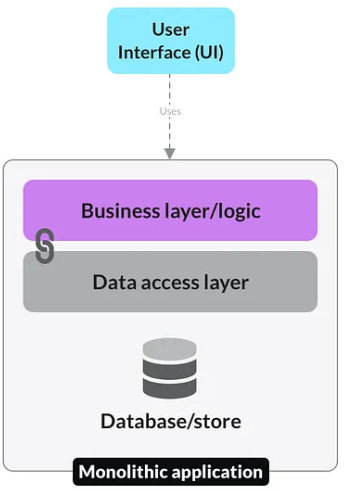
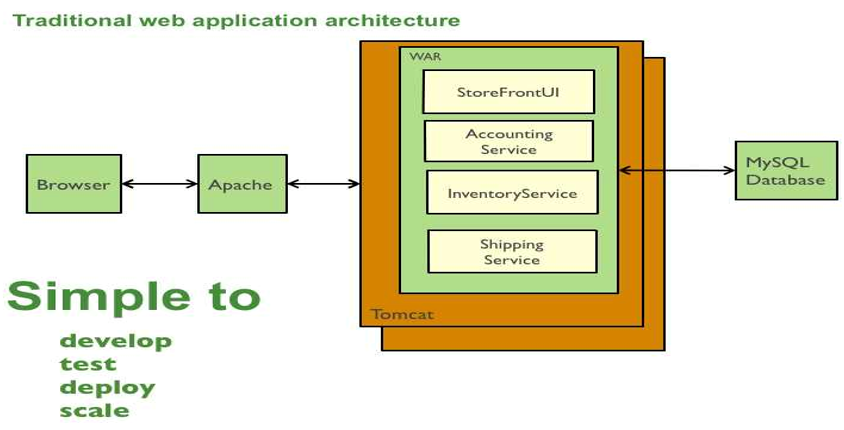
A monolithic application is built as a single unit. Enterprise applications are built in three parts:



* A database — consisting of many tables usually in a relational database management system
* A client-side user interface — consisting of HTML pages and/or JavaScript running in a browser)
* A server-side application — which will handle HTTP requests, execute domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser.



it is a single logical executable. To make any changes to the system, a developer must build and deploy an updated version of the server-side application.

## Pros of Monolithic Applications

**Easier to develop and deploy:**

**A simple codebase that’s easy to work on.** Modular codebases inside monolithic applications are quick to set up and easy for developers to reason about. The whole app is compiled into a single executable, making it easy to access data and functions without worrying about inter-process communication

**Quick to deploy with fewer security concerns.** Setting up your [continuous integration and delivery](https://en.wikipedia.org/wiki/CI/CD) pipeline is much simpler for a monolithic application, as there is only one executable and one service to deploy. It’s far easier to make security considerations as there are fewer services that could be compromised, with less sensitive data flowing through the network.

## Cons of Monolithic Applications

**Complex maintenance:** As an application grows and adds features, a monolithic codebase may become enormous and complex. This makes the program difficult to maintain, especially as the number of developers working on the single codebase increases.

**Difficulty in implementing changes:** Changes to one component of the program may accidentally affect other parts of the codebase, needing greater wo rk to identify problems.

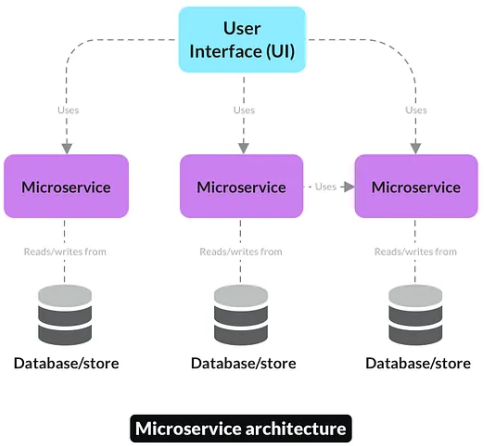
**Scaling horizontally becomes a challenge as the load increases.**

Monolithic applications are typically [scaled vertically](https://www.section.io/blog/scaling-horizontally-vs-vertically) by adding CPU and memory to serve more concurrent requests. Scaling monolithic applications horizontally (adding identical instances) is inefficient, making it difficult to scale them beyond a certain threshold.

**Single point of failure:** Since all parts of a program are inextricably connected, a problem anywhere in the code may bring the entire application down.

**Difficult to adopt and test new technologies.** Switching programming languages or frameworks requires rewriting the entire monolithic application, which may be impossible due to time or funding limitations. This makes it difficult to test new technologies without spending a lot of effort on transitioning to them.

Microservice architectures include a series of independently developed and deployable services. They are [loosely coupled](https://en.wikipedia.org/wiki/Loose_coupling) with separate responsibilities and concerns.



Microservices can be written in different programming languages and use different technologies, as long as they can communicate with each other through a common API.

Basically, Microservice following characteristics:

* Each microservice can have its own database.
* Each microservice should be developed independently
* Each microservice should be deployed independently
* Each microservice should be scaled independently

## Pros of Microservices Applications:

1. **Microservices are self-contained:** Because microservices are self-contained, they may be debugged, deployed, and maintained separately from other modules.
2. **Services can handle a large number of requests with horizontal scaling.**

Microservices can auto scale horizontally once they have been made [stateless](https://sparkequation.com/2020/11/12/stateless-vs-stateful-microservices-addressing-the-benefits-and-quandaries), allowing them to handle much higher request volume at a lower cost.

1. **Multiple languages and frameworks can be used.** [Polyglot](https://www.dictionary.com/browse/polyglot) architectures allow teams to choose their own programming languages and technologies. This means they can select the best tool for the job rather than being constrained by languages that have been used in the past.

## Cons of Microservices Applications:

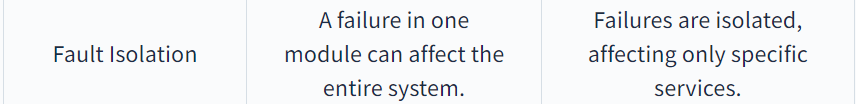
1. **Making changes across multiple services is complex.** When making a change you must consider the impact it will have on all the upstream and downstream services involved.
2. **Database transactions across many services are difficult.**
3. **Debugging and testing services can be challenging.**

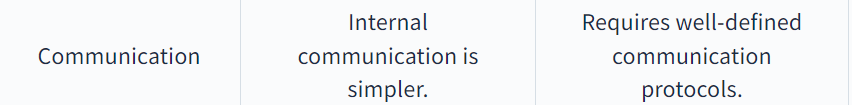
## Key Difference Between Monolithic Vs Microservices Architecture

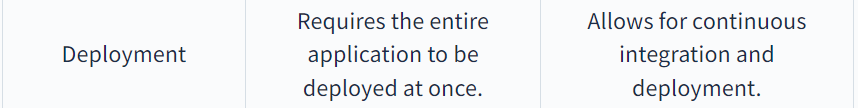
In Monolithic Architecture, the entire application is built as a single, tightly integrated unit, making it simpler to develop and initially deploy. However, it can become complex to maintain and scale as the application grows.

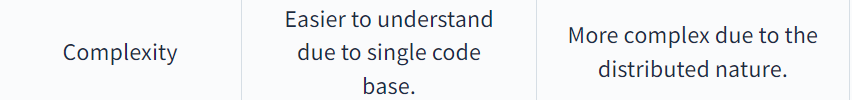
Microservices Architecture divides the application into small, independent modules that can be developed, deployed, and scaled individually.

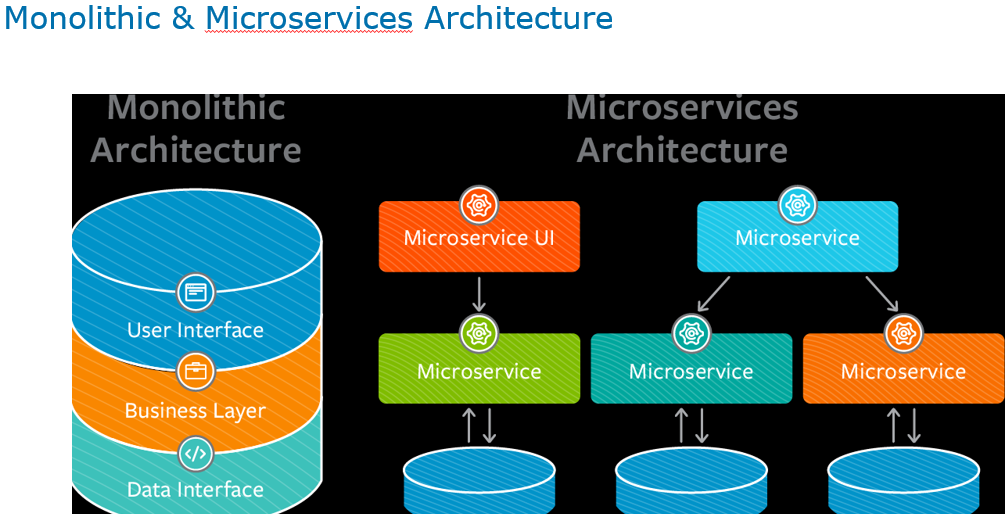




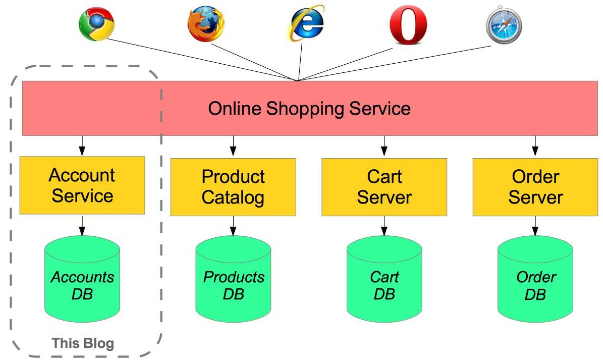








**online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:**



### **Spring Cloud**

**Spring Cloud provides various tools or modules for developers to build common design patterns to solve different infrastructural concerns in Microservices projects and focus on their main business problems.**

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems or Microservices projects.

For example, common patterns such as configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, and cluster state.

Spring Cloud offers below features:

* Distributed/versioned configuration
* Service registration and discovery
* Routing
* Service-to-service calls
* Load balancing
* Circuit Breakers
* Distributed messaging
* API Gateway
* Distributed tracing

**Create Microservice #1(Eureka Server)**

In order to discover and communicate Microservices with each other, we need to create a Eureka Server Service. Creating a Eureka Server is itself similar to creating a Microservice. Moreover, it is just a Spring Boot Project that incorporates Spring Cloud’s Eureka Server dependency. In ‘application. properties’ file we will have some specific properties that will indicate that this application/microservice is a Eureka server

### Step #1: Create a Spring Boot Project

While creating a project in STS, add starter ‘Eureka Server’ in order to get features of it.

### Step #2: Apply Annotation @EnableEurekaServer at the main class

In order to make your application/microservice acts as Eureka server, you need to apply @EnableEurekaServer at the main class of your application



### Step #3: Modify application.properties file

**server.port=8761**

**eureka.client.register-with-eureka=false**

**eureka.client.fetch-registry=false**

#### eureka.client.register-with-eureka=false

Default value of property ‘eureka.client.register-with-eureka’ is true. Please note that this property is mandatory to include in Eureka Server in order to make its value as false. However, this is optional to add in case of other microservices/applications that are not Eureka server. Moreover, every microservice project is connected to Spring Cloud project that provides default value to true. Therefore, We should include ‘eureka.client.register-with-eureka=false’ for one time only in case of Eureka server as Eureka Server itself can’t be registered.

#### eureka.client.fetch-registry=false

This property indicates that Eureka Server is supported to fetch instance details of microservice to make intra-communication between microservices happen. If one microservice wants to communicate with another microservice by using Eureka then inside microservice we should add this property (eureka.client.fetch-registry) and set it to true. However, inside Eureka Server we should include this property with a value as false. However, Eureka server will never try to fetch registry as it is itself having a registry. Hence the value of this property in case of Eureka server will be false. Moreover, every microservice project is connected to Spring Cloud project that provides default value to true.

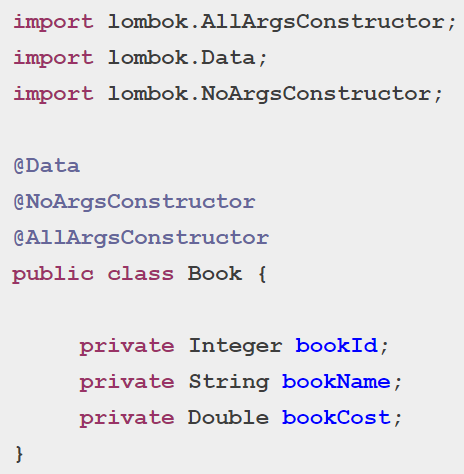
Default port for Eureka Server is 8761.

## Create Microservice #2(Producer Service)

### Step #1: Create a Spring Boot Project

While creating a project in STS, add starter ‘Eureka Discovery Client’ , ‘Spring Web’ and ‘Lombok’ in order to get all required features.

### Step #2: Create Model class as Book.java



### Step #3: Create a RestContoller class as BookRestController.java

**import java.util.List;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.core.env.Environment;**

**import org.springframework.http.HttpStatus;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.PathVariable;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import com.dev.springcloud.feign.model.Book;**

**@RestController**

**@RequestMapping("/book")**

**public class BookRestController {**

**@Autowired**

**Environment environment;**

**@GetMapping("/data")**

**public String getBookData() {**

**return "data of BOOK-SERVICE, Running on port: "**

**+environment.getProperty("local.server.port");**

**}**

**@GetMapping("/{id}")**

**public Book getBookById(@PathVariable Integer id) {**

**return new Book(id, "Head First Java", 500.75);**

**}**

**@GetMapping("/all")**

**public List<Book> getAll(){**

**return List.of(**

**new Book(501, "Head First Java", 439.75),**

**new Book(502, "Spring in Action", 340.75),**

**new Book(503, "Hibernate in Action", 355.75)**

**);**

**}**

**@GetMapping("/entity")**

**public ResponseEntity<String> getEntityData() {**

**return new ResponseEntity<String>(**

**"Hello from BookRestController",**

**HttpStatus.OK);**

**}**

**}**

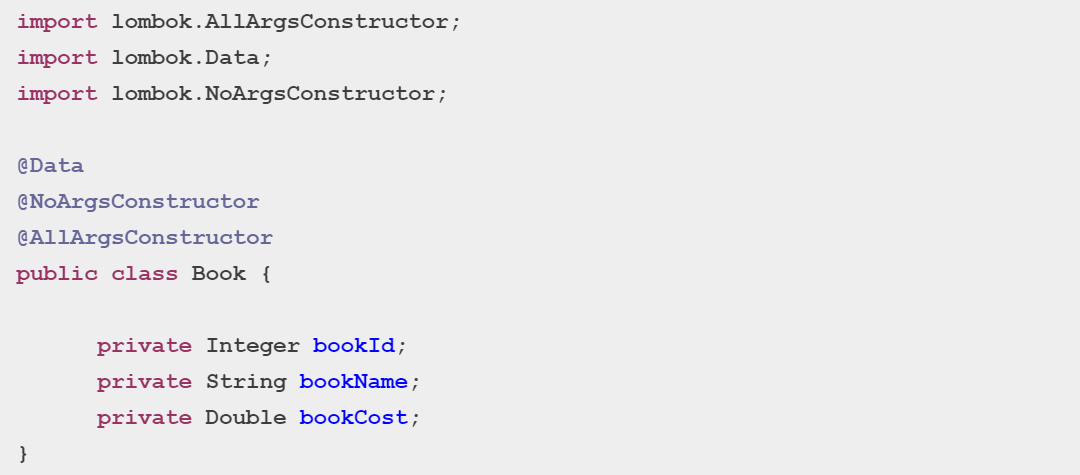
## Create Microservice #3(Consumer Service)

### Step #1: Create a Spring Boot Project

While creating a project in STS, add starter ‘OpenFeign’, ‘Eureka Discovery Client’ , ‘Spring Web’ and ‘Lombok’ in order to get all required features.

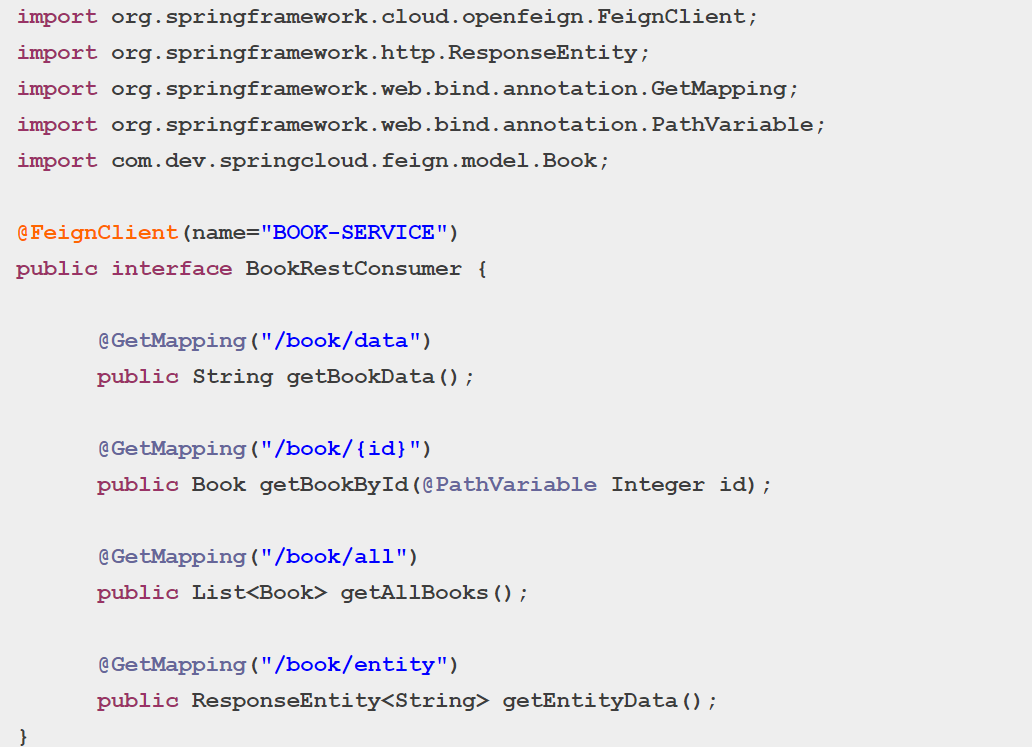
### Step #2: Create Model class as Book.java

Let’s replicate the model class as Book.java as it is in the producer service.



### Step #3: Create an interface as BookRestConsumer.java

This is the most important file when talking about Feign Client. At Interface level, we need to apply @FeignClient annotation and provide the name of producer service/application as below



### Step #4: Create a RestController as StudentRestController.java

At the end, create a RestController as StudentRestController to receive the data from Book service as below. The important point to note here is that we need to auto-wire the BookRestConsumer here in this class and then use it.

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import com.dev.springcloud.feign.consumer.BookRestConsumer;**

**@RestController**

**@RequestMapping("/student")**

**public class StudentRestController {**

**@Autowired**

**private BookRestConsumer consumer;**

**@GetMapping("/data")**

**public String getStudentInfo() {**

**System.out.println(consumer.getClass().getName());** //prints as a proxy class

**return "Accessing from STUDENT-SERVICE ==> " +consumer.getBookData();**

**}**

**@GetMapping("/allBooks")**

**public String getBooksInfo() {**

**return "Accessing from STUDENT-SERVICE ==> " + consumer.getAllBooks();**

**}**

**@GetMapping("/getOneBook/{id}")**

**public String getOneBookForStd(@PathVariable Integer id) {**

**return "Accessing from STUDENT-SERVICE ==> " + consumer.getBookById(id);**

**}**

**@GetMapping("/entityData")**

**public String printEntityData() {**

**ResponseEntity<String> resp = consumer.getEntityData();**

**return "Accessing from STUDENT-SERVICE ==> " + resp.getBody() +" , status is:" + resp.getStatusCode();**

**}**

**}**

## How to test FeignClient Enabled Microservice?

It’s time to test our FeignClient enabled Microservice. Please follow below steps:

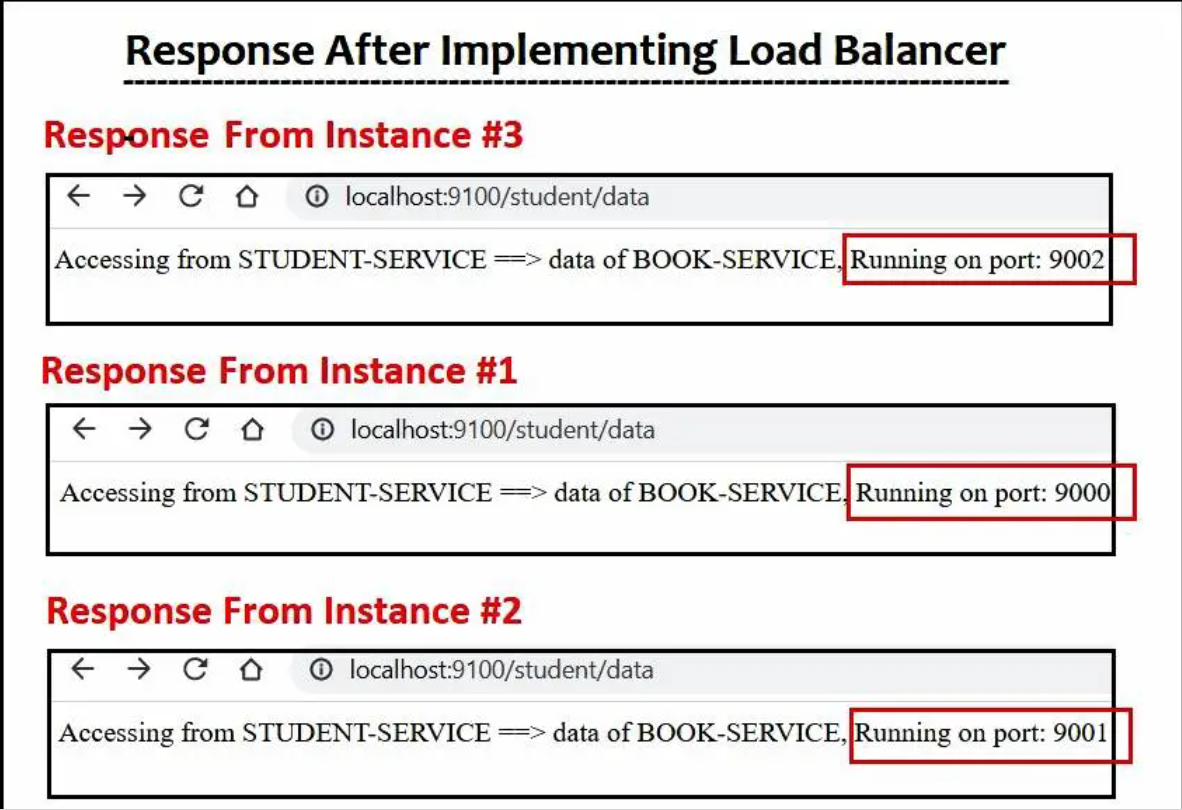
1) Start Service/Application containing Eureka Server  
2) Start Producer Service/Application (Book)  
3) Start Consumer Service/Application (Student)  
4) Open a browser window, hit below URLs and observe the results.

http://localhost:9100/student/data  
http://localhost:9100/student/allBooks  
http://localhost:9100/student/getOneBook/501  
http://localhost:9100/student/entityData

## How to test FeignClient Enabled Microservice as a Load Balancer?

Now we are going to test load-balancing functionality of FeignClient enabled Microservice. Please follow below steps:

1) Start Service/Application containing Eureka Server  
2) Start multiple instances of Producer Service/Application (Book). In order to get it, change the server port in application.properties file, save the file and then start the application. Let’s assume that we need three instances of the application. Port 9000 is already configured, now configure two more 9001 and 9002.  
3) Start Consumer Service/Application (Student)  
4) Now, open a browser window, hit  URL ‘http://localhost:9100/student/data‘ which contains server port information. Further, refresh the browser multiple times and observe the port number in the results. It will randomly pick the server from 9000, 9001 and 9002 as shown below.



## FAQ

### **Is Feign Client Limited to HTTP Communication?**

Yes, Feign Client primarily supports HTTP-based protocols and is built for HTTP communication.

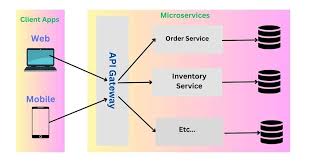
### **What is difference between feign client and RestTemplate?**

**RestTemplate:** RestTemplate is a synchronous HTTP client provided by Spring Framework. It requires developers to manually create HTTP requests and handle the responses.

**Feign Client:** Feign Client is a declarative web service client developed by Netflix. It allows developers to define interfaces with annotated methods representing the HTTP API of the target microservice. Feign Client handles the elementary details of making HTTP requests, including request marshalling, load balancing. It simplifies the development process and focus on a more concise and declarative coding style.

**Spring Cloud API Gateway**

**Spring API Gateway** is a server that acts as an entry point for all client requests to access the microservices in the backend.



Key Features and Benefits of Spring API Gateway:

 **Routing**: It routes incoming requests to appropriate microservices. This is done based on request path, headers, or other conditions.

 **Load Balancing**: Integrates with load balancing mechanisms like **Ribbon** or **Spring Cloud LoadBalancer** to distribute requests to multiple instances of a microservice.

 **Security**: Handles authentication and authorization, ensuring secure communication between clients and microservices.

 **API Gateway Patterns**: Supports various patterns like **Request Routing**, **Rate Limiting**, **Circuit Breaker**, **API Aggregation**, and **Centralized Logging**.

 **Filters**: It provides pre-filters and post-filters for request/response transformations, which are useful for tasks such as logging, metrics collection, or adding security headers.

 **Cross-cutting concerns**: API Gateway can be used to centralize management of common concerns such as:

* Authentication and authorization (OAuth, JWT)
* Logging, monitoring, and auditing
* Caching
* Request/response transformations (JSON to XML, etc.)

 **API Aggregation**: It can aggregate responses from multiple microservices into a single response, simplifying client-side logic.

 **Fault Tolerance**: Integration with patterns like **Circuit Breakers** (e.g., **Hystrix** or **Resilience4j**) to ensure resilience during failures.

**Key Components in Spring Cloud Gateway:**

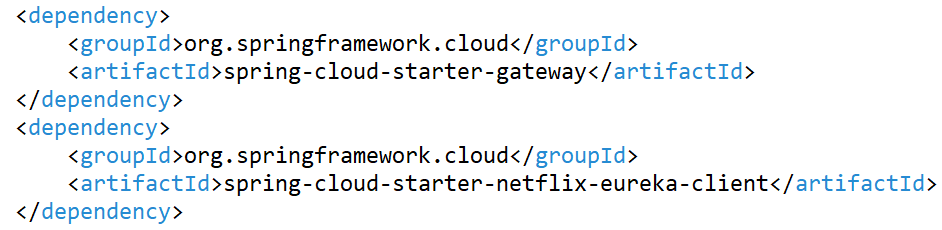
1. **Gateway Filter**: A Spring bean that provides custom pre- or post-processing of the requests. Filters are typically used for tasks like authentication, logging, modifying headers, etc.
2. **Routes**: Configurations that define the route mappings between incoming requests and microservices. Each route maps a URL pattern to a service.
3. **Predicate**: A function that is used in conjunction with routes to define conditions that determine whether a route will be applied (e.g., path match, header match).

**Step 1: Set up the Spring API Gateway**

**1.1 Create the Spring Gateway project**

Start by creating a new Spring Boot project for the API Gateway.

**pom.xml for Spring API Gateway**





server.port = 9191

spring.cloud.gateway.routes[0].id=PRODUCT-SERVICE

spring.cloud.gateway.routes[0].uri=lb://PRODUCT-SERVICE

spring.cloud.gateway.routes[0].predicates[0]=Path=/product/\*\*

spring.cloud.gateway.routes[1].id=CUSTOMER-SERVICE

spring.cloud.gateway.routes[1].uri=lb://CUSTOMER-SERVICE

spring.cloud.gateway.routes[1].predicates[0]=Path=/customer/\*\*

spring.cloud.gateway.routes[2].id=ORDER-SERVICE

spring.cloud.gateway.routes[2].uri=lb://ORDER-SERVICE

spring.cloud.gateway.routes[2].predicates[0]=Path=/order/\*\*

**What Is Load Balancing?**

**Load balancing is the process of distributing traffic among different instances of the same application.**

One of the most prominent **reasons of evolution from monolithic to microservices architecture is horizontal scaling**. It is required in modern day applications to improve user experience in the case of higher traffic for a particular service. We create multiple instances of the service in order to handle the large traffic of requests. But if the requests are not distributed among the instances effectively, then horizontal scaling is of no use.

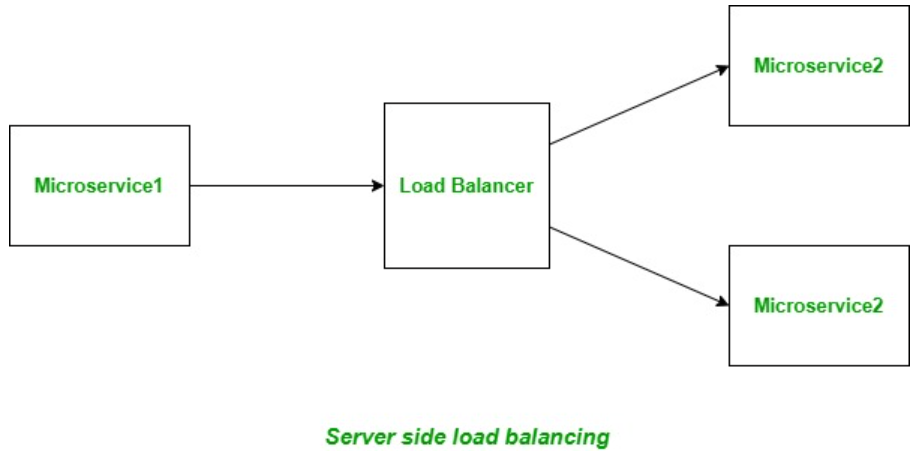
Load balancing refers to **efficiently distributing the incoming network traffic across a group of backend servers** (multiple instances of the service).

## Types of load-balancing

1. Server-side Load Balancing
2. Client-side Load Balancing

## 1. Server-side Load Balancing (Traditional server-side load balancing)

In Server-side load balancing, the **instances of the service are deployed on multiple servers** and then a **load balancer is put in front** of them. It is generally a hardware load balancer. All the incoming requests traffic firstly comes to this load balancer acting as a middle component. It then decides to which server a particular request must be directed to based on some algorithm.



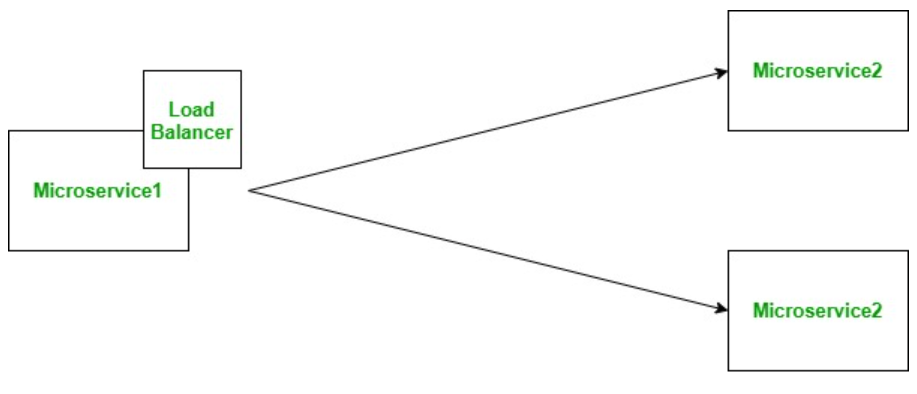
Server-side load balancing is involved in monolithic applications where we have limited number of application instances behind the load balancer. We deploy our war/ear files into multiple server instances which are basically a pool of server having the same application deployed and we put a load balancer in front of it.

Load balancer has a public IP and DNS. The client makes a request using that public IP/DNS. Load balancer decides to which internal application server request will be forwarded to. It mainly uses round robin or sticky session algorithm. We call it server-side load balancing.

### **Disadvantages of Server-side load balancing**

1. Mostly server-side load balancing is a manual effort and we need to add/remove instances manually to the load balancer to work. So ideally, we are losing the todays on demand scalability to auto-discover and configure when any new instances will be spinned of.
2. Server-side load balancer acts as a **single point of failure** as if it fails, all the instances of the microservice becomes inaccessible as only load balancer has the list of servers.

### **Client-side Load Balancing**



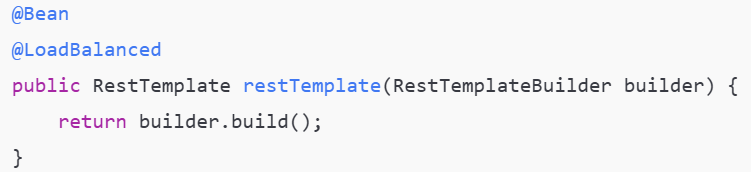
They reside in the application as inbuilt component and bundled along with the application, so we don’t have to deploy them in separate servers.

In microservice architecture, we will have to develop many microservices and each microservice may have multiple instances in the ecosystem. To overcome this complexity, we have already one popular solution to use **service discovery pattern.**

Now if one microservice wants to communicate with another microservice, it generally looks up the service registry using discovery client and Eureka server returns all the instances of that target microservice to the caller service. Then it is the responsibility of the caller service to choose which instance to send request.

Here the client-side load balancing comes into picture and automatically handles the complexities around this situation and delegates to proper instance in load balanced fashion. Note that we can specify the load balancing algorithm to use.

Set Up Load Balancer Using @LoadBalanced RestTemplate



**@RestController**

**@RequestMapping("/book")**

**public class BookRestController {**

**@Autowired**

**Environment environment;**

**@GetMapping("/data")**

**public String getBookData() {**

**return "data of BOOK-SERVICE, Running on port: "**

**+environment.getProperty("local.server.port");**

**}**