HSBC

Infosys
be more

About Spring Microservices

Microservice is becoming a popular choice for implementing applications which need to be highly scalable, deployable, reliable etc. Many organizations have slowly started moving towards Microservices architecture based applications. Microservice concept was created around 2012 and the term was first coined by Martin Fowler. Early adopters of microservices are Netflix, Amazon, Uber, LinkedIn, Twitter etc. Netflix also contributed heavily to the developer community by outsourcing their microservices solutions.

In Amazon, a single click may involve execution of hundreds of services. Uber has more than 2000 microservices as part of their application.

Spring framework provides required components for implementing microservices in a simpler way.

In this course, we'll discuss about creating, packaging and deploying microservices using Spring Boot and Spring Cloud components.

Target Audience: Developers

Prerequisite

Concepts you must know before doing this course

- Creating a simple application using Spring Core
- Performing CRUD operations using Spring Data JPA
- Creating a REST service through Spring Boot
- OAuth2 concepts

Recommended resources to learn the prerequisite concepts

- Spring Boot
- OAUTH2.0 from tools.ietf and aaronparecki.com

Prerequisite

Software Requirements

- Spring Tool Suite
  - The most widely used Eclipse-based Integrated Development Environment that is customized to implement, debug, run and deploy the Spring applications
- Java 8
  - JDK (Java Development Kit) provides tools for developing, debugging and monitoring, as well as the Java runtime environment (JRE) for Java applications
- Maven 3.0
  - The most popular software project management and comprehensive tool to simplify the build process of any Java-based project
- Google Chrome
  - The most widely used web browser for viewing the web pages
- Postman/SoapUI
  - Powerful HTTP client to develop and test the web services.
- MySQL Workbench
  - Workbench for MySQL

Hardware requirements:

- Any standard Desktop/Laptop

Learning Outcomes

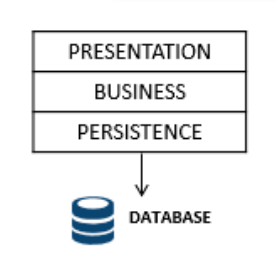
When the entire codebase is a monolith, there are several problems related to maintenance, scalability, deployment, reliability and also making it cloud native. Microservices is an architectural style of developing an application as a suite of small services, each capable of independently deployable. In this course you will learn how to create cloud native (i.e. build for cloud) microservices using Spring. You will learn using a InfyTel telecom application and implement the InfyGo system in incremental exercises as part of the course.

Specifically, you will be able to:

- Split a monolithic application into microservices.
- Store configuration details on cloud using CloudConfig
- Load balance requests between microservices using Ribbon
- Discover services in cloud using Eureka
- Increase resilience through Hystrix
- Use asynchronous communication to improve performance
- Create a API gateway using Zuul
- Simplify REST calls through Feign
- Secure microservices using OAuth through Spring Cloud Security
- Monitor your microservices through Turbine, Sleuth and Zipkin

What is a Monolithic Application?

Have you ever written an application which looks like this?



Do you deploy the entire application as a single WAR file or JAR?
Well, it means you have been writing monolithic apps!

What is a Monolithic Application?

Mono – single. If your entire codebase is a single war/jar file it means it's a monolithic app.
In monolithic apps all the functionalities are part of a single process. Such an application may consume API's of other applications and may also expose some of its functionalities as services.
Twitter, Facebook, Amazon, etc all started off as monolithic applications.

Disadvantages of Monolithic Applications

If thousands of monolithic applications are running out there without a problem, why is it suddenly considered as not an ideal solution?

Monolithic application were a good solution for the requirements of earlier times. But with recent needs for better user experience, monolithic applications fail to deliver especially as:

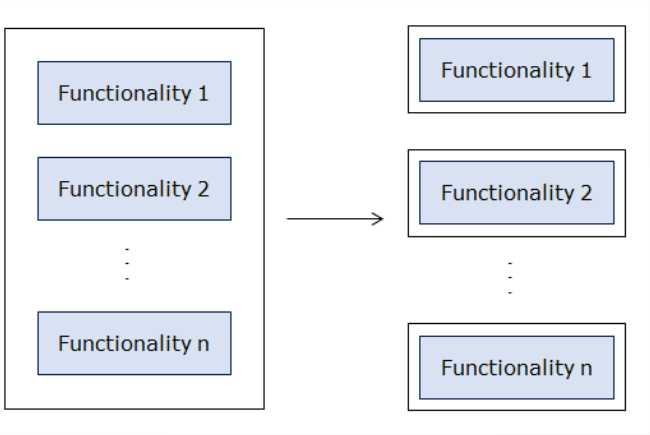
- Even for a small change the entire app has to be redeployed thus increasing downtime of the app
- Larger the app, larger the deployment time and startup time
- Even if only specific parts of the application experiences a larger load, we have to deploy the entire app in multiple servers to take care of scaling. This takes up resources and increases maintenance problems
- A large code becomes very intimidating
- We are stuck with chosen technology. If later on we find a particular functionality can be better written using python or scala or C#, modification is extremely difficult.

Solutions to Monolithic Applications

People have come up with alternatives for monolithic architecture, including Service Oriented Architecture, Mircoservices, etc.
In this course, we will be looking at Microservices.

What is a Microservice?

Microservice is an architectural style. In this style, the application is made up of smaller (or micro) apps which communicate with each other, through open protocols like HTTP. Microservices are by nature distributed in nature. We can either decompose an monolithic app into a microservice one or we can develop a microservice based solution right from beginning.

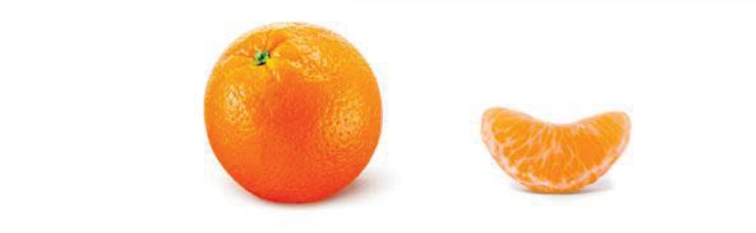


What is a Microservice?

Microservices have key characteristics such as:
•Individually developed - The development of one microservice need not depend on the completion of another microservice
•Individually deployed - One microservice can be deployed in a separate docker image with different environment from another
•Individually maintained - Changes to a microservice need not affect the entire application.
•Individually scaled - If one of the microservice is experiencing a increase in load, that alone can be scaled
•Communicated through light weight protocols like REST
•Individually monitored - We can find out the errors and performance of a separate microservice without bothering about performance of other microservices
•Organized around business capabilities. In other words, the application is split into smaller services not based on technology stack (like presentation, DB, etc) but on specific business functionalities.
•Uses "Dumb Pipes and Smart Endpoints" - Dumb pipes are the communication channels between microservices. They are dumb in the sense that they do not differentiate between communication originating from one microservice and another. The microservices act as smart endpoints as they determine who should be the specific recipient of a communication.
•They have decentralized databases
•They strongly support devops.

SOA and Microservices

What is the difference between an Orange and an Orange segment? You will find more similarities than differences. The same is the case with SOA (Service Oriented Architecture) and Microservices. You cannot talk about microservices without talking about SOA.



Microservices are fine grained SOA. A SOA has ESB (Enterprise Service Bus) whereas Microservices have a much simpler and direct communication mechanism.

SOA and Microservices

Adrian Cockcroft at Netflix : "We used to call the things we were building on the cloud "cloud-native" or "fine-grained SOA," and then the ThoughtWorks people came up with the word "microservices." It's just another name for what we were doing anyways, so we just started calling it microservices, as well. … You're trying to find words for things that resonate"

Steve Jones, MDM at Capegemini: Microservices is SOA, for those who know what SOA is.

Amazon: Amazon's successful evolution from retailer to technology platform is its SOA (service-oriented architecture), which broke new technological ground and proved that SOAs can deliver on their promises.

Uber: We decided to follow the lead of other hyper-growth companies—Amazon, Netflix, SoundCloud, Twitter, and others—and break up the monolith into multiple codebases to form a service-oriented architecture (SOA). Specifically, since the term SOA tends to mean a variety of different things, we adopted a microservice architecture.

SOA and Microservices

Similarities:

- Both of them are intended to address issues of monolithic architecture
- Both focus on splitting the application into smaller individual components
- Both use service end points for communication

Though they are similar, SOA failed whereas Microservices are trending. This is due to two reasons:

- SOA has been there for a decade and a lot of lessons were learnt on the mistakes arising from it
- The tools needed for SOA to succeed are available only now. Hence one can say that Microservice is SOA done right.

SOA and Microservices

Differences:

Communication: Microservices use simpler communication protocols like REST, AMQP (*Advanced Message Queuing Protocol)*. SOA accommodated many more with SOAP based communication protocol being very popular.

Security: SOAP has very complex security standards. Also, XML can contain even executable data thus increasing security risks. Microservices now use advanced resilience concepts like circuit breaker, fallback, OAuth2 integration, etc.

Complexity: SOA had a more complex service orchestration as the services have strong contractual agreements. SOA used ESB (Enterprise Service Bus) as the central command mechanism. Microservices have a simpler contract agreement

Implementation: SOA implementation differed from each other and many were very vendor dependent

DevOps: In SOA, one had control over the development aspects. One was still dependent on other teams for operations. Microservice is best suited for DevOps as each team not only develops and maintains, but also deploys. Thus one team is responsible for entire life cycle of the microservice

Monolithic vs Microservices

If monolithic was bad, people would still not be doing it :)
Monolithic has its own advantages (as long as it is reasonable in size)*, such as :

- Easier to get the bigger picture of the application, as the entire code base is in one place
- You have to deploy only one jar/war file
- Relatively simple and straightforward to develop
- The dependency on network latency and security is greatly reduced

Microservices is a buzz word technology gaining a lot of traction. Based on the need, competence and the size of the applications, one may have to choose between Monolithic, SOA, Microservice, etc.

***What is a reasonable size? Unfortunately there is no common agreement on the same.**

Quick Summary

•Monolithic applications are deployed as a single file

•They have problems in deployment, scaling, technology, etc.

•Microservices is an architectural style in which the bigger single application is broken down into loosely coupled services

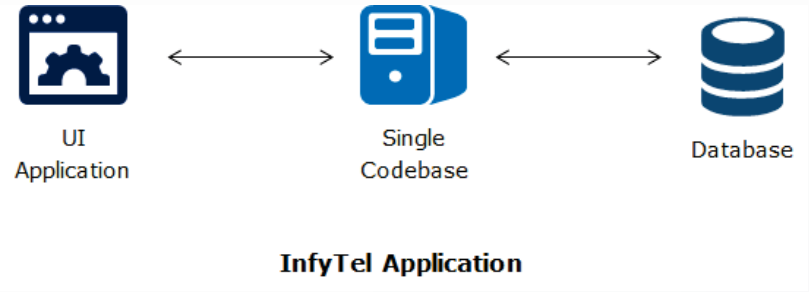
•These services are individually maintained, deployed, developed, tested

•These services intercommunicate using light weight protocols like REST.

•Each service can use its own set of technologies.

Infytel – Stage 1

In this course we will look at a telecom application called Infytel. As the course proceeds we will evolve this app from its monolithic architecture to full fledged microservice version in multiple stages. The first stage of the application is its monolithic version.



Issues with stage1

The monolithic version of our application, though working flawlessly, has issues. For example,

•If we have to scale up the add friend functionality what should we do?
- We will have to scale the entire application though only add friend functionality is facing a peak request rate. This leads to loss of resources

•If the get plan functionality can be better implemented in python, what should we do?
- This is difficult as we are stuck with one code base written in Java.

•If we need to update the logic of call details, what should we do?
- We end up stopping the entire application when we deploy the newer version. Thus even unrelated functionalities get affected

•If we make a modification to Customer functionality, what is the impact?
- We have to do a regression testing of the whole application

Infytel - Stage 2

How does one go about splitting a huge monolithic application into microservices? There are several considerations to choose from. Briefly they are:

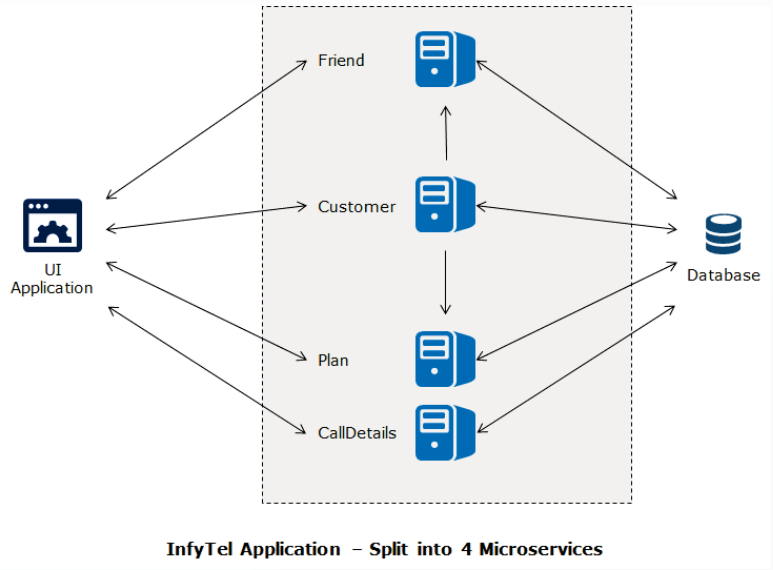
Service Boundaries: We have to split functionalities based on their bounded contexts. Bounded contexts define the scope of the services such that a change in one does not affect other services. This ensures that the services are autonomous and are loosely coupled with others. This is based on the Domain Driven design principles. Each application/domain is split into subdomains.

Single Responsibility: Each Microservice should do only one thing. It should be responsible for only thing.

Common Closure Principle: Functionalities that change should be packaged together so that changes to one does not percolate to other teams.

Infytel - Stage 2

In order to avoid the issues of the monolithic version, we are going to split the single monolithic application into 4 microservices (Customer, FriendFamily, Plan, Call-details). The application now looks like below:



InfyTel Application – Split into 4 Microservices

Infytel - Stage 2

Why did we split our monolithic into 4 microservices?

•The **call details** has a 'single-responsibility' of only dealing with call details of a given customer.

•The **plan** has a 'single-responsibility' of only dealing with plan related functionality of getting all plans and getting a specific plan.

•The **friend family** has a 'single-responsibility' of only dealing with adding and retrieving friends.

•The **customer** has a 'single-responsibility' of only dealing with customer functionality like login, view profile, register. Can we have add family in another and fetch family in another? The answer is maybe not, because if Family structure changes the both have to be changed.

The granularity of services is an architecture choice.

How to convert Infytel Monolithic to Microservices?

1.Create 4 different spring boot applications:

1. infytel-customer
2. infytel-calldetails
3. infytel-plans
4. infytel-friend-family

2.All of them will point to the same database, but they have their **own port and application names**

3.There is no change in the table script

1. The functionalities retained in different microservices are:

Functionality	Microservice
register,login,view profile	infytel-customer
view call details	infytel-calldetails
view all plans, view specification	infytel-plans
add friend, get all friends for a number	infytel-friend-family

Issues with Stage 2

Microservices are by nature distributed applications.
One immediate impact of microservices is whether the database should be shared or distributed?
Having a single shared DB is dangerous.
If the DB fails the entire application fails.

Hence the best practice is to not use a shared DB.

Also, microservices are supposed to be independent of each other with loose coupling. But in the approach we have taken,

- the Customer microservice is using FriendFamily table which actually belongs to FriendFamily microservice.
- the Customer microservice is also using the Plan table which belongs to the Plan microservice.

This is because Customer has a one to one relationship with plan table and one to many relationship with FriendFamily table through foreign keys.

This is tight coupling and not a good approach.

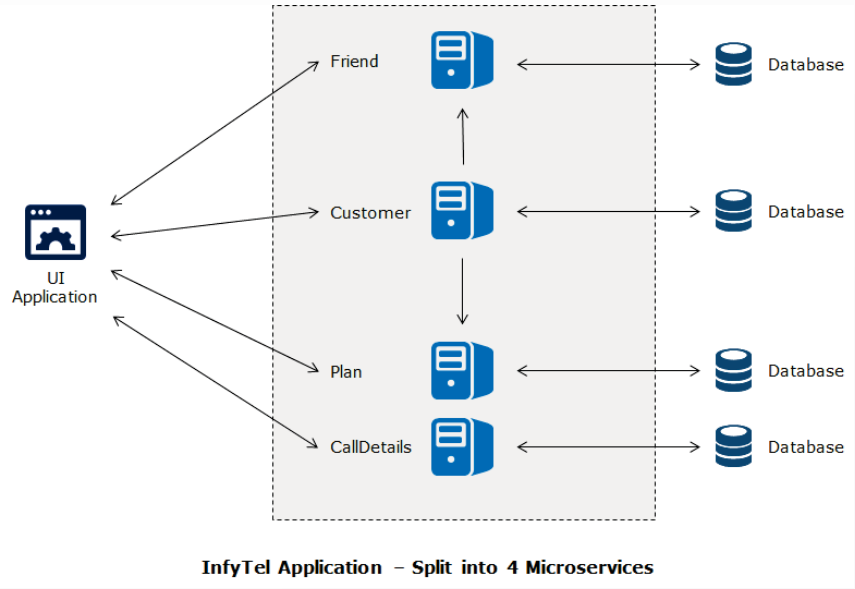
Solutions for DB issues

The right approach in dealing with databases in a microservice architecture is:

- Each microservice should have its own set of tables in a separate database or schema.
- One microservice should not try to directly access tables owned by another microservice. If there is a data dependency then a **microservice** should call another service through **REST api** to fetch data.
- Foreign key constraints should not be there. We can still have columns to establish link between entities.
- We should not use joins between tables owned by two different microservices.
- If you need data from both tables make a database call to fetch data owned by own service and make REST call to fetch data from the other. This pattern is called API Composition.

Infytel - Stage 3

In the next stage of our application, we are going to use individual schema for each microservice and the tables are placed in relevant schema. The updated application looks like below:



InfyTel Application – Split into 4 Microservices

How to use independent DB for Infytel microservice?

1.Remove foreign key constraints from all the tables

2.All microservices should have only entity classes related to their own functionality alone. Therefore remove other entity classes from the customer microservice

3.Since there is no foreign key constraints and no other entity classes in the microservices, add end points in the microservices which will provide relevant data.

4.View Profile end point in customer microservice currently fetches data from Friend and Family and Plan table directly. Now it needs to get these data from other microservices.

5.In Friend and Family microservice, create a new end point for fetching friend and family details for a customer

6.In Plan Microservice, create a new end point for fetching plan details for a customer

7.Modify View Profile end point in Customer Microservice to invoke these two new end points for fetch relevant details

Observations from Stage 3

So even though we have avoided joins in the table level, we have ended up increasing the REST end points.
Because of this the complexity in the application level has increased. Also, the number of network calls needed also has increased.
Initially the profile request involved one call to the customer REST endpoint and one call to the DB. Now it has become 3 calls to DB and 3 calls to endpoints in different microservices. Also, joining at API level is slower than table level join.

But despite these issues, we use this approach as the benefit of loose coupling and independence of microservices far outweigh these cons.

Why Cloud?

Deploying our application on cloud makes hardware maintenance much easier as it becomes the responsibility of the cloud provider. But just deploying it in cloud does not mean our application is taking advantages of the facilities given by cloud. For example, moving our application to cloud only relieves us from the problems of maintaining the underlying infrastructure. We can set up virtual machines with any configuration we need.

But cloud-enabled applications are not cloud-native applications. Cloud native applications are specifically written in such a way that it takes advantage of cloud. For example,

- Will your application take advantage of multiple instances of related services and load balance across them?
- In cloud, lot of things can go wrong. If a service is suddenly unavailable, is your application resilient enough?
- In cloud, instances can be increased, decreased, moved around and so on. Will your application automatically adapt to such changes?

If yes, then the application is cloud-native as it was designed to work on cloud advantages.
Our applications have to be cloud native, in other words, they should be written in such a way that it can make use of the advantages of cloud ecosystem. Elasticity, availability, security are some of the features we get by deploying our application in a cloud environment.
We have just seen how to create microservices using spring boot and how they can interact among themselves. As part of this course we are also going to see how we can develop such microservices such that they are cloud enabled.

Challenges with Cloud

•Since the services are deployed at random hosts and ports, how can Customer microservice know where to find the other microservices?

•Most of the configuration details of the 4 microservices are similar. How can we avoid this duplication?

•Since in cloud plan and friend-family are dynamically deployed, how will the customer microservice know where to find them?

•When we scale friend-family in cloud, how customer service can load balance the load across multiple instances of them?

•What if while fetching the profile the request friend-family service fails. How to bring in resilience?

•Since the application is now spread across as multiple microservices, how do we trace the flow of a request?

Cloud Solutions

We have many solutions for the challenges posed by cloud native applications. Some of them are:

- Netflix OSS
- Spring Cloud
- Vertx
- Restlet
- Akka
- Ninja

We will be using Spring Cloud in this course. Spring Cloud also uses open source components from Netflix.

What is Netflix OSS?

Netflix, the online entertainment industry giant catering to 62 million users, is also a tech giant. The company came up with many in house solutions to cloud microservice problems and they made their solutions open source. They are the pioneers of Microservice Architecture. Netflix OSS is the open source project for a variety of solutions including build, deployment, data analytics, etc.

All their solutions are currently being used in their product and have proven their effectiveness. Many organizations have adopted Netflix OSS solutions for their projects.

What is Spring Cloud?

Netflix OSS components are tough to use in its raw form. This is where Spring Cloud comes into the picture. Spring Cloud is actually a suite of projects from Spring. Some of the major projects under the Spring Cloud umbrella project are:

- Spring Cloud Config
- Spring Cloud Netflix
- Spring Cloud Security
- Spring Cloud Sleuth, etc

The Spring Cloud Netflix provides spring integration for the common components from Netflix OSS with Spring Boot

Storing Configuration

Let us consider configuration details of our microservices. All our microservices have the below configuration, apart from application name and port:

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/infytel
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
```

All these are placed inside the application itself. The problem with this approach is that if we have to modify any configuration then we have to change these configuration in multiple services and redeploy all these services. If a service has multiple instances then all the instances have to be redeployed.

Storing Configuration

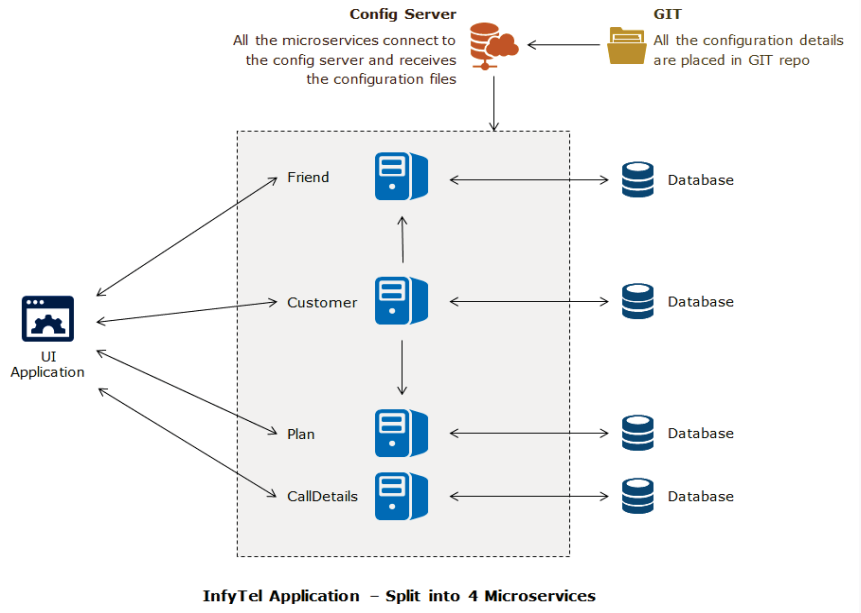
What are some of the alternatives in placing our configuration details?

- Placing the configuration as environment variables - but there is a limit on creation of environment variables
- Placing them in external files – but access to such file system is difficult in cloud

An ideal solution would be to use an external version control system like GIT, as it not only avoids the above mentioned problems but also gives us traceability of changes.

Cloud Storage - Stage 4

In the next stage of our application, we will use Cloud Config to access our configurations from GIT.



How to use Cloud Config?

1.Create a GIT repository
2.Create a file called application.properties in the repository
3.Place the common properties in application.properties file

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
```

4.Create separate properties file for each microservice with name of the file matching the spring.application.name of the service
5.Place the respective properties in their respective files
6.Create a spring starter project for config server with relevant dependencies

```
<dependencyManagement>
        <dependencies>
        <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Dalston.RELEASE</version>
        <type>pom</type>
```

```
<scope>import</scope>
        </dependency>
        </dependencies>
</dependencyManagement>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

How to use Cloud Config?

7.Add information of the git server in the properties file of config server

```
server.port=1111
spring.application.name=InfytelCloudConfigServer
spring.cloud.config.server.git.uri=https://github.com/<<username>>/infy-2.git
```

8.Add @EnableConfigServer annotation in the application file of config server

9.Add relevant dependencies to all the microservices

```
<dependencyManagement>
<dependencies>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Dalston.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
```

```
</dependency>
            </dependencies>
</dependencyManagement>
<dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

How to use Cloud Config?

10.Create a bootstrap.properties file in each microservice with a property for the config server

```
spring.application.name=CustomerMS
spring.cloud.config.uri=http://localhost:1111
server.port=8001
```

11.Remove all properties except spring.application.name from the individual services.
12.Run the application

Note on Cloud Configuration

When we use Cloud-Config, each microservice gets two properties files with the name 'application.properties'.
One in GIT and the other one present locally. Which one will take precedence?
In order to ensure that the properties file from GIT takes higher priority, it needs to be accessed first before the local properties file.
For this we use the bootstrap context.
Spring cloud creates a parent context to the spring application context called the ‘bootstrap’ context.
This context takes precedence over application context.
This context is responsible for loading configuration details from external source. Both these contexts share the Spring Environment, thus making configuration usage seamless.
Since the bootstrap context takes precedence, we have to mention the URI of the **config-server** in a **bootstrap.properties/yaml** in our clients.

Note on Cloud Configuration

Spring uses Environment to get the configuration details from various sources such as the environment variables, properties files, yaml files, etc. These are built in property sources. When we use cloud config, it just adds an additional property source to the Environment such that it takes the properties from the cloud config server.
When we use cloud config, this property source takes the higher priority. That means any duplicate properties in other property sources are ignored.
We can access the configuration files by using endpoint on the config server in any one of the below patterns:

```
http://<config_server_host>:<port>:/<application>-<profile>.yml
http://<config_server_host>:<port>/<application>-<profile>.properties
```

Note on Cloud Configuration

The properties file used in the GIT must have the same name as that of the clients **spring.application.name**
We can have multiple profiles as well and if a profile is not mentioned it will load the default profile, which is the same name as that of the **spring.application.name.**

If we try to access a microservice property for a given profile, the order of files used will be:

1.Yaml file for that profile
2.Properties file for that profile
3.application.yml file
4.application.properties file

For example, if we have a property called x=10 in application.yml and x=20 in application.properties, the final value used will be 10

The config-server is contacted by the clients only once, during the start of the project. Therefore any changes made to the configuration after the application starts will not be reflected in the application.

Configuring Cloud Config

We can also configure the clients retry attempts to contact the config server using the properties like

Property	Usage
initialInterval	Initially retry interval in milliseconds default 1000ms
multiplier	Multiplier for next interval. Default 1.1 times
maxInterval	Max interval for backoff. Default 2000ms
maxAttempts	Maximum number of attempts. Default 6 times

Configuring Cloud Config

If a port is not specified for the config server, it runs in its default port 8888. Also, if the cloud-config server is down, then the client will throw an error not during startup, but while trying to access a property at runtime. To avoid this we can have the failFast property set to true. By this the client will fail at startup time rather than at run time.

```
spring.cloud.config.failFast=true
```

Also, in order to avoid config-server to be a single point of failure, we usually deploy multiple instances of it to ensure high availability. If the cloud config server is unavailable, it will use the properties files in the individual applications as a fallback

Dynamic Configuration Changes

When we make any changes to the properties file in GIT, the config-clients automatically do not update themselves with the modified values. **This is because the configurations are taken only once at the time of startup.**

For example, if we modify the property of CustomerMS, we have to restart CustomerMS so that it can again fetch the properties from the config server. However, this is not a practical approach.

Dynamic Configuration Changes

When we make any changes to the properties file in GIT, the config-clients automatically do not update themselves with the modified values. **This is because the configurations are taken only once at the time of startup.** To overcome this we need to :

- Add @RefreshScope annotation on the bean which is using the property
- Add spring-boot-actuator end point dependency

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- Disable security to the refresh end point by adding the below property in the relevant microservices:

```
management.security.enabled=false
```

** We should not disable the management security in deployment. We will look at security of cloud-config later in the course.*

- Send a **POST** request to the /refresh end point of the service. This refreshes the microservice without restarting/redpolying it.

In this case we would add @RefreshScope on the CustomerController, add actuator dependency on the CustomerMS and send a POST request to http://localhost:8001/refresh which is where the CustomerMS is running

RefreshScope

The @RefreshScope annotation is needed only if we are using @Value. If we are getting the values from spring environment variable directly, we don't need this annotation. Every time we use getProperty() on spring environment variable, it fetches the latest value. However, we would still need to have the /refresh actuator endpoint and a send a POST request to it.

Spring Cloud Bus

The problem with using the /refresh endpoint is that we have to manually fire a POST request to it.
It is not automatically done when the configuration changes. Also we need to fire a request to /refresh of all the services which might get affected by the change in the property. That means we have to keep track of which property is used in which application. If we have 100 microservices using the property, we need to fire /refresh to all those microservices.
The solution is to use Spring-Cloud-Bus. This, along with Queuing service like RabbitMQ, will trigger refresh events on all dependent microservices. However, Spring-Cloud-Bus is beyond the scope of the curriculum.

Quick Summary

- Spring Cloud project allows us to write cloud compatible code
- One of its sub-project is Spring Cloud Config
- Cloud-Config allows to keep all configurations in central place
- Since this uses the Bootstrap context, we need to use bootstrap.properties file
- A config server contacts the GIT repo.
- The config clients contact the config server at start up to gather the configurations
- Changes made to configurations don't automatically reflect in the clients
- The configuration files use a specific order of priority.

Cloud Config - Exploration

Explore further concepts on Cloud Config:

How to use DB instead of GIT for cloud config?

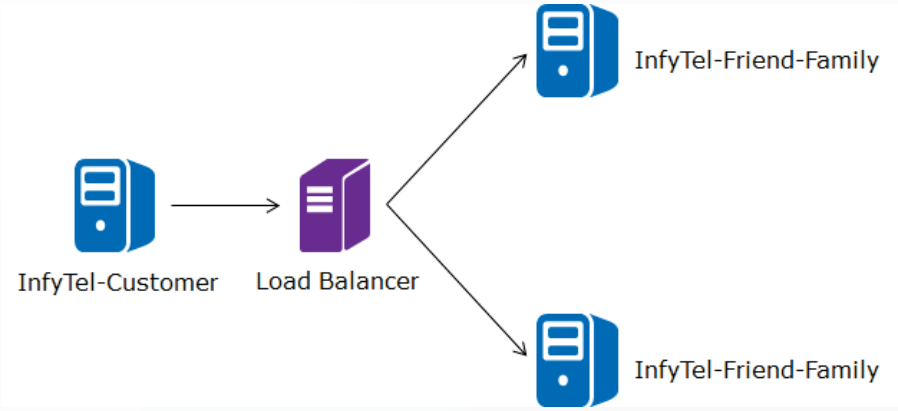
How to serve the properties in different formats?

Need for Load Balancing

Let us say that Infytel has announced an offer that customers with 5 friends will get a 5% discount. Since this is a limited time offer, there is a surge in people adding their friend details. To manage the load we are scaling by increasing the instances of infytel-friend-family from one to two. Now we have to find a way to balance the load of requests between these two instances. Else only one instance will get more or all the load and the other instance will become redundant.

Client Side vs Server Side Load Balancing

Since the number of instances has increased, the infytel-customer has to send requests to both these instances so that the load is properly balanced.
One way is to put a load balancer in front of the infytel-friend-family instances. This is usually a hardware load balancer. The infytel-customer would simply send the request to the load balancer and it is the responsibility of the load balancer to decide whom it should forward the request to. This concept is also known as **Server-Side load balancing**.



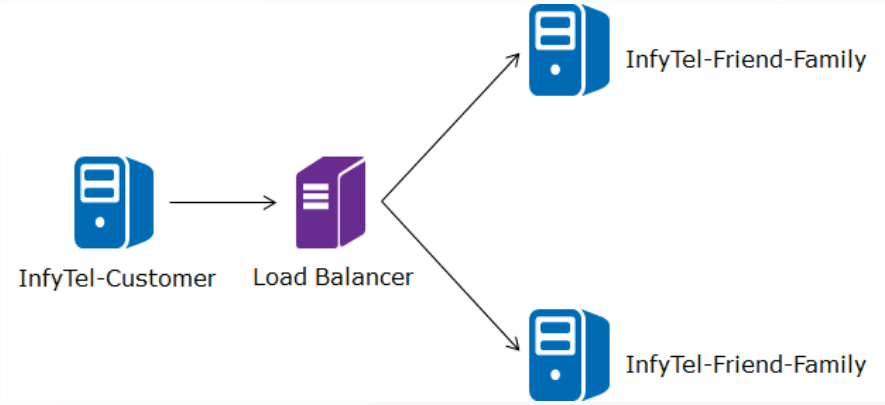
Client Side vs Server Side Load Balancing

Server side load balancing has several problems:

•If the load balancer fails, then we don't have access to any of the instances of the microservice

•Since each microservice would have a dedicated load balancer, we have to manage, track and maintain hundreds of such load balancers.

•It increases network latency. Now it would take two hops to reach the service. One to the load balancer and another from the load balancer



Client Side vs Server Side Load Balancing

Server side load balancers can either be hardware or software load balancers. They are usually hardware load balancers. Some examples of Hardware load balancers are:

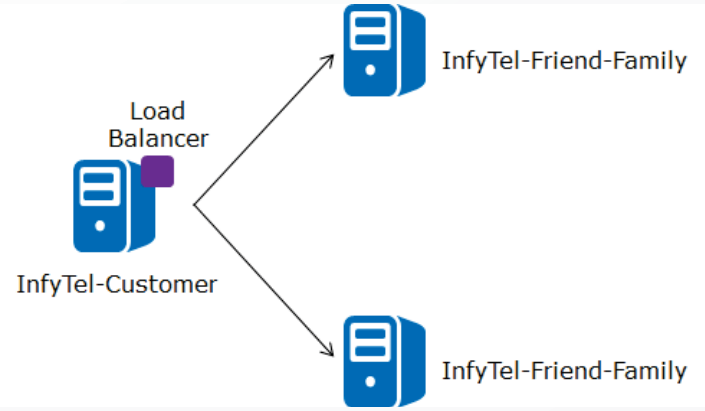
- F5 BIG-IP load balancer
- CISCO system catalyst
- Barracuda load balancer
- Coytepoint load balancer

Client Side vs Server Side Load Balancing

Client side load balancing is a natural solution to this. The client is responsible for deciding to whom it will send the request to. The client side load balancers are thus software load balancers and not traditional hardware load balancers. Of course the downside is we are mixing our application code with load balancing code.
Spring-Cloud Netflix has a client load balancer called Ribbon. We will use Ribbon to perform client side load balancing in our application.

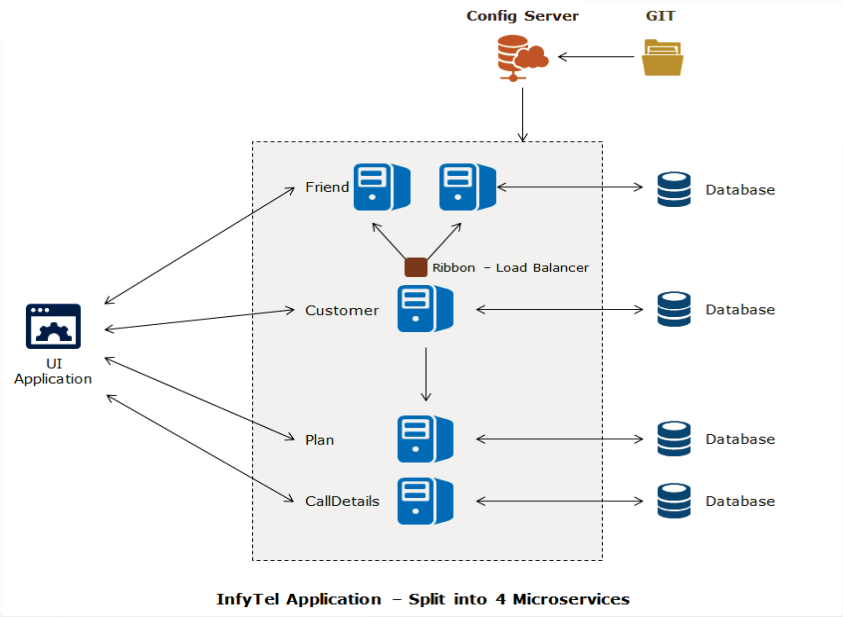
Some examples of software load balancers are:

- HAProxy
- NGINX
- mod_athena
- Varnish



Load Balancing using Ribbon - Stage 5

The below illustration shows our application using Ribbon.



How to use Ribbon?

1.Add dependency in the infytel-customer

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

2.Create a Configuration class with the below bean

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

3.Autowire the RestTemplate as @Autowired RestTemplate template; in the CustomerController
4.Add @RibbonClient(name="custribbon") annotation on the CustomerController class

How to use Ribbon?

5.Add the below properties in the properties file:

```
custribbon.ribbon.eureka.enabled=false
custribbon.ribbon.listOfServers=http://localhost:8001,http://localhost:8002
```

6.Autowire the rest template and update the infytel-friend-family invocation in CustomerController as

```
List<Long> friends =
template.getForObject("http://custribbon/customers/" +
phoneNo+"/friends", List.class);
```

7.Run the application, with the two instances of FriendFamilyMS running in two different ports

Notes on Ribbon

By default ribbon uses the NoOpPing strategy for checking if the services are up.
However, the NoOpPing is a dummy strategy. It assumes that all services are up. Thus it will keep pinging the services even if they are down.
We can configure the Ping strategy so that we stop sending requests to services which are down.
Also, Ribbon by default uses Round Robin load balancing strategy.
These things can be modified by adding a configuration file.

Configuring Ribbon

Ribbon can be configured as:

```
<clientName>.<nameSpace>.<propertyName>=<value >
```

For example:

```
custribbon.default.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule
```

Need for Service Discovery

In our CustomerMS, we had got the FriendMS URI from the cloud-config server. In a cloud situation, a instance may be provisioned and deprovisioned randomly. At one time the service may run in one port and after sometime, the cloud provider may shift it to another.

How can infytel-customer find out the current port and host where infytel-friend-family is running?

Even if we mention the port and path in a cloud-config server, it will not help our case as the values change dynamically. The service cannot go and modify the properties which are in GIT every time it changes and refresh all the relevant microservices. That will be very cumbersome.

Need for Service Discovery

The problem of dynamic configuration is very similar to that of the problem of changing phone numbers. Typically we store the phone numbers of our friends and colleagues in our mobile itself. It is like storing the configuration in the app itself.

Need for Service Discovery

If our friends change their phone numbers and don't inform us, we will get an error message when trying to dial the old number. This is like a service trying to contact another service on its outdated address.

Now imagine a global telephone directory app like Truecaller, etc. We get to know the details of a number even though we don't have that number stored with us. How does a global telephone directory work?

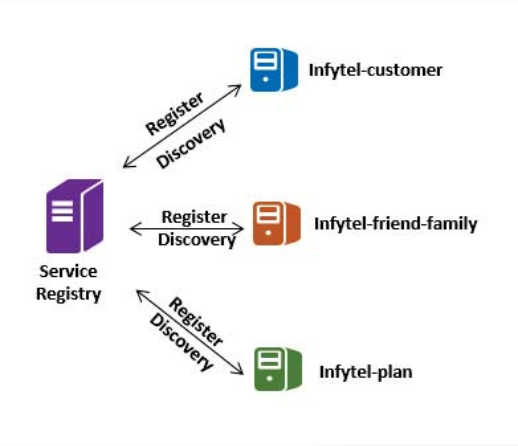
Need for Service Discovery

You have a service registry to which you can register yourself. To register yourself you have to provide your details like first name, last name, phone no, etc. Once you register you can discover details of all other registered numbers as well. Now if someone changes their number, if they register their new number with such a global directory, you again get their updated details automatically. This is how Service discovery works in microservices as well.

Service Registration and Discovery

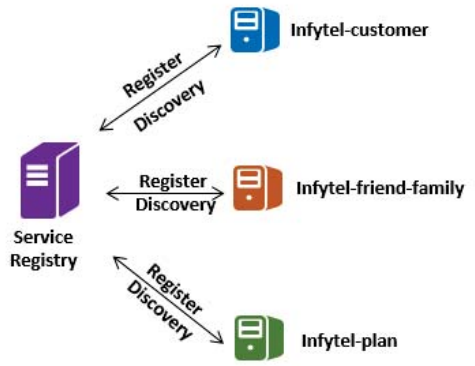
In Microservices, the solution to this problem is what is called as the service discovery pattern.
In this pattern, a service registers itself with a central server called the Service Registry. Now once it registers itself with the Service Registry two things happen:

- Its details like name, port, host, etc are stored in the service registry
- A list of other registered services become available to it



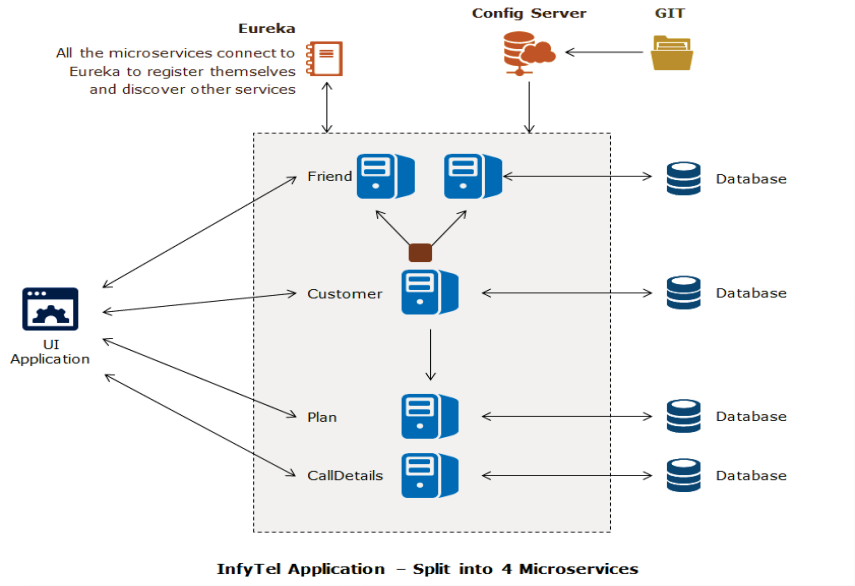
Service Registration and Discovery

Thus even if one if the services were to get redeployed at a different host and port, the other services need not worry about it. When the service redeploys, it would simply update its information in the service registry again. The other services would discover about its updated details through the service registry.
There are many service registry solutions like Netflix Eureka, etcd, ZooKeeper, consul, etc. In this course we will learn about Netflix Eureka.



Eureka – Stage 6

In the third stage of the application, we will register our microservices with a Eureka Service Discovery server. The details of the Eureka are also stored in the GIT repo which can be accessed using the ConfigServer



InfyTel Application – Split into 4 Microservices

How to use Eureka?

1.Create a Spring Starter project with name infytel-eureka
2.Add the below dependencies:

```
<dependencyManagement>
      <dependencies>
        <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-dependencies</artifactId>
          <version>Dalston.RELEASE</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
      </dependencies>
</dependencyManagement>
<dependency>
           <groupId>org.springframework.cloud</groupId>
           <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

How to use Eureka?

3.Add the below properties in the application.properties file of infytel-eureka.

```
spring.application.name=Eureka1
server.port=5555
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
eureka.client.service-
url.defaultZone=http://localhost:5555/eureka
```

4.Add @EnableDiscoveryServer annotation in the application file of infytel-eureka

5.Add the below dependencies in the microservices:

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

6.Add @EnableDiscoveryServer in all microservices application file

How to use Eureka?

7.Add the below property in the application.properties file in git

```
eureka.client.service-url.defaultZone=http://localhost:5555/eureka
```

8.Autowire Discovery client in CustomerController class as @Autowired DiscoveryClient client;
9.Remove the String friendUri; from CustomerController and update the code in accessing the friend-family-service as:

```
List<ServiceInstance>
instances=client.getInstances("FriendFamilyMS");
ServiceInstance instance=instances.get(0);
URI friendUri = instance.getUri();
```

Note: The service discovery happens through the spring.application.name value of the services. Hence they should not change.

10.Run the application

Notes on Eureka

@EnableDiscoveryClient annotation makes an application an Eureka instance as well as an Eureka Client. Every application registered with Eureka is an Eureka instance. Since every Eureka instance can also get details about other registered Eureka instances, it also becomes a client.

register-with-eureka property when set to true (which is by default), will register an application with the Eureka Sever. Such an application is also called a Eureka Instance. The Eureka Instance will start sending heartbeats to the Eureka Server. If the Eureka server does not receive heartbeats from a Instance within a configurable time limit (every 30 secs by default) , it considers the Instance to be down and deregisters it from the registry.

fetch-registry property will fetch the registry from the Eureka Sever once at startup time and will cache it. It will check the Eureka Server at regular intervals (by default at every 30 secs) to see if there are any changes. If there are changes, it fetches only the updates and the unchanged parts will be continued to be accessed from cache.

Notes on Eureka

Every registered client gets access to what is known as a Discovery client. The discovery client is actually a service endpoint, which returns an enum of all ServiceInstance instances of the clients registered with the service registry. You can take a look at all the instances registered with the service registry using this end point as follows:

http://{eureka-host}:{eureka-port}/{eureka}/apps/{spring-application-name}

Quick Summary

- In a cloud environment, we cannot predict the host and port of the different microservices.
- By using the service discovery pattern, we can dynamically find the services registered
- Netflix Eureka allows us to create a Service Registry
- Clients can register themselves and discovery other instances

Using Ribbon With Eureka

Ribbon is typically used along with Eureka. Earlier we had seen how we can use Ribbon with a static list of servers. Instead of using a static list, we can get a dynamic list of servers by using it with Eureka.
When used with Eureka, not only will Ribbon get the server list, but also will depend on the Eureka to know if a service is up or not.

How to use Ribbon with Eureka?

1.Remove Autowire of DiscoverClient and Autowire load balanced RestTemplate
2.Remove code for getting URI from discovery client
3.Access the PlanMS and FriendMS through rest template object and use the names of the service instead of the URI. Since Eureka is used, the URI will be picked automatically based on the service name.

```
PlanDTO planDTO=template.getForObject("http://PLANMS"+"/plans/"+custDTO.getCurrentPlan().getPlanId(),
PlanDTO.class);
List<Long> friends=template.getForObject("http://FRIENDFAMILYMS"+"/customers/"+phoneNo+"/friends",
List.class);
```

Eureka Cluster

Eureka is rarely run as a single instance, as it would become a single point of failure. Typically we run multiple instances of Eureka forming a cluster. In a cluster, each Eureka server replicates the information in the other servers.

How to cluster Eureka?

1.Open the hosts file in C:\Windows\System32\drivers\etc
2.Add the below hostnames:

```
127.0.0.1    Eur1
127.0.0.1    Eur2
127.0.0.1    Eur3
```

3.Use the given yml file in the infytel-eureka server

```
spring:
  profiles: Eureka1
  application:
    name: Eureka
server:
  port: 2222
eureka:
  instance:
    hostname: Eur1
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://Eur2:2223/eureka/,http://Eur3:2224/eureka/
```

How to cluster Eureka?

```
spring:
  profiles: Eureka2
  application:
    name: Eureka
server:
  port: 2223
eureka:
  instance:
    hostname: Eur2
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
   defaultZone: http://Eur1:2222/eureka/,http://Eur3:2224/eureka/
```

```
spring:
  profiles: Eureka3
  application:
    name: Eureka
server:
  port: 2224
eureka:
  instance:
    hostname: Eur3
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
  defaultZone: http://Eur1:2222/eureka/,http://Eur2:2223/eureka/
```

4.Comma separated values in the defaultZone indicate peer awareness. Eureka1, Eureka2 and Eureka3 are peers of each other and hence will replicate the details across each other.

How to cluster Eureka?

5.Update the application.properties file in GIT for the below property:

```
eureka.client.service-
url.defaultZone=http://Eur1:2222/eureka,http://Eur3:2223/eureka,http://Eur3:2224/eureka
```

6.Run all the three profiles of the Eureka server and restart all the microservices
7.You will get three dashboards in three different Eureka ports. Since we have a cluster, each dashboard will have the same details of microservices as the other two Eureka Servers in the cluster.
8.Bring down a microservice. You will find that since each Eureka server in a cluster replicates itself, all Eureka servers in the cluster will now have the same updated information.

Need for Resilience

Imagine a situation where the CustomerController is trying to contact the infytel-friend-family service. Due to either a network issue or the server is slow or the server is down or there is some error the CustomerController is unable to contact the service.

Now what should be the course of action? If we keep sending the requests to infytel-friend-family thinking that it will start working after sometime, that is insanity. We need to add resilience to our application such that the application can deal with such error situations. A try catch block can handle errors. But if a request is repeatedly causing an error, should the request even be continued to be sent?

Cascading Failure

You are hurrying in the morning, leaving home to office. You try starting your vehicle, and it does not start. What do you do?

1.You keep trying till it works
2.You try a few times. If it does not work, you stop trying.
3.You try a few times. If it does not work, you stop trying. You come in the evening and try it again a few times, just to be sure.

Cascading Failure

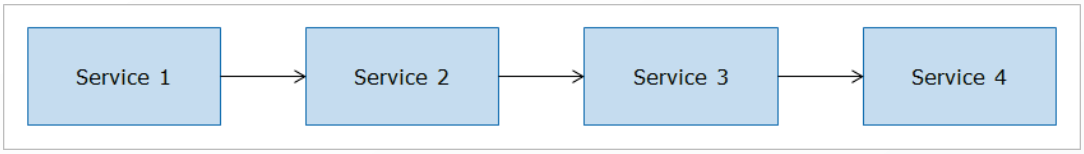
If you chose option A, then

- You miss your meeting with the VP that morning.
- Since you miss your meeting, you get rated bad in your work
- Since you get rated bad in your work, your work deteriorates further
- Since your work deteriorates further, you lose your promotion

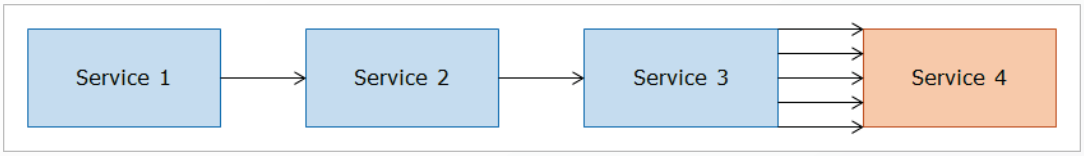
To prevent such cascading failure, walk away when you can!

Cascading Failure

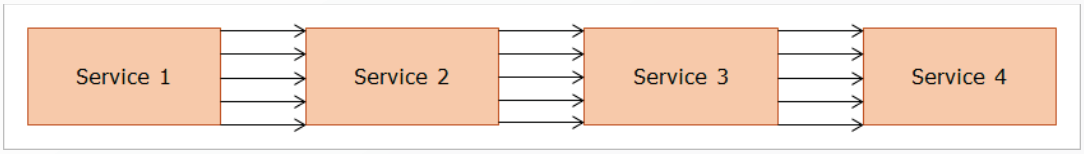
Cascading Failure may happen in microservices as well. Imagine, the below scenario:



Let us say that Service D has become slow. Thus the requests to service D start queuing up.

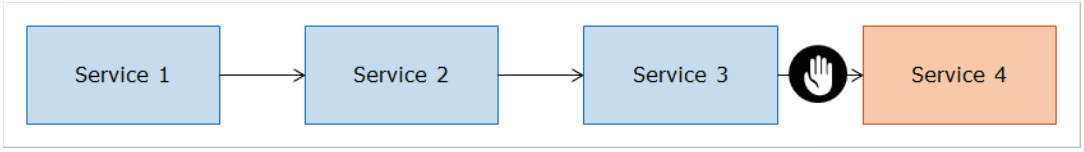


Because of this latency, the requests start queuing up in all the related services slowing everything down. This is cascading failure.

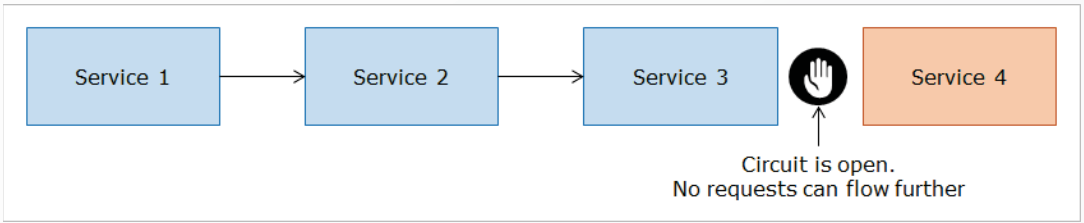


Cascading Failure

The better approach would be that if a particular service is taking more time than usual, then don't send anymore requests to that service. Stop requests to that service. This prevents an increase in slowing down of other services.



The same principle is used in our electrical appliances as well. When a high voltage passes, the fuse trips thereby preventing other appliances from burning out. When the fuse trips, the circuit is open and no current passes through it. When the fuse is reset, the circuit is closed and the current starts passing through it again. This pattern when applied in fault tolerance is called as the Circuit breaker pattern.



Circuit Breaker Pattern

When does a fuse trip in a house? When the current flow exceeds a threshold. Similarly, in microservices communication, when the number of errors in a given time frame is beyond an acceptable limit, the circuit opens, thereby preventing further flow and protecting other parts of the application.

Hystrix -Stage 7

Using Netflix Hystrix, we can apply a circuit breaker pattern in our application. Hystrix will open the circuit when the numbers of failures in a given time frame are more. Hystrix uses the Fail Fast approach. It is better to fail fast than to fail big time later.

After opening the Circuit, Hystrix will attempt to close the circuit again. Just like you coming and checking the bike in the evening, Hystrix checks if there is any change in the status quo by sending a single request again. If that fails, it opens the circuit again and waits again.

The error threshold, waiting time, retry attempts, etc are all configurable in Hystrix.

Fallback

When you find that your vehicle is not starting after a few attempts, do you just stay back at home? No. You try to salvage the situation. You try to take a taxi or a bus. Of course, this means you have to either sacrifice your money or time. Not the ideal situation you would want yourself in. But hey!, at least you still get to meet the VP!

Fallback

This sort of alternate arrangement when the circuit is open is called a Fallback pattern. Hystrix allows you to mention any alternate piece of code that you wish to run if a service is down. Obviously you don't get the same result as you wish you had. But, providing some form of data instead of an error is better.

Thus Hystrix allows you to degrade gracefully. The user continues to have access to the application, but some of the features will be unavailable temporarily. Compared to FailFast this is the FailSilent pattern.

Fallback executes when:

- An error occurs
- A timeout occurs
- Circuit opens

How to use Hystrix?

1.Add dependency in the infytel-customer service

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

2.Add the @EnableCircuitBreaker annotation in the application class of infytel-customer. This tells Spring Cloud that the application uses the Circuit Breaker pattern

3.Create a service class called CustomerCircuit with @service annotation

4.Add the below method in the service class with @HystrixCommand annotation. This annotation makes the method call fault tolerant. This annotation will work only if the class is component or service

```
@HystrixCommand(fallbackmethod="getFriendsFallback")
public List<Long> getFriends(Long phoneNo){
                List<Long> numbers=
template.getForObject("http://FriendFamilyMS/friends/"+phoneNo, List.class);
                return numbers;
}
```

How to use Hystrix?

5.Create the below method in the same class:

```
public List<Long> getFriendsFallback(Long phoneNo){
                        return new ArrayList<Long>();
}
```

6.Add the below properties in the CustomerMS.properties

```
hystrix.command.default.circuitBreaker.requestVolumeThreshold=10
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=10000
```

7.Add @Autowired CustomerCircuit circuit; in the CustomerController
8.Update the call from CustomerController as shown below:

```
List<Long> friends=circuit.getFriends(phoneNo);
```

9.Run the code

Notes on Hystrix

Once the specified number of requests (threshold volume) are sent within the specified time (rolling stats time window) and if the specified percentage of requests end up as errors (error percentage), the hystrix opens the circuit. Once the circuit is open, no more requests will be sent to the infytel-friend-family service. After a specified time interval (sleep window), hystrix will again close the circuit and pass one request. If that request fails, then again circuit is automatically closed for the specified time again. This repeats in a cycle.

Configuring Hystrix

Hystrix can be configured using @HystrixProperty annotation and using some of the properties like:

•execution.isolation.thread.timeoutInMilliseconds - Time in milliseconds at which point the command will timeout and halt execution.

•circuitBreaker.sleepWindowInMilliseconds- The time in milliseconds after a HystrixCircuitBreaker trips open that it should wait before trying requests again.

•circuitBreaker.errorThresholdPercentage- Error percentage threshold (as whole number such as 50) at which point the circuit breaker will trip open and reject requests.

Service Communication

The customer microservice talks to both the friend-family service and the plan service. These communications are happening synchronously. That means, only after friend-family service request completes, we can send the request to the plan service. In situations where one service can be completed independent of the other, we can use asynchronous communication. Though Asynchronous communications reduce overall time, we cannot use it indiscriminately. For example, it cannot be used if one service depends on the data from another service call.

Asynchronus Communication - Steps

1.Create the below method in the hystrix service class:

```
public Future<PlanDTO> getSpecificPlans(int planId) {
                        return new AsyncResult<PlanDTO>() {
                                @Override
                                public PlanDTO invoke() {
                                        return
template.getForObject("http://PLANMS"+"/plans/"+planId, PlanDTO.class);
}};}
```

2.The method returns a Future. Future is a datatype which indicates that the result will be available in future. You can read about it here

3.To return a future value, the method returns a AsyncResult. AsyncResult is an abstract class. Hence we are returning an anonymous implementation of it, while overriding its invoke method. The invoke method does the actual code execution.

Asynchronus Communication - Steps

4.In the CustomerController, we process the result of the Future as :

```
Future<PlanDTO>
planDTOFuture=hystService.getSpecificPlans(custDTO.getCurrentPlan().getPlanId());
custDTO.setCurrentPlan(planDTOFuture.get());
```

5.The same is done for FriendFamilyMS invocation as well.

Quick Summary

•In a distributed microservice application, failure can result in cascading effect

•The app should be resilient to failures. In other words, a failure of one system should not bring down the entire app

•Hystrix uses the circuit breaker pattern to take care of latency issues and failures.

•When it detects that the threshold of failures has breached, it opens the circuit and prevents further request from reaching it.

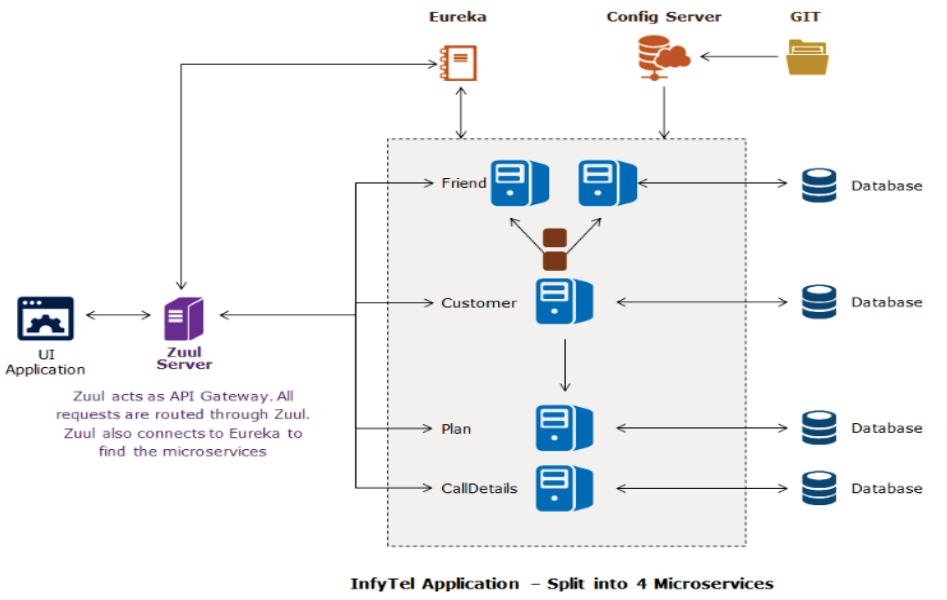
•It uses fail first approach.

•We can adopt a fail silent approach by using a fall back

Need for API Gateway

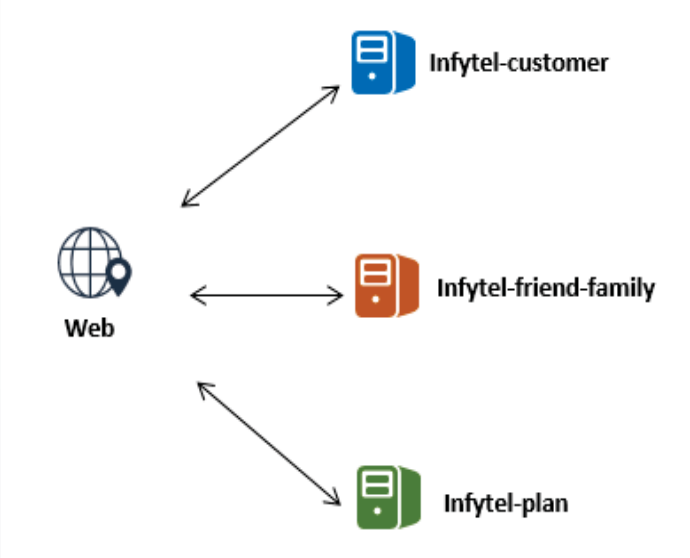
Need for API Gateway

- In our application, the UI application has to send requests to the different microservices directly. But since the microservices are in cloud, the host and port are changing frequently. Hence it is not sustainable for the UI application to talk to the microservices directly.
- The better approach is the UI application sends its request to a proxy server which then forwards the request to appropriate microservices. This type of proxy is also called as reverse proxy.



Need for API Gateway

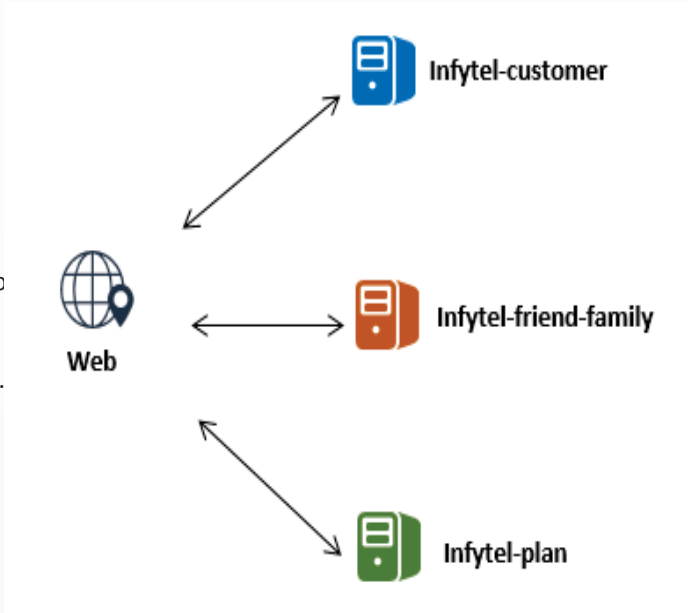
- In a microservice architecture, the client may have to send requests to multiple services directly to get the final data.



Need for API Gateway

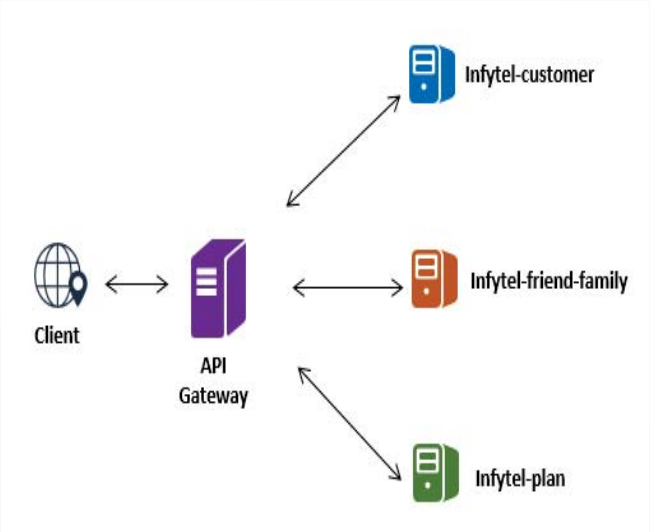
There are several problems with this approach:

- The client is now responsible for both gathering, aggregating and formatting the content
- The client must know the port and host of the services. If they change constantly then the client code also has to change
- If we decide to refractor the services later on by merging two services together, it would break the client
- The services may not always respond back in HTTP protocols. Thus the protocols may be incompatible with a browser
- The number of requests fired by the client is more
- Different clients need different data



API Gateway Pattern

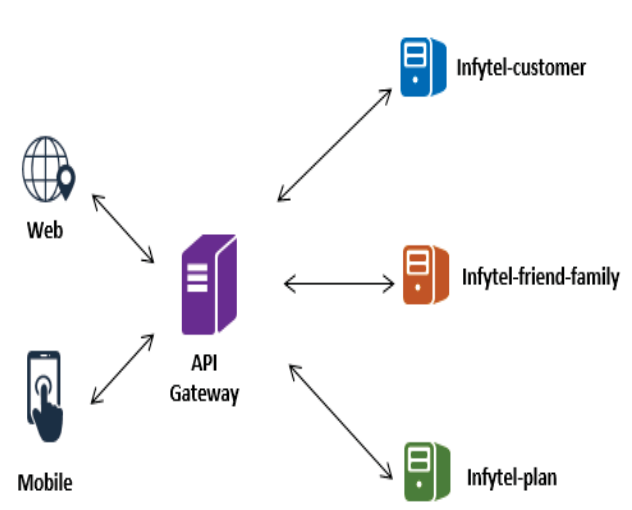
The API gateway pattern is a solution to the problems of a client invoking microservices. In a API gateway pattern you have a API gateway server which comes in between the client and the services.



API Gateway Pattern

The API gateway performs complex tasks including:

Intelligent routing. When a request comes from the client it intelligently routes it to the appropriate services



API Gateway Pattern

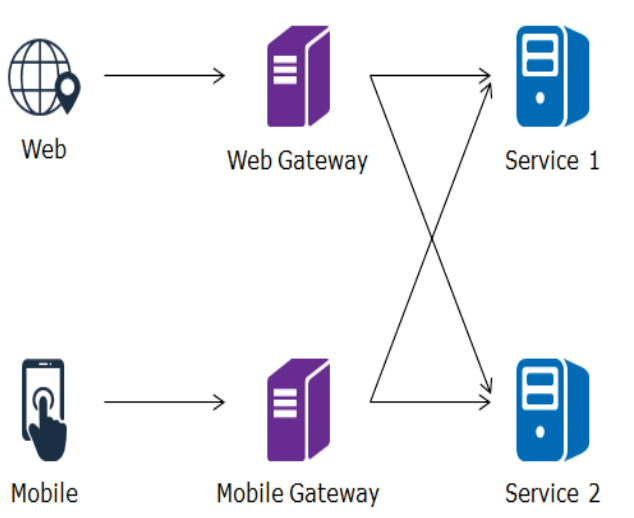
It also does:

- Request Aggregation. Based on a single request from the client, it invokes multiple services, aggregates the result and sends it back
- Protocol translation. It will be responsible for taking data from a service through let's say AMQP and sends the data to the client over HTTP
- Security
- Load balancing

In this course we will focus on creating a API gateway with only the Intelligent routing and load balancing capabilities through Netflix Zuul library.

What is BFF?

A BFF or Backend for Front End is a variation of the API gateway pattern. Instead of having a single API gateway for all types of UI applications, we can have dedicated API gateway for Dekstop, mobile, etc. The advantage is that the UI application and the API gateway will be maintained by the same team and they can configure the routes to their specific needs.



How to use Zuul?

1. Create a new spring boot starter application called infytel-zuul

2. Add the below dependencies apart from the dependency management, eureka and cloud config dependencies.

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

3. Add the below annotations in the application file of infytel-zuul

```
@EnableZuulProxy
@EnableDiscoveryClient
```

4. Routes can be added using the zuul.routes.<routeName>.path=/<URI> and zuul.routes.<routeName>.service-id=<ServiceName> . by using strip-prefix=false we can avoid repetition in the path.

5. /<URI>/** will match anything after the given URI. Sometimes a single * can be used as a wildcard character : /URI/*/hello. You can refer more here

How to use Zuul?

6. Add the below routes to the properties file

```
spring.application.name=ZuulServer
server.port=8001
zuul.routes.customer_create.path=/customers
zuul.routes.customer_create.service-id=CustomerMS
zuul.routes.customer_login.path=/customers/*/login/*
zuul.routes.customer_login.strip-prefix=false
zuul.routes.customer_login.service-id=CustomerMS
zuul.routes.customer_profile.path=/customers/*
zuul.routes.customer_profile.strip-prefix=false
zuul.routes.customer_profile.service-id=CustomerMS
zuul.routes.friends_customer.path=/customers/*/friends
zuul.routes.friends_customer.strip-prefix=false
zuul.routes.friends_customer.service-id=FriendMS
zuul.routes.calldetails_customer.path=/customers/*/calldetails
zuul.routes.calldetails_customer.service-id=CallDetailsMS
zuul.routes.plan.path=/plan/**
zuul.routes.plan.service-id=PlanMS
zuul.ignored-patterns.customer=/customerms/**
zuul.ignored-patterns.plans=/planms/**
eureka.client.service-url.defaultZone=http://localhost:5555/eureka
```

How to use Zuul?

7. Update the UI application code to use the host of Zuul

8. Run the application

Notes on Zuul

Zuul automatically uses Ribbon. Thus all requests through Zuul are load balanced. Zuul also automatically uses Hystrix for resilience. We can add fallbacks for Zuul routes.

Zuul Configuration

The below is one of the common set of configurations you can have on Zuul:

Adding a prefix:

```
zuul.prefix=/infytel
```

Through this configuration, all requests must be prefixed with infytel.

Modifying the route names:

By default the name of the route should match the name of the Service registered in Eureka. If we want we can map a name to the existing service. For example, in the below configuration, we have configured that any request of path /cust/* must be routed to CustomerMS

```
zuul.routes.CustomerMS.path=/cust/**
```

How to use Zuul and Hystrix

1. Create the below configuration class.

2. The class should have a ZuulFallbackProvider bean. This bean should implement two methods. getRoute() method should return "*" or null if we wish to apply this for all routes. Else we should mention the specific route name alone.

3. The other method should be fallbackResponse() which return a ClientHttpResponse() object.

How to use Zuul and Hystrix

```
@Configuration
public class ZuulConfigClass {

        @Bean
        public ZuulFallbackProvider zuulFallbackProvider() {
                return new ZuulFallbackProvider() {

                        @Override
                        public String getRoute() {
                                // TODO Auto-generated method stub
                                return null;}

                        @Override
                        public ClientHttpResponse fallbackResponse() {
                                // TODO Auto-generated method stub
                                return new ClientHttpResponse() {

                                        @Override
                                        public HttpHeaders getHeaders() {
                                                // TODO Auto-generated method stub
                                                HttpHeaders headers = new HttpHeaders();
                                                headers.setContentType(MediaType.TEXT_PLAIN);
                                                return headers;
                                        }
```

How to use Zuul and Hystrix

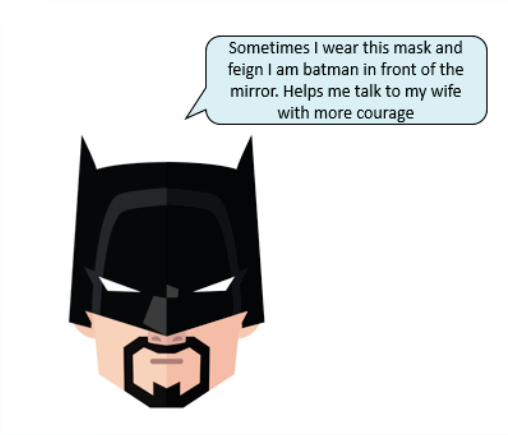
```
@Override
public InputStream getBody() throws IOException {
		// TODO Auto-generated method stub
		return new ByteArrayInputStream("Sorry. Something went wrong".getBytes());		}

@Override
public String getStatusText() throws IOException {
		// TODO Auto-generated method stub
		return "OK";}

@Override
public HttpStatus getStatusCode() throws IOException {
		// TODO Auto-generated method stub
		return HttpStatus.OK;}
@Override
public int getRawStatusCode() throws IOException {
		// TODO Auto-generated method stub
		return 200;}
@Override
public void close() {
		// TODO Auto-generated method stub
}};}};}}
```

Zuul - Exploration

Explore further about Zuul here:

1. How to intercept requests in Zuul?
2. How to get details of all mapped routes?

Need for Declarative Client



Problems with RestTemplate

We have been using RestTemplate to talk to other microservices. But there are several problems with that approach:

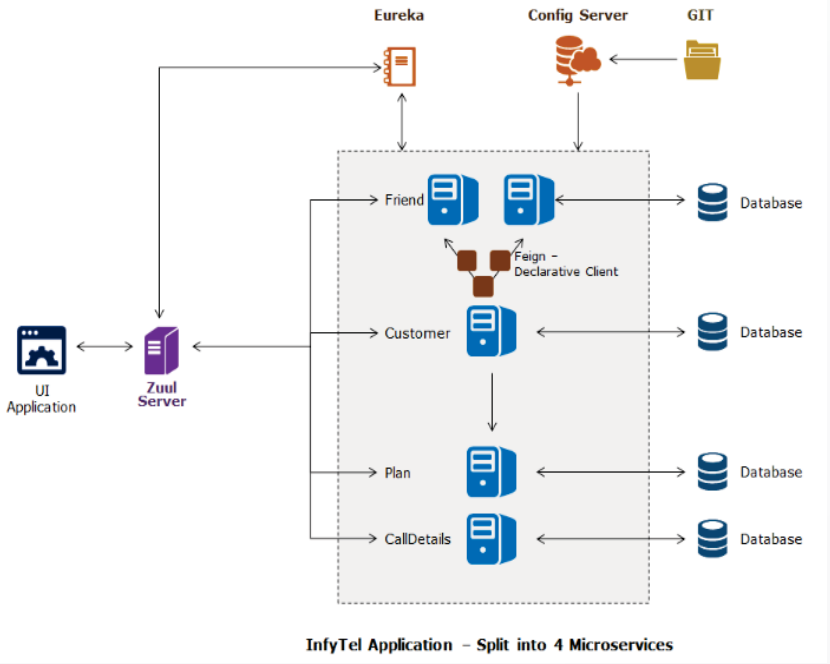
- You have to be aware of the various methods of the Rest Template API to use it.
- You need a separate bean for Load balancing
- You need a separate service for Circuit breaker
- The header details of a request from Zuul are not forwarded to the other microservices using RestTemplate

Hence we need a form of contacting other microservices which makes it easier by avoiding the above mentioned problems

What is Feign - Stage 11

- Feign is a declarative client from Netflix. It is declarative because we as developers declare the api's for contacting other microservices. We define the rules in the form of our own interfaces. At runtime, Feign will create implementation for our interfaces automatically Thus with minimal code and self made interfaces, we can have greater control on how one microservice communicates with the other.

- Feign automatically uses Ribbon. Thus all calls are automatically load balanced. Feign also works well with Hystrix. With appropriate dependency and configuration, Feign will automatically use circuit breaker and fallback fo all calls without the need for a separate service class.



How to use Feign?

We have been using the RestTemplate for making our calls to other services. But when used with Ribbon and Hystrix, it becomes cumbersome. Netflix provides a easier to use client called Feign. The Feign client automatically integrates with Ribbon and provides load balancing.

It can also be Hystrix enabled so that the calls are wrapped around Hystrix commands..

How to use Feign?

1. Create an interface

```
@FeignClient("PlanMS")
public interface CustPlanFeign {

        @RequestMapping(value="/plans/{planId}")
        PlanDTO getSpecificPlan(@PathVariable("planId") int planId);

}
```

2. The interface must be annotated with @FeignClient and must mention the service name for which we are writing the Feign Client.
3. We need not provide an implementation for this interface. The framework will provide a dynamic implementation at run time.
4. Autowire the feign client as:

```
@Autowired
  CustPlanFeign planFeign;
```

5. Invoke the plan microservice as:

```
planFeign.getSpecificPlan(planId);
```

Security in Microservices

Security in Microservices

Securing a monolith is very different from securing a Microservice application. In a monolith the client access the application providing the credentials and this is used in a session and checked with a database. As long as the user data is in session, he is authorized to access the functionalities exposed by the app.

In a microservice, the client contacts only the API gateway server. Though securing the API gateway is good, it is not good enough against internal threats. The services cannot know the source of the request. They cannot determine if the request is coming from a user or another service or some malicious code.

How to secure API Gateway?

- Add the below dependency in the infytel-zuul server

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

- Add the below annotations to the infytel-zuul application

```
@EnableOAuth2Sso
@EnableZuulProxy
```

- Add the below to the application.yml file

```
client:
   clientId: bd1c0a783ccdd1c9b9e4
   clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
   accessTokenUri: https://github.com/login/oauth/access_token
   userAuthorizationUri: https://github.com/login/oauth/authorize
   clientAuthenticationScheme: form
resource:
   userInfoUri: https://api.github.com/user
```

- Run the application

How to secure individual services?

1. Add the below dependencies to all microservices

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

2. Add the @EnableResourceServer annotation to all microservices application file

3. Add the below configuration in the application.yml file

```
client:
   accessTokenUri: https://github.com/login/oauth/access_token
   userAuthorizationUri: https://github.com/login/oauth/authorize
resource:
   userInfoUri: https://api.github.com/user
```

Securing Individual microservices

When we secure individual microservices, we face the issue of secure communication between these services. The authentication header which flows from the Zuul server to the microservice does not get automatically forwarded when one microservice talks to another. We have to programaticaly extract the headers received from Zuul and add it when we make a request through RestTemplate.

Securing Cloud Config

While we have secured the individual services, we also need to secure the configuration server so that one does not directly access the property files by using the endpoints of the config server. This can be done by using basic authentication and configuring username and password for it.

Encrypt Decrypt properties

We are storing sensitive information, including DB credentials, in a public repository. This is not very secure. The solution is to encrypt the values and place them in Git. Once encrypted we can either make the config server decrypt the values for us or it can be left to the individual services to decrypt it themselves.

Spring Cloud Microservices Overview

Now that we have seen several components provided by Spring Cloud, lets do a recap of how all of them work together. We will look at two things:

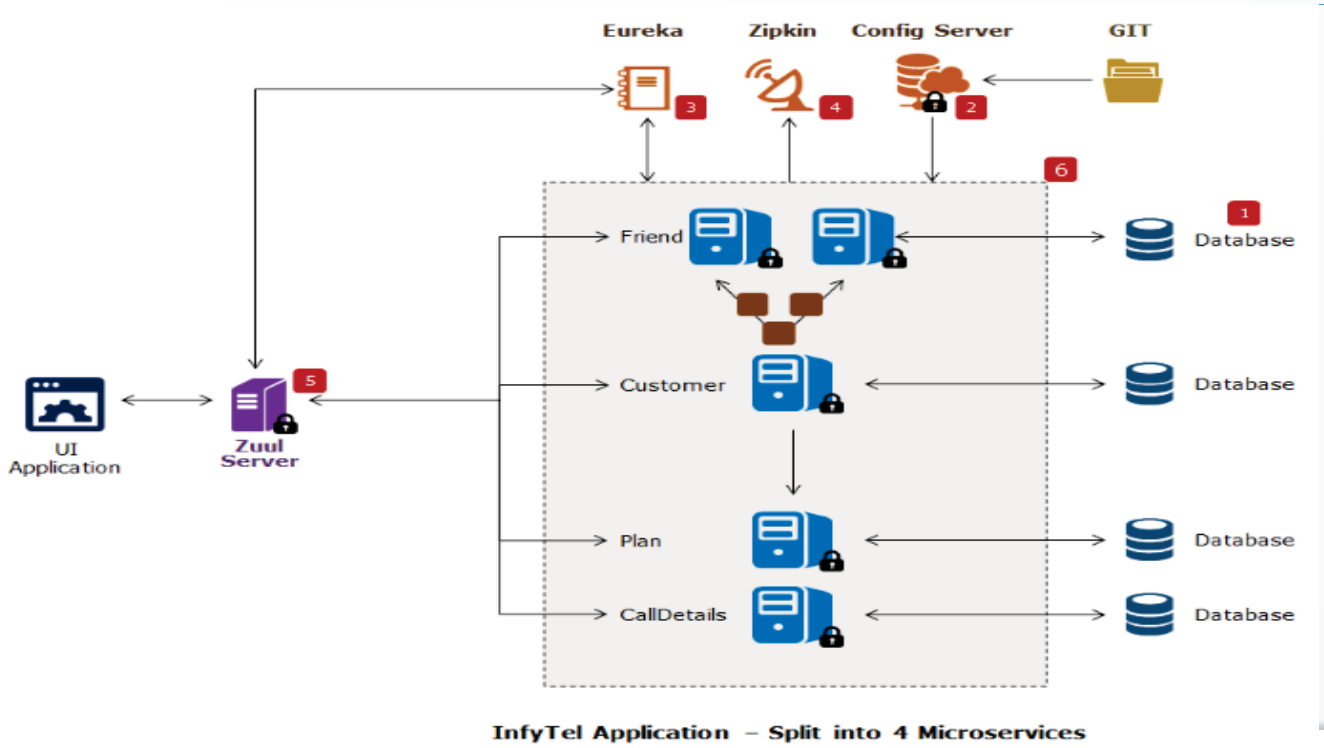
1. What happens when a Spring Cloud system starts? Since these servers have interdependencies, there is a specific order in which these servers have to be started.

2. What happens when a request hits this Spring Cloud system?

Startup order

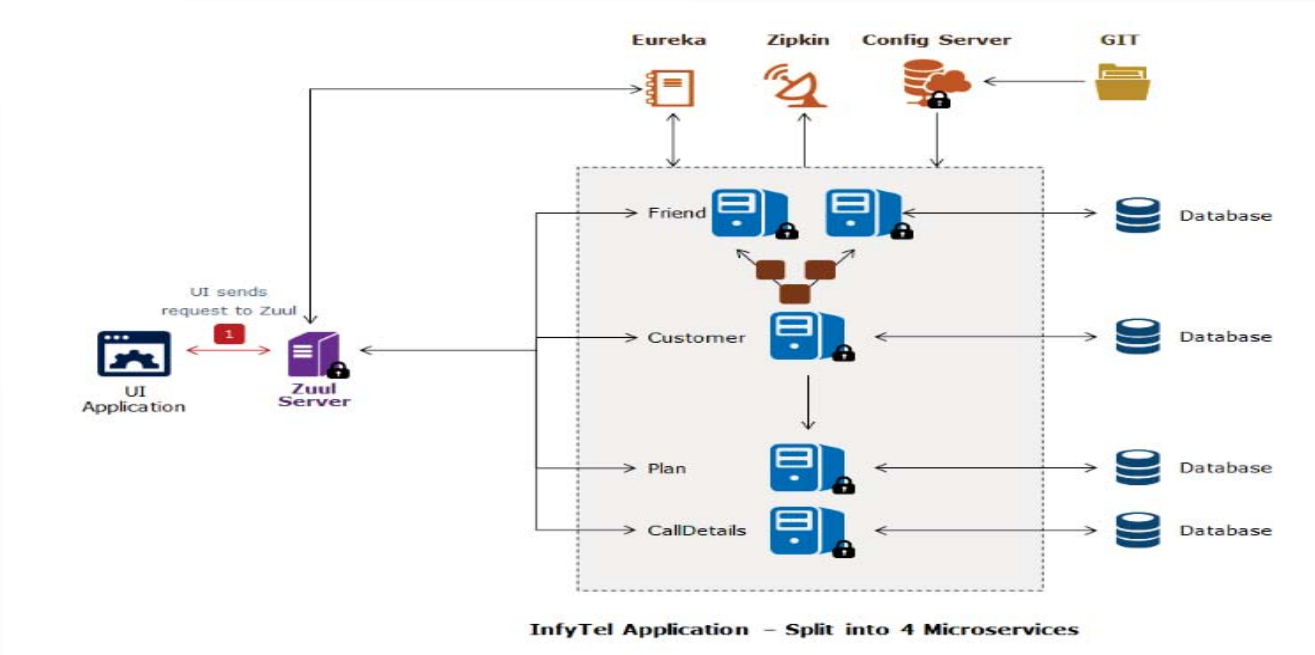
- The startup order of various components are very important. For example, if the microservices are started before the config server, then the microservices will not get the property files from the git repo.
- The diagram shown in the next slide shows the right startup order of various components in the application.

Startup order

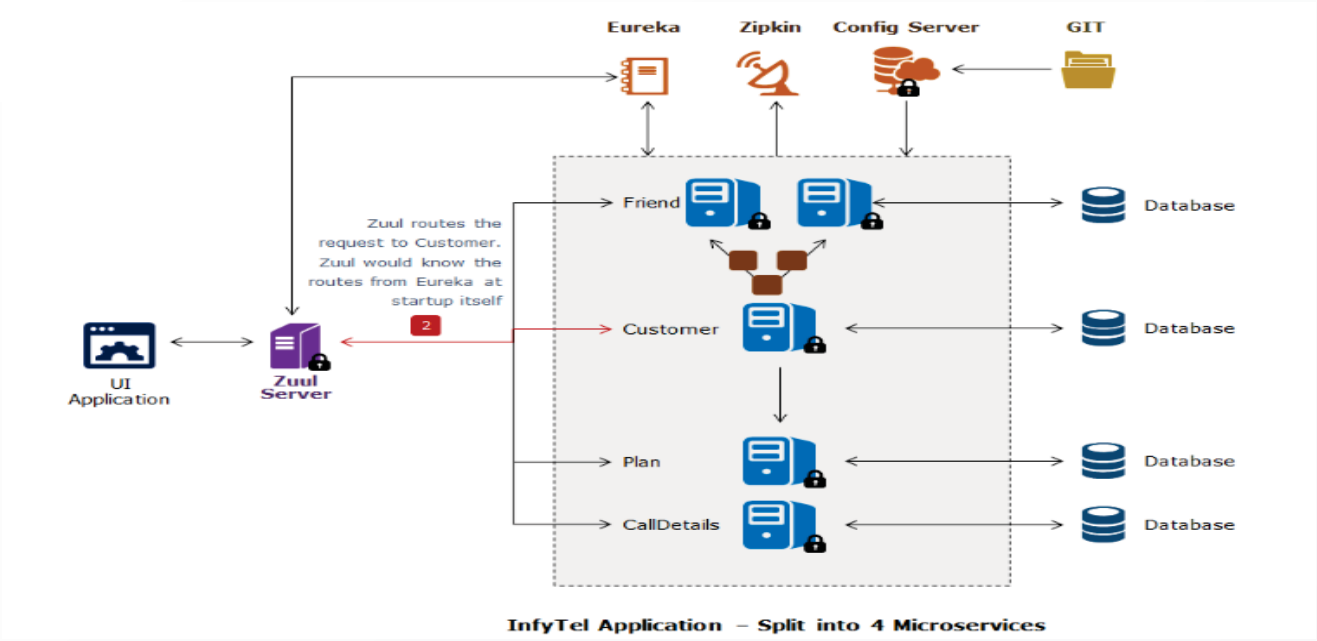


InfyTel Application – Split into 4 Microservices

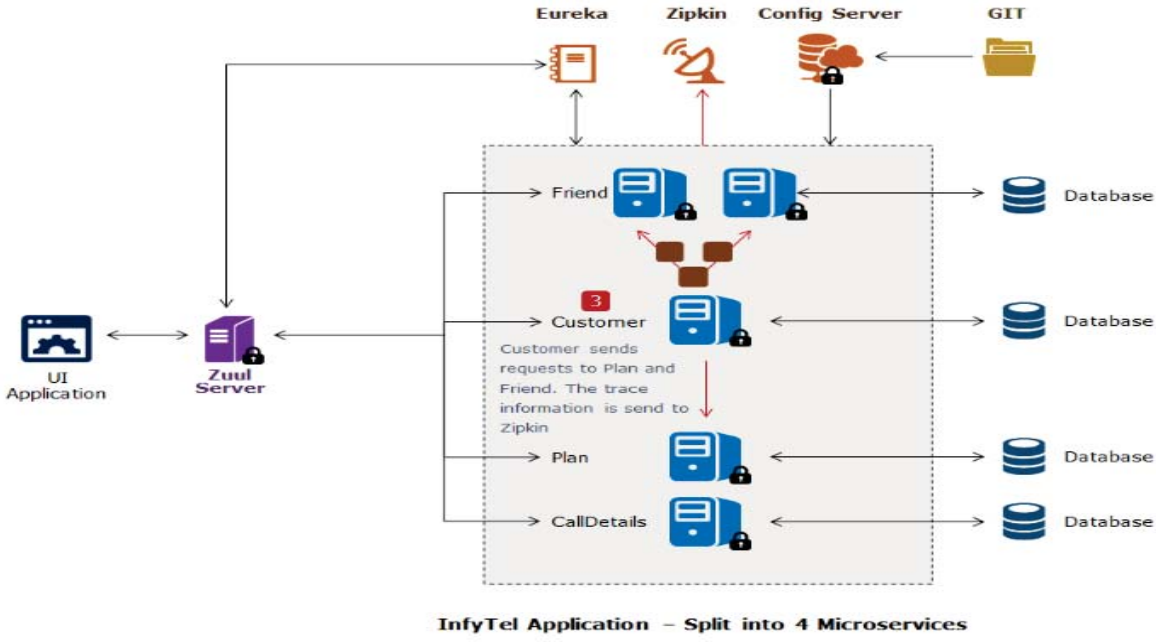
Flow of control - Step 1



Flow of Control - Step 2



Flow of Control - Step 3



Challenges

Creating, deploying and maintaining a microservice application is not easy. It has many technical and practical considerations. Some of them are:

- As the number of microservices increases, managing them and scaling them becomes difficult
- Failure points are more
- Since the attack area increases, security challenges also increase
- Since the number of microservices are high, keeping track of what is happening where becomes difficult
- We need to monitor the different services and find out which are up, which are down, which need scaling, etc
- Log data will be immense and we need to find out meaningful information from them
- A single faulty service can bring down all the other services
- Coordinating with external services is a challenge
- Managing databases becomes increasingly difficult
- Since each service may use different technology stack, deploying them also becomes a challenge
- Testing microservices is a challenge as one microservice may depend on data from other microservices

Patterns

Many challenges of microservices can be solved by using appropriate design patterns. We have already seen some of them in the course. The challenges can be addressed by the patterns such as:

- As the number of microservices increases, managing them and scaling them becomes difficult
  - Load Balancing pattern (Ribbon), Messaging Patterns
- Failure points are more
  - Circuit Breaker Pattern, Fallback pattern (Hystrix)
- Since the attack area increases, security challenges also increase
  - Spring-Cloud-Security, Spring-Cloud-OAuth2
- Since the number of microservices are high, keeping track of what is happening where becomes difficult
  - Service Discovery(Eureka Dashboard), Circuit Breaker (Hystrix, Turbine), Distributed Tracing Pattern (Zipkin)
- We need to monitor the different services and find out which are up, which are down, which need scaling, etc
  - Service Discovery(Eureka Dashboard), Circuit Breaker (Hystrix, Turbine), Distributed Tracing Pattern (Zipkin)
- Log data will be immense and we need to find out meaningful information from them
  - Distributed Tracing (Sleuth, ELK Stack)

Patterns

- A single faulty service can bring down all the other services
  - Circuit Breaker (Hystrix)
- Coordinating with external services is a challenge
  - API Gateway, Backend For Frontend
- Managing databases becomes increasingly difficult
  - Single DB, DB Per Service, API composition, Saga, CQRS
- Since each service may use different technology stack, deploying them also becomes a challenge
  - Multiple service instances per host, Service instance per host , Service instance per Container

Database Patterns

Some of the Database patterns are:

Single DB - There is only one DB which is shared by all the microservices

DB Per Service - Each microservice has its own DB. This throws challenges on what if there is data related across databases. To address this we have other patterns like:

API composition - This is the pattern we saw in the beginning of the course, where the microservice was aggregating the result from other microservices by adding additional REST endpoints in those microservices instead of performing a table join.

Saga - In this pattern a sequence of individual or local Database operations are carried out through event driven database management model. For example, if we delete a customer, the related friend and family should also be deleted. These two operations can be considered a saga. In this model, when a customer is deleted, an delete event is raised on the friend service. The friend service listens to such events and once it receives the delete event, it performs a local delete transaction on the friend table

CQRS - This is called the Command Query Responsibility Segregation. In this pattern, we maintain two copies or views of the database. One view is based on the common queries that get executed on the database. Thus this view can be a join of different tables. The other model of the same database is based on the update operations that get executed on the database. Thus the database is segregated based on the common commands and queries.

Deployment Pattern

Deploying a microservice app is different from deploying a monolithic app. In a monolithic app, you typically have only one executable and it is straightforward and simple to deploy it.

However, in a microservice architecture, there may be hundreds of microservices. We have to deploy them in such a way that they are isolated, can be monitored, maintained, redeployed, etc.

Deployment Pattern

Two ways in which they can be deployed are:

- **Multiple Service per host:**
  - In this pattern multiple service instances are deployed on the same host. The advantage is that it takes lesser resource, however, this also has the problem of monitoring. It is difficult to monitor which service is consuming more resource since multiple services are running in the same host
- **Single service per host:**
  - In this pattern, each instance of a service is individually deployed on a separate host. This allows us to monitor which service is consuming more resource. However, this is complex as the number of hosts are more.

Deployment Pattern

The microservices can also be packaged in the below ways:

- **container-less :**
  - The executable is a JAR or WAR but the dependent frameworks are separate. For example, it does not include Tomcat, Jboss, etc
- **self-contained:**
  - This contains the executable as well as the frameworks.
- **in-container:**
  - The executable, frameworks and the entire JVM is packaged in one go, using a container technology like Docker, Kubernetes, etc

The best way is to go for in-container as this ensures that the development, testing, deployment environments are same for every one and it is best from Devops perspective.

Testing Challenges

Testing a microservice involves:

1. Testing individual classes/functionalities inside a microservice - Unit Test
2. Testing whether it is able to receive message in the right format - Contract test
3. Testing whether it is able to respond back with right format - Contract test
4. Testing whether it is able to communicate with other microservices and pass messages to them - Integration test
5. Testing whether it is able to aggregate the results from other microservices - Component test

In a microservice architecture, each microservice is maintained by a different team. This microservice may make calls to other microservices and aggregate the data. In such a situation, testing a microservice becomes challenging. Because testing one microservice means we may have to run ten other related microservices and start three other related data bases. Though such end to end testing is more robust and reliable, it is not feasible and very brittle. If something fails, we can't know what is causing the failure.

We can use mocks and stubs, but even they have their drawbacks. We have open source projects like Spring Cloud Contract to address this. Spring Cloud Contract helps in not only unit test the service, but also perform integration testing.

Thank You