders/oas.yaml

GetOrderSchema:

**additionalProperties: false**

type: object

required:

- order

- id

- created

- status

properties:

id:

type: string

format: uuid

...

Check out the orders API specification under ch06/orders/oas.yaml in the GitHub repository for this book to see additional examples of additionalProperties.

The tolerant reader pattern is useful when an API is not fully consolidated or is likely to change frequently and when we want to be able to make changes to it without breaking integrations with existing clients. However, in other cases, like we saw in chapter 2 (section 2.5), the tolerant reader pattern can introduce new bugs or lead to unexpected integration issues.

For example, OrderItemSchema has three properties: product, size, and quantity. product and size are required properties, but quantity is optional, and if missing, the server assigns to it the default value of 1. In some scenarios, this can lead to confusing situations. Imagine a client sends a payload with a typo in the representation of the quantity property, for example with the following payload:

{

"order": [

{

"product": "capuccino",

"size": "small",

**"quantit": 5**

}

]

}

Using the tolerant reader implementation, we ignore the field quantit from the payload, and we assume that the quantity property is missing and set its value to the default of 1. This situation can be confusing for the client, who intended to set a different value for quantity.

**THE API CLIENT SHOULD’VE TESTED THEIR CODE!** You can argue that the client should’ve tested their code and verified that it works properly before calling the server. And you’re right. But in real life, code often goes untested, or is not properly tested, and a little bit of extra validation in the server will help in those situations. If we check the payload for the presence of illegal properties, this error will be caught and reported to the client.

How can we accomplish this using pydantic? To disallow unknown attributes, we need to define a Config class within our models and set the extra property to forbid.

Listing 6.4 Disallowing additional properties in models

# file: orders/orders/api/schemas.py

from datetime import datetime

from enum import Enum

from typing import List, Optional

from uuid import UUID

from pydantic import BaseModel**, Extra,** conint, conlist, validator

...

class OrderItemSchema(BaseModel):

product: str

size: Size

quantity: int = Optional[conint(ge=1, strict=True)] = 1

**class Config:**  ①

**extra = Extra.forbid**

class CreateOrderSchema(BaseModel):

order: List[OrderItemSchema]

**class Config:**

**extra = Extra.forbid**

class GetOrderSchema(CreateOrderSchema):

id: UUID

created: datetime

status: StatusEnum

① We use Config to ban properties that haven’t been defined in the schema.

Let’s test this new functionality. Run the following command to start the server:

$ uvicorn orders.app:app --reload

As we saw in chapter 2, FastAPI generates a Swagger UI from the code, which we can use to test the endpoints. We’ll use this UI to test our new validation rules with the following payload:

{

"order": [

{

"product": "string",

"size": "small",

"**quantit**": 5

}

]

}

**DEFINITION** A *Swagger UI* is a popular style for representing interactive visualizations of REST APIs. They provide a user-friendly interface that helps us understand the API implementation. Another popular UI for REST interfaces is Redoc (<https://github.com/Redocly/redoc>).

To get to the Swagger UI, visit http://127.0.0.1:8000/docs and follow the steps in figure 6.2 to learn how to execute a test against the POST /orders endpoint.

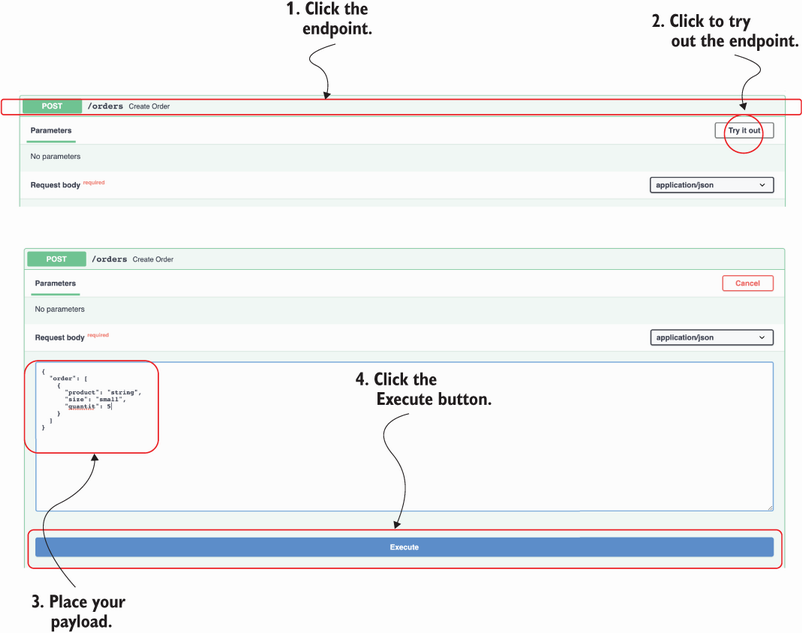


Figure 6.2 Testing the API with the Swagger UI: to test an endpoint, click the endpoint itself, then click the Try it Out button, then click the Execute button.

After running this test, you’ll see that now FastAPI invalidates this payload and returns a helpful 422 response with the following message: “extra fields not permitted.”

**6.4 Overriding FastAPI’s dynamically generated specification**

So far, we’ve relied on FastAPI’s dynamically generated API specification to test, visualize, and document the orders API. The dynamically generated specification is great to understand how we’ve implemented the API. However, our code can contain implementation errors, and those errors can translate to inaccurate documentation. Additionally, API development frameworks have limitations when it comes to generating API documentation, and they typically lack support for certain features of OpenAPI. For example, a common missing feature is documenting OpenAPI links, which we’ll add to our API specification in chapter 12.

To understand how the API is supposed to work, we need to look at our API design document, which lives under orders/oas.yaml, and therefore is the specification we want to show when we deploy the API. In this section, you’ll learn to override FastAPI’s dynamically generated API specification with our API design document.

To load the API specification document, we need PyYAML, which you can install with the following command:

$ pipenv install pyyaml

In the orders/app.py file, we load the API specification, and we overwrite our application’s object openapi property.

Listing 6.5 Overriding FastAPI’s dynamically generated API specification

# file: orders/orders/app.py

from pathlib import Path

**import yaml**

from fastapi import FastAPI

app = FastAPI(debug=True)

**oas\_doc = yaml.safe\_load(**

**(Path(\_\_file\_\_).parent / '../oas.yaml').read\_text()**

**)**  ①

**app.openapi = lambda: oas\_doc**  ②

from orders.api import api

① We load the API specification using PyYAML.

② We override FastAPI’s openapi property so that it returns our API specification.

To be able to test the API using the Swagger UI, we need to add the localhost URL to the API specification. Open the orders/oas.yaml file and add the localhost address to the servers section of the specification:

# file: orders/oas.yaml

servers:

**- url: http://localhost:8000**

**description: URL for local development and testing**

- url: https://coffeemesh.com

description: main production server

- url: https://coffeemesh-staging.com

description: staging server for testing purposes only

By default, FastAPI serves the Swagger UI under the /docs URL, and the OpenAPI specification under /openapi.json. That’s great when we only have one API, but CoffeeMesh has multiple microservice APIs; therefore, we need multiple paths to access each API’s documentation. We’ll serve the orders API’s Swagger UI under /docs/orders, and its OpenAPI specification under /openapi/orders.json. We can override those paths directly in FastAPI’s application object initializer:

# file: orders/app.py

**app = FastAPI(**

**debug=True, openapi\_url='/openapi/orders.json', docs\_url='/docs/orders'**

**)**

This concludes our journey through building the orders API with FastAPI. It’s now time to move on to building the API for the kitchen service, for which we’ll use a new stack: Flask + marshmallow. Let’s get on with it!

**6.5 Overview of the kitchen API**

In this section, we analyze the implementation requirements for the kitchen API. As you can see in figure 6.3, the kitchen service manages the production of customer orders. Customers interface with the kitchen service through the orders service when they place an order or check its status. CoffeeMesh staff can also use the kitchen service to check how many orders are scheduled and to manage them.

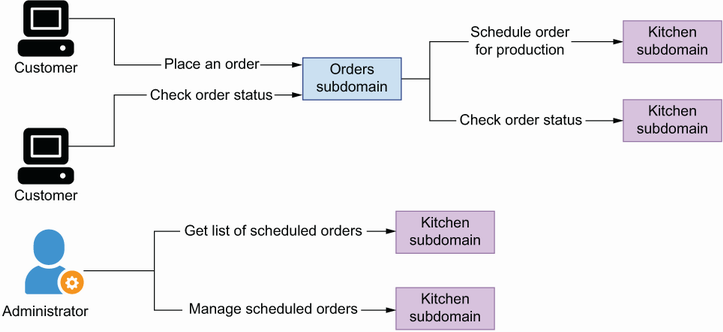


Figure 6.3 The kitchen service schedules orders for production, and it tracks their progress. CoffeeMesh staff members use the kitchen service to manage scheduled orders.

The specification for the kitchen API is provided under ch06/kitchen/oas.yaml in the repository provided with this book. The kitchen API contains four URL paths (see figure 6.4 for additional clarification):

* /kitchen/schedules—Allows us to schedule an order for production in the kitchen (POST) and to retrieve a list of orders scheduled for production (GET)
* /kitchen/schedules/{schedule\_id}—Allows us to retrieve the details of a scheduled order (GET), to update its details (PUT), and to delete it from our records (DELETE)
* /kitchen/schedules/{schedule\_id}/status—Allows us to read the status of an order scheduled for production
* /kitchen/schedules/{schedule\_id}/cancel—Allows us to cancel a scheduled order

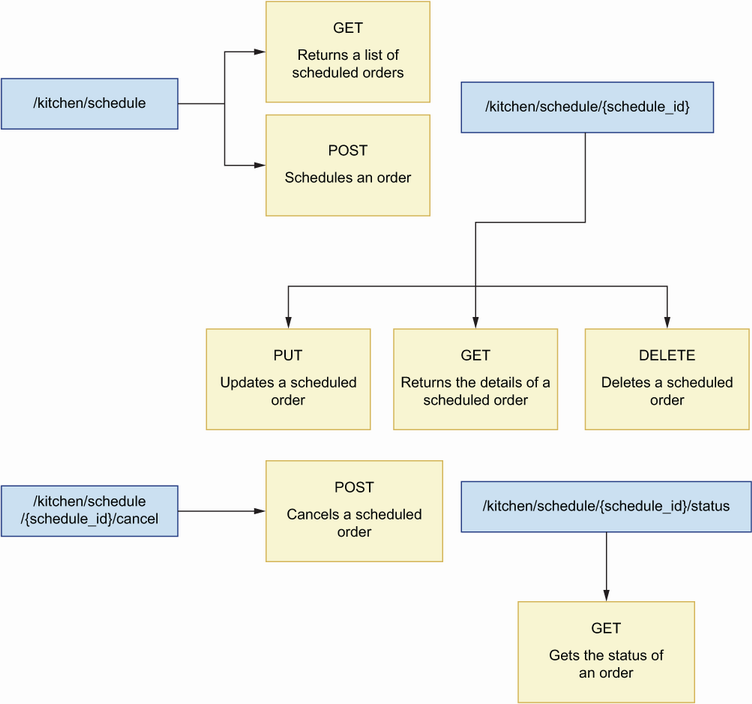


Figure 6.4 The kitchen API has four URL paths: /kitchen/schedules exposes a GET and a POST endpoint; /kitchen/schedules/{schedule\_id} exposes PUT, GET, and DELETE endpoints; /kitchen/schedules/{schedule\_id}/cancel exposes a POST endpoint; and /kitchen/schedules/{schedule\_id}/status exposes a GET endpoint.

The kitchen API contains three schemas: OrderItemSchema, ScheduleOrderSchema, and GetScheduledOrderSchema. The ScheduleOrderSchema represents the payload required to schedule an order for production, while the GetScheduledOrderSchema represents the details of an order that has been scheduled. Just like in the orders API, OrderItemSchema represents the details of each item in an order.

Just as we did in chapter 2, we’ll keep the implementation simple and focus only on the API layer. We’ll mock the business layer with an in-memory representation of the schedules managed by the service. In chapter 7, we’ll learn service implementation patterns that will help us implement the business layer.

**6.6 Introducing flask-smorest**

This section introduces the framework we’ll use to build the kitchen API: flask-smorest (<https://github.com/marshmallow-code/flask-smorest>). Flask-smorest is a REST API framework built on top of Flask and marshmallow. Flask is a popular framework for building web applications, while marshmallow is a popular data validation library that handles the conversion of complex data structures to and from native Python objects. Flask-smorest builds on top of both frameworks, which means we implement our API schemas using marshmallow, and we implement our API endpoints following the patterns of a typical Flask application, as illustrated in figure 6.5. As you’ll see, the principles and patterns we used when we built the orders API with FastAPI can be applied regardless of the framework, and we’ll use the same approach to build the kitchen API with flask-smorest.

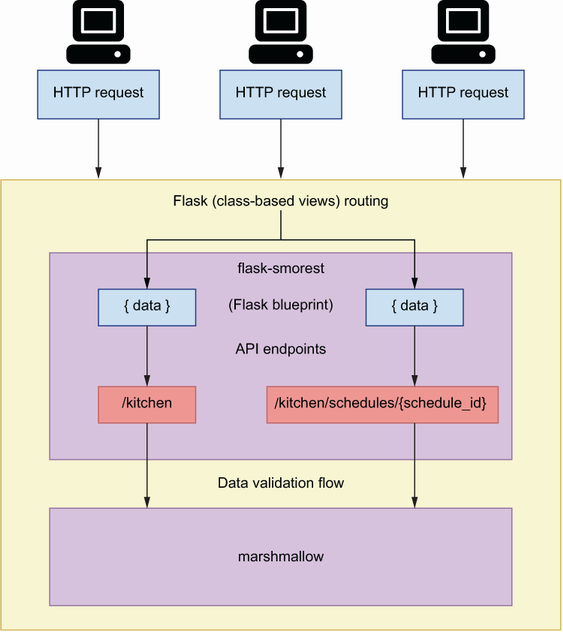


Figure 6.5 Architecture of an application built with flask-smorest. Flask-smorest implements a typical Flask blueprint, which allows us to build and configure our API endpoints just as we would in a standard Flask application.

Building APIs with flask-smorest offers an experience similar to building them with FastAPI, with only two major differences:

* *FastAPI uses pydantic for data validation, while flask-smorest uses marshmallow*. This means that with FastAPI we use native Python-type hints to create data validation rules, while in marshmallow we use field classes.
* *Flask allows us to implement API endpoints with class-based views*. This means that we can use a class to represent a URL path and implement its HTTP methods as methods of the class. Class-based views help you write more structured code and encapsulate the specific behavior of each URL path within the class. In contrast, FastAPI allows you only to define endpoints using functions. Notice that Starlette allows you to implement class-based routes, so this limitation of FastAPI may go away in the future.

With this covered, let’s kick off the implementation of the kitchen API!

**6.7 Initializing the web application for the API**

In this section, we set up the environment to start working on the kitchen API. We’ll also create the entry point for the application and add basic configuration for the web server. In doing so, you’ll learn how to set up a project with flask-smorest and how to inject configuration objects into your Flask applications.

Flask-smorest is built on top of the Flask framework, so we’ll lay out our web application following the patterns of a typical Flask application. Create a folder called ch06/kitchen for the kitchen API implementation. Within that folder, copy the kitchen API specification, which is available under ch06/kitchen/oas.yaml in this book’s GitHub repository. oas.yaml contains the API specification for the kitchen API. Use the cd command to navigate into the ch06/kitchen folder, and run the following commands to install the dependencies that we’ll need to proceed with the implementation:

$ pipenv install flask-smorest

**NOTE** If you want to ensure that you’re installing the same version of the dependencies that I used when writing this chapter, copy the ch06/kitchen/ Pipfile and the ch06/kitchen/Pipfile.lock files from the GitHub repository onto your local machine, and run pipenv install.

Also, run the following command to activate the environment:

$ pipenv shell

Now that we have the libraries we need, let’s create a file called kitchen/app.py. This file will contain an instance of the Flask application object, which represents our web server. We’ll also create an instance of flask-smorest’s Api object, which will represent our API.

Listing 6.6 Initialization of the Flask application object and the Api object

# file: kitchen/app.py

from flask import Flask

from flask\_smorest import Api

app = Flask(\_\_name\_\_) ①

kitchen\_api = Api(app) ②

① We create an instance of the Flask application object.

② We create an instance of flask-smorest’s Api object.

Flask-smorest requires some configuration parameters to work. For example, we need to specify the version of OpenAPI we are using, the title of our API, and the version of our API. We pass this configuration through the Flask application object. Flask offers different strategies for injecting configuration, but the most convenient method is loading configuration from a class. Let’s create a file called kitchen/config.py for our configuration parameters. Within this file we create a BaseConfig class, which contains generic configuration for the API.

Listing 6.7 Configuration for the orders API

# file: kitchen/config.py

class BaseConfig:

API\_TITLE = 'Kitchen API' ①

API\_VERSION = 'v1' ②

OPENAPI\_VERSION = '3.0.3' ③

OPENAPI\_JSON\_PATH = 'openapi/kitchen.json' ④

OPENAPI\_URL\_PREFIX = '/' ⑤

OPENAPI\_REDOC\_PATH = '/redoc' ⑥

OPENAPI\_REDOC\_URL = 'https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js' ⑦

OPENAPI\_SWAGGER\_UI\_PATH = '/docs/kitchen' ⑧

OPENAPI\_SWAGGER\_UI\_URL = 'https://cdn.jsdelivr.net/npm/swagger-ui-

➥ dist/' ⑨

① The title of our API

② The version of our API

③ The version of OpenAPI we are using

④ Path to the dynamically generated specification in JSON

⑤ URL path prefix for the OpenAPI specification file

⑥ Path to the Redoc UI of our API

⑦ Path to a script to be used to render the Redoc UI

⑧ Path to the Swagger UI of our API

⑨ Path to a script to be used to render the Swagger UI

Now that the configuration is ready, we can pass it to the Flask application object.

Listing 6.8 Loading configuration

# file: kitchen/app.py

from flask import Flask

from flask\_smorest import Api

**from config import BaseConfig**  ①

app = Flask(\_\_name\_\_)

**app.config.from\_object(BaseConfig)** ②

kitchen\_api = Api(app)

① We import the BaseConfig class we defined earlier.

② We use the from\_object method to load configuration from a class.

With the entry point for our application ready and configured, let’s move on to implementing the endpoints for the kitchen API!

**6.8 Implementing the API endpoints**

This section explains how we implement the endpoints of the kitchen API using flask-smorest. Since flask-smorest is built on top of Flask, we build the endpoints for our API exactly as we’d do any other Flask application. In Flask, we register our endpoints using Flask’s route decorator:

@app.route('/orders')

def process\_order():

pass

Using the route decorator works for simple cases, but for more complex application patterns, we use Flask blueprints. Flask blueprints allow you to provide specific configuration for a group of URLs. To implement the kitchen API endpoints, we’ll use the flask-smorest’s Blueprint class. Flask-smorest’s Blueprint is a subclass of Flask’s Blueprint, so it provides the functionality that comes with Flask blueprints, enhances it with additional functionality and configuration that generates API documentation, and supplies payload validation models, among other things.

We can use Blueprint’s route decorators to create an endpoint or URL path. As you can see from figure 6.6, functions are convenient for URL paths that only expose one HTTP method. When a URL exposes multiple HTTP methods, it’s more convenient to use class-based routes, which we implement using Flask’s MethodView class.

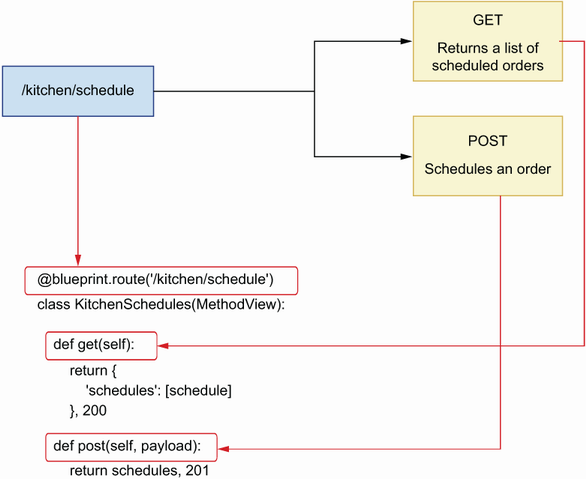


Figure 6.6 When a URL path exposes more than one HTTP method, it’s more convenient to implement it as a class-based view, where the class methods implement each of the HTTP methods exposed.

As you can see in figure 6.7, using MethodView, we represent a URL path as a class, and we implement the HTTP methods it exposes as methods of the class.

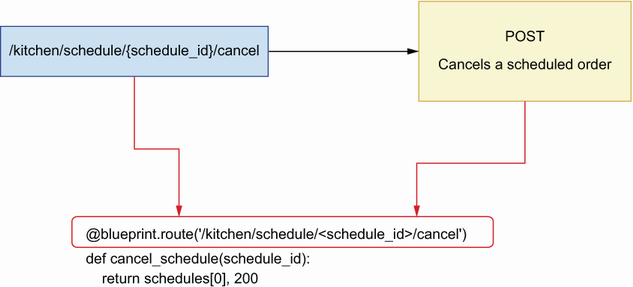


Figure 6.7 When a URL path exposes only one HTTP method, it’s more convenient to implement it as a function-based view.

For example, if we have a URL path /kitchen that exposes GET and POST endpoints, we can implement the following class-based view:

class Kitchen(MethodView):

def get(self):

pass

def post(self):

pass

Listing 6.9 illustrates how we implement the endpoints for the kitchen API using class-based views and function-based views. The content in listing 6.9 goes into the kitchen/api/api.py file. First, we create an instance of flask-smorest’s Blueprint. The Blueprint object allows us to register our endpoints and add data validation to them. To instantiate Blueprint, we must pass two required positional arguments: the name of the Blueprint itself and the name of the module where the Blueprint’s routes are implemented. In this case, we pass the module’s name using the \_\_name\_\_ attribute, which resolves to the name of the file.

Once the Blueprint is instantiated, we register our URL paths with it using the route() decorator. We use class-based routes for the /kitchen/schedules and the /kitchen/schedules/{schedule\_id} paths since they expose more than one HTTP method, and we use function-based routes for the /kitchen/schedules/{schedule\_ id}/cancel and /kitchen/schedules/{schedule\_id}/status paths because they only expose one HTTP method. We return a mock schedule object in each endpoint for illustration purposes, and we’ll change that into a dynamic in-memory collection of schedules in section 6.12. The return value of each function is a tuple, where the first element is the payload and the second is the status code of the response.

Listing 6.9 Implementation of the endpoints of the orders API

# file: kitchen/api/api.py

import uuid

from datetime import datetime

from flask.views import MethodView

from flask\_smorest import Blueprint

blueprint = Blueprint('kitchen', \_\_name\_\_, description='Kitchen API') ①

schedules = [{ ②

'id': str(uuid.uuid4()),

'scheduled': datetime.now(),

'status': 'pending',

'order': [

{

'product': 'capuccino',

'quantity': 1,

'size': 'big'

}

]

}]

@blueprint.route('/kitchen/schedules') ③

class KitchenSchedules(MethodView): ④

def get(self): ⑤

return {

'schedules': schedules

}, 200 ⑥

def post(self, payload):

return schedules[0], 201

@blueprint.route('/kitchen/schedules/<schedule\_id>') ⑦

class KitchenSchedule(MethodView):

def get(self, schedule\_id): ⑧

return schedules[0], 200

def put(self, payload, schedule\_id):

return schedules[0], 200

def delete(self, schedule\_id):

return '', 204

@blueprint.route(

'/kitchen/schedules/<schedule\_id>/cancel', methods=['POST']

) ⑨

def cancel\_schedule(schedule\_id):

return schedules[0], 200

@blueprint.route('/kitchen/schedules/<schedule\_id>/status, methods=[GET])

def get\_schedule\_status(schedule\_id):

return schedules[0], 200

① We create an instance of flask-smorest’s Blueprint class.

② We declare a hardcoded list of schedules.

③ We use the Blueprint’s route() decorator to register a class or a function as a URL path.

④ We implement the /kitchen/schedules URL path as a class-based view.

⑤ Every method view in a class-based view is named after the HTTP method it implements.

⑥ We return both the payload and the status code.

⑦ We define URL parameters within angle brackets.

⑧ We include the URL path parameter in the function signature.

⑨ We implement the /kitchen/schedules/<schedule\_id>/cancel URL path as a function-based view.

Now that we have created the blueprint, we can register it with our API object in the kitchen/app.py file.

Listing 6.10 Registering the blueprint with the API object

# file: kitchen/app.py

from flask import Flask

from flask\_smorest import Api

**from api.api import blueprint**  ①

from config import BaseConfig

app = Flask(\_\_name\_\_)

app.config.from\_object(BaseConfig)

kitchen\_api = Api(app)

**kitchen\_api.register\_blueprint(blueprint)** ②

① We import the blueprint we defined earlier.

② We register the blueprint with the kitchen API object.

Using the cd command, navigate to the ch06/kitchen directory and run the application with the following command:

$ flask run --reload

Just like in Uvicorn, the --reload flag runs the server with a watcher over your files so that the server restarts when you make changes to the code.

If you visit the http://127.0.0.1:5000/docs URL, you’ll see an interactive Swagger UI dynamically generated from the endpoints we implemented earlier. You can also see the OpenAPI specification dynamically generated by flask-smorest under http://127.0.0.1:5000/openapi.json. At this stage in our implementation, it’s not possible to interact with the endpoints through the Swagger UI. Since we don’t yet have marshmallow models, flask-smorest doesn’t know how to serialize data and therefore doesn’t return payloads. However, it’s still possible to call the API using cURL and inspect the responses. If you run curl http://127.0.0.1:5000/kitchen/schedules, you’ll get the mock object we defined in the kitchen/api/api.py module.

Things are looking good, and it’s time to spice up the implementation by adding marshmallow models. Move on to the next section to learn how to do that!

**6.9 Implementing payload validation models with marshmallow**

Flask-smorest uses marshmallow models to validate request and response payloads. In this section, we learn to work marshmallow models by implementing the schemas of the kitchen API. The marshmallow models will help flask-smorest validate our payloads and serialize our data.

As you can see in the kitchen API specification under ch06/kitchen/oas.yaml in this book’s GitHub repository, the kitchen API contains three schemas: ScheduleOrderSchema schema, which contains the details needed to schedule an order; GetScheduledOrderSchema, which represents the details of a scheduled order; and OrderItemSchema, which represents a collection of items in an order. Listing 6.11 shows how to implement these schemas as marshmallow models under kitchen/api/ schemas.py.

To create marshmallow models, we create subclasses of marshmallow’s Schema class. We define the models’ properties with the help of marshmallow’s field classes, such as String and Integer. Marshmallow uses these property definitions to validate a payload against a model. To customize the behavior of marshmallow’s models, we use the Meta class to set the unknown attribute to EXCLUDE, which instructs marshmallow to invalidate the payload with unknown properties.

Listing 6.11 Schema definitions for the orders API

# file: kitchen/api/schemas.py

from marshmallow import Schema, fields, validate, EXCLUDE

class OrderItemSchema(Schema):

class Meta: ①

unknown = EXCLUDE

product = fields.String(required=True)

size = fields.String(

required=True, validate=validate.OneOf(['small', 'medium', 'big'])

)

quantity = fields.Integer(

validate=validate.Range(1, min\_inclusive=True), required=True

)

class ScheduleOrderSchema(Schema):

class Meta:

unknown = EXCLUDE

order = fields.List(fields.Nested(OrderItemSchema), required=True)

class GetScheduledOrderSchema(ScheduleOrderSchema): ②

id = fields.UUID(required=True)

scheduled = fields.DateTime(required=True)

status = fields.String(

required=True,

validate=validate.OneOf(

["pending", "progress", "cancelled", "finished"]

),

)

class GetScheduledOrdersSchema(Schema):

class Meta:

unknown = EXCLUDE

schedules = fields.List(

fields.Nested(GetScheduledOrderSchema), required=True

)

class ScheduleStatusSchema(Schema):

class Meta:

unknown = EXCLUDE

status = fields.String(

required=True,

validate=validate.OneOf(

["pending", "progress", "cancelled", "finished"]

),

)

① We use the Meta class to ban unknown properties.

② We use class inheritance to reuse the definitions of an existing schema.

Now that our validation models are ready, we can link them with our views. Listing 6.12 shows how we use the models to add validation for request and response payloads on our endpoints. To add request payload validation to a view, we use the blueprint’s arguments() decorator in combination with a marshmallow model. For response payloads, we use the blueprint’s response() decorator in combination with a marshmallow model.

By decorating our methods and functions with the blueprint’s response() decorator, we no longer need to return a tuple of payload plus a status code. Flask-smorest takes care of adding the status code for us. By default, flask-smorest adds a 200 status code to our responses. If we want to customize that, we simply need to specify the desired status code using the status\_code parameter in the decorator.

While the blueprint’s arguments() decorator validates and deserializes a request payload, the blueprint’s response() decorator doesn’t perform validation and only serializes the payload. We’ll discuss this feature in more detail in section 6.11, and we’ll see how we can ensure that data is validated before being serialized.

Listing 6.12 Adding validation to the API endpoints

# file: kitchen/api/api.py

import uuid

from datetime import datetime

from flask.views import MethodView

from flask\_smorest import Blueprint

**from api.schemas import (**

**GetScheduledOrderSchema,**

**ScheduleOrderSchema,**

**GetScheduledOrdersSchema,**

**ScheduleStatusSchema,** ①

**)**

blueprint = Blueprint('kitchen', \_\_name\_\_, description='Kitchen API')

...

@blueprint.route('/kitchen/schedulles')

class KitchenSchedules(MethodView):

**@blueprint.response(status\_code=200, schema=GetScheduledOrdersSchema)** ②

def get(self):

**return {'schedules': schedules}**

**@blueprint.arguments(ScheduleOrderSchema)**  ③

**@blueprint.response(status\_code=201, schema=GetScheduledOrderSchema)** ④

def post(self, payload):

return schedules[0]

@blueprint.route('/kitchen/schedules/<schedule\_id>')

class KitchenSchedule(MethodView):

**@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)**

def get(self, schedule\_id):

return schedules[0]

**@blueprint.arguments(ScheduleOrderSchema)**

**@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)**

def put(self, payload, schedule\_id):

return schedules[0]

**@blueprint.response(status\_code=204)**

def delete(self, schedule\_id):

return

**@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)**

@blueprint.route(

'/kitchen/schedules/<schedule\_id>/cancel', methods=['POST']

)

def cancel\_schedule(schedule\_id):

return schedules[0]

**@blueprint.response(status\_code=200, schema=ScheduleStatusSchema)**

@blueprint.route(

'/kitchen/schedules/<schedule\_id>/status', methods=['GET']

)

def get\_schedule\_status(schedule\_id):

return schedules[0]

① We import our marshmallow models.

② We use the blueprint’s response() decorator to register a marshmallow model for the response payload.

③ We use the blueprint’s arguments() decorator to register a marshmallow model for the request payload.

④ We set the status\_code parameter to the desired status code.

To see the effects of the new changes in the implementation, visit http://127.0.0.1:5000/docs URL again. If you’re running the server with the --reload flag, the changes will be automatically reloaded. Otherwise, stop the server and run it again. As you can see in figure 6.8, flask-smorest now recognizes the validation schemas that need to be used in the API, and therefore they’re represented in the Swagger UI. If you play around with the UI now, for example by hitting the GET /kitchen/schedules endpoint, you’ll be able to see the response payloads.

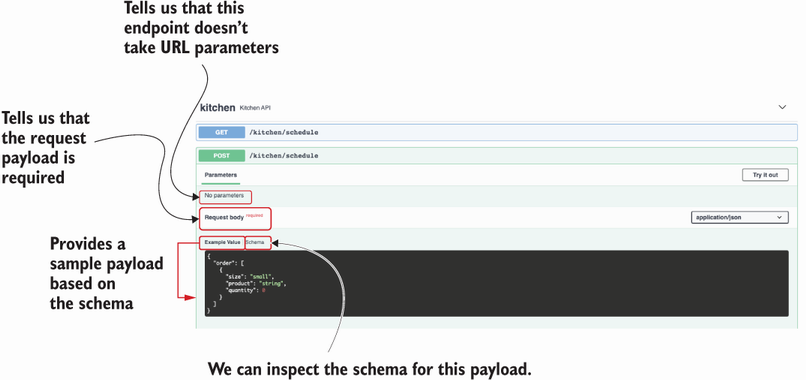


Figure 6.8 The Swagger UI shows the schema for the request for the POST /kitchen/schedules endpoint’s payload and provides an example of it.

The API is looking good, and we are nearly finished with the implementation. The next step is adding URL query parameters to the GET /kitchen/schedules endpoint. Move on to the next section to learn how to do that!

**6.10 Validating URL query parameters**

In this section, we learn how to add URL query parameters to the GET /kitchen/ schedules endpoint. As shown in listing 6.13, the GET /kitchen/schedules endpoint accepts three URL query parameters:

* progress *(Boolean)*—Indicates whether an order is in progress.
* limit *(integer)*—Limits the number of results returned by the endpoint.
* since *(date-time)*—Filters results by the time when the orders were scheduled. A date in date-time format is an ISO date with the following structure: YYYY-MM-DDTHH:mm:ssZ. An example of this date format is 2021-08-31T01:01:01Z. For more information on this format, see [https://tools.ietf.org/html/rfc3339 #section-5.6](https://tools.ietf.org/html/rfc3339#section-5.6).

Listing 6.13 Specification for the GET /kitchen/schedules URL query parameters

# file: kitchen/oas.yaml

paths:

/kitchen/schedules:

get:

summary: Returns a list of orders scheduled for production

parameters:

- name: progress

in: query

description: >-

Whether the order is in progress or not.

In progress means it's in production in the kitchen.

required: false

schema:

type: boolean

- name: limit

in: query

required: false

schema:

type: integer

- name: since

in: query

required: false

schema:

type: string

format: 'date-time'

How do we implement URL query parameters in flask-smorest? To begin, we need to create a new marshmallow model to represent them. We define the URL query parameters for the kitchen API using marshmallow. You can add the model for the URL query parameters to kitchen/api/schemas.py with the other marshmallow models.

Listing 6.14 URL query parameters in marshmallow

# file: kitchen/api/schemas.py

from marshmallow import Schema, fields, validate, EXCLUDE

...

**class GetKitchenScheduleParameters(Schema):**

**class Meta:**

**unknown = EXCLUDE**

**progress = fields.Boolean()** ①

**limit = fields.Integer()**

**since = fields.DateTime()**

① We define the fields of the URL query parameters.

We register the schema for URL query parameters using the blueprint’s arguments() decorator. We specify that the properties defined in the schema are expected in the URL, so we set the location parameter to query.

Listing 6.15 Adding URL query parameters to GET /kitchen/schedules

# file: kitchen/api/api.py

import uuid

from datetime import datetime

from flask.views import MethodView

from flask\_smorest import Blueprint

from api.schemas import (

GetScheduledOrderSchema, ScheduleOrderSchema, GetScheduledOrdersSchema,

ScheduleStatusSchema, **GetKitchenScheduleParameters**  ①

)

blueprint = Blueprint('kitchen', \_\_name\_\_, description='Kitchen API')

...

@blueprint.route('/kitchen/schedules')

class KitchenSchedules(MethodView):

**@blueprint.arguments(GetKitchenScheduleParameters, location='query')** ②

@blueprint.response(status\_code=200, schema=GetScheduledOrdersSchema)

def get(self, **parameters**): ③

return schedules

...

① We import the marshmallow model for URL query parameters.

② We register the model using the arguments() decorator and set the location parameter to query.

③ We capture URL query parameter in the function signature.

If you reload the Swagger UI, you’ll see that the GET /kitchen/schedules endpoint now accepts three optional URL query parameters (shown in figure 6.9). We should pass these parameters to our business layer, which will use them to filter the list of results. URL query parameters come in the form of a dictionary. If the user didn’t set any query parameters, the dictionary will be empty and therefore and evaluate to False. Since URL query parameters are optional, we check for their presence by using the dictionary’s get() method. Since get() returns None when a parameter isn’t set, we know that a parameter is set when its value isn’t None. We won’t be implementing the business layer until chapter 7, but we can use query parameters to filter our in-memory list of schedules.

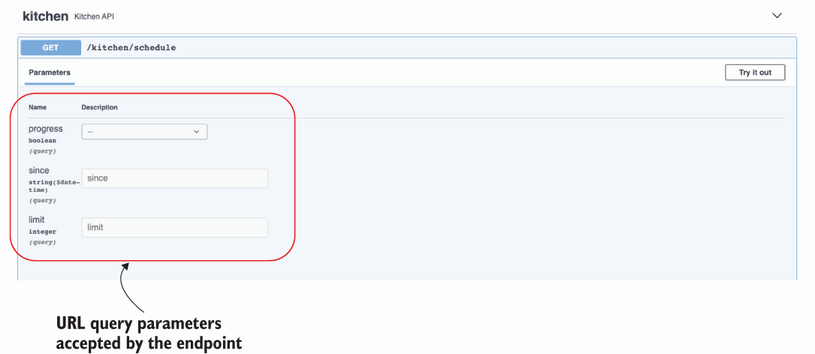


Figure 6.9 The Swagger UI shows the URL query parameters of the GET /kitchen/schedules endpoint, and it offers form fields that we can fill in to experiment with different values.

Listing 6.16 Use filters in GET /kitchen/schedules

# file: kitchen/api/api.py

...

@blueprint.route('/kitchen/schedules')

class KitchenSchedules(MethodView):

@blueprint.arguments(GetKitchenScheduleParameters, location='query')

@blueprint.response(status\_code=200, schema=GetScheduledOrdersSchema)

def get(self, parameters):

**if not parameters:**  ①

**return {'schedules': schedules}**

**query\_set = [schedule for schedule in schedules]**  ②

**in\_progress = parameters.get(progress)**  ③

**if in\_progress is not None:**

**if in\_progress:**

**query\_set = [**

**schedule for schedule in schedules**

**if schedule['status'] == 'progress'**

**]**

**else:**

**query\_set = [**

**schedule for schedule in schedules**

**if schedule['status'] != 'progress'**

**]**

**since = parameters.get('since')**

**if since is not None:**

**query\_set = [**

**schedule for schedule in schedules**

**if schedule['scheduled'] >= since**

**]**

**limit = parameters.get('limit')**

**if limit is not None and len(query\_set) > limit:** ④

**query\_set = query\_set[:limit]**

**return {'schedules': query\_set}**  ⑤

**...**

① If no parameter is set, we return the full list of schedules.

② If the user set any URL query parameters, we use them to filter the list of schedules.

③ We check for the presence of each URL query parameter by using the dictionary’s get() method.

④ If limit is set and its value is lower than the length of query\_set, we return a subset of query\_set.

⑤ We return the filtered list of schedules.

Now that we know how to handle URL query parameters with flask-smorest, there’s one more topic we need to cover, and that is data validation before serialization. Move on to the next section to learn more about this!

**6.11 Validating data before serializing the response**

Now that we have schemas to validate our request payloads and we have hooked them up with our routes, we have to ensure that our response payloads are also validated. In this section, we learn how to use marshmallow models to validate data. We’ll use this functionality to validate our response payloads, but you could use the same approach to validate any kind of data, such as configuration objects.

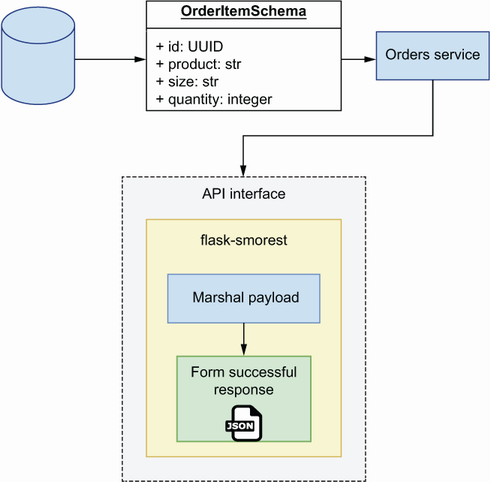


Figure 6.10 Workflow of a data payload with the flask-smorest framework. Response payloads are supposed to come from a “trusted zone,” and therefore are not validated before marshalling.

When we send a payload in a response, flask-smorest serializes the payload using marshmallow. However, as shown in figure 6.10, it doesn’t validate if it’s correctly formed.[**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/06.htm#pgfId-1081568) As you can see in figure 6.11, in contrast to marshmallow, FastAPI does validate our data before it’s serialized for a response.

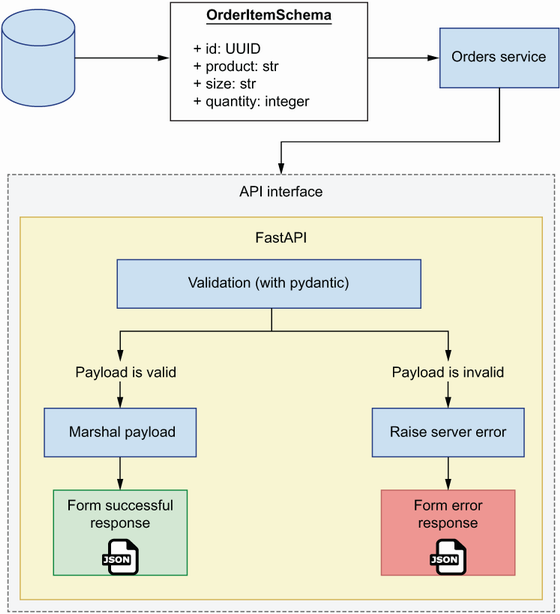


Figure 6.11 Workflow of a data payload with the FastAPI framework. Before marshalling a response, FastAPI validates that the payload conforms to the specified schema.

The fact that marshmallow doesn’t perform validation before serialization is not necessarily undesirable. In fact, it can be argued that it’s a desirable behavior, as it decouples the task of serializing from the task of validating the payload. There are two rationales to justify why marshmallow doesn’t perform validation before serialization (<http://mng.bz/9Vwx>):

* It improves performance, since validation is slow.
* Data coming from the server is supposed to be trusted and therefore shouldn’t require validation.

The reasons the maintainers of marshmallow use to justify this design decision are fair. However, if you’ve worked with APIs, and websites in general, long enough, you know there’s generally very little to be trusted, even from within your own system.

**ZERO-TRUST APPROACH FOR ROBUST APIS** API integrations fail due to the server sending the wrong payload as much as they fail due to the client sending malformed payloads to the server. Whenever possible, it’s good practice to take a zero-trust approach to our systems design and validate all data, regardless of its origin.

The data that we send from the kitchen API comes from a database. In chapter 7, we’ll learn patterns and techniques to ensure that our database contains the right data in the right format. However, and even under the strictest access security measures, there’s always a chance that malformed data ends up in the database. As unlikely as this is, we don’t want to ruin the user experience if that happens, and validating our data before serializing helps us with that.

Thankfully, it’s easy to validate data using marshmallow. We simply need to get an instance of the schema we want to validate against and use its validate() method to pass in the data we need to validate. validate() doesn’t raise an exception if it finds errors. Instead, it returns a dictionary with the errors, or an empty dictionary if no errors are found. To get a feeling for how this works, open a Python shell by typing python in the terminal, and run the following code:

>>> from api.schemas import GetScheduledOrderSchema

>>> GetScheduledOrderSchema().validate({'id': 'asdf'})

{'order': ['Missing data for required field.'], 'scheduled': ['Missing

➥ data for required field.'], 'status': ['Missing data for required

➥ field.'], 'id': ['Not a valid UUID.']}

After importing the schema on line 1, in line 2 we pass a malformed representation of a schedule containing only the id field, and in line 3 marshmallow helpfully reports that the order, scheduled, and status fields are missing, and that the id field is not a valid UUID. We can use this information to raise a helpful error message in the server, as shown in listing 6.17. We validate schedules in the GET /kitchen/schedules method view before building and returning the query set, and we iterate the list of schedules to validate one at a time. Before validation, we make a deep copy of the schedule so that we can transform its datetime object into an ISO date string, since that’s the format expected by the validation method. If we get a validation error, we raise marshmallow’s ValidationError exception, which automatically formats the error message into an appropriate HTTP response.

Listing 6.17 Validating data before serialization

# file: kitchen/api/api.py

import copy

import uuid

from datetime import datetime

from flask.views import MethodView

from flask\_smorest import Blueprint

**from marshmallow import ValidationError**  ①

...

@blueprint.route('/kitchen/schedules')

class KitchenSchedules(MethodView):

@blueprint.arguments(GetKitchenScheduleParameters, location='query')

@blueprint.response(status\_code=200, schema=GetScheduledOrdersSchema)

def get(self, parameters):

for schedule in schedules:

schedule = copy.deepcopy(schedule)

schedule['scheduled'] = schedule['scheduled'].isoformat()

**errors = GetScheduledOrderSchema().validate(schedule)**  ②

**if errors:**

**raise ValidationError(errors)**  ③

**...**

return {'schedules': query\_set}

...

① We import the ValidationError class from marshmallow.

② We capture validation errors in the errors variable.

③ If validate() finds errors, we raise a ValidationError exception.

Please be aware that there are known issues with validation in marshmallow, especially when your models contain complex configurations for determining which fields should be serialized and which fields shouldn’t (see <https://github.com/marshmallow-code/marshmallow/issues/682> for additional information). Also, take into account that validation is known to be a slow process, so if you are handling large payloads, you may want to use a different tool to validate your data, validate only a subset of your data, or skip validation altogether. However, whenever possible, you’re better off performing validation on your data.

This concludes the implementation of the functionality of the kitchen API. However, the API is still returning the same mock schedule across all endpoints. Before concluding this chapter, let’s add a minimal implementation of an in-memory list of schedules so that we can make our API dynamic. This will allow us to verify that all endpoints are functioning as intended.

**6.12 Implementing an in-memory list of schedules**

In this section, we implement a simple in-memory representation of schedules so that we can obtain dynamic results from the API. By the end of this section, we’ll be able to schedule orders, update them, and cancel them through the API. Because the schedules are managed as an in-memory list, any time the server is restarted, we’ll lose information from our previous session. In the next chapter, we’ll address this problem by adding a persistence layer to our service.

Our in-memory collection of schedules will be represented by a Python list, and we’ll simply add and remove elements from it in the API layer. Listing 6.18 shows the changes that we need to make to kitchen/api/api.py to make this possible. We initialize an empty list and assign it to a variable named schedules. We also refactor our data validation code into an independent function named validate\_schedule() so that we can reuse it in other view methods or functions. When a schedule payload arrives in the KitchenSchedules’ post() method, we set the server-side attributes, such as the ID, the scheduled time, and the status. In the singleton endpoints, we look for the requested schedule by iterating the list of schedules and checking their IDs. If the requested schedule isn’t found, we return a 404 response.

Listing 6.18 In-memory implementation of schedules

# file: kitchen/api/api.py

**import copy**

import uuid

from datetime import datetime

**from flask import abort**

...

**schedules = []**  ①

**def validate\_schedule(schedule):**  ②

**schedule = copy.deepcopy(schedule)**

**schedule['scheduled'] = schedule['scheduled'].isoformat()**

**errors = GetScheduledOrderSchema().validate(schedule)**

**if errors:**

**raise ValidationError(errors)**

@blueprint.route('/kitchen/schedules')

class KitchenSchedules(MethodView):

@blueprint.arguments(GetKitchenScheduleParameters, location='query')

@blueprint.response(GetScheduledOrdersSchema)

def get(self, parameters):

...

@blueprint.arguments(ScheduleOrderSchema)

@blueprint.response(status\_code=201, schema=GetScheduledOrderSchema,)

def post(self, payload):

**payload['id'] = str(uuid.uuid4())**  ③

**payload['scheduled'] = datetime.utcnow()**

**payload['status'] = 'pending'**

**schedules.append(payload)**

**validate\_schedule(payload)**

**return payload**

@blueprint.route('/kitchen/schedules/<schedule\_id>')

class KitchenSchedule(MethodView):

@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)

def get(self, schedule\_id):

**for schedule in schedules:**

**if schedule['id'] == schedule\_id:**

**validate\_schedule(schedule)**

**return schedule**

**abort(404, description=f'Resource with ID {schedule\_id} not found')**④

@blueprint.arguments(ScheduleOrderSchema)

@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)

def put(self, payload, schedule\_id):

**for schedule in schedules:**

**if schedule['id'] == schedule\_id:**

**schedule.update(payload)**  ⑤

**validate\_schedule(schedule)**

**return schedule**

**abort(404, description=f'Resource with ID {schedule\_id} not found')**

@blueprint.response(status\_code=204)

def delete(self, schedule\_id):

**for index, schedule in enumerate(schedules):**

**if schedule['id'] == schedule\_id:**

**schedules.pop(index)**  ⑥

**return**

**abort(404, description=f'Resource with ID {schedule\_id} not found')**

@blueprint.response(status\_code=200, schema=GetScheduledOrderSchema)

@blueprint.route(

'/kitchen/schedules/<schedule\_id>/cancel', methods=['POST']

)

def cancel\_schedule(schedule\_id):

**for schedule in schedules:**

**if schedule['id'] == schedule\_id:**

**schedule['status'] = 'cancelled'**  ⑦

**validate\_schedule(schedule)**

**return schedule**

**abort(404, description=f'Resource with ID {schedule\_id} not found')**

@blueprint.response(status\_code=200, schema=ScheduleStatusSchema)

@blueprint.route(

'/kitchen/schedules/<schedule\_id>/status', methods=['GET']

)

def get\_schedule\_status(schedule\_id):

**for schedule in schedules:**

**if schedule['id'] == schedule\_id:**

**validate\_schedule(schedule)**

**return {'status': schedule['status']}**

**abort(404, description=f'Resource with ID {schedule\_id} not found')**

① We initialize schedules as an empty list.

② We refactor our data validation code into a function.

③ We set the server-side attributes of a schedule, such as the ID.

④ If a schedule isn’t found, we return a 404 response.

⑤ When a user updates a schedule, we update the schedule’s properties with the contents of the payload.

⑥ We remove the schedule from the list and return an empty response.

⑦ We set the status of the schedule to cancelled.

If you reload the Swagger UI and test the endpoints, you’ll see you’re now able to add schedules, update them, cancel them, list and filter them, get their details, and delete them. In the next section, you’ll learn to override flask-smorest’s dynamically generated API specification to make sure we serve our API design instead of our implementation.

**6.13 Overriding flask-smorest’s dynamically generated API specification**

As we learned in section 6.4, API specifications dynamically generated from code are good for testing and visualizing our implementation, but to publish our API, we want to make sure we serve our API design document. To do that, we’ll override flask-smorest’s dynamically generated API documentation. First, we need to install PyYAML, which we’ll use to load the API design document:

$ pipenv install pyyaml

We override the API object’s spec property with a custom APISpec object. We also override APISpec’s to\_dict() method so that it returns our API design document.

Listing 6.19 Overriding flask-smorest’s dynamically generated API specification

# file: kitchen/app.py

**from pathlib import Path**

**import yaml**

**from apispec import APISpec**

from flask import Flask

from flask\_smorest import Api

from api.api import blueprint

from config import BaseConfig

app = Flask(\_\_name\_\_)

app.config.from\_object(BaseConfig)

kitchen\_api = Api(app)

kitchen\_api.register\_blueprint(blueprint)

**api\_spec = yaml.safe\_load((Path(\_\_file\_\_).parent / "oas.yaml").read\_text())**

spec = APISpec(

title=api\_spec["info"]["title"],

version=api\_spec["info"]["version"],

openapi\_version=api\_spec["openapi"],

)

**spec.to\_dict = lambda: api\_spec**

**kitchen\_api.spec = spec**

This concludes our journey through implementing REST APIs using Python. In the next chapter, we’ll learn patterns to implement the rest of the service following best practices and useful design patterns. Things are spicing up!

**Summary**

* You can build REST APIs in Python using frameworks like FastAPI and flask-smorest, which have great ecosystems of tools and libraries that make it easier to build APIs.
* FastAPI is a modern API framework that makes it easier to build highly performant and robust REST APIs. FastAPI is built on top of Starlette and pydantic. Starlette is a highly performant asynchronous server framework, and pydantic is a data validation library that uses type hints to create validation rules.
* Flask-smorest is built on top of Flask and works as a Flask blueprint. Flask is one of Python’s most popular frameworks, and by using flask-smorest you can leverage its rich ecosystem of libraries to make it easier to build APIs.
* FastAPI uses pydantic for data validation. Pydantic is a modern framework that uses type hints to define validation rules, which results in cleaner and easy-to-read code. By default, FastAPI validates both request and response payloads.
* Flask-smorest uses marshmallow for data validation. Marshmallow is a battle-tested framework that uses class fields to define validation rules. By default, flask-smorest doesn’t validate response payloads, but you can validate responses by using marshmallow models’ validate() method.
* With flask-smorest, you can use Flask’s MethodView to create class-based views that represent URL paths. In a class-based view, you implement HTTP methods as methods of the class, such as get() and post().
* The tolerant reader pattern follows Postel’s law, which recommends being tolerant with errors in HTTP requests and validating response payloads. When designing your APIs, you must balance the benefits of the tolerant reader pattern with risk of integration failure due to bugs like typos.

**1** Jon Postel, Ed., “Transmission Control Protocol,” RFC 761, p. 13, <https://tools.ietf.org/html/rfc761>.

**2** To understand why additionalProperties doesn’t work when using model composition, see the excellent discussion about this topic in JSON Schema’s GitHub repository: <https://github.com/json-schema-org/json-schema-spec/issues/556>.

**3** Before version 3.0.0, marshmallow used to perform validation before serialization (see the change log: <https://github.com/marshmallow-code/marshmallow/blob/dev/CHANGELOG.rst#300-2019-08-18>)