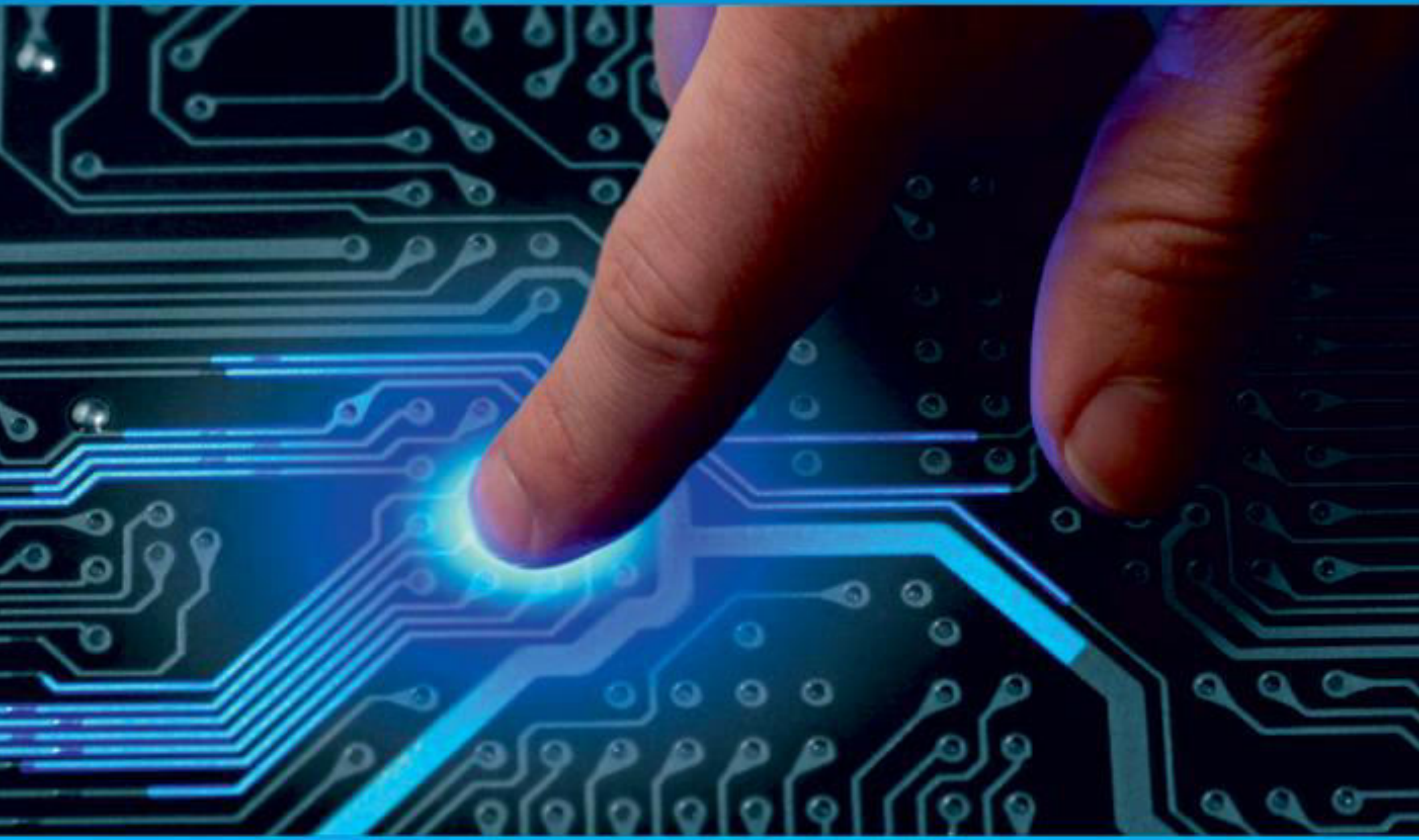
IJIRCCE

e-ISSN: 2320-9801 | p-ISSN: 2320-9798



INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

Volume 10, Issue 7, July 2022

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.165

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165 |

|| Volume 10, Issue 7, July 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1007008|

Placement Application using Microservice Architecture and Spring Framework

Narendra V, Nikhil H P, Nikhil V, R Sai Vamshi, DR. Harish Kumar B T

UG Student, Dept. of CSE, Bangalore Institute of Technology, Bangalore, Karnataka of India

UG Student, Dept. of CSE, Bangalore Institute of Technology, Bangalore, Karnataka of India

UG Student, Dept. of CSE, Bangalore Institute of Technology, Bangalore, Karnataka of India

UG Student, Dept. of CSE, Bangalore Institute of Technology, Bangalore, Karnataka of India

Assistant Professor, Dept. of CSE, Bangalore Institute of Technology, Bangalore, Karnataka of India

ABSTRACT: Placement Manager application based on spring framework and Micro Services paradigm had been designed and developed in this paper. This system was designed to overcome the difficulty of filtering the students after the registration to visit takes place through a google form, automate the student list generation process, and keep track of the offers issued by the company in the respective drives conducted by them. This application is designed to overcome the difficulties mentioned. Since monolith applications have scaling issues and must be managed by a single team, we use microservices for developing this application so that each microservice can be managed and owned by a single team. Model-View-Controller design patterns of spring have been used in the architecture. Spring Provides the best code reuse and has the property Inversion-of-Control. The developed system is a multitier system including a Controller layer, Service layer, Repository layer, and database layer, which can separate controller logic from business logic and improve the system's reusability, reliability, and maintainability with low coupling

KEYWORDS: Microservices, Spring, Multitier System, MVC Architecture, Inversion of Control, Placement Application

I. INTRODUCTION

The rapid evolution of internet technology has made greater demand to develop applications that could run on their browses so that people have the entire world in their small hand-held devices. Spring is an open-source framework based on J2EE and uses a layered approach to integrate well with the internet. Spring is a Lightweight J2EE application development framework that uses MVC (Model-View-Controller) to separate business logic from the view and the roles of controllers, services, repository, and modals objects.

Microservices are software architecture that allows decomposing a system, its components, or its functionalities into a set of small services that are implemented, deployed, and managed independently. It also helps to develop the heterogeneous development of services using the required language based on the functionality requirement, where one microservice could be developed using Node, another using Spring, and another using Flask similarly it can be used.

In each professional course offering colleges/institutions, placement activity will be going on every year, and it is a bad practice to manage the students participating in drive activities and offer management manually. Since there may be mistakes, errors, or even data may get lost. It is tough to search and filter out students during the drive based on their offers and the minimum eligibility specified by the company during the drive. This application tries to solve problems like student data management, offer management, profile management, resume management, and student list generation that take place automatically once the registration has been closed.

Section II of this paper discussed the spring framework working mechanism. Section III of this paper discusses what microservices are, their advantages, difficulties while developing applications using microservices, and how we can resolve those difficulties with existing dependencies and working mechanisms. Section IV and V of this paper discussed system design and implementation. Section VI and VII of this paper discussed the throughput of microservices and monolith applications. Sections II and III are related works. Sections IV and V are proposed solutions.

.

II. SPRING WORKING MECHANISM

Spring is a framework used for creating web applications in Java language. It was created as an alternative to programming applications that use EJB (Enterprise Java Beans) technology (Enterprise Java Beans), which is not liked by programmers [1]. The one main advantage is that we can only include the modules needed. We need to include all modules even though they are not needed.

The most critical group of modules that forms the core of the entire application is Core Container. It consists of the following modules [2]:

- spring-core
- spring-beans
- spring-context
- spring-context-support
- spring-expression.

The core modules are depicted as shown in figure 1.

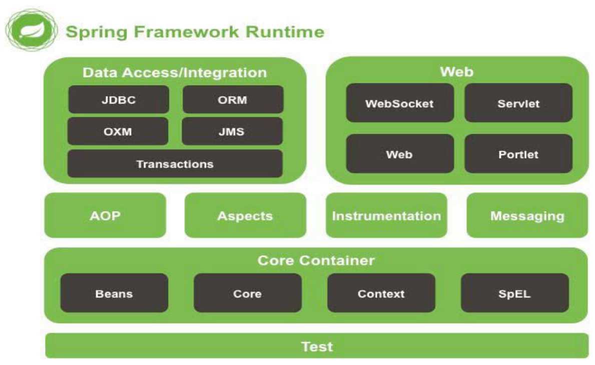


Fig.1. Spring Framework Structure

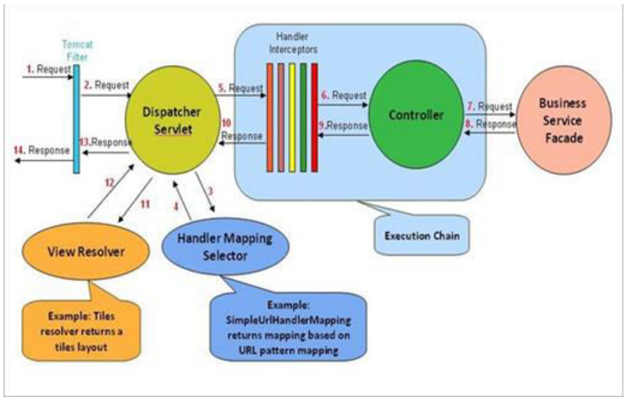


Fig.2. Spring Application Work Flow

Figure 2 shows the workflow of the spring application, where first, the client initiates a request, and that request gets received in Tomcat Filter. The request goes to the dispatcher servlet, where the Handler Mapping service is taken to check to which controller the request needs to be sent. After that, the request must go through a series of filters like a security check. After passing all filters designed, the request will reach the controller, which needs to be executed. The Business Service is called from the controller for that functionality to be executed. Then the business service will send the response, and that response is sent back to the dispatcher servlet, from there a view file is selected, the controller specifies that then the view is rendered, and finally, the response is sent back. This application does not use the view part since it sends a response in JSON format of the modal class. The above things are mentioned in [3].

III. MICROSERVICES

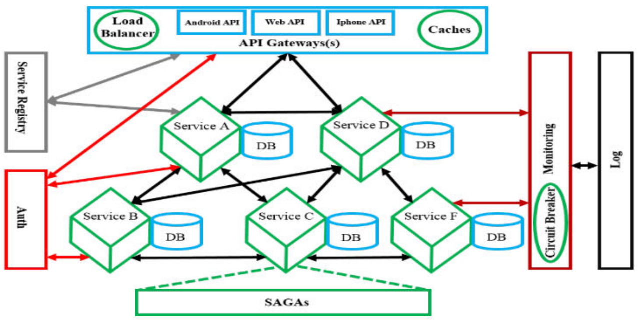


Fig.3. Micro Services Conceptual Design

Now let us look at how a request goes in the microservices architecture pattern. Figure 3 shows the overall conceptual design of the microservices. First, the request goes to the API Gateway; the gateway checks the URL and determines the microservice to which the request must be served. Then, the gateway contacts the Service Registry to get the address where the microservice is present. Then after getting the location gateway, redirects the request to that microservice. In Figure 3, there are five microservices present there, and API Gateway (which redirects the request to

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165 |

|| Volume 10, Issue 7, July 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1007008|

the microservice to which it is intended for execution and has a load balancer module, which is used for equal distribution of load among multiple instances of a microservice present), Service Registry (which keeps in track of each service present there. When a service starts, it first must register itself to Service Registry by specifying the service name and location details. The service must send its presence after a stipulated amount of time as specified by the Service Registry. If the service does not send its presence, the Service Registry marks that instance as down and does not forward requests to that service. If the service does not respond after a threshold, then that instance is dropped. The primary purpose of the Service registry is to keep track of services running up), Log represents the server where all microservices log their content there so that the logs can be easily accessible, SAGAs [4] is a module used to deal with transactions that span multiple microservices as specified, and the last Module is Circuit Breaker, this module uses the pattern to check if the dependency microservice is working fine or not based on the last specified number of requests and short the requests, this process provides the characteristics Fault tolerance.

The Steps in migrating monolith to microservice architecture are specified [5], and the respective migration steps, technology, and each step performance analysis [6]. [7] Shows how Heterogenous design of Micro-Services. [8] Shows how the complete functionality can be divided into microservices so that each microservice can be owned by a single team and can be implemented and reduce the application delivery time to the customer. The Comparison of the Service Registry technology of microservice is discussed in [8]. [11] It shows how deployment patterns and performance analysis of the microservices could be made in AWS by taking a policy example, where the functionality is divided into two parts. The first part or service fetches the policy from the database. In contrast, the second took custom input from the user and calculated the respective parameters needed based on application functionality.

IV. SYSTEM ARCHITECTURE

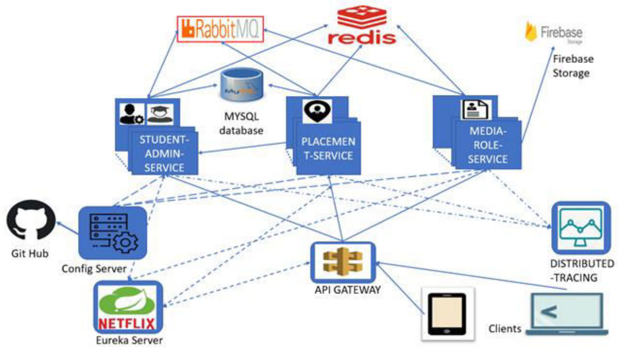


Fig.4. Placement Application Microservices Architecture

Figure 4 shows us the detailed architecture of the placement application. It consists of 10 servers with three microservices: Student Admin Service, Placement Service, and Media Role Service. Each microservice performs unique tasks and only interacts with each other only when necessary. Whenever multiple user requests service from student admin and placement services, the API gateway will redirect the request to service so that both tasks are performed simultaneously. Due to this simultaneous process, we can get higher throughput than monolith applications. We have divided the entire functionality of the application into three modules or microservices. The description of each module is as follows:

A. *Student Admin Service*

This service or module deals with the entities like students, admin, and branch and storing their respective details.This module is responsible for locking and unlocking student details during the list generation.

B. *Placement Server*

This service or module deals with entities like companies, their visits, activities, and their registrations, along with company offers. This module is also responsible for generating the student list after the closing registration time passes of a particular visit. When the registration for the company starts, it fetches student details from the student admin service server. It checks if the student is eligible to participate in the given drive by checking eligibility criteria and the offers held by the student. It registers students only if they are eligible to participate in the drive.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165 |

|| Volume 10, Issue 7, July 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1007008|

C. *Media Role Service*

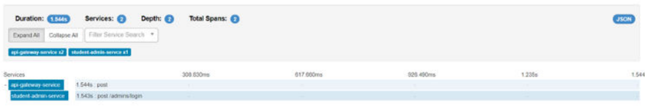
This service or module is responsible for storing all the application's media requirements like the students' resume and profile, Job Description, and the student list generated for a visit. This module also has an asynchronous activity; when a student or company visit is deleted, all their respective files are deleted based on notifications sent by the respective microservices.

D. *Config Server*

This module is responsible for taking the configurations of the respective service based on the stages of the application and the environment in which the application or service is running.

E. *Distributed Tracing Server*

We know that whenever a request comes, we need to track where the request has been spanned and where errors occurred. The tracked details are the customer IP, micro service IP, request path, and time spent by request in each service. This server or module is responsible for keeping track of these details. An example is shown in figure 5. This shows the details timeline as specified with the above details.



```
api-gateway-service      1.544s : post
student-admin-service    1.543s : post /admins/login
```

Fig.5.Distributed tracing of a request

F. *Rabbit MQ Server*

In our application, we need to have asynchronous activities to be performed. With the help of this server, we can perform these activities. This server acts as a message broker between microservices.

G. *Redis Server*

This server acts as a centralized place for cache management.

H. *Firebase Storage*

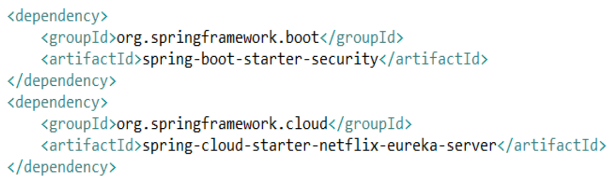
Google Firebase storage acts as a centralized cloud service, storing all the application's required media files.

V. IMPLEMENTATION

In this section, we go to the implementation part of the microservices that we have designed, or framework is used, and minor customization is performed as required by the application. Here we will discuss the spring framework concerning the implementation part. Here, we also take basic auth as security for the applications.

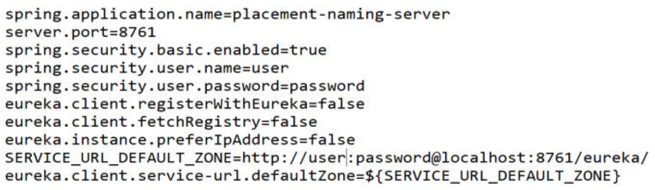
A. Service Registry

Here, we have used the Eureka server to do the service registry job. To create this, create a basic spring boot starter project and add the two dependencies specified in figure 6. The first dependency is used to add spring boot security, and the second is to add the eureka server module to an application.properties file in src/resources section add the two lines as specified in figure 7 for the basic authentication and default properties to be mentioned for the eureka server such as fetchRegistry property must be set to false, because this server need not fetch the instances of different microservice, perferIpAddress is set to true if we need to if registry must contain a map of service name and IP-address:port as value format, registerWithEureka is set to false since the server should not register to itself. Finally, we need to add an annotation to the main class of the spring boot project, as shown in figure 8.



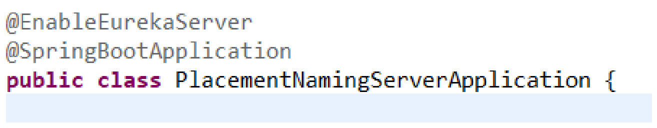
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Fig.6. Service Registry Dependencies



```
spring.application.name=placement-naming-server
server.port=8761
spring.security.basic.enabled=true
spring.security.user.name=user
spring.security.user.password=password
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
eureka.instance.preferIpAddress=false
SERVICE_URL_DEFAULT_ZONE=http://user:password@localhost:8761/eureka/
eureka.client.service-url.defaultZone=${SERVICE_URL_DEFAULT_ZONE}
```

Fig.7. Eureka Server Properties file

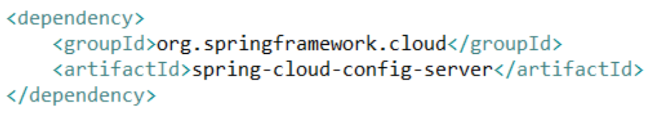


```
@EnableEurekaServer
@SpringBootApplication
public class PlacementNamingServerApplication {
```

Fig.8. Eureka Server Enable

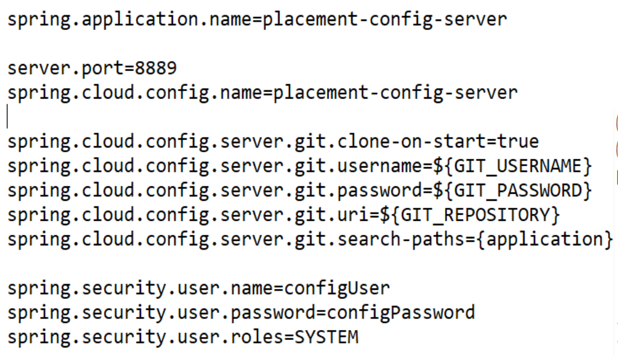
B. *Config Server*

We Create a new spring-boot starter project, then add the dependency as specified in figure 9. Then we need to configure a directory as a store of all properties of microservices running; here, we have used Git Repository. We need to mention the values for the properties, as shown in figure 10. The search-paths property is set to the application specified to search for a directory of the application name, then takes the environment value, then sends the property file, and the security properties are added. The last thing is to add annotation, as specified in figure 11, to the main class of the spring-boot project created.

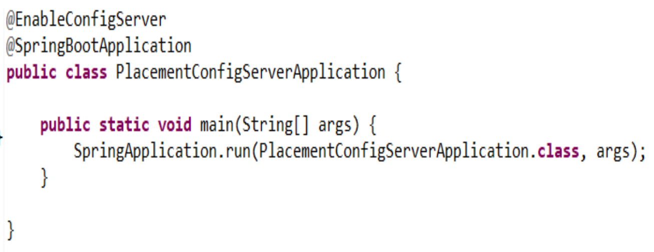


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Fig.9. Config Server Dependency



```
spring.application.name=placement-config-server

server.port=8889
spring.cloud.config.name=placement-config-server

spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.username=${GIT_USERNAME}
spring.cloud.config.server.git.password=${GIT_PASSWORD}
spring.cloud.config.server.git.uri=${GIT_REPOSITORY}
spring.cloud.config.server.git.search-paths={application}

spring.security.user.name=configUser
spring.security.user.password=configPassword
spring.security.user.roles=SYSTEM
```

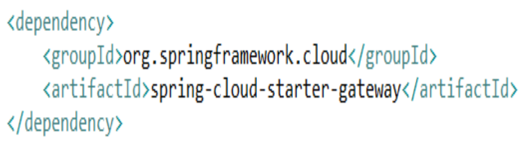


```
@EnableConfigServer
@SpringBootApplication
public class PlacementConfigServerApplication {

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

}
```

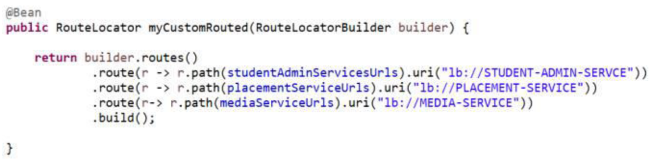
Fig.10. Config Server properties Fig.11. Config Server Enable Annotation

C. *API Gateway*

Create a new spring-boot starter project, add the dependency specified in figure 12, and add spring security dependency. Here it is optional to add security based on the requirement. We have chosen to add the security in the respective microservice and create a new bean as specified in figure 13, where it takes RouteLocatorBuilder as an argument, then uses the object then calls routes() function, then route function where the inner function takes the parameter where we need to specify the URL patterns as an array, then the microservice to which it needs to be redirected, where need to precede the microservice name by lb:// to indicate to contact service registry and do load balancing between the instances. Here in our application, we have created three arrays, specified the pattern of routes, and then added them. One such example of the array of the student-micro-service is shown in figure 14.

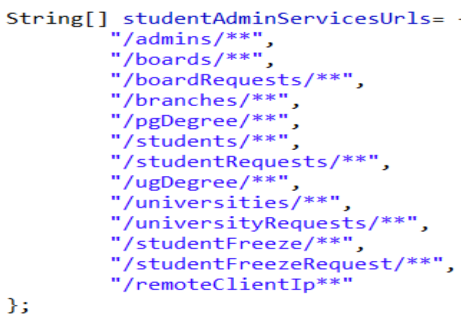


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```



```
@Bean
public RouteLocator myCustomRouted(RouteLocatorBuilder builder) {

    return builder.routes()
            .route(r -> r.path(studentAdminServicesUrls).uri("lb://STUDENT-ADMIN-SERVCE"))
            .route(r -> r.path(placementServiceUrls).uri("lb://PLACEMENT-SERVICE"))
            .route(r-> r.path(mediaServiceUrls).uri("lb://MEDIA-SERVICE"))
            .build();

}
```

Fig.12. API Gateway Dependency Fig.13. Route Locator Bean



```
String[] studentAdminServicesUrls= {
        "/admins/**",
        "/boards/**",
        "/boardRequests/**",
        "/branches/**",
        "/pgDegree/**",
        "/students/**",
        "/studentRequests/**",
        "/ugDegree/**",
        "/universities/**",
        "/universityRequests/**",
        "/studentFreeze/**",
        "/studentFreezeRequest/**",
        "/remoteClientIp**"
};
```

Fig.14. Student Admin Service routes

D. *Micro Service (Student Admin Service, Placement Service, Media-role Service)*

Here, we will consider a generalized way to define a microservice; we will hide the business login implementation. Figure 15 shows the dependencies needed for each microservice containing the business functionality to be included in pom.xml.

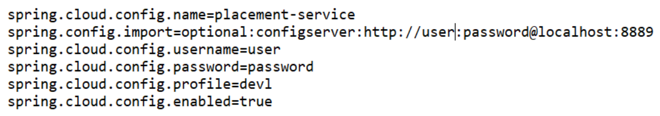
TABLE 1. DEPENDENCIES LIST

No	Dependencies	
	Group ID	*Artifact ID*
1	org.springframework.boot	spring-boot-starter-parent
2	org.springframework.boot	spring-boot-starter-actuator
3	org.springframework.boot	spring-boot-starter-data-jpa
4	org.springframework.boot	spring-boot-starter-security
5	org.springframework.boot	spring-boot-starter-validation
6	org.springframework.boot	spring-boot-starter-amqp
7	org.springframework.boot	spring-boot-starter-web
8	org.springframework.boot	spring-boot-starter
9	org.springframework.boot	spring-security-test
10	org.springframework.boot	spring-boot-starter-aop
11	org.springframework.boot	spring-boot-starter-data-redis
12	org.springframework.boot	spring-boot-devtools
13	org.springframework.boot	spring-security-test
14	org.springframework.cloud	spring-cloud-starter-netflix-eureka-client
15	org.springframework.cloud	spring-cloud-starter-openfeign
16	org.springframework.cloud	spring-cloud-starter-config
17	org.springframework.cloud	spring-cloud-starter-bootstrap
18	org.springframework.cloud	spring-cloud-starter-sleuth
19	org.springframework.cloud	spring-cloud-sleuth-zipkin
20	mysql	mysql-connector-java
21	org.projectlombok	lombok
22	org.apache.poi	poi-ooxml
23	io.github.resilience4j	resilience4j-spring-boot2
24	redis.clients	Jedis

No	Dependencies	
	Group ID	*Artifact ID*
25	com.googlecode.json-simple	json-simple
26	io.jsonwebtoken	Jjwt
27	javax.xml.bind	jaxb-api
28	com.google.firebase	firebase-admin
29	org.springframework.boot	spring-boot-starter-mail

Here we will explain only the essential dependencies, table 1 shows the 28 dependencies used in 3 micro services, 6th dependency is used to add dependencies to send message to a Message brocker,10th dependency to include Aspect-oriented-Programing that is needed for 23rd dependency, 11th dependency is used for adding redis server contact implementing functionalities for application to call for cache, 14th dependency is added to include the functionality for registering to the service registry, 15th dependency to add inter communication between the micro services, 16th dependency to include the modules that re needed to contact the config server and get the required properties to start microservice,18th and 19th dependency are added to have distributed tracing to establish between the microservices here we use ZIPKIN as distributed tracing server, 20th to have mysql connected since here we have used MySQL database, 21st dependency to have lambok to generate getters, setters and builder just by adding annotations to class, 22nd dependency to create Excel files, 26th dependency to support JSON-web token type of tracking of the clients to get their identity and security parameters, 28th dependency is used to have access to firebase storage for uploading and downloading of the media files required by the application, 29th dependency to include mail functionalities.Here we will discuss the essential properties that need to be added for each functionality to work correctly.

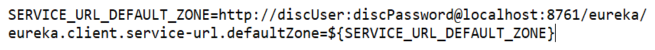
Figure 15 shows the properties that need to be added to the property file to get the configuration file from the config server. The first property is the spring.cloud.config.name, which specifies the application name, spring.cloud.import specifies the URL of the config server running, username and password are used for primary authentication purpose, spring.cloud.config.profile property specifies the environment in which the application is running to get the environment properties, and the last property is spring.cloud.config.enable must be set to true to retrieve the property file from the config server or else false to indicate using the same property values present in the property file locally with the project.



```
spring.cloud.config.name=placement-service
spring.config.import=optional:configserver:http://user:password@localhost:8889
spring.cloud.config.username=user
spring.cloud.config.password=password
spring.cloud.config.profile=devl
spring.cloud.config.enabled=true
```

Fig.15. Config Client Properties

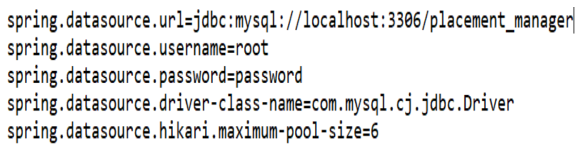
Figure 16 shows the properties for the Service Registry client implementation, the only property where we need to specify the URL of the eureka server and add annotation @EnableEurekaClient to the project's main class.



```
SERVICE_URL_DEFAULT_ZONE=http://discUser:discPassword@localhost:8761/eureka/
eureka.client.service-url.defaultZone=${SERVICE_URL_DEFAULT_ZONE}
```

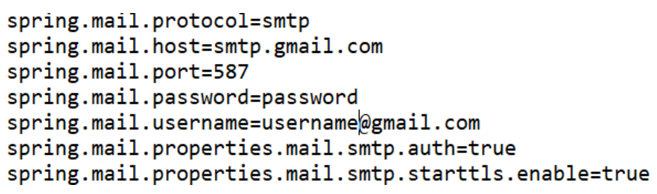
Fig.16. Eureka Client Properties

Figure 17 specifies the properties that need to be added for the spring-data JPA to work concerning the MySQL database.



```
spring.datasource.url=jdbc:mysql://localhost:3306/placement_manager
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.hikari.maximum-pool-size=6
```

Fig.17. MySQL Properties



```
spring.mail.protocol=smtp
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.password=password
spring.mail.username=username@gmail.com
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

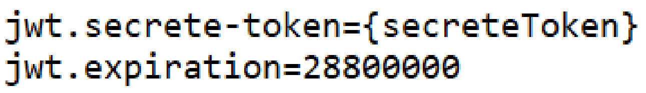
Fig.18. Spring Mail Properties

Figure 18 specifies the properties that need to be added so that the dependency added will use these properties while sending mail. Here we use this for the forgotten password to retrieve the passcode to their respective user mailbox.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165 |

|| Volume 10, Issue 7, July 2022 ||

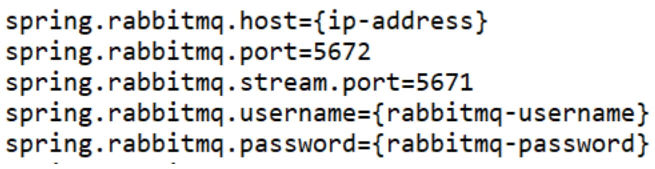
| DOI: 10.15680/IJIRCCE.2022.1007008|



```
jwt.secrete-token={secreteToken}
jwt.expiration=28800000
```

Fig.19. JWT Properties

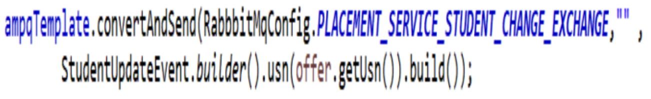
Figure 19 shows the properties required for implementing JSON Web token, the first property is secret-key specified in hex, and the following property is expiration in milliseconds.



```
spring.rabbitmq.host={ip-address}
spring.rabbitmq.port=5672
spring.rabbitmq.stream.port=5671
spring.rabbitmq.username={rabbitmq-username}
spring.rabbitmq.password={rabbitmq-password}
```

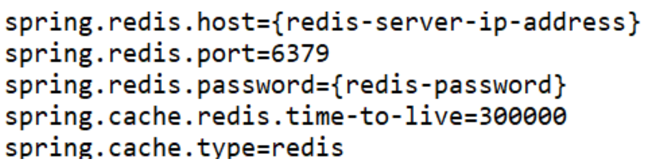
Fig.20. Rabbit MQ Properties

In this application, we use the Rabbit-MQ server for implementing the asynchronous messaging between the applications. The properties that need to be added to the client file are shown in figure 20; The first property specifies the host IP address where the rabbit-MQ server is running, the second property is the port to which we establish a connection, 3rd the property specifies the port in which we can publish and receive messages, the following two properties are used to authenticate to message broker. Figure 21 shows an example of sending a message via message broker, convertAndSend is the function used to send a message. The first argument is the exchange name, the second argument is the topic, and the third argument is the object that needs to be sent a message.



```
ampqTemplate.convertAndSend(RabbbitMqConfig.PLACEMENT_SERVICE_STUDENT_CHANGE_EXCHANGE,"" ,
        StudentUpdateEvent.builder().usn(offer.getUsn()).build());
```

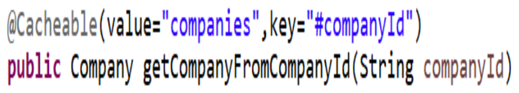
Fig.21. Publish Message



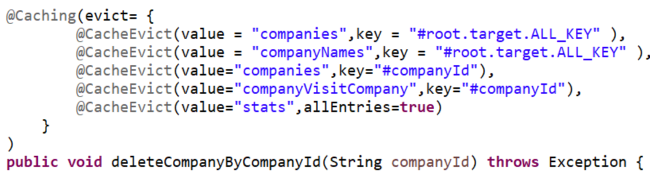
```
spring.redis.host={redis-server-ip-address}
spring.redis.port=6379
spring.redis.password={redis-password}
spring.cache.redis.time-to-live=300000
spring.cache.type=redis
```

Fig.22. Redis Server Client Properties

Figure 22 specifies the essential properties that need to be added for the centralized caching purpose. Figure 23 specifies how we could cache the result using @Cacheable annotation. The first value in the annotation is the domain class, and the second value is the key that specifies the unique value of a key that needs to be stored in the Redis server. Figure 24 specifies the annotation called @CacheEvict removed the specified key from the Redis server, and it takes the same arguments as that of the @Cacheable annotation.

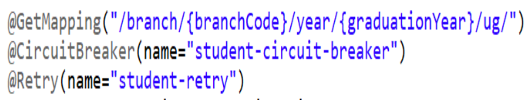


```
@Cacheable(value="companies",key="#companyId")
public Company getCompanyFromCompanyId(String companyId)
```

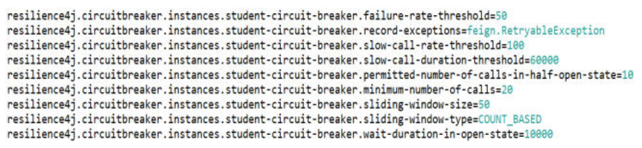


```
@Caching(evict= {
        @CacheEvict(value = "companies",key = "#root.target.ALL_KEY" ),
        @CacheEvict(value = "companyNames",key = "#root.target.ALL_KEY" ),
        @CacheEvict(value="companies",key="#companyId"),
        @CacheEvict(value="companyVisitCompany",key="#companyId"),
        @CacheEvict(value="stats",allEntries=true)
    }
)
public void deleteCompanyByCompanyId(String companyId) throws Exception {
```

Fig.23. Caching ExampleFig.24. Specifies Cache Multiple Caching and Evict



```
@GetMapping("/branch/{branchCode}/year/{graduationYear}/ug/")
@CircuitBreaker(name="student-circuit-breaker")
@Retry(name="student-retry")
```



```
resilience4j.circuitbreaker.instances.student-circuit-breaker.failure-rate-threshold=50
resilience4j.circuitbreaker.instances.student-circuit-breaker.record-exceptions=feign.RetryableException
resilience4j.circuitbreaker.instances.student-circuit-breaker.slow-call-rate-threshold=100
resilience4j.circuitbreaker.instances.student-circuit-breaker.slow-call-duration-threshold=60000
resilience4j.circuitbreaker.instances.student-circuit-breaker.permitted-number-of-calls-in-half-open-state=10
resilience4j.circuitbreaker.instances.student-circuit-breaker.minimum-number-of-calls=20
resilience4j.circuitbreaker.instances.student-circuit-breaker.sliding-window-size=50
resilience4j.circuitbreaker.instances.student-circuit-breaker.sliding-window-type=COUNT_BASED
resilience4j.circuitbreaker.instances.student-circuit-breaker.wait-duration-in-open-state=10000
```

Fig.25. Circuit Breaker AnnotationFig.26. Circuit Breaker Properties

To add a circuit breaker to the application, we first need to define the properties needed. Those properties are specified in figure 26; sliding-window-size indicates the maximum number of calls that need to be taken into

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165 |

|| Volume 10, Issue 7, July 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1007008|

consideration to make a decision; sliding-window-type can take two values one is count_based (based on previous requests) and time(based on the previous requests in specified seconds),minimum-number-of-calls specifies the minimum number of requests that need to be considered for analysis, failure-rate-threshold this specifies maximum failure requests on which we short incoming requests, and permitted-number-of-calls-in-half-open-state specifies the number of calls to retry to check if the dependent microservice is available and wait-duration-in-open-state specifies the duration in which the calls are short-circuited (default response is sent).

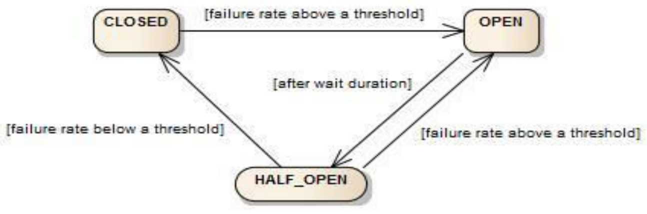

CLOSED
[failure rate above a threshold]
OPEN
[after wait duration]
[failure rate below a threshold]
[failure rate above a threshold]
HALF_OPEN

Fig.27. State Chart Diagram of Circuit Breaker

VI. RESULTS

Here, the test performance was performed by dividing the HTTP requests into two sections, one for the student admin service and the next for the placement service. In both the requests, some request to the media service was also performed. As a result, this test gives the overall performance of the application.

The tests were performed on a cluster capacity of 6GB ram; the database has a capacity of 16GB ram, 4VCPUS, and 20GB of storage capacity.

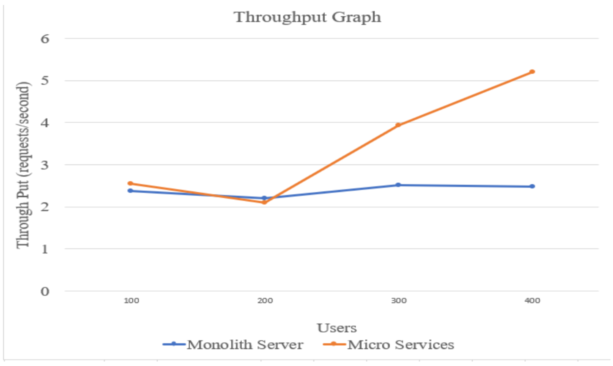


Fig.28. Throughput Graph

TABLE 2. TESTS RESULT DATA

Users	Type of Architecture	
	Monolith Server	*Microservices*
100	2.38584	2.55177
200	2.2	2.097
300	2.513235	3.92997
400	2.48`965	5.20776

Tests show that after some threshold amount of user throughput of microservice increases. Here we compare the Through Put for some of the functionality between the microservices and monolith applications. Figure 27 and Table 2 show the throughput test for the different users. From table 2, we can see that for 100 users, the throughput for microservices was more, but for 200 users, monolith works better than microservices; for 300 users (we scaled monolith sever two times, and only placement-service was replicated two times for load balancing), we can see that monolith application throughput does not increase, whereas the microservice throughput increases linearly.

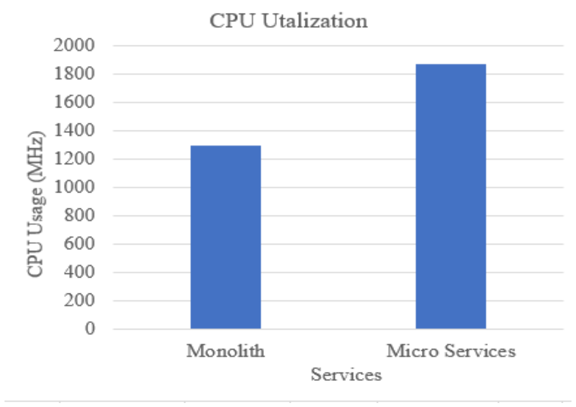


Fig.29. CPU Utilization

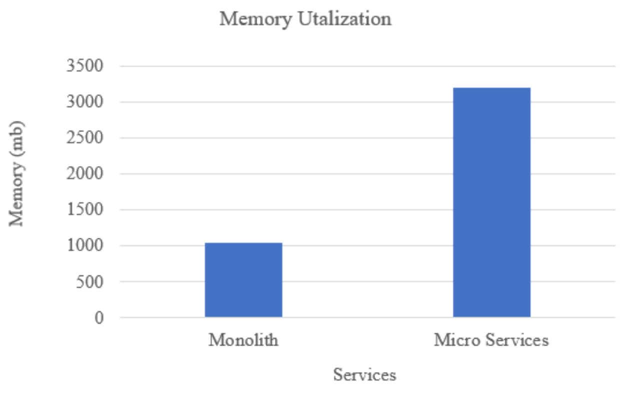


Fig.30. Memory Utilization

Figure 29 and Figure 30 show the CPU and memory utilization for monolith and microservices. We can see that microservices utilize the maximum amount of memory and CPU provided by the server to provide maximum efficiency.

VII. CONCLUSION

From the above analysis, we can say that a monolith server can be used where the numbers of users are less. In contrast, microservicesmust be used when different functions require different scaling and more users.

From this, we can understand the importance of microservices. This project tries to solve the problem of the students violating the rules and regulations of the placement cell. This project tries to solve the problem of filtering the student list after accepting student responses.

REFERENCES

1. Arthur and S. Azadegan, "Spring framework for rapid open source J2EE Web application development: a case study," *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*, 2005, pp. 90-95, DOI: 10.1109/SNPD-SAWN.2005.74
2. M. Gajewski and W. Zabierowski, "Analysis and Comparison of the Spring Framework and Play Framework Performance, Used to Create Web Applications in Java," *2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, 2019, pp. 170-173, DOI: 10.1109/MEMSTECH.2019.8817390.
3. N. Sharma, B. K. Murthy and P. N. Barwell, "Development of ePPMS for research proposals based on integrated spring and Hibernate framework," *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, pp. 247-250.
4. R. M. Munaf, J. Ahmed, F. Khakwani and T. Rana, "Microservices Architecture: Challenges and Proposed Conceptual Design," *2019 International Conference on Communication Technologies (ComTech)*, 2019, pp. 82-87, DOI: 10.1109/COMTECH.2019.8737831.
5. X. Larrucea, I. Santamaria, R. Colomo-Palacios and C. Ebert, "Microservices," *in IEEE Software*, vol. 35, no. 3, pp. 96-100, May/June 2018, doi: 10.1109/MS.2018.2141030.
6. D. Guaman, L. Yaguachi, C. C. Samanta, J. H. Danilo, and F. Soto, "Performance evaluation in the migration process from a monolithic application to microservices," *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*, 2018, pp. 1-8, DOI: 10.23919/CISTI.2018.8399148.
7. A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *in IEEE Internet Computing*, vol. 23, no. 6, pp. 19-27, 1 Nov.-Dec. 2019, DOI: 10.1109/MIC.2019.2951094.
8. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 000149-000154, DOI: 10.1109/CINTI.2018.8928192.

9. S. He, L. Zhao, and M. Pan, "The Design of Inland River Ship Microservice Information System Based on Spring Cloud," *2018 5th International Conference on Information Science and Control Engineering (ICISCE)*, 2018, pp. 548-551, DOI: 10.1109/ICISCE.2018.00120.
10. Y. Jayawardana, R. Fernando, G. Jayawardena, D. Weerasooriya and I. Perera, "A Full Stack Microservices Framework with Business Modelling," *2018 18th International Conference on Advances in ICT for Emerging Regions (ICT)*, 2018, pp. 78-85, DOI: 10.1109/ICTER.2018.8615473.
11. M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference(10CCC)*, 2015, pp.583-590, DOI: 10.1109/ColumbianCC.2015.7333476.