Monolithic Architecture

- Developing all functionalities in single application is called Monolithic Architecture.
  - products
  - cart
  - checkout
  - payment
  - orders
  - tracking
  - cancellation
  - reports
  - admin
- We will package our application as jar/war => fat jar or fat war

Advantages:

- Simple to develop.
- Everything is available at once place.
- Configurations required only once.

Dis-Advantages:

- Difficult to maintain.
- Single Point of failure.
- Re-Deploy entire application.

To overcome the problems of monolithic architecture, Microservices architecture came into picture,

Microservices:

- Microservices is not a programming language.
- Microservices is not a framework.
- Microservices is not an API.
- Microservices is an Architectural Design Pattern.
- Microservices architecture is used to develop application functionalities with loosely coupling.
- Instead of developing all functionalities in single project, we will divide functionalities and we will develop multiple APIs.
  Note: Every API is called as REST API / Backend API / Service/ Microservice.
- Microservices are independently deployable components.

Note: Microservices architecture is universal. Any programming language project also can be developed using this architecture.

Advantages:

1. Loosely coupling
2. Easy To Maintain
3. Faster Development
4. Quick Release
5. Technology independence
6. Less Downtime

Challenges with Microservices:

1. Bounded Context (deciding number of services we need to develop in the application)
2. Lot of configurations
3. Visibility
4. Infrastructure cost

Microservices Architecture:

- There is no standard / fixed architecture for microservices development.
- People are customizing microservices architecture based on their requirement.
- Below are the Microservices architecture components.
    1. Service Registry / Service Discovery
    2. Admin Server
    3. Zipkin Server
    4. Config Sever
    5. Backend APIs / Rest API / Services
    6. Feign Client
    7. API Gateway

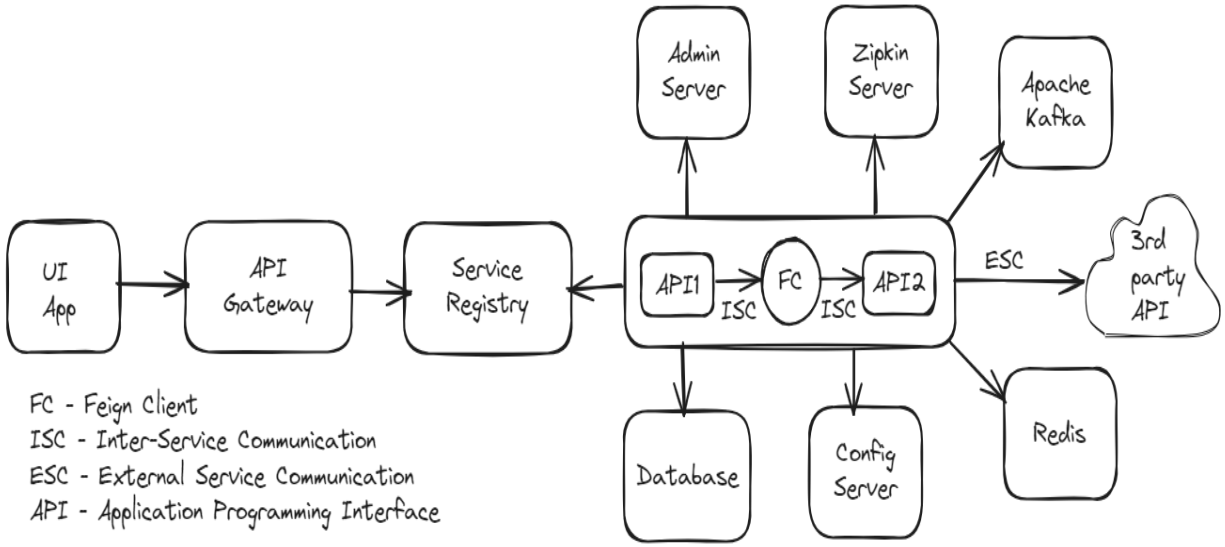

Fig. General Microservices Architecture

Service Registry:

- Service Registry is used to maintain list of services available in the project.
- It provides information about registered services like,
  Name of service, URL of service, status of service
- It provides number of instances available for each service.
- We can use Eureka Server as a service registry.
- Eureka server provided by Spring Cloud Netflix library.

Admin Server:

- Actuators are used to monitor and manage our applications.
- Monitoring and managing all the APIs separately is a challenging task.
- Admin Server Provides an user interface to monitor and manage all the APIs at one place using actuator endpoints.

Zipkin Server:

- It is Used for Distributed tracing.
- Using zipkin server, we can monitor which API is taking more time to process request.
- Using Zipkin we can understand how many APIs involved in request processing.

Backend APIs:

- Backend APIs contains business logic.
- Backend APIs are also called as REST APIs / services / microservices,
  Ex: Payment-API, Cart-API, Flights- API, Hotels- API

  Note: Backend API can register as client for Service Registry, Admin server & Zipkin server (It is optional)

FeignClient:

- It is provided by spring cloud libraries.
- It is used for Inter Service Communication.
- Inter service communication means one API is accessing another API using Service Registry.

Note: External communication means accessing third party APIs.

- When we are using FeignClient we no need mention URL of the API to access. Using service name feign client will get service URL from service registry.
- Feign Client uses Ribbon to perform Client-side load balancing.

API Gateway:

- API Gateway is used to manage our project backend APIs.

- API Gateway acts as mediator between user requests and backend APIs.
- API Gateway acts as entrypoint for all backend APIs.
- In API Gateway we will have 2 types of logics,
    1. Request Filter : To validate the request (go / no-go)
    2. Request Router : forward request to particular Backend-API based on URL Pattern

        /hotels => hotels – api
        /flights => flights - api
        /trains => trains - api

Config Server:

- Config Server is part of Spring Cloud Library.
- Config Server is used to externalize config properties of application.

Note: In real-time we will keep app config properties outside of the project to simply application maintenance.

Steps to develop Service Registry Application (Eureka Server):

1. Create Service Registry application with below dependency,

    ```
    spring-boot-starter-actuator
    spring-cloud-starter-netflix-eureka-server
    ```

2. Configure **`@EnableEurekaServer`** annotation in boot start main class.
3. Configure below properties in application.yml file,

    ```
    server:
      port: 8761

    eureka:
      client:
        register-with-eureka: false
    ```

Note:

- If Service-Registry project port is 8761 then clients can discover service-registry and will register automatically with service-registry.
- If service-registry project running on any other port number then we have to register clients with service-registry manually.

    ```
    eureka:
       client:
         serviceUrl:
           defaultZone: http://localhost:8761/
    ```

4. Once application started we can access Eureka Dashboard using below URL

    URL : http://localhost:8761/

Steps to develop Spring Admin-Server:

1. Create Boot application with admin-server dependency

```
spring-boot-admin-starter-server
```

2. Configure **`@EnableAdminServer`** annotation at start main class.
3. Change Port Number (Optional) in properties file.
4. Run the boot application,
5. Access application URL in browser (We can see Admin Server UI)

Steps to work with Zipkin Server:

1. Download Zipkin Jar file,

   URL : https://zipkin.io/pages/quickstart.html
2. Go to the downloaded folder and Run Zipkin jar file ,

   ```
   $ java -jar <jar-name>
   ```
3. By default Zipkin Server Runs on Port Number 9411.
4. Access Zipkin server dashboard

   URL : http://localhost:9411/

Steps to develop WELCOME-API:

1. Create Spring Boot application with below dependencies,

   ```
   - spring-boot-starter-actuator
   - spring-boot-starter-web
   - spring-boot-admin-starter-client
   - spring-cloud-starter-netflix-eureka-client
   - zipkin
   - spring-boot-devtools
   ```
2. Configure **`@EnableDiscoveryClient`** annotation at start main class,
3. Create RestController with required method.
4. Configure below properties in application.yml file,

```
server:
  port: 8081

spring:
  application:
    name: WELCOME-API
  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka

management:
  endpoints:
    web:
      exposure:
        include: '*'
```

5. Run the application and check in Eureka Dashboard (It should display in eureka dashboard)
6. Check Admin Server Dashboard (It should display) (we can access application details from here)

   Ex: Beans, loggers, heap dump, thread dump, metrics, mappings etc...
7. Send Request to REST API method.
8. Check Zipkin Server UI and click on Run Query button (it will display trace-id with details).

Steps to develop GREET-API:

1. Create Spring Boot application with below dependencies,
   - spring-boot-starter-actuator
   - spring-boot-starter-web
   - spring-boot-admin-starter-client
   - spring-cloud-starter-netflix-eureka-client
   - zipkin
   - spring-boot-devtools
   - open-feign
2. Configure **`@EnableDiscoveryClient`** & **`@EnableFeignClients`** annotation at start main class.
3. Create Feign Client to access WELCOME-API

```
@FeignClient(name = "WELCOME-API")
public interface WelcomeApiClient {

	@GetMapping("/welcome")
	public String invokeWelcomeApi();
}
```

4. Create RestController with required methods and inject feign-client to access welcome-api

```
package in.nk.tech.sbm.api.restcontroller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import in.nk.tech.sbm.api.cleint.WelcomeApiClient;

@RestController
public class GreetRestController {

	@Autowired
	private WelcomeApiClient apiClient;

	@GetMapping("/greet")
	public String getGreetMsg() {
		String msg = "Good Morning...";
		String welcomeResponse =
apiClient.invokeWelcomeAPi();
```

```
            return welcomeResponse +" , "+msg;
        }
    }
```

5. Configure below properties in application.yml file,

```
server:
  port: 8081

spring:
  application:
    name: GREET-API

  boot:
    admin:
      client:
        url: http://localhost:1111/


management:
  endpoints:
    web:
      exposure:
        include: '*'
```

6. Run the application and check in Eureka Dashboard (It should display in eureka dashboard)
7. Check Admin Server Dashboard (It should display) (we can access application details from here)

   Ex: Beans, loggers, heap dump, thread dump, metrics, mappings etc...
8. Send Request to REST API method.
9. Check Zipkin Server UI and click on Run Query button (it will display trace-id with details).

Working with Spring Cloud API Gateway:

1. Create Spring boot application with below dependencies,

```
- web-stater
- eureka-client
- cloud-gateway
- devtools
```

2. Configure **`@EnableDiscoveryClient`** annotation at boot start main class.
3. Configure API Gateway Routings in application.yml file like below

```
spring:
  cloud:
    gateway:
      discovery.locator:
        enabled: true
        lowerCaseServiceId: true
      routes:
      - id: welcome-api
        uri: lb://WELCOME-API
```

```
          predicates:
          - Path=/welcome
        - id: greet-api
          uri: lb://GREET-API
          predicates:
          - Path=/greet
    application:
      name: CLOUD-API-GATEWAY
server:
  port: 3333
```

In API gateway we will have 3 types of logics:

1. Route
2. Predicate
3. Filters

- **Routing:** is used to defined which request should be processed by which REST API in backend. Routes will be configured using Predicate.
- **Predicate:** This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filters:** are used to manipulate incoming request and outgoing response of our application

Note:

- Using Filters we can implement security also for our application.
- We can validate client given token in the request using Filter for security purpose.
- We can write request and response tracking logic in Filter.
- Filters are used to manipulate request & response of our application.
- Any cross-cutting logics like security, logging, monitoring can be implemented using Filters.

What is Cloud Config Server:

- We are configuring our application config properties in application.properties or application.yml file.

  Ex: DB Props, SMTP props, Kafka Props, App Messages etc...
- application.properties or application.yml file will be packaged along with our application (it will be part of our app jar file).
- If we want to make any changes to properties then we have to re-package our application and we have to re-deploy our application.

Note: If any changes required in config properties, then We have to repeat the complete project build & deployment which is time consuming process.

- To avoid this problem, we have to separate our project code and project config properties files.
- To externalize config properties the application we can use Cloud Config Server.
- Cloud Config Server is part of Spring Cloud Library.

Note: Application config properties files we will maintain in git hub repo and config server will load them and will give to our application based on our application-name.

- Our microservices will get config properties from Config server and config server will load them from git hub repo.

Developing Config Server App:

1. Create Git Repository and keep ymls files required for projects

Note: We should keep file name as application name
app name : greet then file name : greet.yml
app name : welcome then file name : welcome.yml

Git Repo : `https://github.com/Honnur6268/Configuration_Properties.git`

2. Create Spring Starter application with below dependency,
   - `spring-cloud-config-server`
3. Write **`@EnableConfigServer`** annotation at boot start main class.

```
@SpringBootApplication
@EnableConfigServer
public class Application {

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

4. Configure below properties in application.yml file

```
spring:
  cloud:
    config:
      server:
        git:
```

```
          uri:
https://github.com/Honnur6268/Configuration_Properties.git
          clone-on-start: true
management:
  security:
    enabled: false
```

5. Run Config Server application.

Config Server Client Development:

1. Create Spring Boot application with below dependencies.
   - web-starter
   - spring-cloud-starter-config
   - dev-tools
2. Create Rest Controller with Required methods

```
@RestController
@RefreshScope
public class WelcomeRestController {

      @Value("${msg}")
      private String msg;

      @GetMapping("/")
      public String getWelcomeMsg() {
            return msg;
      }
}
```

3. Configure ConfigServer url in application.yml file like below

```
server:
  port: 9090
spring:
  config:
    import: optional:configserver:http://localhost:8080
  application:
    name: greet
```

4. Run the application and test it.
5. Change app-name to 'welcome' and test it .
   Note: yml file must be same name as app name

Circuit Breaker Design Pattern:

- Circuit Breaker => It is an electric concept.
- It is used to protect us from high voltage or low voltage power.
- It is used to divert traffic when some problem detected in normal execution flow.
- We can use Circuit Break concept in our microservices to implement fault tolerance systems / Resilience systems.

Note: When main logic is failing continuously then we have to execute fallback logic for some time.

Circuit Breaker Implementation:

1. Create Spring Boot project with below dependencies,

```
- web-starter
- actuator
- aop
- spring-cloud-starter-config

<dependency>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-spring-boot3</artifactId>
      <version>2.0.2</version>
</dependency>
```

2. Create Model class to represent Bored API response,

```
@Data
public class Activity {
      private String activity;
      private String type;
      private String link;
      private String key;
      private Integer participants;
      private Double price;
      private Double accessibility;
}
```

3. Create Rest Controller to consume Bored API,

```
@RestController
public class ActivityRestController {

      private final String BORED_API =
"https://www.boredapi.com/api/activity";

      @GetMapping
      @CircuitBreaker(name = "randomActivity", fallbackMethod =
"fallbackRandomActivity")
      public String getRandomActivity() {

            RestTemplate rt = new RestTemplate();

            ResponseEntity<Activity> responseEntity =
rt.getForEntity(BORED_API, Activity.class);
            Activity activity = responseEntity.getBody();
            System.out.println("Activity Recieved::" +
activity.getActivity());

            int i = 10/0;
            return activity.getActivity();
      }
```

```
        public String fallbackRandomActivity(Throwable throwable)
{
              return "Service Down. Please Try Again Later!!!";
        }
}
```

4. Configure Circuit Breaker Properties,

```
spring:
  application.name: resilience4j-demo

management:
  endpoints.web.exposure.include:
    - '*'
  endpoint.health.show-details: always
  health.circuitbreakers.enabled: true

resilience4j.circuitbreaker:
  configs:
    default:
      registerHealthIndicator: true
      slidingWindowSize: 10
      minimumNumberOfCalls: 5
      permittedNumberOfCallsInHalfOpenState: 3
      automaticTransitionFromOpenToHalfOpenEnabled: true
      waitDurationInOpenState: 5s
      failureRateThreshold: 50
      eventConsumerBufferSize: 10
```

5. Test The application and monitor actuator health endpoint.

Example: Develop Backend API's Using Spring Boot and Microservices as shown in below diagram. (Include Service Registry, Admin Server, Zipkin Server, Config Server. API Gateway, Feign Clients, and Circuit Breaker). No need of Frontend UI.

Title: Company Stock Price Calculator

API-1: Stock-Price-API (Stores Current Stock Price info about companies)

API-2: Stock-Calci-API (Fetches Current Stock Price info based on Company name and calculates Total Stock Price based on number of quantities provided and returns total stock price)