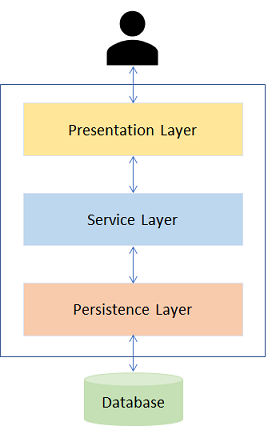
|  |
| --- |
| Micoservices |

Consider an online shopping application with the following functionalities:

1. Search products
2. Place order
3. View all orders

The most common approach to develop this application is to implement all these functionalities in a single application. This application is then deployed as one application for all different types of users. This type of architecture where all the functionalities are implemented in one single application is called as **Monolithic Architecture**. In this architecture the application is divided into different layers. Some common layers in this architecture are as follows:

* **Presentation Layer** — It responsible for handling HTTP requests and sending respose.
* **Service Layer** — In this layer business logic of the application is implemented.
* **Persistence layer** — In this layer logic for accessing the database is implemented.



The applications developed using this architecture are easy to develop, test and deploy. However, there are certain issues that tag along with monolithic applications. As the application becomes larger and complex, we will face difficulties in the following aspects:

1. Deployment will take a long time.
2. Scalability is an issue as it is not possible to scale any single functionality alone. For example, the 'Search' functionality maybe used more often than the 'Book' functionality. Hence, we would like to scale the 'Search' functionality and not the 'Book' functionality. This is not possible when we go with the monolithic approach.
3. Failure of a single functionality will lead to failure of entire application in monolithic architecture.
4. New technologies or frameworks cannot be used in the existing application. If new technology is needed, then complete re-write must be done.
5. It is not very reliable as a single bug in any module can bring down the entire application.

How do we resolve these issues?

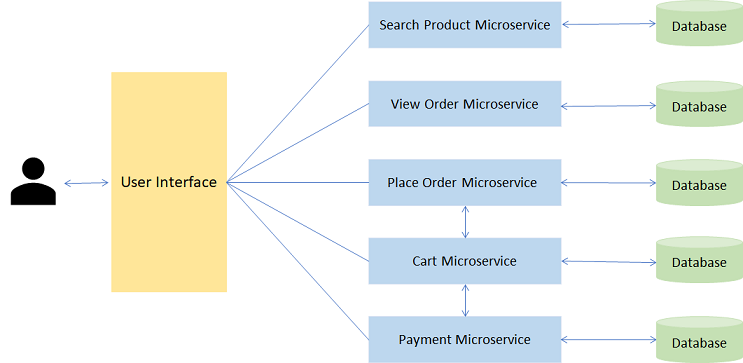
The answer is **Microservices**. It is another architecture of application development. Now let us explore it in detail.

We have learned that in monolithic architecure all the fuctionalities are implemented in one single application and then this application is deployed. But it has some drawbacks which can be addressed if we develop the application using another architecure called as **microservices**.  In this architecture we split the application into a set of smaller services instead of building a single monolithic application. Each of these services implements specific business requirement and can communicate with each other through well defined interfaces. These services have their own database and can be deployed independently.

Consider the online shopping application which we have developed using monolithic architecure. Now if you have do develop same application using microservices, then you need to split the business requirements or functonalities into several services communicating with each other and having its own database. You can have following services for this application:

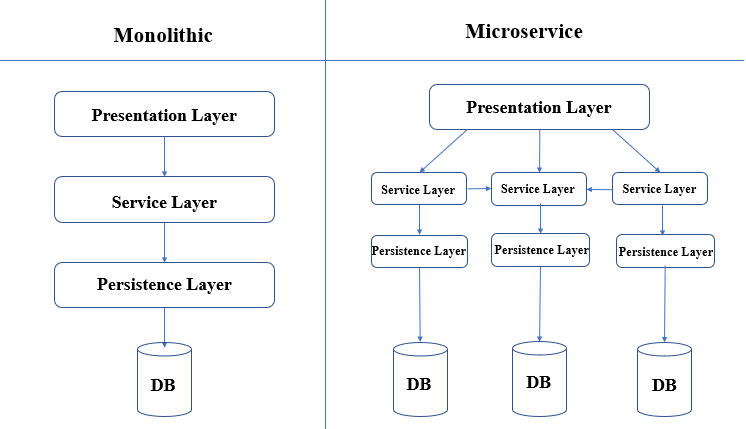
* Search Product Microservice - Responsible for seaching a product.
* View Order Microservice - View orders placed by customer.
* Place Order Microservice - Takes an order and process it.
* Cart Microservice - Manage user cart, this service can utilize Catalog service as a data source.
* Payment Microservice - Manage payments.

Each of these microservices also interact with each other to transfer data which is required by another microservice. For example, Place Order microservice will interact with Card and Order Microservice.



Now that we have seen a real-life example of Microservice, let us understand why we should choose Microservices for developing enterprise applications.

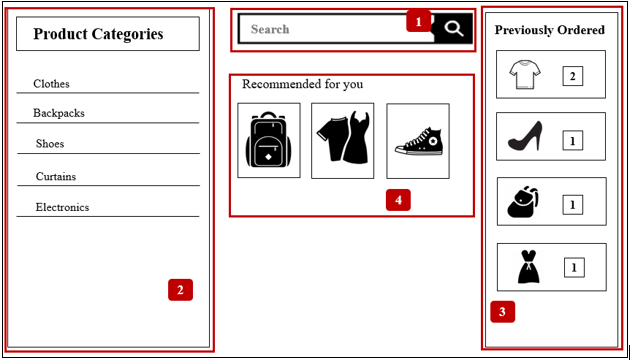
You have learnt the architectural differences between Monolithic Applications and Microservices. Now let us see how functionally wise they are different.



|  |  |  |
| --- | --- | --- |
| **Parameter** | **Monolithic** | **Microservice** |
| Development | Needs to be developed in a single language | Each service can be written in different languages |
| Testing | For any change, the entire application needs to be tested again | Only the modified service needs to be tested again |
| Runtime | Monolithic application runs as a single process | Each service runs its own process |
| Packaging | Packaged as a single JAR/WAR file | Each service is packaged as single JAR/WAR file |
| Scalability | Entire application needs to be scaled or replicated on multiple servers | Only the service that requires scaling, can be scaled |
| Minor Changes | For any modification. entire application needs to be re-built and re-deployed | Only the modified service needs to be re-built and re-deployed |

Now we will understand more about Microservices with an example.

An online shopping application can be divided into a number of microservices based on the functionality and business requirements.



This application can be divided into the following microservices:

1. **Product Microservice**: This microservice can be used to help the User search for any specific product that they are looking for.
2. **Product Categories Microservice**: This microservice shows all product categories available to the User.
3. **Orders Microservice**: This Microservice is responsible for placing orders and the User can view the order history as well.
4. **Recommended Products Microservice**: This microservice shows a list of recommended products to the User based on purchase history.

Now that we have seen a real-life example of Microservice, let us understand why we should choose Microservices for developing enterprise applications.

The following are the top reasons for choosing Microservice architecture:

* **Easy deployment**: A large code base will make the IDE slow and build time increases. In Microservice architecture, each project is independent and small in size. So overall build and development time gets reduced.
* **Small teams**: Microservices have single-purpose design, which means they can be built and maintained by smaller teams. Moreover, each team can be cross-functional i.e each team can have their own developers, testers and operations team and can be responsible for a single microservice.
* **Loose coupling**: It extremely difficult to change technology or language or framework when everything is tightly coupled and dependent on each other. In microservices, it is easy to change technology or framework because every module and project is independent and loosely coupled.
* **Domain-driven design**: Although microservices are supposed to be small in size, it is not the only criteria. The services should have well-defined boundaries centered around business requirements. Domain-driven design, helps design software systems based on the underlying model of the business domain, which is the requirement of microservices.

As we have seen, there are multiple reasons for choosing Microservice architecture. The following are the scenarios in which microservices can be used.

1. Migrating a monolithic application due to improvements required in scalability, manageability and agility.
2. Rewriting a heavily used legacy application. (A legacy application is a software program that is outdated or obsolete.)
3. Highly agile applications due to domain-driven design of microservices.
4. Applications that demand speed of delivery as microservices are easy to build and maintain.
5. New product development where a new product is conceived and brought to market.

Does this mean we should use microservice architecture always?

The answer is No.

If we look at the evolution of software, we started with unstructured code. Then we moved on to packages for codes, which slowly moved on to creating libraries for reuse. Then we had multiple applications and multiple libraries which had services added to them.

So, when should we choose Microservices and when should we choose Monolithic applications

We should choose Monolithic when:

1. We have little knowledge of the market
2. The application we are developing is small

We should choose Microservices when:

1. We have more knowledge of the market
2. The application being developed is a large enterprise application and needs to be highly scalable.

There are some advantages of using microservices.

1. **Scalability**: It is easier to scale only the required service based on demand.
2. **Fault Isolation**: Failure in one service does not affect the working of other services. So the entire application does not fail due to failure of single functionality.
3. **Speed of deployment**: As services are small, it can be built and deployed faster.
4. **Freedom of technology**: Each service can be written in different technology, using different frameworks based on requirement.
5. **Autonomous teams**: Microservices grant the developers more independence to work autonomously and make technical decisions quickly in smaller groups.

Although microservices have advantages, there are a few disadvantages as well. The following are the disadvantages of using microservices.

1. **Performance**: As we have a more distributed and complex application, performance could be affected.
2. **Maintenance**: An application can have hundreds of microservices and maintaining all of them can be a challenge. This can be overcome by de-centralized management.
3. **Infrastructure**: Initially, microservices require a huge infrastructure setup.
4. **Cost**: Due to huge infrastructure setup huge cost may also be incurred. Also, services need to communicate with each other. This can increase network latency and processing costs.

Once you have deployed Spring Boot application into production environment, you always want to monitor the application. This is beacause you want to ensure that the application must be always up and running and also in case of any issues you want to fix it quickly. Therefore, an application monitoring tool is required so that you can analyse the health of your application.

Spring Boot has an in-built mechanism for monitoring application called as Actuator. It is a sub-project of Spring Boot. It offers several production grade features to monitor the application. Once you enable actuator in your Spring Boot application, a set of endpoints are exposed using which you can monitor and manage your application.  You can also integrate these endpoints with other application monitoring tools such Prometheus, Graphite etc.

Now let us see how you can enable Actuator in your Spring Boot application.

**Enabling Spring Boot Actuator**

Actuators can be easily enabled in your application by adding following spring-boot-actuator dependency in pom.xml file:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

**Actuator Endpoints**

Once Actuator is enabled in your application, using actuator endpoints you can monitor and manage your application. These endpoints are exposed over HTTP in Spring MVC application using the 'id' of endpoints as the URL path along with /acuator as prefix. The following table shows some important actuator endpoints :

|  |  |
| --- | --- |
| **id** | **Description** |
| /beans | Provides list of all Spring beans available in the application |
| /configprops | Provides a collated list of all @ConfigurationProperties |
| /env | Exposes all the properties fromSpring's ConfigurableEnvironment |
| /info | Displays arbitrary application information |
| /metrics | Displays metric information for the current application |
| /mappings | Displays a collated list of all request mapping paths |
| /shutdown | Allows the application to shutdown |
| /trace | Displays trace information, by default latest 100 HTTP requests |
| /health | Provides applications health information |

These endpoints contain sestive information so all of them are not exposed by default. Only /health and /info endpoints are exposed by default. You can enable all other endpoints by adding following property in the application.properties file:

1. management.endpoints.web.exposure.include=\*

You also restrict the exposure of specific endpoint. For example, to expose all endpoints except env add the the following property in application.properties file:

1. management.endpoints.web.exposure.include=\*
2. management.endpoints.web.exposure.exclude=env

Any specific endpoint can be exposed as shown below:

1. management.endpoints.web.exposure.include=env,beans

If your application is running on port number 8756, you can get list of all the actuator endpoints using URL http://localhost:8765/actuator. It will give you the following response:

1. {
2. "\_links": {
3. "self": {
4. "href": "http://localhost:8765/actuator",
5. "templated": false
6. },
7. "health": {
8. "href": "http://localhost:8765/actuator/health",
9. "templated": false
10. },
11. "health-path": {
12. "href": "http://localhost:8765/actuator/health/{\*path}",
13. "templated": true
14. },
15. "info": {
16. "href": "http://localhost:8765/actuator/info",
17. "templated": false
18. }
19. }
20. }

If all the endpoints are exposed, then you will get following response:

1. {
2. "\_links": {
3. "self": {
4. "href": "http://localhost:8765/actuator",
5. "templated": false
6. },
7. "beans": {
8. "href": "http://localhost:8765/actuator/beans",
9. "templated": false
10. },
11. "caches": {
12. "href": "http://localhost:8765/actuator/caches",
13. "templated": false
14. },
15. "caches-cache": {
16. "href": "http://localhost:8765/actuator/caches/{cache}",
17. "templated": true
18. },
19. "health": {
20. "href": "http://localhost:8765/actuator/health",
21. "templated": false
22. },
23. "health-path": {
24. "href": "http://localhost:8765/actuator/health/{\*path}",
25. "templated": true
26. },
27. "info": {
28. "href": "http://localhost:8765/actuator/info",
29. "templated": false
30. },
31. "conditions": {
32. "href": "http://localhost:8765/actuator/conditions",
33. "templated": false
34. },
35. "configprops": {
36. "href": "http://localhost:8765/actuator/configprops",
37. "templated": false
38. },
39. "env": {
40. "href": "http://localhost:8765/actuator/env",
41. "templated": false
42. },
43. "env-toMatch": {
44. "href": "http://localhost:8765/actuator/env/{toMatch}",
45. "templated": true
46. },
47. "loggers": {
48. "href": "http://localhost:8765/actuator/loggers",
49. "templated": false
50. },
51. "loggers-name": {
52. "href": "http://localhost:8765/actuator/loggers/{name}",
53. "templated": true
54. },
55. "heapdump": {
56. "href": "http://localhost:8765/actuator/heapdump",
57. "templated": false
58. },
59. "threaddump": {
60. "href": "http://localhost:8765/actuator/threaddump",
61. "templated": false
62. },
63. "metrics": {
64. "href": "http://localhost:8765/actuator/metrics",
65. "templated": false
66. },
67. "metrics-requiredMetricName": {
68. "href": "http://localhost:8765/actuator/metrics/{requiredMetricName}",
69. "templated": true
70. },
71. "scheduledtasks": {
72. "href": "http://localhost:8765/actuator/scheduledtasks",
73. "templated": false
74. },
75. "mappings": {
76. "href": "http://localhost:8765/actuator/mappings",
77. "templated": false
78. }
79. }
80. }

Now, let us see how you can monitor InfyBank application using some of these endpoints.

**Monitoring application through Actuator Endpoints**

* **/health endpoint**

This endpoint gives you the information about health of application. You can acces this endpoint using the URL http://localhost:8765/actuator/health. It will give you following response:

1. {
2. "status": "UP"
3. }

The status will be UP if application is running and healthy. If application has some issues such as database is down etc. then you will get following response:

1. {
2. "status": "DOWN"
3. }

It tells only whether the status of application is UP or DOWN.  If you want to get complete details about health of application then add the following property in the application.properties file:

1. management.endpoint.health.show-details=always

After this you will get the more detailed response as follows:

1. {
2. "status": "UP",
3. "components": {
4. "db": {
5. "status": "UP",
6. "details": {
7. "database": "MySQL",
8. "validationQuery": "isValid()"
9. }
10. },
11. "diskSpace": {
12. "status": "UP",
13. "details": {
14. "total": 262143995904,
15. "free": 156223340544,
16. "threshold": 10485760,
17. "exists": true
18. }
19. },
20. "ping": {
21. "status": "UP"
22. }
23. }
24. }

* **/metrics endpoint**

This endpoint displays various metrics that can be checked for your application. You can acces this endpoint using the URL http://localhost:8765/actuator/metrics. It will give you following response :

1. {
2. "names": [
3. "hikaricp.connections",
4. "hikaricp.connections.acquire",
5. "hikaricp.connections.active",
6. "hikaricp.connections.creation",
7. "hikaricp.connections.idle",
8. "hikaricp.connections.max",
9. "hikaricp.connections.min",
10. "hikaricp.connections.pending",
11. "hikaricp.connections.timeout",
12. "hikaricp.connections.usage",
13. "http.server.requests",
14. "jdbc.connections.active",
15. "jdbc.connections.idle",
16. "jdbc.connections.max",
17. "jdbc.connections.min",
18. "jvm.buffer.count",
19. "jvm.buffer.memory.used",
20. "jvm.buffer.total.capacity",
21. "jvm.classes.loaded",
22. "jvm.classes.unloaded",
23. "jvm.gc.live.data.size",
24. "jvm.gc.max.data.size",
25. "jvm.gc.memory.allocated",
26. "jvm.gc.memory.promoted",
27. "jvm.gc.pause",
28. "jvm.memory.committed",
29. "jvm.memory.max",
30. "jvm.memory.used",
31. "jvm.threads.daemon",
32. "jvm.threads.live",
33. "jvm.threads.peak",
34. "jvm.threads.states",
35. "log4j2.events",
36. "process.cpu.usage",
37. "process.start.time",
38. "process.uptime",
39. "system.cpu.count",
40. "system.cpu.usage",
41. "tomcat.sessions.active.current",
42. "tomcat.sessions.active.max",
43. "tomcat.sessions.alive.max",
44. "tomcat.sessions.created",
45. "tomcat.sessions.expired",
46. "tomcat.sessions.rejected"
47. ]
48. }

The above response has the name of individual metric. To get more information about these metrics you need to append the metric name to the URL. For example if you want to know more about jvm.memory.used metric then the URL will be http://localhost:8765/actuator/metrics/jvm.memory.used. This URL will give the following response:

1. {
2. "name": "jvm.memory.used",
3. "description": "The amount of used memory",
4. "baseUnit": "bytes",
5. "measurements": [
6. {
7. "statistic": "VALUE",
8. "value": 118873840
9. }
10. ],
11. "availableTags": [
12. {
13. "tag": "area",
14. "values": [
15. "heap",
16. "nonheap"
17. ]
18. },
19. {
20. "tag": "id",
21. "values": [
22. "tenured-SOA",
23. "class storage",
24. "nursery-survivor",
25. "miscellaneous non-heap storage",
26. "tenured-LOA",
27. "JIT code cache",
28. "JIT data cache",
29. "nursery-allocate"
30. ]
31. }
32. ]
33. }

According to above response, the application is using 118873840 bytes or 118.87384MB of memory.

You can also create your own Actuator endpoints. Let us see how to do this.

​In Spring Boot you can easily customize existing Actuator endpoints and can also create new endpoints. To create a new endpoint you have to create a class and annotate it with @Component and @Endpoint annotation. The @Endpoint annotation has a parameter id which determines the URL path of endpoint. This class contains the methods which returns the response of the endpoint. For example, consider the following endpoint bean class:

1. @Component
2. @Endpoint(id = "customers")
3. public class MyCustomerEndpoint {

6. @ReadOperation
7. public List<CustomerDTO> getAllCustomers(){
8. *// rest of the code*
9. }
11. @WriteOperation
12. public String updateCustomer(Integer customerId, String emailId) throws InfyBankException {
13. *// rest of the code*
14. }
16. @DeleteOperation
17. public String deleteCustomer(@Selector Integer customerId) {
18. *// rest of the code*
19. }
20. }

In above class,

* The value of id parameter of @Endpoint is customers*.*So this endpoint is accessible by URL /actuator/customers.
* a method defined with @ReadOperation which will be mapped to HTTP GET method and automatically be exposed over HTTP.
* a method defined with @WriteOperation which will be mapped to HTTP POST method and automatically be exposed over HTTP. The methods that are annotated with @WriteOperation can take parameters in JSON format alone.
* a method defined with @DeleteOperation which will be mapped to HTTP DELETE method and automatically be exposed over HTTP.
* technology-specific endpoints defined with @WebEndpoint/@JmxEndpoint. For ex, @WebEndpoint is exposed over HTTP only.

**Objective**:

To enable and observe actuator endpoints.

**Steps**:

**Step 1**: Import the Spring REST demo which was done for implementing CRUD operations in Spring Boot application.

**Step 2**: Add the following Spring Boot starter dependency in pom.xml.

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

**Step 3**: Add the below property in the application.properties file to enable all the actuator endpoints:

1. management.endpoints.web.exposure.include=\*

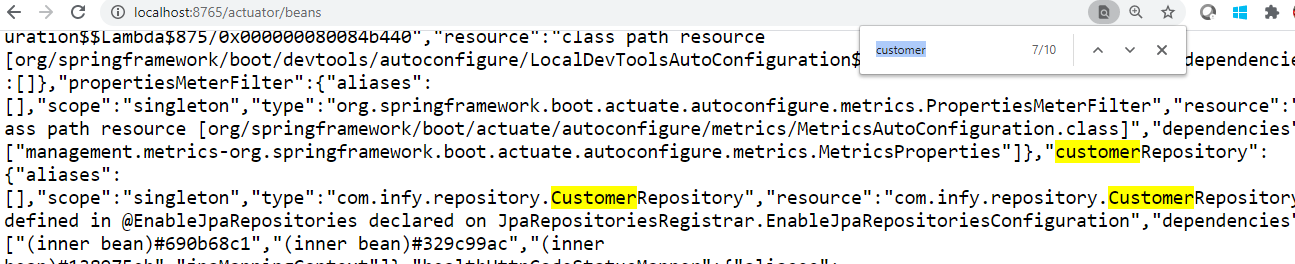
**Step 4:**Start the Spring Boot application.

**Step 5:**In Google chrome, go to the URL **http://localhost:8765/actuator**. Below screen will appear.



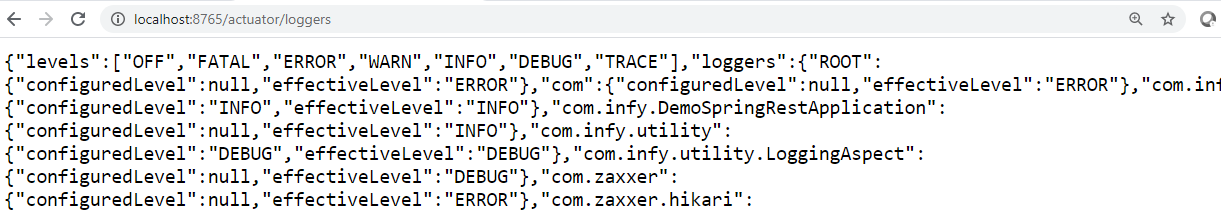
Lets see output of some of the endpoints.

**beans:**http://localhost:8765/actuator/beans

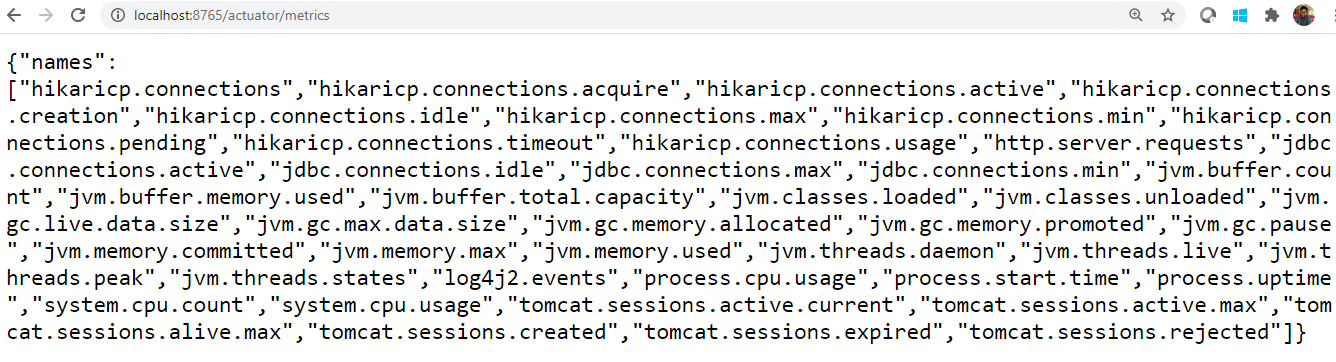


In the above image, the customerRepository bean is highlightedwith some basic information like scope, type, resource and so on.

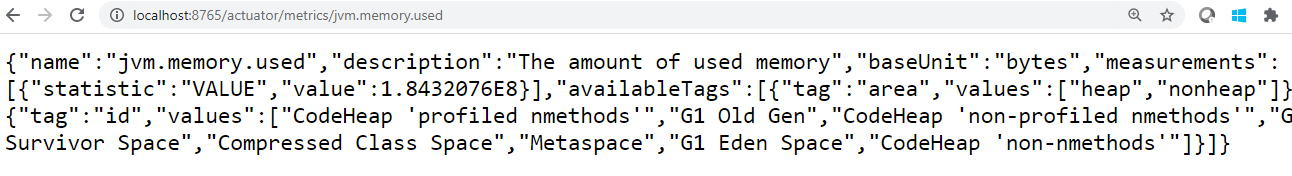
**loggers:**http://localhost:8765/actuator/loggers



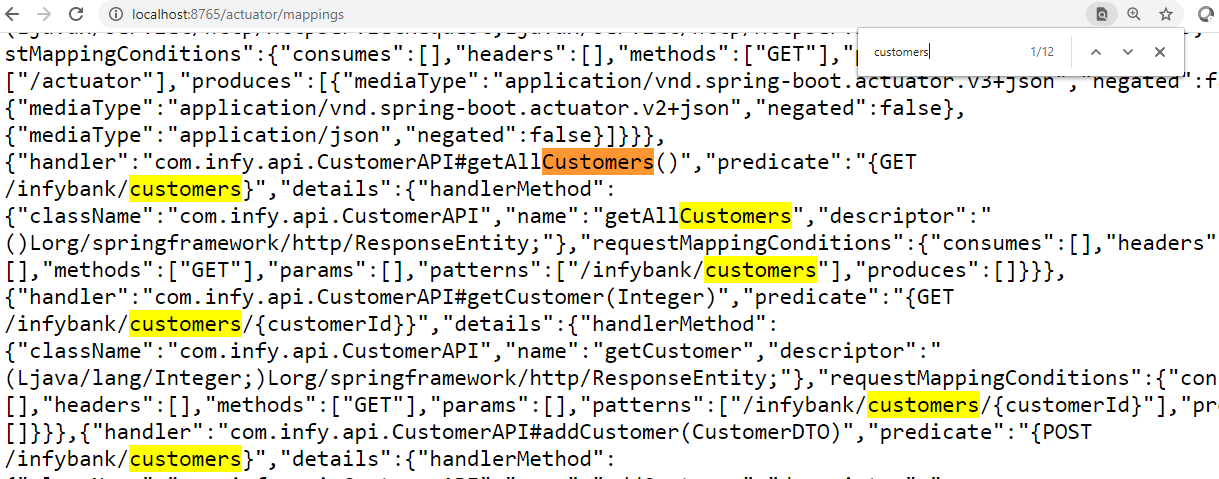
**metrics:**http://localhost:8765/actuator/metrics will provide all the metrics name



**metrics/jvm.memory.used:**http://localhost:8765/actuator/metrics/jvm.memory.used provides details of jvm memory used metric



**mappings:**http://localhost:8765/actuator/mappings



You can download the demo from [here.](https://academy.onwingspan.com/apis/authContent/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0131291254633594882277/web-hosted/assets/DemoSpringREST0216517346583761658989845341.zip)

To create a custom endpoint follow the below steps:

**Step 6:**Create MyCustomerEndpoint class in com.infy.endpoint package as shown below:

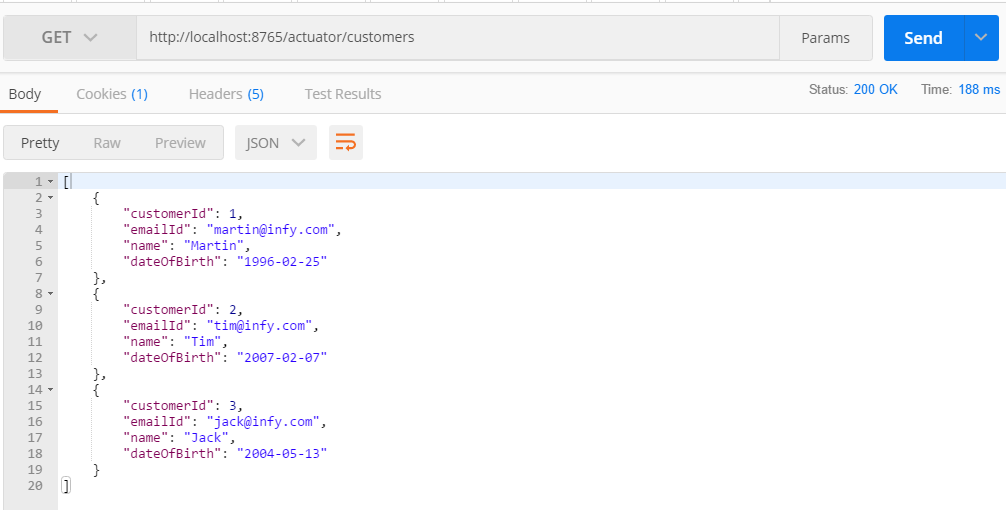
1. @Component
2. @Endpoint(id = "customers")
3. public class MyCustomerEndpoint {
5. @Autowired
6. private CustomerRepository customerRepository;
8. @Autowired
9. private Environment environment;
11. @ReadOperation
12. public List<CustomerDTO> getAllCustomers(){
13. List<Customer> customers = (List<Customer>) customerRepository.findAll();
14. List<CustomerDTO> customerDTOs = new ArrayList<>();
15. customers.forEach((customer) -> {
16. CustomerDTO customerDTO = new CustomerDTO();
17. customerDTO.setCustomerId(customer.getCustomerId());
18. customerDTO.setDateOfBirth(customer.getDateOfBirth());
19. customerDTO.setEmailId(customer.getEmailId());
20. customerDTO.setName(customer.getName());
22. customerDTOs.add(customerDTO);
23. });
24. return customerDTOs;
25. }
27. @WriteOperation
28. public String updateCustomer(Integer customerId, String emailId) throws InfyBankException {
29. Optional<Customer> optional = customerRepository.findById(customerId);
30. Customer customer = optional.orElseThrow(() -> new InfyBankException("Service.CUSTOMER\_NOT\_FOUND"));
31. customer.setEmailId(emailId);
33. return environment.getProperty("API.UPDATE\_SUCCESS");
34. }
36. @DeleteOperation
37. public String deleteCustomer(@Selector Integer customerId) {
38. customerRepository.deleteById(customerId);
39. return environment.getProperty("API.DELETE\_SUCCESS");
40. }
41. }

**Step 7:**Restart the application and test the below requests on Postman.

* **GET request:**

URL - http://localhost:8765/actuator/customers

Response -



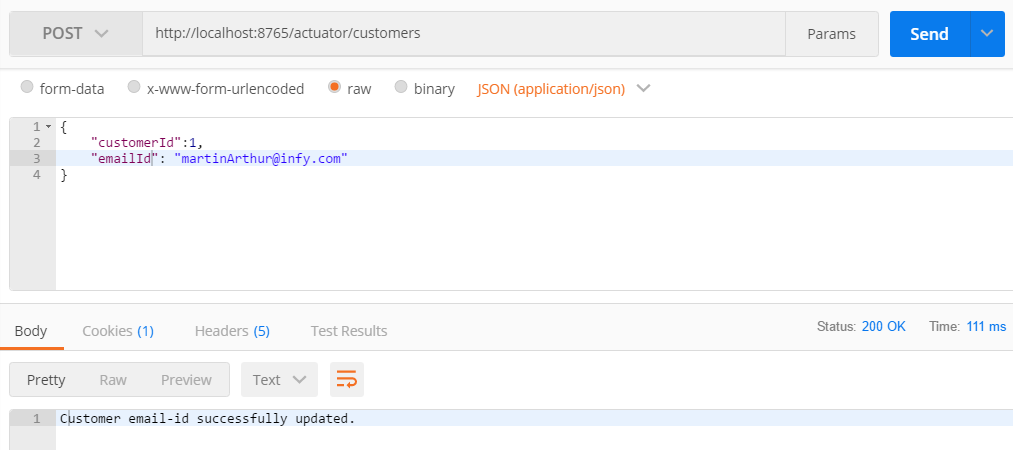
* **POST request:**

URL - http://localhost:8765/actuator/customers

Request Body -

1. {
2. "customerId":1,
3. "emailId": "martinArthur@infy.com"
4. }

Response -



* **DELETE request:**

URL - http://localhost:8765/actuator/customers/1

Response -

