BLOG ARTICLES

A microservices example: writing a simple to-do application

By Saurabh Badhwar
September 15, 2016

Microservices are becoming a new trend, thanks to the modularity and granularity they provide on top of advantages like releasing applications in a continuous manner. There are various platforms and projects that are rising which aims to make writing and managing microservices easy.

Keeping that in mind, I thought, why not make a demo application that can give an example of how microservices are built and how they interact. In this article, I will be building a small application using the Microservice Architecture (MSA).

The application will be a super simple To-Do management list. So, let's have take a look at what we are going to build and how we are going to build.

Editor's note: This article references Fedora, which is the upstream project for Red Hat Enterprise Linux (RHEL) – now free for developers. This tutorial should also work on RHEL, just replace 'dnf' with 'yum' wherever appropriate.

To-Do Manager: A super simple Microservices Example

So, talking about our application, it is leveraging the MSA where the whole application is divided into a set of services that specialize in doing a specific task using a simple set of protocols. All the communication between different services occur over the network.

Now, for building our Application, we will be making use of Python. Our application uses Flask as the framework for getting the basic things up. Currently, the application uses a few files in the JSON format which serves as the database for our application. For the time being, the application to most part is static in nature.

So, let's talk about the architecture of the application. Currently, our application consists of two services namely the User service and the To-Do service:

- User Service: The user service provides a RESTful endpoint to list the users in our application and also allows to query the user lists based on their usernames. This service currently runs on port 5000 of our server.

- To-Do Service: The ToDo service provides a RESTful endpoint to list all the lists as well as providing the list of projects filtered on the basis of usernames. This service runs on port 5001 of our server.

The application can be located at ToDo Manager, feel free to clone, fork, modify, and extend.

So, let's setup our development environment and get set up with the application.

Set up the development environment

As a personal preference, I use virtual environments for building different python applications since it removes the risk of messing up with the libraries globally on my development system. For this tutorial, I will be using Fedora 24 Workstation. So, let's setup our environment by getting the required tools and setting our virtual environment. Run the following command to get the virtual environment.

```
sudo dnf install python-virtualenv
```

The next step is to create our project directory

```
mkdir todo && cd todo
```

Now, let's setup our virtual environment and install the required dependencies for our application:

```
virtualenv venv
```

The above command will create a virtual environment named venv under the application directory.

Next, we need to install the dependencies for our application. Our application is current dependent on two libraries – Flask and requests. Here, Flask as I have introduced is a web framework and requests is a library which allows us to make HTTP requests.

Before installing the dependencies, we need to activate our virtual environment. So, let's do it.

```
source venv/bin/activate
```

The above command activates our virtual environment and now we need to install the dependencies. Run the below commands to get the dependencies installed in our virtual environment.

```
pip install flask requests
```

So, now we are done with setting up our development environment. The next step is to setup our directory structure for the application. Currently, our application has two directories namely database and services. The database directory hosts the files containing some dummy data for the users and todo lists made by the users.

The services directory holds the code for our individual services – in this case the user service and todo service.

So, before we start coding our services, let's get the database setup.

Create a file named *users.json* under the database directory and add the following to the file:

```
{
    "saurabh_badhwar": {
        "id":1,
        "name":"Saurabh Badhwar",
        "verified":1
```

```
    },
    "aniket": {
        "id":2,
        "name":"Aniket Sharma",
        "verified":1
    },
    "luckas": {
        "id":4,
        "name":"Luckas Friendel",
        "verified":0
    }
}
```

The next thing we have to do is create another file named *todo.json* which contains the data of our lists. Create the file and add the following data to it:

```
{
    "saurabh_badhwar": {
        "home": [
            "Buy milk",
            "Look for pest control service",
            "Get a new carpet"
        ],
        "work": [
            "Complete the blogpost",
            "Create presentation for meeting"
        ]
    },
    "aniket": {
        "school": [
            "Complete homework",
            "Prepare for test"
        ]
    }
}
```

So, now we are done with the database part for our application. Next, we have to build our services. So, let's start with writing our User service.

Under the services directory, create a file named users.py and write the code for it:

So, let's start with the code, first we import the dependencies for the service

```
from flask import Flask, jsonify, make_response
import requests
```

```
import os
import simplejson as json
```

The next we do is to initialize our flask service

*app = Flask(**name**)*

Now, we import our users database in the service and parse it as a JSON file

```
with open("{}/database/users.json".format(database_path), "r") as f:
    usr = json.load(f)
```

The next step is to write our application endpoints, a simple hello world entrypoint can be created as:

```
@app.route("/", methods=['GET'])
def hello():
    ''' Greet the user '''

    return "Hey! The service is up, how about doing something useful"
```

The @app.route helps set the application route. The above example helps us setup a hello world application point. When a user visits our application at http://localhost:5000 the user will be greeted with the message we have specified.

Using the same technique we can come up with the other end points for our service. Writing the complete code doesn't seem feasible for this post. You can refer to the repository link provided above for the complete code of the application.

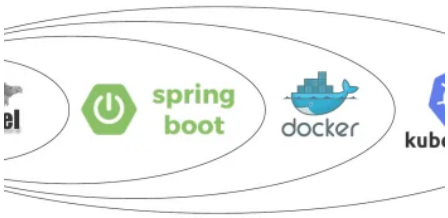
The final step in writing the service will be to run the server as soon as the application is called. This can be achieved by the following set of code for our user microservice

```
if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

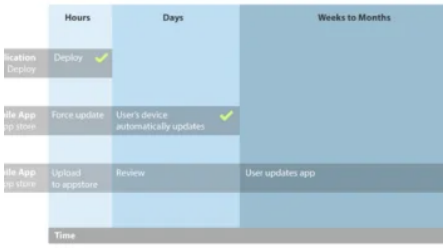
For the ToDo service and rest of the code for the User service, you can look up the repository.

If you found this article interesting, or built something that you want to deploy, head over and look at Red Hat OpenShift, which provides a great platform to host and manage your Microservices.

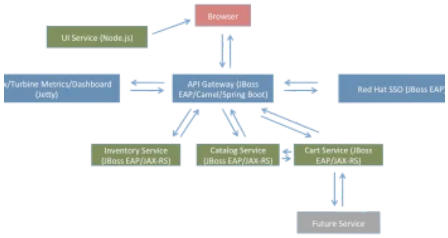
Related



Spring Cloud for Microservices Compared to Kubernetes
December 9, 2016
In "Containers"



Evolving a Mobile-centric Architecture: The Microservices Way
April 15, 2015
In "Mobile"



Microservices: Comparing DIY with Apache Camel
November 7, 2016
In "Accelerated Development and Management"

Categorized In

Microservices, Python, Red Hat OpenShift Container Platform

Tags

Applications microservices python RHEL

Red Hat Developer Comment Policy

Please keep your comments relevant and polite. Opinions shared in comments are not official Red Hat news or policy. Please read our Comment Policy before commenting.

Recent Posts

Broadening compiler checks for buffer overflows in _FORTIFY_SOURCE
Using the SystemTap Dyninst runtime environment
Deploying the Mosquitto MQTT message broker on Red Hat OpenShift, Part 1
Containerize .NET for Red Hat OpenShift: Linux containers and .NET Core
Vulnerability analysis for Golang applications with Red Hat CodeReady Dependency Analytics
Fail fast with Opossum circuit breaker in Node.js
Using a custom devfile registry and C++ with Red Hat CodeReady Workspaces
Mandrel: A specialized distribution of GraalVM for Quarkus
How to pick the right container base image
C# 9 new features for methods and functions

Top Posts

How to install Python 3 on Red Hat Enterprise Linux

Top 10 must-know Kubernetes design patterns

How to install Java 8 and 11 on Red Hat Enterprise Linux 8

Introduction to Linux interfaces for virtual networking

Podman and Buildah for Docker users

Feeds

RSS ATOM

LOGIN TO ADMIN YOUR BLOG