

Spring Boot and Flask Microservice Discovery with Netflix Eureka



David Landup

Introduction

In this guide, we'll be utilizing [Netflix Eureka](#), a microservice discovery service to combine a Spring Boot microservice with a Flask microservice, bridging services written in totally different programming languages and frameworks.

We'll be building two services - The *End-User Service*, which is a Spring Boot service oriented at the end-user, that collects data and sends it to the *Data-Aggregation Service* - a Python service, using Pandas to perform data aggregation, and return a JSON response to the *End-User Service*.

Netflix Eureka Service Discovery

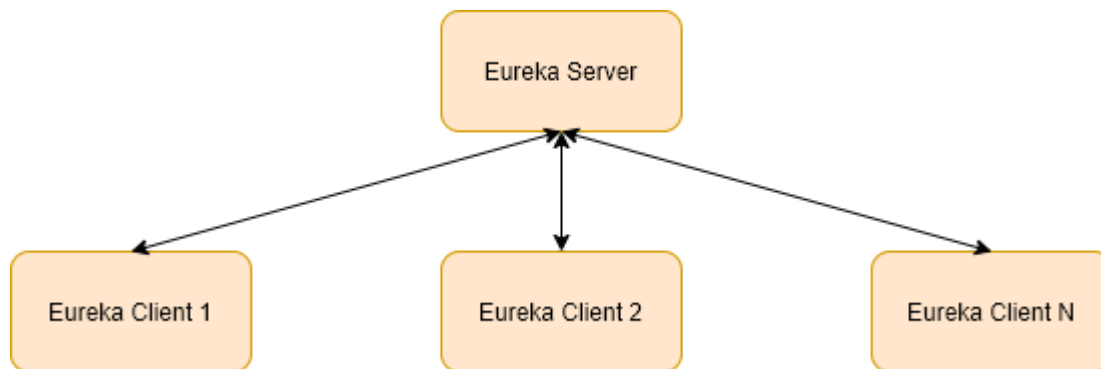
When switching from a monolith codebase to a microservice-oriented architecture - Netflix built a plethora of tools that helped them overhaul their entire architecture. One of the in-house solutions, which was subsequently released to the public is known as *Eureka*.

Netflix Eureka is a *service discovery* tool (also known as a *lookup server* or *service registry*), that allows us to register multiple microservices, and handles request

routing between them.

It's a central hub where each service is registered, and each of them communicates with the rest through the hub. Instead of sending REST calls via hostnames and ports - we delegate this to Eureka, and simply call the *name* of the service, as registered in the hub.

To achieve this, a typical architecture consists of a few elements:



You can spin off the Eureka Server in any language that has a Eureka wrapper, though, it's most naturally done in Java, through Spring Boot, since this is the original implementation of the tool, with official support.

Each Eureka Server can register *N* Eureka Clients, each of which is typically an individual project. These can also be done in any language or framework, so each microservice uses what's most suitable for their task.

We'll have two clients:

- **End-User Service** (Java-based Eureka Client)

- **Data-Aggregation Service** (Python-based Eureka Client)

Since Eureka is a Java-based project, originally meant for Spring Boot solutions - it doesn't have an *official* implementation for Python. However, we can use a community-driven Python wrapper for it:

- **Netflix Eureka**
- **Python's Eureka Client**

With that in mind, let's create a *Eureka Server* first.

Creating a Eureka Server

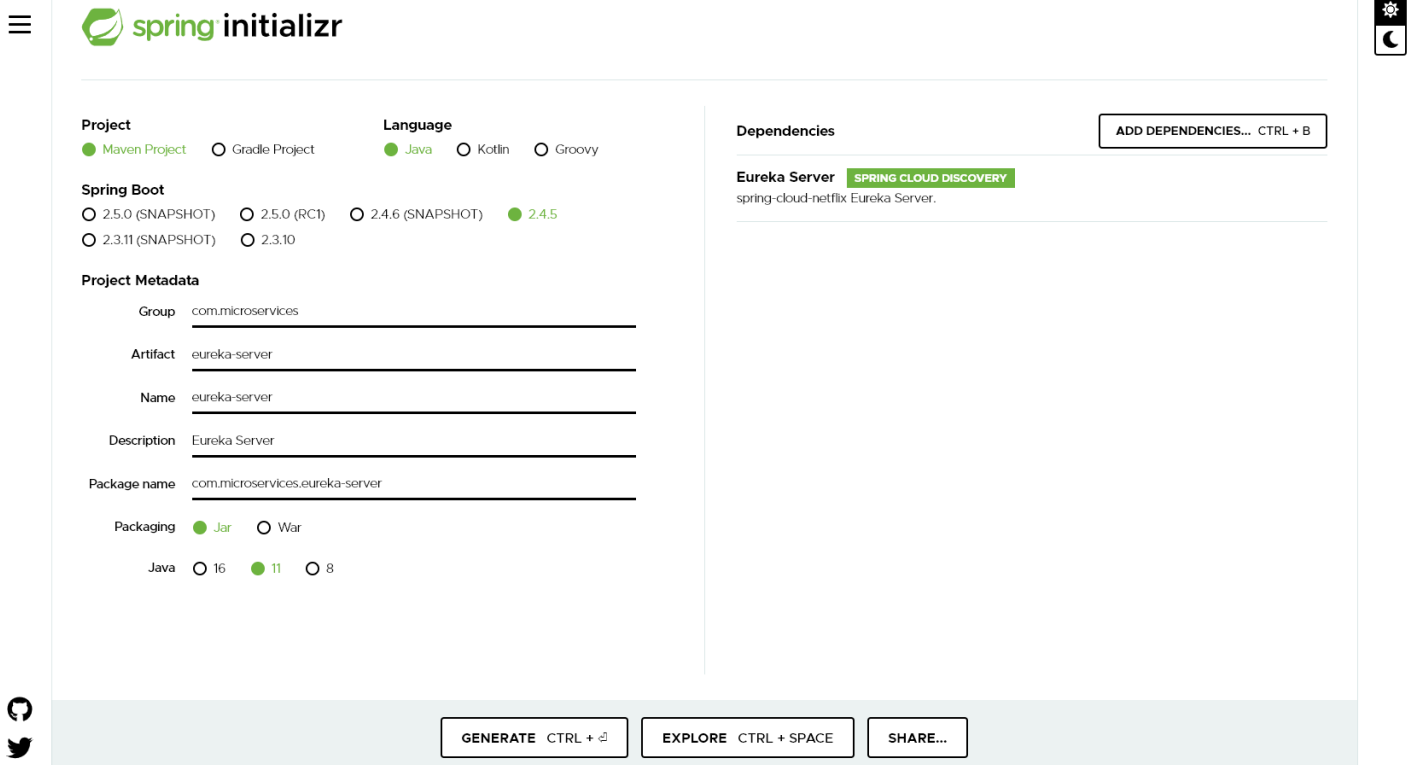
We'll be using Spring Boot to create and maintain our Eureka Server. Let's start off by making a directory to house our three projects, and within it a directory for our server:

```
$ mkdir eureka-microservices
$ cd eureka-microservices
$ mkdir eureka-server
$ cd eureka-server
```

The **eureka-server** directory will be the root directory of our Eureka Server. You can start a Spring Boot project here through the CLI:

```
$ spring init -d=spring-cloud-starter-eureka-server
```

Alternatively, you can use **Spring Initializr** and include the *Eureka Server* dependency:



The image shows the Spring Initializr web interface for configuring a new project. The 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.4.5' selected. The 'Project Metadata' section includes fields for Group (com.microservices), Artifact (eureka-server), Name (eureka-server), Description (Eureka Server), and Package name (com.microservices.eureka-server). The 'Packaging' section has 'Jar' selected, and the 'Java' version is set to '11'. The 'Dependencies' section shows 'Eureka Server' with the 'SPRING CLOUD DISCOVERY' dependency. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'.

If you already have a project and just wish to include the new dependency, if you're using Maven, add:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
  <version>${version}</version>
</dependency>
```

Or if you're using Gradle:

```
implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-eureka-server'
```

Regardless of the initialization type - the Eureka Server requires a *single* annotation to be marked as a server.

In your `EndUserApplication` file class, which is our entry-point with the `@SpringBootApplication` annotation, we'll just add an `@EnableEurekaServer`:

```
@SpringBootApplication
@EnableEurekaServer
```

```
public class EurekaServerApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(EurekaServerApplication.class, args);  
    }  
}
```

The default port for Eureka Servers is **8761**, and it's also recommended by the Spring Team. Though, for good measure, let's set it in the **application.properties** file as well:

```
server.port=8761
```

With that done, our server is ready to run. Running this project will start up the Eureka Server, available at **localhost:8761**:

The screenshot shows the Spring Eureka web interface. The top navigation bar includes the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment and data center details (all N/A) and system metrics like current time, uptime, lease expiration, and renewals.
- DS Replicas:** A section showing the local host 'localhost' as a data source.
- Instances currently registered with Eureka:** A table with columns for Application, AMIs, Availability Zones, and Status. It currently shows 'No instances available'.
- General Info:** A table listing various system metrics such as total-avail-memory, num-of-cpus, current-memory-usage, server-upptime, registered-replicas, unavailable-replicas, and available-replicas.
- Instance Info:** A section at the bottom for detailed instance information.



Note: Without registering any services, Eureka may incorrectly claim an *UNKNOWN* instance is up.

Creating a Eureka Client - End User Service in Spring Boot

Now, with our server spun up and ready to register services, let's go ahead and make our *End-User Service* in Spring Boot. It'll have a single endpoint that accepts JSON data regarding a *Student*. This data is then sent as JSON to our *Data Aggregation Service* that calculates general statistics of the grades.

In practice, this operation would be replaced with much more labor-intensive operations, which make sense to be done in dedicated data processing libraries and which justify the use of another service, rather than performing them on the same one.

That being said, let's go back and create a directory for our *End-User Service*:

```
$ cd..
$ mkdir end-user-service
$ cd end-user-service
```

Here, let's start a new project via the CLI, and include the `spring-cloud-starter-netflix-eureka-client` dependency. We'll also add the `web` dependency since this application will actually be facing the user:

```
$ spring init -d=web, spring-cloud-starter-netflix-eureka-client
```

Alternatively, you can use [Spring Initializr](#) and include the *Eureka Discovery Client* dependency:

The screenshot shows the Spring Initializr web application. The interface includes a sidebar with a hamburger menu and a settings icon. The main content area is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for **2.5.0 (SNAPSHOT)**, **2.5.0 (RC1)**, **2.4.6 (SNAPSHOT)**, **2.4.5** (selected), **2.3.11 (SNAPSHOT)**, and **2.3.10**.
- Project Metadata:** Includes input fields for **Group** (com.microservices), **Artifact** (end-user-service), **Name** (end-user-service), **Description** (Java-Based End User Service), and **Package name** (com.microservices.end-user-service).
- Dependencies:** Includes a button **ADD DEPENDENCIES... CTRL + B**. Below this, there are two dependency cards:
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Eureka Discovery Client** (SPRING CLOUD DISCOVERY): A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

The footer includes social media icons (GitHub, Twitter) and three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

If you're using Maven, add the following:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  <version>${version}</version>
</dependency>
```

Or if you're using Gradle:

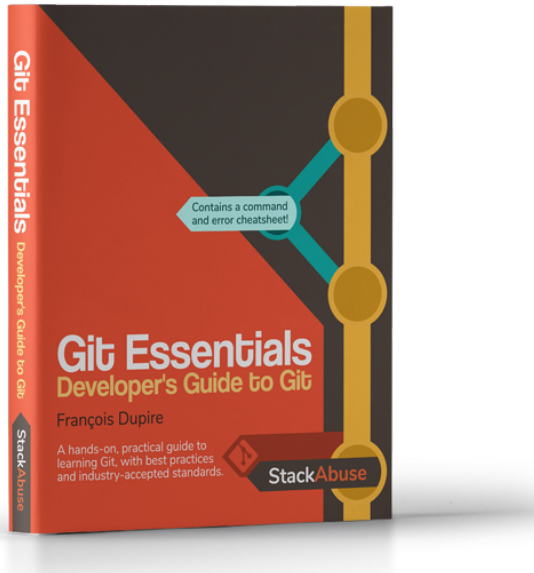
```
implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-netf
```

Regardless of the initialization type - to mark this application as a Eureka Client, we simply add the `@EnableEurekaClient` annotation to the main class:

```
@SpringBootApplication
@EnableEurekaClient
public class EndUserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(EndUserServiceApplication.class, args);
    }

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Note: Alternatively, you can use the `@EnableDiscoveryClient` annotation, which is a more wide-encompassing annotation. It can refer to Eureka, Consul or Zookeeper, depending on which tool is being used.



Free eBook: Git Essentials

Check out our hands-on, practical guide to learning Git, with best-practices, industry-accepted standards, and included cheat sheet. Stop Googling Git commands and actually *learn* it!

[Download the eBook](#)

We've also defined a `@Bean` here, so that we can `@Autowired` the `RestTemplate` later on in our controller. This `RestTemplate` will be used to send a `POST` request to the *Data Aggregation Service*. The `@LoadBalanced` annotation signifies that our `RestTemplate` should use a `RibbonLoadBalancerClient` when sending requests.

Since this application is a Eureka Client, we'll want to give it a *name* for the registry. Other services will refer to this name when relying on it. The name is defined in the `application.properties` or `application.yml` file:

```
server.port = 8060
spring.application.name = end-user-service
eureka.client.serviceUrl.defaultZone = http://localhost:8761/eureka
```

```
server:
  port: 8060
spring:
  application:
```

```

    name: end-user-service
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

```

Here, we've set the port for our application, which Eureka needs to know to route requests to it. We've also specified the name of the service, which will be referenced by other services.

Running this application will register the service to the Eureka Server:

```

INFO 3220 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat s
INFO 3220 --- [          main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating
INFO 3220 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    : Discover
INFO 3220 --- [          main] c.m.e.EndUserServiceApplication        : Started
INFO 3220 --- [tbeatExecutor-0] com.netflix.discovery.DiscoveryClient    : Discover
INFO 3220 --- [tbeatExecutor-0] com.netflix.discovery.DiscoveryClient    : Discover
INFO 3220 --- [tbeatExecutor-0] com.netflix.discovery.DiscoveryClient    : Discover

```

Now, if we visit `localhost:8761`, we'll be able to see it registered on the server:

The screenshot shows the Spring Eureka web interface. The top navigation bar includes the Spring Eureka logo and links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into several sections:

- System Status:** A table showing environment (N/A), data center (N/A), current time (2021-04-29T10:49:34 +0200), uptime (00:00), lease expiration enabled (false), renew threshold (5), and renewals (last min) (0).
- DS Replicas:** A section showing the local host (localhost) as a data center replica.
- Instances currently registered with Eureka:** A table with columns for Application, AMIs, Availability Zones, and Status. It shows one instance of END-USER-SERVICE registered at localhost:8060.
- General Info:** A table showing various system metrics such as total-avail-memory (304mb), num-of-cpus (16), current-memory-usage (40mb (13%)), server-uptime (00:00), registered-replicas (http://localhost:8761/eureka/), unavailable-replicas (http://localhost:8761/eureka/), and available-replicas.

Now, let's go ahead and define a `Student` model:

```
public class Student {  
    private String name;  
    private double mathGrade;  
    private double englishGrade;  
    private double historyGrade;  
    private double scienceGrade;  
  
    // Constructor, getters and setters and toString()  
}
```

For a student, we'll want to calculate some *summary statistics* of their performance, such as the *mean*, *minimum* and *maximum* of their grades. Since we'll be using Pandas for this - we'll leverage the very handy `DataFrame.describe()` function. Let's make a `GradesResult` model as well, that'll hold our data once returned from the *Data Aggregation Service*:

```
public class GradesResult {  
    private Map<String, Double> mathGrade;  
    private Map<String, Double> englishGrade;  
    private Map<String, Double> historyGrade;  
    private Map<String, Double> scienceGrade;  
  
    // Constructor, getters, setters and toString()  
}
```

With the models done, let's make a really simple `@RestController` that accepts a `POST` request, deserializes it into a `Student` and sends it off to the *Data Aggregation* service, which we haven't made yet:

```
        return ResponseEntity
            .status(HttpStatus.OK)
            .body(String.format("Sent the Student to the Data Aggregation Service:
    }
}
```

This `@RestController` accepts a `POST` request, and deserializes its body into a `Student` object. Then, we're sending a request to our `data-aggregation-service`, which isn't yet implemented, as it will be registered on Eureka, and we pack the JSON results of that call into our `GradesResult` object.

Note: If the serializer has issues with constructing the `GradesResult` object from the given result, you'll want to manually convert it using Jackson's `ObjectMapper`:

```
String result = restTemplate.postForObject("http://data-aggregation-service/
ObjectMapper objectMapper = new ObjectMapper();
GradesResult gradesResult = objectMapper.readValue(result, GradesResult.class);
```

Finally, we print the `student` instance we've sent as well as the `grades` instance we constructed from the result.

Now, let's go ahead and create the *Data Aggregation Service*.

Creating a Eureka Client - Data Aggregation Service in Flask

The only missing component is the *Data Aggregation Service*, which accepts a *Student*, in JSON format and populates a Pandas `DataFrame`, performs certain operations and returns the result back.

Let's create a directory for our project and start a virtual environment for it:

```
$ cd..  
$ mkdir data-aggregation-service  
$ python3 -m venv flask-microservice
```

Now, to activate the virtual environment, run the `activate` file. On Windows:

```
$ flask-microservice/Scripts/activate.bat
```

On Linux/Mac:

```
$ source flask-microservice/bin/activate
```

We'll be spinning up a simple Flask application for this, so let's install the dependencies for both Flask and Eureka via `pip` in our activated environment:

```
(flask-microservice) $ pip install flask pandas py-eureka-client
```

And now, we can create our Flask application:

```
$ touch flask_app.py
```

Now, open the `flask_app.py` file and import Flask, Pandas and the Py-Eureka Client libraries:

```
from flask import Flask, request  
import pandas as pd  
import py_eureka_client.eureka_client as eureka_client
```

We'll be using Flask and `request` to handle our incoming requests and return a response, as well as spin up a server. We'll be using Pandas to aggregate data, and we'll use the `py_eureka_client` to register our Flask application to the Eureka Server on `localhost:8761`.

Let's go ahead and set this application up as a Eureka Client and implement a `POST` request handler for the student data:

```
rest_port = 8050
eureka_client.init(eureka_server="http://localhost:8761/eureka",
                  app_name="data-aggregation-service",
                  instance_port=rest_port)

app = Flask(__name__)

@app.route("/calculateGrades", methods=['POST'])
def hello():
    data = request.json
    df = pd.DataFrame(data, index=[0])
    response = df.describe().to_json()
    return response

if __name__ == "__main__":
    app.run(host='0.0.0.0', port = rest_port)
```

Note: We have to set the host to `0.0.0.0` to open it to external services, lest Flask refuse them to connect.

This is a pretty minimal Flask app with a single `@app.route()`. We've extracted the incoming `POST` request body into a `data` dictionary through `request.json`, after which we've made a `DataFrame` with that data.

Since this dictionary doesn't have an index at all, we've manually set one.

Finally, we've returned the `describe()` function's results as JSON. We haven't used `jsonify` here since it returns a `Response` object, not a String. A `Response` object, when sent back would contain extra `\` characters:

```
{\\"mathGrade\\":...}  
vs  
{"mathGrade":...}
```

These would have to be escaped, lest they throw off the deserializer.

In the `init()` function of `eureka_client`, we've set the URL to our Eureka Server, as well as set the name of the application/service for discovery, as well as supplied a port on which it'll be accessible. This is the same information we've provided in the Spring Boot application.

Now, let's run this Flask application:

```
(flask-microservice) $ python flask_app.py
```

And if we check our Eureka Server on `localhost:8761`, it's registered and ready to receive requests:

The screenshot shows the Spring Eureka web interface. The top navigation bar includes the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details.

Environment	N/A	Current time	2021-04-29T13:37:13 +0200
Data center	N/A	Uptime	02:48
		Lease expiration enabled	true
		Renews threshold	5
		Renews (last min)	12
- DS Replicas:** A section showing the local data store status, currently 'localhost'.
- Instances currently registered with Eureka:** A table listing registered services.

Application	AMIs	Availability Zones	Status
DATA-AGGREGATION-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.2:data-aggregation-service:8050
END-USER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-8HAKM3G:end-user-service:8060
- General Info:** A table showing system metrics.

Name	Value
total-avail-memory	304mb
num-of-cpus	16
current-memory-usage	81mb (26%)
server-uptime	02:48
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/
available-replicas	

Calling Flask Service from Spring Boot Service using Eureka

With both of our services up and running, registered to Eureka and able to communicate with each other, let's send a **POST** request to our *End-User Service*, containing some student data, which will in turn send a **POST** request to the *Data Aggregation Service*, retrieve the response, and forward it to us:

```
8\"", \"englishGrade\" : \"10\", \"historyGrade\" : \"7\", \"scienceGrade\" : \"10\"]
```

This results in a response from the server to the end-user:

```
Sent the Student to the Data Aggregation Service: Student{name='David', mathGrade=8
And got back:
GradesResult{mathGrade={count=1.0, mean=8.0, std=null, min=8.0, 25%=8.0, 50%=8.0, 7
```

Conclusion

In this guide, we've created a microservice environment, where one service relies on another, and hooked them up using Netflix Eureka.

These services are built using different frameworks, and different programming languages - though, through REST APIs, communicating between them is straightforward and easy.

The source code for these two services, including the Eureka Server is available on [GitHub](#).

#python #java #flask #spring boot #microservices #eureka

Last Updated: March 20th, 2023

Was this article helpful? ☆☆☆☆☆



You might also like...

- [Guide to Spring Cloud Task: Short-Lived Spring Boot Microservices](#)
- [Build a Spring Boot REST API with Java - Full Guide](#)
- [Spring Boot with Redis: HashOperations CRUD Functionality](#)
- [Thymeleaf Path Variables with Spring Boot](#)
- [Spring Security: In-Memory Invalidation of JWT Tokens During User Logout](#)

Improve your dev skills!

Get tutorials, guides, and dev jobs in your inbox.

Enter your email

[Sign Up](#)

No spam ever. Unsubscribe at any time. Read our [Privacy Policy](#).

David Landup *Author*



Entrepreneur, Software and Machine Learning Engineer, with a deep fascination towards the application of Computation and Deep Learning in Life Sciences (Bioinformatics, Drug Discovery, Genomics), Neuroscience (Computational Neuroscience), robotics and BCIs.

Great passion for accessible education and promotion of reason, science, humanism, and progress.



Matplotlib and Pandas

Course

Data Visualization in Python with Matplotlib and Pandas

#python #pandas #matplotlib

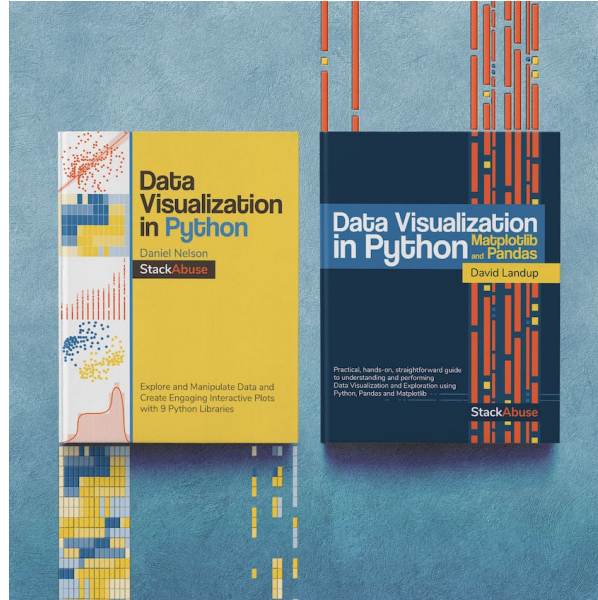
Data Visualization in Python with Matplotlib and Pandas is a course designed to take absolute beginners to Pandas and Matplotlib, with basic Python knowledge, and...



David Landup

Details →

Make Clarity from Data - Quickly Learn Data Visualization with Python



Learn the landscape of Data Visualization tools in Python - work with **Seaborn**, **Plotly**, and **Bokeh**, and excel in **Matplotlib**!

From simple plot types to ridge plots, surface plots and spectrograms - understand your data and learn to draw conclusions from it.

[Learn more](#)



© 2013-2023 Stack Abuse. All rights reserved.

[About](#) | [Disclosure](#) | [Privacy](#) | [Terms](#)