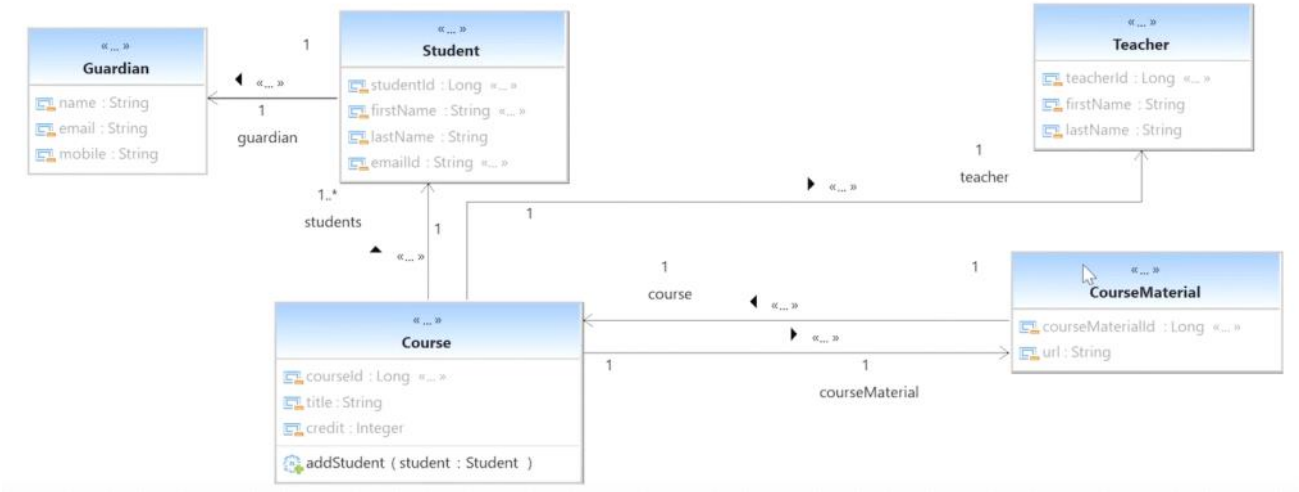
# Spring-boot Misc.

Tuesday, December 17, 2024 3:19 PM

## Spring-data-JPA

While working with Java, we often need to connect to a database, and for that, Java provides the JDBC template. The challenge with Java-DB connectivity is that Java works with objects and classes, while databases operate on tables (comprising rows and columns). In the traditional approach using JDBC, we fetch data from the database and manually map it to Java objects. However, there should be a framework to automate this process. Such frameworks are called ORM (Object-Relational Mapping). ORM essentially means that for every table in the database, there is a corresponding Java object to represent that table. Some popular ORM frameworks include Hibernate and iBatis.
However, each ORM framework has its own way of connecting to the database and its own methods for performing database operations. As a result, migrating from one ORM framework to another can be quite challenging.
To streamline this process, Java introduced the concept of JPA (Java Persistence API). JPA is a framework provided by Java that acts as a template for implementing database operations. ORM frameworks can use this template to implement their own methods.

Project architecture -




Use below application.properties to connect to db -

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/schooldb
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name =com.mysql.jdbc.Driver
spring.jpa.show-sql: true
```

We are using hibernate implementation here.
**spring.jpa.show-sql: true** is used to print sql log statements.
**spring.jpa.hibernate.ddl-auto=update** is used to auto-update changes done in spring-boot application to db. (never use this property in prod.)

Student and guardian entity classes ->
Student.java

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Table(
        name = "tbl_student",
        uniqueConstraints = @UniqueConstraint(
                name = "emailid_unique",
                columnNames = "email_address"
        )
)
public class Student {

    @Id
```

Guardian.java

```
@Embeddable
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
@AttributeOverrides({
        @AttributeOverride(
                name = "name",
                column = @Column(name = "guardian_name")
        ),
        @AttributeOverride(
                name = "email",
                column = @Column(name = "guardian_email")
        ),
        @AttributeOverride(
                name = "mobile",
                column = @Column(name = "guardian_mobile")
        )
})
public class Guardian {
```

```java
public class Student {

    @Id
    @SequenceGenerator(
            name = "student_sequence",
            sequenceName = "student_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "student_sequence"
    )
    private Long studentId;
    private String firstName;
    private String lastName;

    @Column(
            name = "email_address",
            nullable = false
    )
    private String emailId;

    @Embedded
    private Guardian guardian;
```

```java
                column = @Column(name = "guardian_mobile")
        )
})
public class Guardian {

    private String name;
    private String email;
    private String mobile;
}
```

Annotations used -
**@Entity** to define the class an Entity class.
**@Data** to generate getters and setters using lombok.
**@Builder** to use builder factory pattern while creating objects.
**@Table** to indicate which table this entity class is representing.
**@UniqueConstraint** to list unique constrained fields from table.
**@Id** to denote as primary key of the table
**@SequenceGenerator** to denote the sequence generator used to generate primary key values.
**@Column** indicates column name of the particular field in table.
As Guardian details are present inside student table only, we cannot create a separate entity class for Guardian details. But guardian details should be maintained in a different class and hence we used **@Embeddable** on Guardian class and embedded the class in Student entity using **@Embeddable**.

**@AttributeOverrides** is used to map the java fields to db table fields. We cannot use

@Builder is used to build the object in this way ->
The only purpose is to increase readability.

```java
@Test
public void saveStudent() {
    Student student = Student.builder()
            .emailId("shabbir@gmail.com")
            .firstName("Shabbir")
            .lastName("Dawoodi")
            .guardianName("Nikhil")
            .guardianEmail("nikhil@gmail.com")
            .guardianMobile("9999999999")
            .build();
}
```

Repository :

StudentRepository.java -

```java
@Repository
public interface StudentRepository extends JpaRepository<Student,Long> {

    List<Student> findByFirstName(String firstName);

    List<Student> findByFirstNameContaining(String name);

    List<Student> findByLastNameNotNull();

    List<Student> findByGuardianName(String guardianName);

    Student findByFirstNameAndLastName(String firstName,
                                       String lastName);
```

@Repository to indicate this class is a repo class.
Repository extends **JpaRepository<Student,Long>**
Long is nothind but primary id data type and Student is the entity for which we will be performing db operations using this repo class.
**JpaRepository** has some inbuilt methods such as findAll(), save(), saveAndFlush() etc.
If we want to create our custom method then format is ->
**findBy<field_name>**
**Eg. findByFirstName (field name must be same as mentioned in entity class)**

```java
//JPQL
@Query("select s from Student s where s.emailId = ?1")
Student getStudentByEmailAddress(String emailId);


//JPQL
@Query("select s.firstName from Student s where s.emailId = ?1")
String getStudentFirstNameByEmailAddress(String emailId);
```

JPQL -
If we want to get a specific data as per our requirement then we can use @Query and define a query directly.
**Note: JPQL query works as per fields names in java objects and not on db tables.**
Here **?1** is nothing but the first parameter passed to a method.

If we want to use traditional sql queries then we can use @Query with natiQuery = true.

```java
@Query("select s.firstName from Student s where s.emailId = ?1")
String getStudentFirstNameByEmailAddress(String emailId);
```

If we want to use traditional sql queries then we can use @Query with natiQuery = true.
**Note: Native query works as per fields names in db tables and not on java objects.**

```java
//Native
@Query(
        value = "SELECT * FROM tbl_student s where s.email_address = ?1",
        nativeQuery = true
)
Student getStudentByEmailAddressNative(String emailId);
```

```java
//Native Named Param
@Query(
        value = "SELECT * FROM tbl_student s where s.email_address = :emailId",
        nativeQuery = true
)
Student getStudentByEmailAddressNativeNamedParam(
        @Param("emailId") String emailId
);
```

Instead of mentioning params as ?1 ?2 we can use **named params**.
We need to use **@Param** on the method parameter.

```java
@Modifying
@Transactional
@Query(
        value = "update tbl_student set first_name = ?1 where email_address = ?2",
        nativeQuery = true
)
int updateStudentNameByEmailId(String firstName, String emailId);
```

While updating the data we need to use @Modifying annotation.
We can use @Transactional as well to roll back the commit in case of failure.

JPA relationships -
Course and CourseMaterial has one-to-one mapping.

```java
public class Course {

    @Id
    @SequenceGenerator(
            name = "course_sequence",
            sequenceName = "course_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "course_sequence"
    )
    private Long courseId;
    private String title;
    private Integer credit;
```

```java
public class CourseMaterial {

    @Id
    @SequenceGenerator(
            name = "course_material_sequence",
            sequenceName = "course_material_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "course_material_sequence"
    )
    private Long courseMaterialId;
    private String url;

    @OneToOne(
            cascade = CascadeType.ALL,
            fetch = FetchType.LAZY,
            optional = false
    )
    @JoinColumn(
            name = "course_id",
            referencedColumnName = "courseId"
    )
    private Course course;
}
```

CourseId is foreign key in CourseMaterial table.
@OneToOne - It denotes on-to-one mapping with Course.
@JoinColum - It denotes foreign key column reference from Course.

Cascading ->
In this case if we try to create a CourseMaterial with some Course which is not yet committed in db, we will get error. Cascading is used in such scenarios.

```java
@Test
public void SaveCourseMaterial() {
    Course course =
            Course.builder()
            .title("DSA")
            .credit(6)
            .build();
```

If we try to perform the Save method, we will get error as Course is not yet saved to db.
When we add **cascade = CascadeType.ALL** then first Course will be saved and then CourseMaterial will be created.
Fetch types - **fetch = FetchType.LAZY or fetch = FetchType.EAGER**
Eager means find operation will return course table data as well. Lazy means until specifically mentioned, it will only fetch CourseMaterial data.

```
        .title("DSA")
        .credit(6)
        .build();

CourseMaterial courseMaterial =
        CourseMaterial.builder()
        .url("www.google.com")
        .course(course)
        .build();
```

```
30
31    @OneToOne(
32            mappedBy = "course"
33    )
34    private CourseMaterial courseMaterial;
35 }
```

CourseMaterial will be created.
Fetch types - **fetch = FetchType.LAZY or fetch = FetchType.EAGER**
Eager means find operation will return course table data as well. Lazy means until specifically mentioned, it will only fetch CourseMaterial data.

Till now, we have not defined any reference to CourseMaterial in Course entity. Hence, while fetching courses we will not get any details related to CourseMaterial. This is called **uni-directional one-to-one mapping.**

Adding @OneToOne in Course entity will map CourseMaterial to Course.
**mappedBy = "course" defines the course variable in CourseMaterial entity. So it means Course is mapped by "course" variable in CourseMaterial.**

Now by creating TeacherRepo we can save teacher and courses under the teacher.
But in this case courses will get saved without any Course material which is not correct.
Currently the relation between Course and CourseMaterial is optional and we need to make it
Mandatory by adding **optional = false**

One-to-many relation :
Teacher to Course relation - One teacher can teach many

```
28      private Long teacherId;
29      private String firstName;
30      private String lastName;
31
32      @OneToMany(
33              cascade = CascadeType.ALL
34      )
35      @JoinColumn(
36              name = "teacher_id",
37              referencedColumnName = "teacherId"
38      )
39      private List<Course> courses;
40  }
```

Course.java | CourseMaterial.java | SpringDataJpaTutorialApplication.java

```
24      )
25      private Long courseMaterialId;
26      private String url;
27
28      @OneToOne(
29              cascade = CascadeType.ALL,
30              fetch = FetchType.LAZY,
31              optional = false
32      )
33      @JoinColumn(
34              name = "course_id",
35              referencedColumnName = "courseId"
36      )
37      private Course course;
38  }
```

Instead of mentioning @OneToMany relation in Teacher entity it is better to mention @ManyToOne relation in Course Entity. This will not change any functionality but it will increase readability -

```
38      @ManyToOne(
39              cascade = CascadeType.ALL
40      )
41      @JoinColumn(
42              name = "teacher_id",
43              referencedColumnName = "teacherId"
44      )
45      private Teacher teacher;
```

```
27      @Test
28      public void saveCourseWithTeacher() {
29          Teacher teacher = Teacher.builder()
30                  .firstName("Priyanka")
31                  .lastName("Singh")
32                  .build();
33
34          Course course = Course
35                  .builder()
36                  .title("Python")
37                  .credit(6)
38                  .teacher(teacher)
39                  .build();
40
41          courseRepository.save(course);
42      }
```

Paging and Sorting -

```
44      @Test
45      public void findAllPagination(){
46          Pageable firstPagewithThreeRecords =
47                  PageRequest.of( page: 0,  size: 3);
48          Pageable secondPageWithTwoRecords =
49                  PageRequest.of( page: 1, size: 2);
50
51          List<Course> courses =
52                  courseRepository.findAll(firstPagewithThreeRecords)
53                          .getContent();
54
55          long totalElements =
56                  courseRepository.findAll(firstPagewithThreeRecords)
57                  .getTotalElements();
```

Create page object by passing required page size and page number.
Let's say there are total 9 records and we passed page size = 3 and page=1 then 4th,5th,6th record will be returned.

Sorting -

```
71      @Test
72      public void findAllSorting() {
73          Pageable sortByTitle =
74                  PageRequest.of(
75                          page: 0,
76                          size: 2,
77                          Sort.by("title")
78                          );
79          Pageable sortByCreditDesc =
80                  PageRequest.of(
81                          page: 0,
82                          size: 2,
83                          Sort.by("credit").descending()
84                  );
85      }
```

Many to many relationship -
Student and course will have many to many relationship because many students can opt many courses.
We have already created Student and Course entity classes. But to maintain many to many relationship we need to create one more table let's say student_course_map.

```
47      @ManyToMany(
48              cascade = CascadeType.ALL
49      )
50      @JoinTable(
51              name = "studen_course_map",
52              joinColumns = @JoinColumn(
53                      name = "course_id",
54                      referencedColumnName = "courseId"
```

Now, to test this out, we need to create a course then create and add teacher to the course.

```
public void saveCourseWithStudentAndTeacher() {

    Teacher teacher = Teacher.builder()
            .firstName("Lizze")
            .lastName("Morgan")
```

```
52              joinColumns = @JoinColumn(
53                      name = "course_id",
54                      referencedColumnName = "courseId"
55              ),
56              inverseJoinColumns = @JoinColumn(
57                      name = "student_id",
58                      referencedColumnName = "studentId"
59              )
60      )
61      private List<Student> students;
62
63      public void addStudents(Student student){
64          if(students == null) students = new ArrayList<>();
65          students.add(student);
66      }
```

```
Teacher teacher = Teacher.builder()
        .firstName("Lizze")
        .lastName("Morgan")
        .build();

Course course = Course
        .builder()
        .title("AI")
        .credit(12)
        .teacher(teacher)
        .build();
```

Then create a student and add it to a course using addStudents() method.
After saving the course below things happen in db -

1) New course gets created with course_id let's say 5.
2) New teacher gets created with teacher_id let's say 3.
3) New student gets created with student_id let's say 10.
4) For course_id 5 there will be teacher_id 3 assigned in course table.
5) In student_course_map table course_id 5 gets mapped with student_id 10.

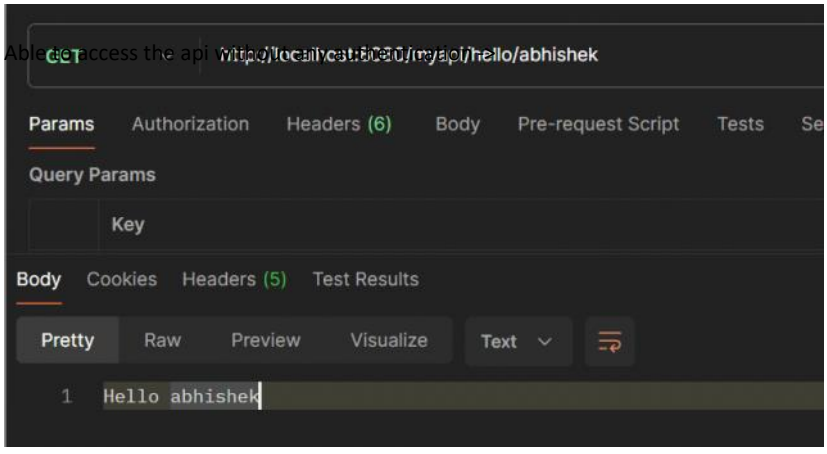
# Spring-security

It takes care of security of application/api. Whenever we implement spring-security, the requests coming to our api need to pass through security layer. Security will check for authentication (who are you?) and authorization (what do you want?).
Session : let's take an example of FB. When we hit a FB website then it will prompt for username and pwd. When we provide username and pwd, then FB will save a sessionId to a browser. Until and unless the sessionId is there, session is active. If we open a different browser then different session is generated.

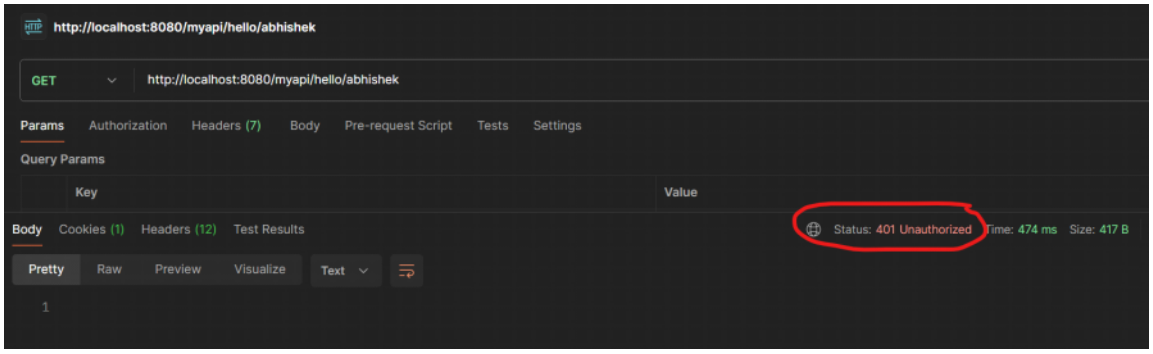
Created a simple rest endpoint without any security ->

```
10  @RestController  no usages
11  @RequestMapping("/myapi")
12  public class HelloController {
13
14      @GetMapping("/hello/{name}")  no usages
15      public String sayHello(@PathVariable String name) { return "Hello " + name; }
18
19  }
```

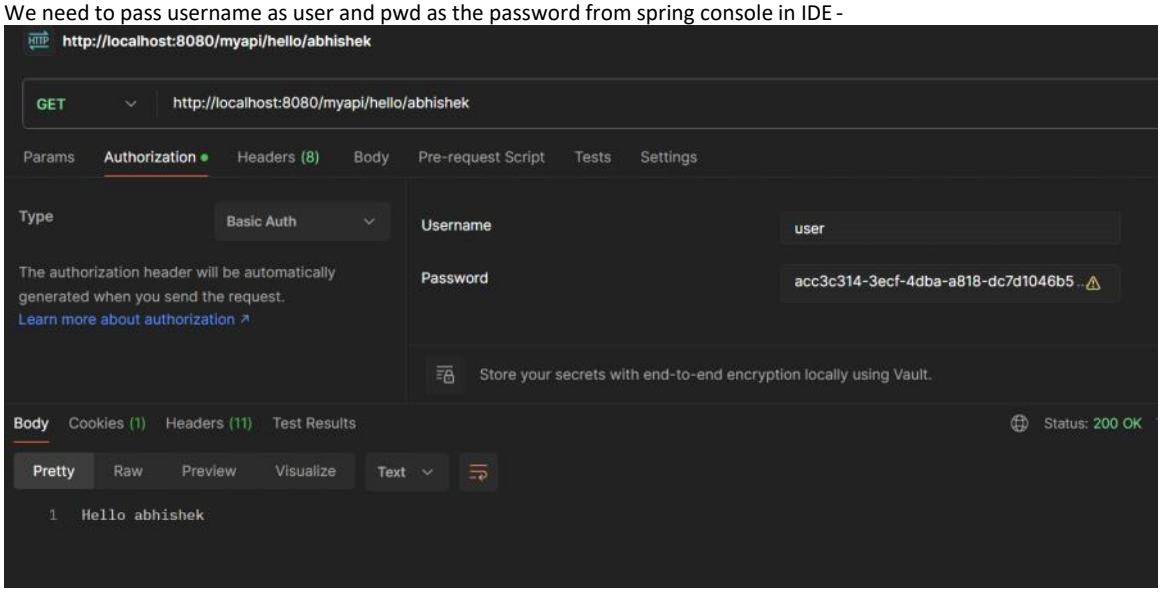
Able to access the api without any authentication



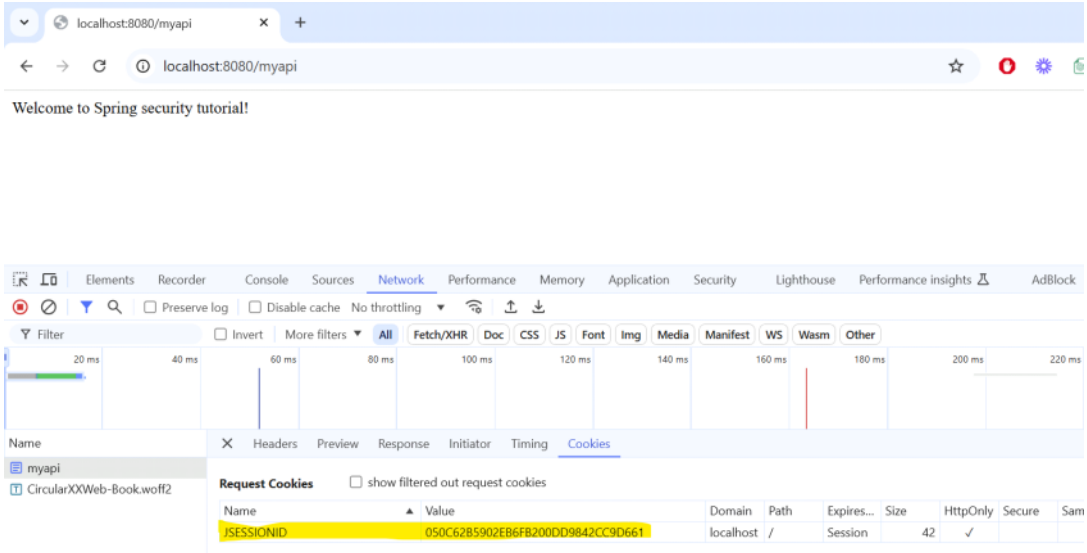

Now if we just add spring-security dependency to a pom.xml ->
And if we try to hit the same api, we will get 401 (unauthorized)




We need to pass username as user and pwd as the password from spring console in IDE -

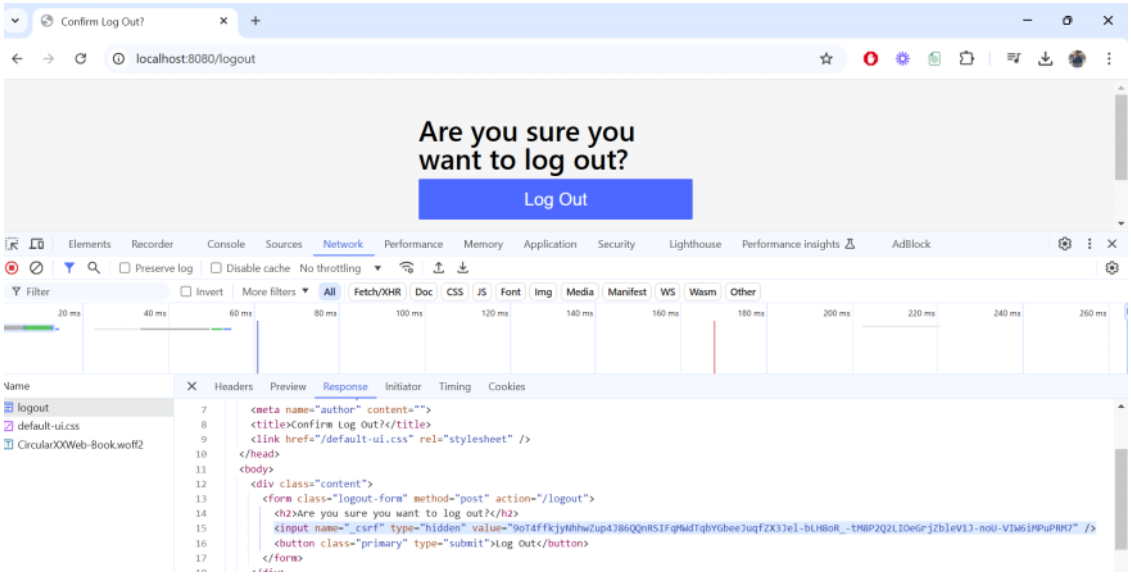



Session ID -



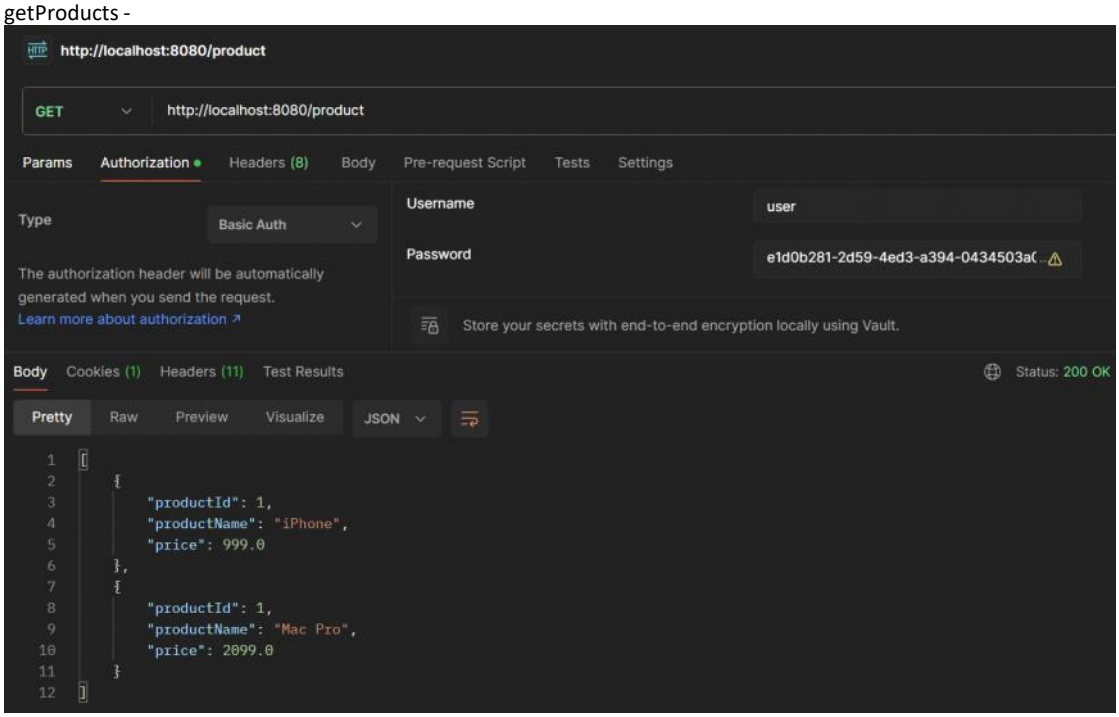

Now, if we hit the same endpoint multiple times, it will not ask to authenticate again because it is using the session id.

CSRF(Cross-Site Request Forgery) token -
If hacker gets session id then hacker can hit our apis directly. To avoid this, there is a CSRF token created along with session ID. And whenever we are trying to hi api endpoint CSRF token also gets shared. So csrf token ensures that the request is coming from same source.



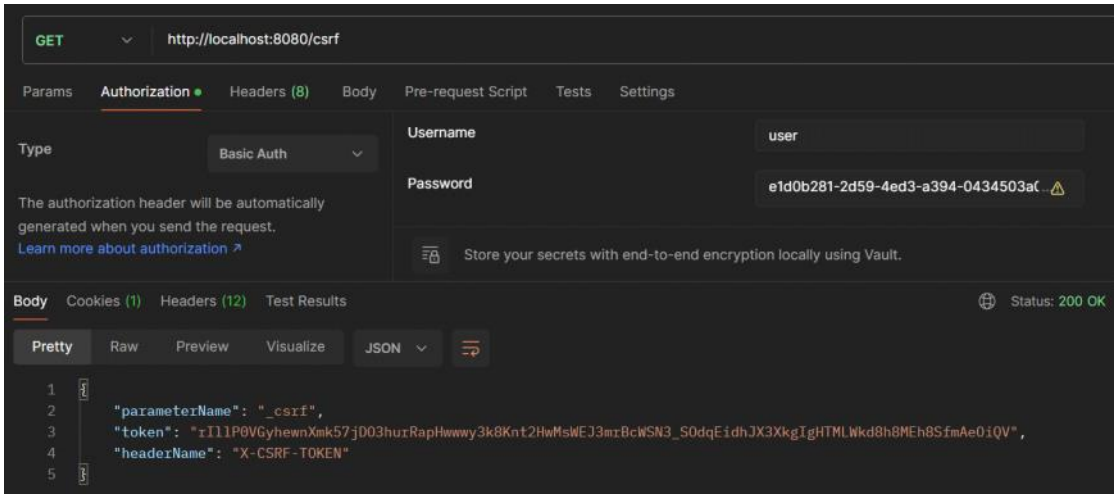

Let's create 3 api endpoints ->
Getmapping - to get list of products
Postmapping - to add a product.
Csrf - to get csrf token

getProducts -



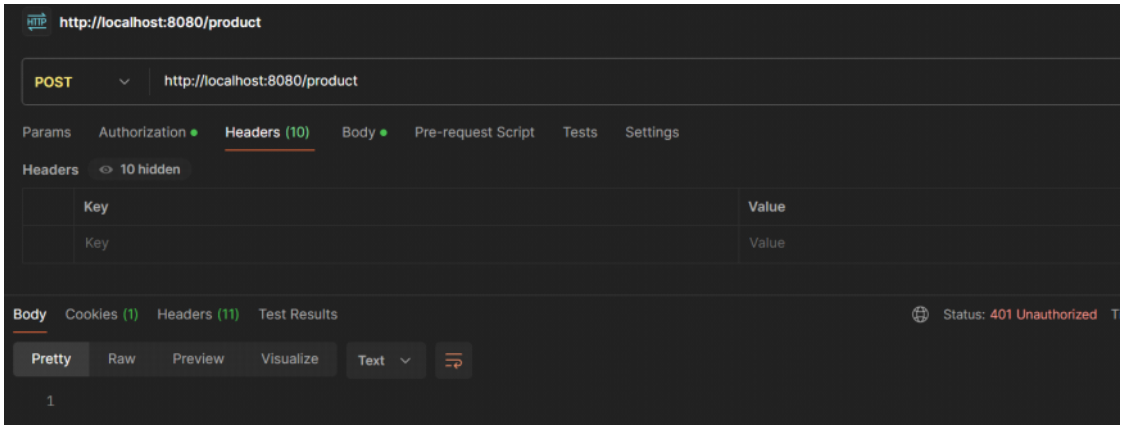

Now if we try to add product using post method then we will get 401. To hit the post request we need to add csrf token.

Get csrf token -

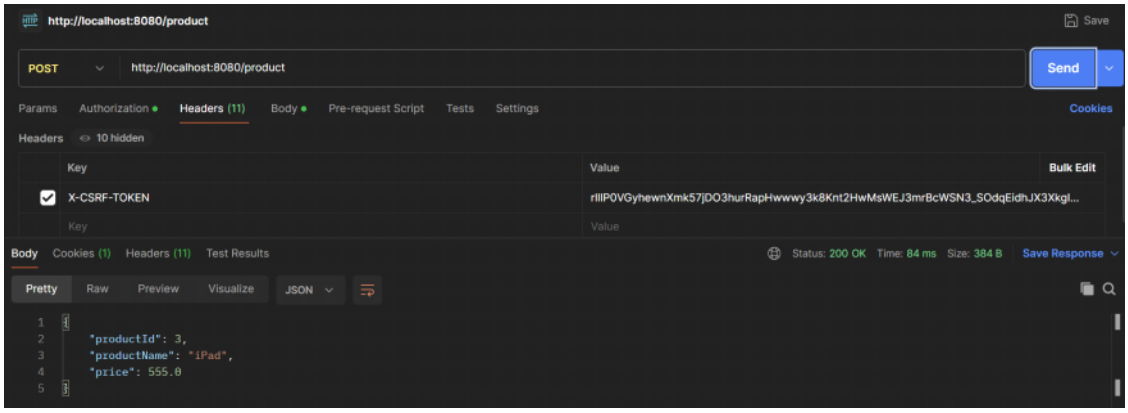



```
@GetMapping("/csrf")  no usages
public CsrfToken getToken(HttpServletRequest request) {
    return (CsrfToken) request.getAttribute( s: "_csrf");
}
```

Now, use the token as header in post request -
Without csrf token -



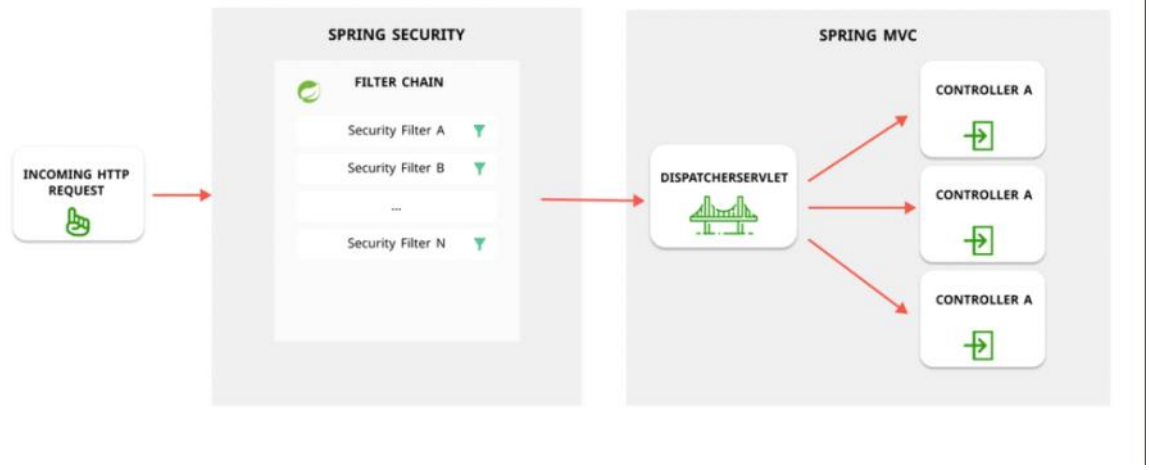

With csrf token -



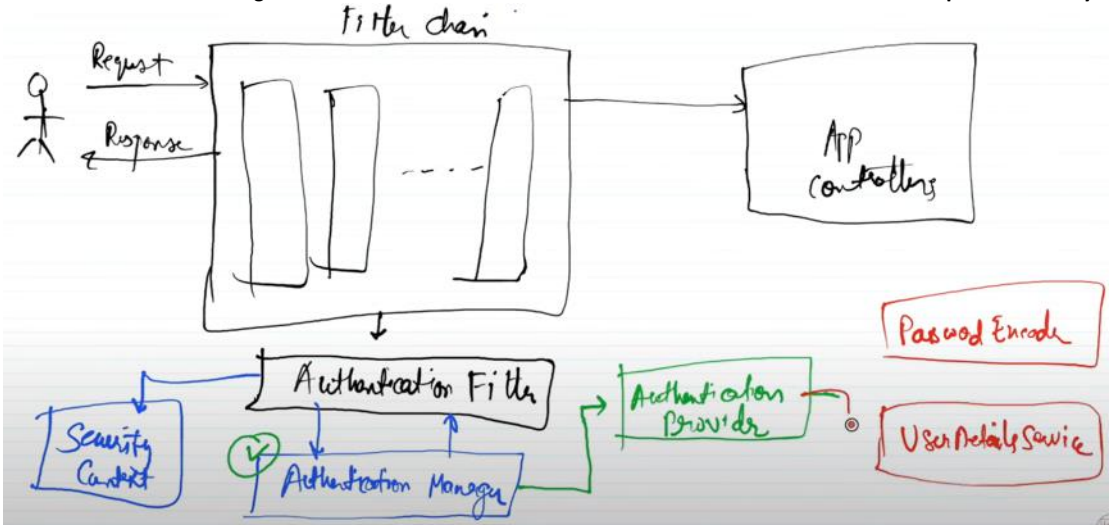

So whenever we are changing the request body either in put, post, delete method, we need to pass CSRF token.

We can add custom credentials as well by mentioning below properties in application.properties ->

Spring security components -
The high level diagram ->




Let's see what exactly happens in Spring security -
Request comes to **filter** -> filter passes it to **Authentication Filter** -> Authentication filter has **Security context** -> Authentication filter communicates with **Authentication manager** -> Authentication manager passes username and password to **Authentication provider** -> Now Authentication provider has 2 components 1) **Password Encoder** -> To know how the password is encoded. 2) **UserDetailsService** gets data from db for the using username and it checks if username and password from db matches with the provided one. If matches then the user is authenticated.
Then authentication manager tells authentication filter if user is valid or not. **Authentication filter updates security context whether user is valid or not**.




Custom configurations -
Custom securityFilterChain ->

```
14    Bean
15 @  ublic SecurityFilterChain securityFilterChain(HttpSecurity httpSecurit
16        httpSecurity
17                .csrf(csrf → csrf.disable())
18                .authorizeHttpRequests(
19                        request → request.anyRequest().authenticated()
20                )
21                .formLogin(Customizer.withDefaults())
22                .httpBasic(Customizer.withDefaults());
23        return httpSecurity.build();
24
```

Custom userDetailsService (In memory) -

```
32        @Bean
33        public UserDetailsService userDetailsService() {
34            UserDetails shabbir
35                    = User.withUsername("shabbir")
36                    .password("{noop}password")
37                    .roles("USER")
38                    .build();
39
```

We can check user details from db as well. For that we need to create a api to register the user first. As the register api will register the user at first place, we should bypass the authentication filter for this particular api. We can do that like ->

```
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity
    httpSecurity
            .csrf(csrf → csrf.disable())
            .authorizeHttpRequests(
                    request → request
                            .requestMatchers( ...patterns: "register").permitAll()
                            .anyRequest().authenticated()
            )
            .formLogin(Customizer.withDefaults())
            .httpBasic(Customizer.withDefaults());
    return httpSecurity.build();
}
```

After registering the user we can authenticate the user from db. We need to implement CustomUserDetails service for the same.

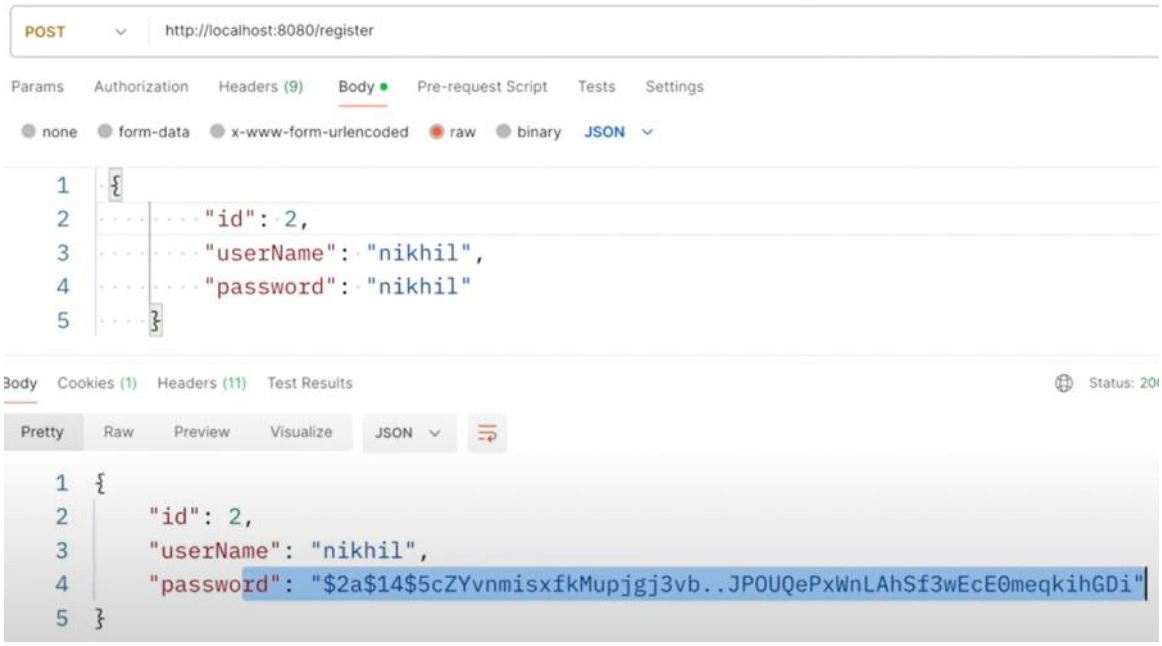
```java
@Component
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByUserName(username);
        if(Objects.isNull(user)) {
            System.out.println("User not available");
            throw new UsernameNotFoundException("User not found");
        }
        return new CustomUserDetails(user);
    }
}
```

```java
@Bean
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider provider
            = new DaoAuthenticationProvider();
    provider.setUserDetailsService(userDetailsService);
    provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
    return provider;
}
```

In above example we have used NoOpPasswordEncoder but we can use password encoder as well and for that we need to set the encoded password while registering the user. Modified register method ->

```java
1 usage   Pieces: Explain
public User register(User user) {
    user.setPassword(bCryptPasswordEncoder
            .encode(user.getPassword()));
    return userRepository.save(user);
}
}
```

So after registering the user we can see encoded password in response ->




Json Web Token (JWT) -
SSO (single sign on) - SSO is used to login with same data to multiple applications. For example, we can use same corporate account creds to access all Microsoft and other company apps.
JWT - When user login with username and password, then JWT is returned to the user. It is user's responsibility to store JWT and this token can be used to access multiple features.
Method to generate JWT token -

```
public String generateToken(User user) {
    Map<String, Object> claims
            = new HashMap<>();
    return Jwts
            .builder()
            .claims()
            .add(claims)
            .subject(user.getUserName())
            .issuer("DCB")
            .issuedAt(new Date(System.currentTimeMillis()))
            .expiration(new Date(System.currentTimeMillis()+ 60
            .and()
            .signWith(generateKey())
            .compact();
}
```

To authenticate JWT token, we need to add a security filter before username and pwd authentication filter. If JWT token is valid then username and pwd filter will be bypassed.

```
            .httpBasic(Customizer.withDefaults())
            .addFilterBefore(jwtAuthenticationFilter,
                    UsernamePasswordAuthenticationFilter.class)
    return httpSecurity.build();
}
```

In the actual logic we need to extract username and expiration time from JWT token. We need to compare username with actual username in db and expiration time should not exceed current time. If that is the case then, user will be authenticated and will bypass next filters.

Logout functionality - In case of logout just delete the JWT token from browser local storage. So when new request comes, new JWT token will be created.

In case of microservices, API gateway should take care of verifying JWT.

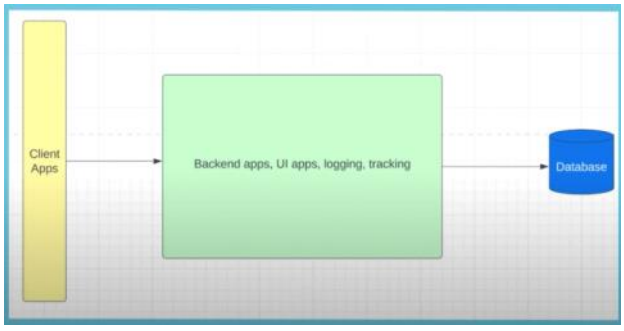
**Note: Once created, JWT should not be changed.**

# Microservices-

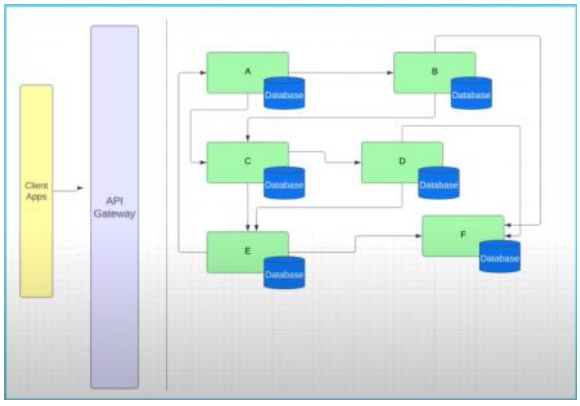
## Key Concepts of Microservices Architecture:

1. **Definition of Microservices**:
   - A microservice is a component or piece of code that performs one specific task well and communicates with other components vi a protocols.
2. **Five Key Concepts** that define microservices:
   - **High Cohesion and Low Coupling**
   - **Independent Deployability**
   - **Business Domain Modeling**
   - **Observability**
   - **Team Organization**

**Typical monolith application -**

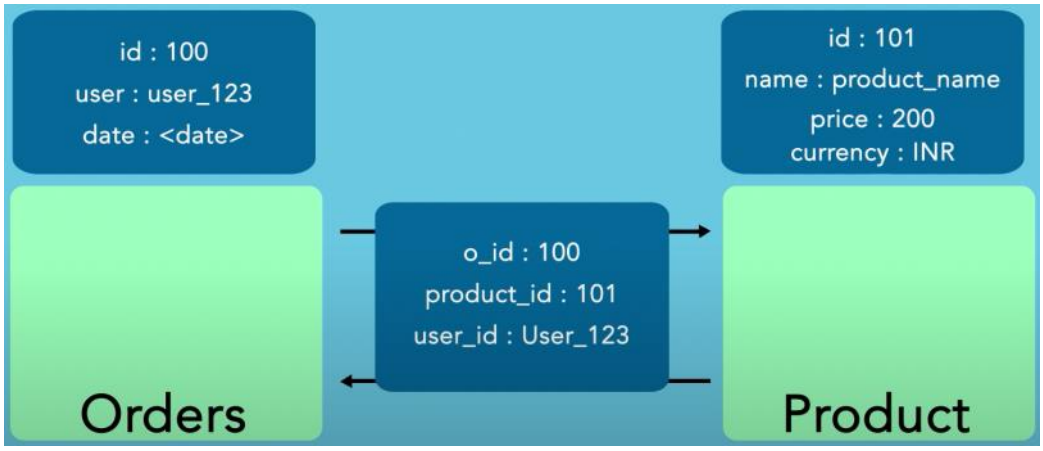



**Typical microservices architecture -**




## Detailed Breakdown of Key Concepts:

1. **High Cohesion and Low Coupling**:
   - **Cohesion**: A service should encapsulate all business logic related to a specific function (e.g., the "order" service should manage eve rything related to orders).
   - **Coupling**: Services should be minimally dependent on one another. For example, services should not share a database or rely on each ot her's internal details. This ensures changes in one service do not affect others.
   - Example: If the "order" service needs data from the "product" service, they should communicate via a decoupled contract (like using different IDs) to prevent changes in one service from breaking others. Which is nothing but **API contract.**




2. **Business Domain Modeling**:
   - Microservices are often structured around business domains. Each service is responsible for one part of the business (e.g., o rder, payment, product, invoices).
   - Example: In an e-commerce system:
     - **Order Service**: Manages order-related functions.

        - **Payment Service**: Manages payment processes.
        - **Product Service**: Manages product information.
    - This is nothing but a **domain-driven-design (DDD).**

3. **Independent Deployability**:
    - Microservices should be deployable independently, unlike monolithic systems where a change in one part requires redeploying the entire system.
    - In microservices, each service has its own deployment pipeline. This allows teams to update services without affecting others, improving efficiency.

4. **Independent Scaling**:
    - Microservices can be scaled individually based on demand. For instance, if the "order" service experiences higher traffic, only that service is scaled, not the entire system.
    - This is not possible in monolithic systems, where scaling often requires scaling the entire application.

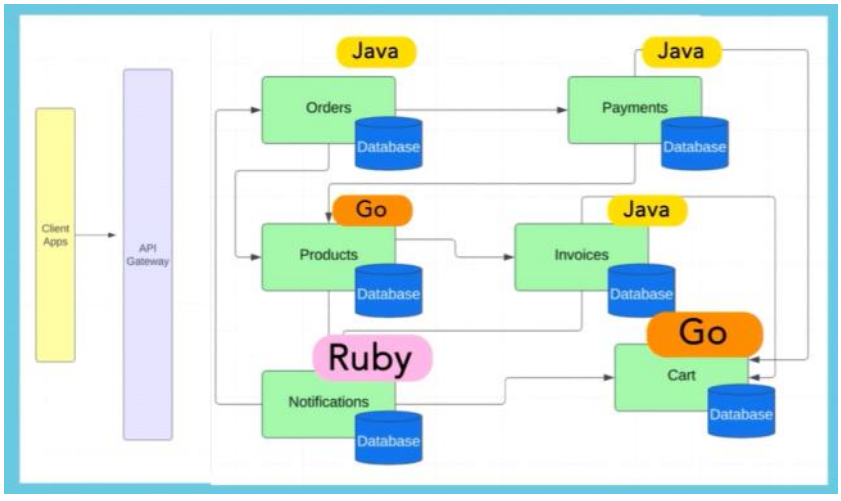
5. **Observability**:
    - Microservices architecture provides better visibility into system health. Monitoring tools can track each service's performance, ensuring they function properly and identifying issues when they arise.
    - It also helps detect patterns of degradation, ensuring system health is maintained.

6. **Resilience**:
    - Microservices improve resilience by isolating failures. If one service fails, it doesn't bring down the entire system. Fault tolerance mechanisms can be implemented to handle failures in a service without affecting the whole system.

7. **Team Organization**:
    - Microservices allow for smaller, focused teams that own individual services. This encourages better ownership, smoother communication, and faster feature delivery.
    - Teams can also be more adaptable, with members able to shift between services as needed.

**Advantages of Microservices:**

1. **Autonomy of Tech Stack**:
    - Teams can choose different technologies for each microservice based on its specific needs (e.g., using Ruby for notifications, Java for payments).
    - This flexibility allows the use of the best tool for each service's requirements.




2. **Reusability**:
    - Services can be reused across different parts of an organization or in different use cases. For example, an "invoices" service could be used for both B2B and B2C models.
3. **Improved Team Performance**:
    - Independent teams working on different services can build features in parallel, reducing conflicts and increasing development speed.
    - Smaller teams can specialize in specific areas, allowing for better focus and faster delivery.
4. **Independent Deployment and Infrastructure**:
    - Microservices allow for independent deployment pipelines and migration to better infrastructure (e.g., from legacy systems to Kubernetes or cloud-based solutions).

**Downsides of Microservices:**

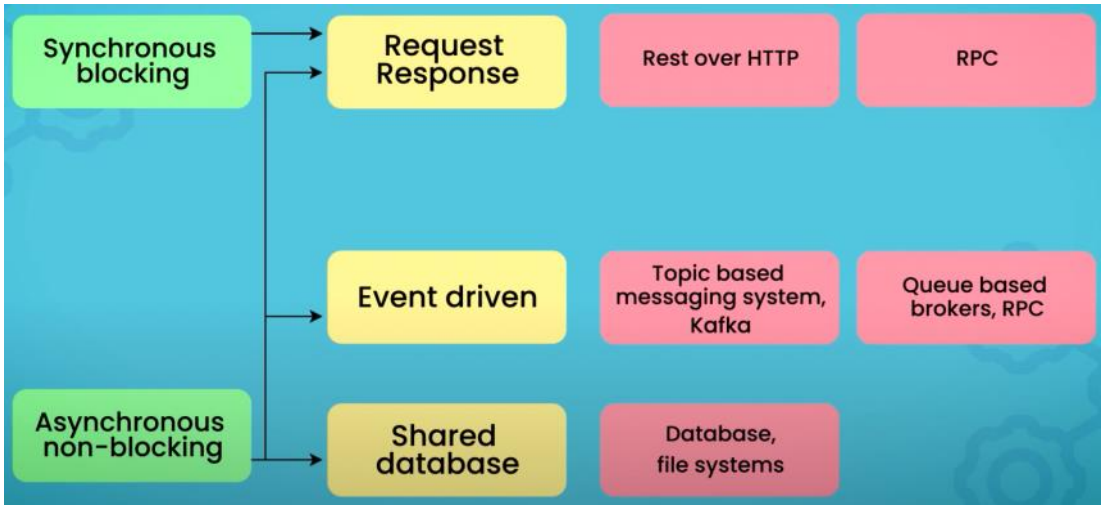
1. **Increased Cost**:
    - Moving to a microservices architecture requires more resources (storage, network, machines) and incurs higher infrastructure costs compared to a monolithic system.

2. **Complexity in Maintenance**:
   - Managing multiple technologies, databases, and services can become complex and require a diverse skill set. This may be challenging for organizations without sufficient expertise.
3. **Data Consistency**:
   - Microservices are prone to data consistency issues due to distributed state management. For example, one service may have outdated data that causes discrepancies in the system (e.g., order state not synchronized across services).
   - Handling this requires careful design, including using eventual consistency and appropriate fault tolerance mechanisms.

**Microservices Communication Overview**

To understand microservices, it is crucial to comprehend how they communicate. This determines the overall architecture and helps in deciding whether moving to microservices is the right choice. Communication between microservices can be classified into two primary styles:

1. **Synchronous Blocking Communication**
2. **Asynchronous Non-blocking Communication**




**1. Synchronous Blocking Communication**

- **Definition**: In synchronous blocking communication, a microservice sends a request to another service and **waits** for the response before proceeding with other tasks. This means that the service is blocked until it receives a reply from the other service.
- **Example**:
  - Order service sends a request to Inventory service to get stock details.
  - The Order service waits for the response before proceeding with further operations.
- **Implementation**:
  - This can be implemented using **REST APIs**, **gRPC**, or **RPC** (Remote Procedure Call).
  - **Request-response** mechanism: The calling service waits for the response before proceeding.
- **Disadvantages**:
  - **Chaining of calls**: If there are multiple services involved in a request-response cycle, it can result in a "chain" of synchronous calls that can become slow, resulting in timeouts.
  - **Scalability issues**: With too many synchronous calls, it may lead to performance bottlenecks.

**2. Asynchronous Non-blocking Communication**

- **Definition**: In this style, a microservice sends a request, event, or data to another microservice and **does not wait** for a response. Instead, it continues its operations, and the other service responds when ready.
- **Example**:
  - **Order service** sends an event (e.g., "order placed") to the **Notification service**.
  - The **Notification service** processes the event asynchronously without blocking the **Order service**.
- **Implementation**:
  - Can be implemented using **event-driven architecture**, **topic-based messaging systems**, **queue-based messaging systems**, and sometimes **RPC**.
  - **Request-response**: Even in asynchronous communication, a response may still be sent, but the calling service does not wait for it.

**Types of Asynchronous Non-blocking Communication**

1. **Event-driven communication**:
   - Microservices communicate via events, and services react to these events without the need for direct synchronization.
   - **Advantages**: High decoupling; the sending service does not need to know about the receiving services.
   - **Example**:
     - The **Order service** sends an event (e.g., order placed) to a queue/topic.
     - Other services (e.g., Inventory, Notification services) consume this event asynchronously.
2. **Shared Database** (not recommended):
   - In this method, multiple microservices share a common database to read/write data.
   - **Use Case**: It works for one-way communication, where one service writes data, and others read it.
   - **Disadvantages**:
     - If multiple services write to the same database, it can lead to **data leaks** or **data inconsistencies**.

- This approach can create tight coupling between services.

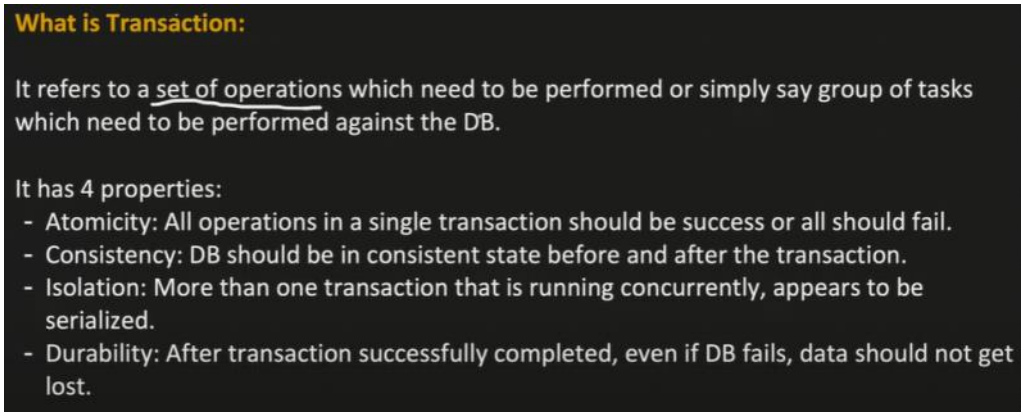
**Detailed Comparison:**

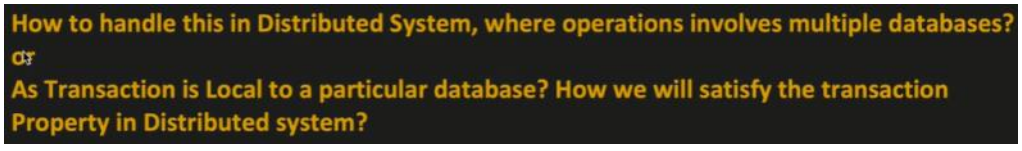
- **Request-Response (Synchronous)**:
  - Services are tightly coupled, as they wait for responses.
  - Commonly used via REST APIs, gRPC, or RPC.
- **Event-driven (Asynchronous)**:
  - Decouples services. A service doesn't need to wait for a response but sends an event and proceeds.
  - Useful for systems that need to scale and operate efficiently.
  - Events can be consumed by multiple services without the sender knowing who the consumers are.
- **Shared Database (not recommended)**:
  - Common data store that services share. Works if communication is one-way (write only).
  - Not ideal for scalable systems with complex data management requirements.

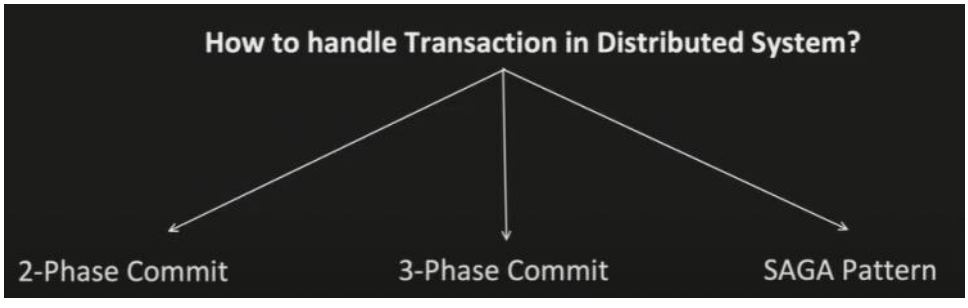
**Key Considerations in Choosing Communication Styles**

- **Synchronous Blocking**:
  - Best for scenarios where immediate feedback is required and the service interaction is simple.
  - Not ideal for systems with high latency or many interdependent services.
- **Asynchronous Non-blocking**:
  - Best suited for systems requiring decoupled services, scalability, and resilience.
  - Suitable for event-driven architectures where services react to events without waiting for synchronous responses.
- **Shared Database**:
  - Avoid when multiple services need to both read and write data. The shared database approach leads to **data consistency issues**.

## How to do distributed transactions in a right way?

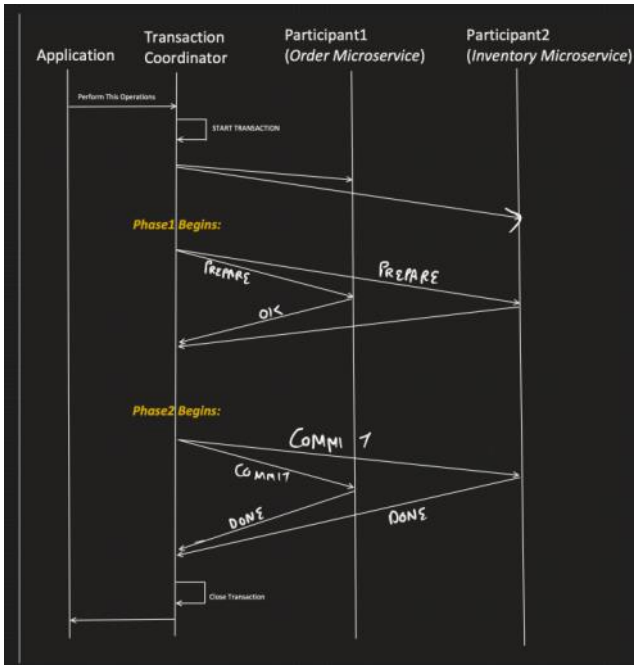



**2-Phase Commit:**

The Two-Phase Commit (2PC) protocol is a consensus algorithm that ensures distributed transactions are completed in a coordinated and reliable manner. It is often used in systems that involve multiple microservices or distributed databases to guarantee that either all participants commit to the transaction or none of them do, preventing data inconsistency.




There are 2 phases as name suggests.

1. **Prepare Phase (Voting Phase) :**

- The **Transaction Coordinator** initiates the process by sending a **prepare** message to all participants (Order and Inventory services in this case).
- **Order Service** receives the request to prepare for the transaction:
  - It ensures the **order** is ready to be placed, meaning it has all the necessary details (e.g., customer info, items, pricing).
  - The **Order Service** may need to check with the **Inventory Service** to ensure that the items are available.
  - The **Order Service** responds to the coordinator with a "yes" vote if it is ready to proceed, or a "no" if something prevents it from proceeding (for example, the order can't be completed due to stock issues).
- **Inventory Service** receives the prepare request:
  - It checks whether the requested items are available in stock and whether it can reserve or update the stock levels for the or der.
  - If the **Inventory Service** can successfully reserve or reduce the stock, it responds with a "yes" vote, indicating that it is ready to commit.
  - If the inventory is insufficient or another issue arises, the **Inventory Service** responds with a "no," indicating it cannot commit to the transaction.

2. **Commit Phase (Decision Phase):**
   Once the coordinator receives all the votes from the participants (Order and Inventory services), it makes a decision:
   - **If both services (Order and Inventory) voted "yes"**:
     - The coordinator sends a **commit** message to both services, instructing them to finalize the transaction (e.g., finalize the order in the **Order Service** and update the inventory in the **Inventory Service**).
     - Both services proceed to **commit** their changes. In this case:
       - **Order Service** creates the order record and finalizes the transaction.
       - **Inventory Service** reduces the stock levels accordingly.
   - **If either service voted "no"**:
     - The coordinator sends an **abort** message to all services, instructing them to roll back any changes that may have been made during the transaction.
     - Both services need to revert to their state before the transaction began.
       - **Order Service** will discard the order and not finalize any process related to it.
       - **Inventory Service** will roll back any reserved inventory changes or other adjustments made during the transaction.

**Failure scenarios:**

1) Failure During the Prepare Phase -

- If a failure occurs during the **Prepare Phase**, it typically happens at the point when the **Transaction Coordinator** sends a **prepare** message to one or more of the participants.
- If a participant **fails to respond** (due to network issues, crash, or timeout), the **Transaction Coordinator** cannot proceed with the transaction and will send an **abort** message to all participants.
- If a participant sends a **"no" vote**, the transaction is also aborted.

2) Failure During the Commit Phase (Decision Phase) -

- In this phase, the **Transaction Coordinator** has received all votes and is ready to issue a **commit** or **abort** message. If all participants voted "yes," the coordinator sends a **commit** message. However, failure may occur at this point.
- **Coordinator crashes after sending commit**: If the **Transaction Coordinator** crashes after sending the **commit** message, some participants may have committed their changes (because they received the commit message), but the coordinator does not know whether all participants succeeded. This creates an issue of uncertainty. The **Transaction Coordinator** may need to recover and retry the decision after a crash, but if recovery is not possible, there could be an incomplete or inconsistent transaction.

- **Network failure while sending commit**: If a **commit** message is not delivered to a participant, that participant may not commit the transaction. The coordinator will not know if the participant committed or not. When the coordinator recovers, it will have to check the status of the participant (usually through logging or a retry mechanism).
- **Participant failure after commit message**: If a participant crashes after receiving the commit message but before it can commit (e.g., after updating the order but before updating inventory), there is a possibility of inconsistent data. The **Inventory Service** might fail to update stock levels, while the **Order Service** might have created the order record. This results in an inconsistent system where the order is created but the stock levels are incorrect.

**Resolving Failures in 2PC:**
If a failure occurs during the commit phase, the system needs to handle it to ensure **data consistency**:

- **Coordinator recovery**: The **Transaction Coordinator** can try to **recover** the transaction by looking at logs or other mechanisms to determine whether the transaction was successfully committed by all participants.
    - If a **participant failed** after committing, the coordinator may need to **re-validate** the participant's commit (e.g., by checking inventory levels or orders).
    - If a **participant failed before committing**, the coordinator may need to **abort** the transaction for that participant.
- **Timeouts and retries**: The coordinator can implement **timeouts** and **retry mechanisms** to handle scenarios where messages are not delivered or participants are unresponsive. This would help ensure the coordinator can eventually make a decision.

**Final Outcome: Inconsistent Data:**
If a failure occurs and the **2PC protocol** cannot resolve it, the result is **inconsistent data**. For example:

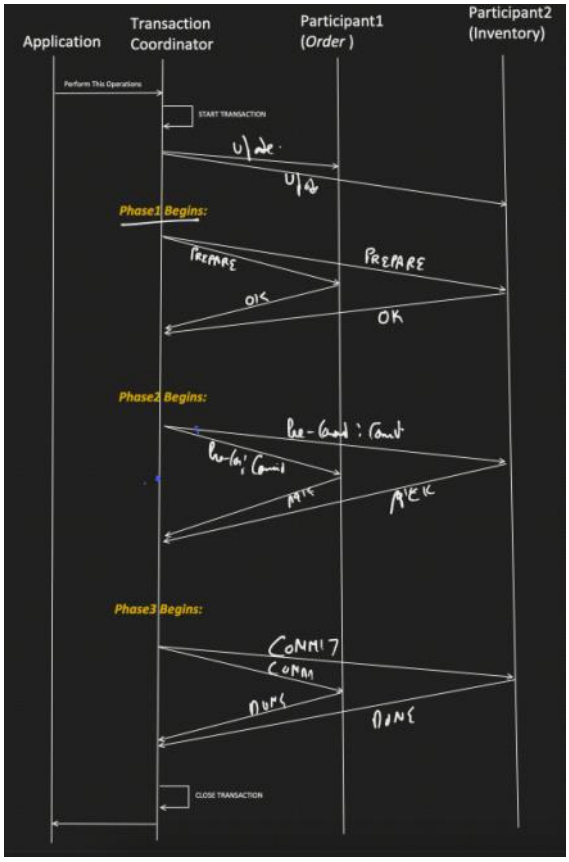
- The **Order Service** may have created an order, but the **Inventory Service** may not have updated the stock level (because it failed to receive the commit message).
- Or, if both services commit and then fail during recovery, the **Transaction Coordinator** might not know that the commit was successful for both services.

In such cases, systems may implement **compensation transactions** or additional protocols (e.g., **Saga pattern**) to handle failures and ensure that the system eventually reaches a consistent state, even if a commit failed.

Disadvantages -

1. **Blocking Nature (Synchronous)**
   Blocking of Participants: The 2PC protocol is synchronous and blocking. Once a participant (a service or resource) has voted in the Prepare Phase, it cannot proceed with any other work until it receives a decision (commit or abort) from the coordinator. If the coordinator crashes or becomes unresponsive, the participants may be left in a blocked state until the coordinator recovers or resolves the issue.
   Coordinator Blockage: The coordinator also cannot proceed unless it has received responses from all participants. If any participant crashes or experiences a failure during the Prepare Phase, the entire transaction can be blocked until the issue is resolved.
2. **Single Point of Failure (Coordinator Dependency)**
   Coordinator as a Bottleneck: The Transaction Coordinator is a critical component in 2PC. If the coordinator fails during the transaction process (e.g., after sending a "prepare" message but before sending the "commit" message), the entire transaction may fail or be in an inconsistent state. The system becomes heavily dependent on the availability and reliability of the coordinator.
   Single Point of Failure: Because the transaction's outcome depends entirely on the coordinator, if the coordinator crashes and cannot recover, the whole transaction might fail or be stuck indefinitely. This creates a single point of failure.

**3-phase commit :**




The **Three-Phase Commit (3PC)** protocol is an extension of the **Two-Phase Commit (2PC)** protocol, designed to overcome some of the key disadvantages of 2PC, particularly the problem of blocking in the case of failures. Below is a detailed explanation of how 3PC addresses the issues in 2PC:

**The Key Disadvantages of Two-Phase Commit (2PC):**

1. **Blocking on Coordinator Failure**:
   In **2PC**, if the **coordinator** crashes after sending the "prepare" message but before sending the "commit" or "abort" decision, the participants are left in a **blocking state**. They are unsure whether they should commit or abort the transaction. This situation can persist indefinitely, resulting in a **liveness issue** where the transaction is not able to proceed.
2. **Blocking on Participant Failure**:
   If a **participant** crashes during the transaction (e.g., after sending a "vote" to the coordinator but before receiving the final "commit" or "abort" message), the coordinator might not be able to decide whether to commit or abort. As a result, other participants are left waiting, blocking the entire transaction.
3. **No Guarantees of Availability**:
   In **2PC**, there is no provision for resolving situations where a failure happens after a participant has committed but the coordinator has not received the commit decision from all participants. This can leave the system in an inconsistent state.

**How Three-Phase Commit (3PC) Resolves These Issues:**

3PC introduces an additional phase (pre-commit) in the protocol to ensure that the system can make a more informed decision in the case of failures and avoid blocking. Here's how 3PC works, and how it resolves the issues present in 2PC:

**Phases of Three-Phase Commit (3PC):**

1. **Phase 1: Can Commit? (Coordinator asks Participants)**
   - The **coordinator** sends a **"can commit?"** message to all participants, asking them if they are ready to commit.
   - Each **participant** responds with either a **"Yes"** (if it is ready to commit) or **"No"** (if it cannot commit, for example, due to internal issues).
2. **Phase 2: Pre-Commit (Participants reply, and the Coordinator makes the decision)**
   - Once the **coordinator** has received a "Yes" from all participants, it sends a **"pre-commit"** message to all participants, indicating that they should prepare to commit but not actually commit yet.
   - If any participant cannot proceed or has indicated "No" in Phase 1, the coordinator will send an **"abort"** message, terminating the transaction early.
3. **Phase 3: Commit or Abort (Final Decision)**
   - Once all participants receive the **"pre-commit"** message, they are expected to acknowledge it and prepare to commit. After this acknowledgment, the coordinator sends the final **"commit"** message to all participants, signaling that they should actually commit the transaction.
   - If there is a failure before the coordinator sends the "commit" message, the transaction is **safe to abort** without blocking, since all participants are in a "pre-commit" state and cannot permanently commit the transaction until the final "commit" message is sent.

**How 3PC Resolves the Disadvantages of 2PC:**

1. **Avoids Blocking on Coordinator Failure:**
   - **In 2PC**, if the coordinator crashes after sending the "prepare" message but before sending the final "commit" or "abort," participants are left

in a **blocking state** because they are unsure whether to commit or abort.
    - **In 3PC**, the **pre-commit phase** ensures that once all participants have agreed to commit, they are prepared to do so and can safely **abort** if the coordinator fails. This ensures that participants are not left in a blocking state.
        - If the coordinator crashes after sending the "pre-commit" message, participants will eventually reach a consensus during the **next coordinator election** (in case of a system restart or recovery) and can decide whether to commit or abort based on the "pre-commit" message.
2. **Avoids Blocking on Participant Failure:**
    - **In 2PC**, if a participant crashes after voting "Yes" but before receiving the "commit" or "abort" decision, the entire system can block.
    - **In 3PC**, the **pre-commit phase** gives the participants an opportunity to acknowledge their readiness to commit. Even if a participant crashes before sending an acknowledgment, it cannot **commit** by itself without the coordinator's final "commit" message. This reduces the risk of blocking because, if a participant does not acknowledge the "pre-commit" message, the coordinator will send an **abort** message and the transaction will be safely rolled back.
3. **Reduces the Likelihood of Inconsistent State (Liveness Guarantees):**
    - **In 2PC**, a failure in the middle of the protocol can leave the system in an inconsistent state (e.g., some participants commit, but others do not).
    - **In 3PC**, the **pre-commit phase** guarantees that, even in the case of a crash, the system can recover in a consistent state because:
        - Participants will only commit after receiving a "commit" message from the coordinator.
        - If the coordinator crashes after sending the "pre-commit" message but before sending "commit," participants will know they should abort the transaction because they never received the final "commit" message.

| Disadvantage | Description |
|---|---|
| Increased Complexity and Overhead | More messages and phases lead to higher communication overhead and longer transaction times. |
| Vulnerable to Network Partitions | Network failures can lead to partitions, which might cause inconsistent states in different parts of the system. |
| Failure Recovery Complexity | Recovery from failures is more complex, as participants need to handle failure scenarios independently. |
| Potential for Inconsistent States | Uncertainty in system state during the pre-commit phase can lead to data inconsistencies. |
| No Full Transaction Atomicity | Does not guarantee full atomicity, as some participants may commit while others do not. |
| Susceptibility to Failures in Pre-Commit Phase | A failure in the pre-commit phase can cause a system to become stuck, requiring timeouts and aborts. |
| Limited Scalability | The additional phases and messages introduce significant overhead, especially in large systems. |
| Reduced Fault Tolerance in Certain Failures | Some failure scenarios (e.g., simultaneous coordinator and participant crashes) can still result in data loss or inconsistency. |

- Coordinator failure during pre-commit or post-pre-commit phases introduces complexity and potential risks in a distributed system. The Three-Phase Commit (3PC) protocol provides a mechanism to ensure that Participants have a better way to handle failures and avoid blocking indefinitely, but it still requires mechanisms like timeouts and coordination for consistency and recovery. The failure of the Coordinator in this protocol increases the complexity of ensuring that all nodes are in a consistent state and may still require manual intervention or complex recovery mechanisms.

### SAGA Pattern:

The **SAGA** pattern is a design pattern used to manage distributed transactions in a **microservices** architecture, specifically when we want to ensure consistency across multiple, independent services without relying on traditional **ACID transactions** (which aren't feasible in distributed systems).
The key idea behind SAGA is that a distributed transaction is broken down into a series of **local transactions** that are managed by each service involved. Each of these local transactions can either **commit** or **rollback** independently. If one of the local transactions fails, the system ensures that the failure is handled by performing **compensating transactions** on the previously completed steps to bring the system back to a consistent state.

**Basic Principles of the SAGA Pattern**

1. **Local Transactions**: Each service involved in the saga performs its own local transaction. These local transactions are typically small, isolated operations like adding an order, debiting an account, or updating inventory.
2. **Coordination**: The SAGA pattern does not rely on a centralized coordinator (like **2PC** or **3PC**) but instead relies on the orchestration or choreography to manage the sequence of transactions.
3. **Compensating Transactions**: If any of the local transactions fail, compensating actions (also called **"rollback"** or **"undo"** operations) are triggered to undo the effects of previously completed local transactions, ensuring that the system remains consistent.

**How Does SAGA Work?**

Here's a simplified view of how the **SAGA pattern** works using both **Choreography** and **Orchestration**:

**1. Example Scenario: Order Processing**

Imagine a scenario where we are processing an order for an e-commerce platform. The following services would be involved:

- **Order Service**: Handles the order creation.
- **Payment Service**: Processes payment for the order.
- **Inventory Service**: Updates the inventory based on the purchased items.

In a typical **SAGA** pattern:

**Step-by-step Execution:**

1. **Order Service**:
   - The saga starts with the **Order Service** creating an order.
   - It performs its **local transaction** (e.g., creating the order in the database).
   - **Choreography**: It emits an event like "OrderCreated".
   - **Orchestration**: The orchestrator sends a "Process Payment" command to the **Payment Service**.
2. **Payment Service**:
   - The **Payment Service** processes the payment (e.g., charges the customer's credit card).
   - If successful, it moves on to the next step.
   - **Choreography**: It emits an event like "PaymentSuccessful".
   - **Orchestration**: The orchestrator sends a "Update Inventory" command to the **Inventory Service**.
3. **Inventory Service**:
   - The **Inventory Service** updates the inventory to reflect that items have been purchased.
   - If successful, the order is marked as complete.
   - **Choreography**: It emits an event like "InventoryUpdated".
   - **Orchestration**: The orchestrator sends a "Commit Order" command to the **Order Service** to finalize the order.

**Failure Handling & Compensating Transactions:**

If something fails at any point, the saga must roll back the earlier transactions to maintain consistency. Let's look at failure scenarios:

- **Failure in Payment**:
  - If the **Payment Service** fails, the saga will **abort**.
  - A compensating transaction might be triggered to **cancel the order** or **reverse the order creation** (in the Order Service).
  - The **Order Service** can send a "Cancel Order" command to **undo** the order creation.
- **Failure in Inventory Update**:
  - If the **Inventory Service** fails (e.g., if there's insufficient stock), the saga should roll back.
  - The **Payment Service** might need to **refund** the payment.
  - The **Order Service** would cancel the order, thus compensating for the incomplete transaction.

Each service in the saga is responsible for performing its part and taking compensating actions if necessary.

## Service discovery and API gateway -


In a microservices architecture, service discovery is the process of automatically detecting and registering services so that they can be accessed dynamically by other services. This is essential in systems with multiple services where each service needs to know the location of others in order to communicate. Spring Boot, in combination with Eureka Server, offers an easy way to implement service discovery, enabling services to register themselves and discover other services dynamically.

- **Eureka Server**:
  - Acts as a **service registry** where all microservices register themselves.
  - It allows services to find each other at runtime without the need for hardcoded URLs or IP addresses.

- **Eureka Client**:
  - Any Spring Boot microservice that wants to register with Eureka needs to have the Eureka client library.
  - It registers with Eureka Server and uses Eureka to discover other services.

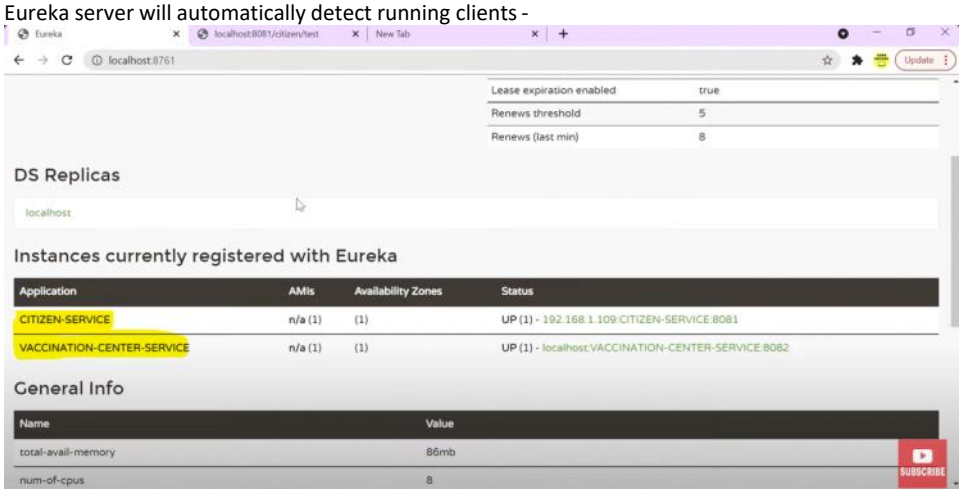
Example -
Let's create 3 spring-boot applications -

1) Eureka server - it will only have an eureka server dependency.
2) Citizen service - it will have eureka client dependency.
3) Vaccination center - it will have eureka client dependency.

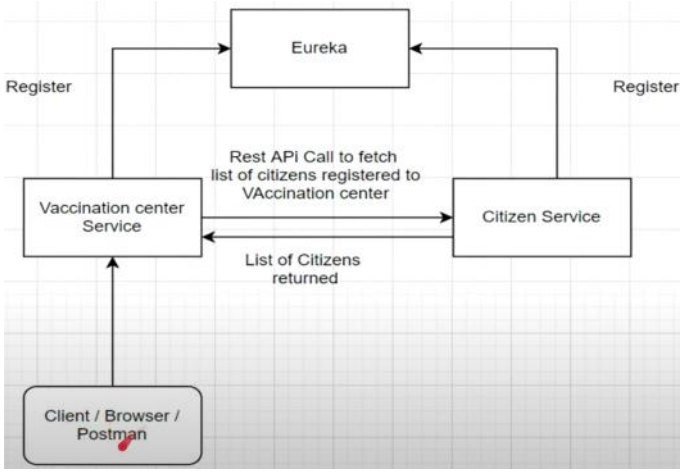
We can register/add Citizen to a particular vaccination center using vaccination center id using Citizen service.
We can add/register a vaccination center using Vaccination center.
Now, if we want to fetch a vaccination center details by passing vaccination center id, it should return vaccination center details along with citizens registered under that vaccination center. For this, we need to communicate with Citizen service.
We can communicate with citizen service using RestTemplate ->

```
@Autowired
private RestTemplate restTemplate;

@PostMapping(path ="/add")
public ResponseEntity<VaccinationCenter> addCitizen(@RequestBody VaccinationCenter vaccinationCenter) {

        VaccinationCenter vaccinationCenterAdded = centerRepo.save(vaccinationCenter);
        return new ResponseEntity<>(vaccinationCenterAdded, HttpStatus.OK);
}

@GetMapping(path = "/id/{id}")
public ResponseEntity<RequiredResponse> getAllDadaBasedonCenterId(@PathVariable Integer id){
        RequiredResponse requiredResponse =  new RequiredResponse();
        //1st get vaccination center detail
        VaccinationCenter center  = centerRepo.findById(id).get();
        requiredResponse.setCenter(center);

        // then get all citizen registerd to vaccination center

        java.util.List<Citizen> listOfCitizens = restTemplate.getForObject("http://CITIZEN-SERVICE/citizen/id/"+id, List.class);
        requiredResponse.setCitizens(listOfCitizens);
        return new ResponseEntity<RequiredResponse>(requiredResponse, HttpStatus.OK);
}
```

Here, we need to not mention any hardcoded url or port number while communicating with citizen service. This happened because of eureka server. So we have 3 applications in running mode - eurekaServer, citizenService(eureka client), vaccinationService(eureka client).
Eureka server will automatically detect running clients -

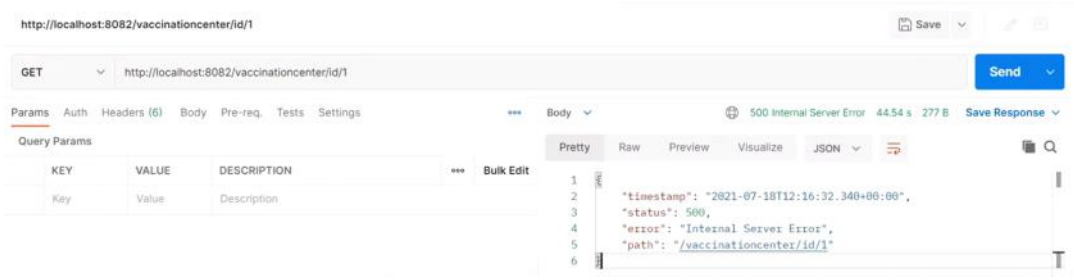



The names are taken from service-specific application.properties file. Any request coming to CITIZEN-SERVICE will be redirected to an available citizen services in a load balanced way.



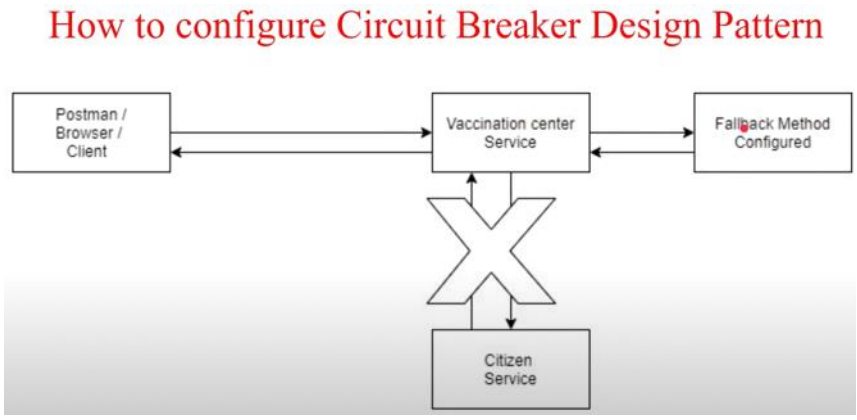

## Fault tolerance in Microservices and Hystrix (Circuit breaker) -

**Fault** - If some part of a system failed then it will cause error to the whole system.




**Fault tolerance** - It refers to the ability of a system or component to continue operating correctly even if some parts of it fail. In a distributed system like microservices, faults are inevitable due to network issues, service failures, or timeouts. A fault-tolerant system can detect failures and either gracefully degrade or attempt recovery, ensuring minimal impact on the overall system.
In Spring Boot, for example, libraries like Resilience4j and Hystrix can be used to achieve fault tolerance through features like retries, timeouts, and fallbacks.
**Circuit Breaker** - It is a design pattern used to detect failures in a service and prevent the system from making repeated requests to a failing service. It acts like an electrical circuit breaker, which opens the circuit when there's an issue, stopping the flow of requests to a failing service, allowing time for recovery, and preventing the service from being overwhelmed.

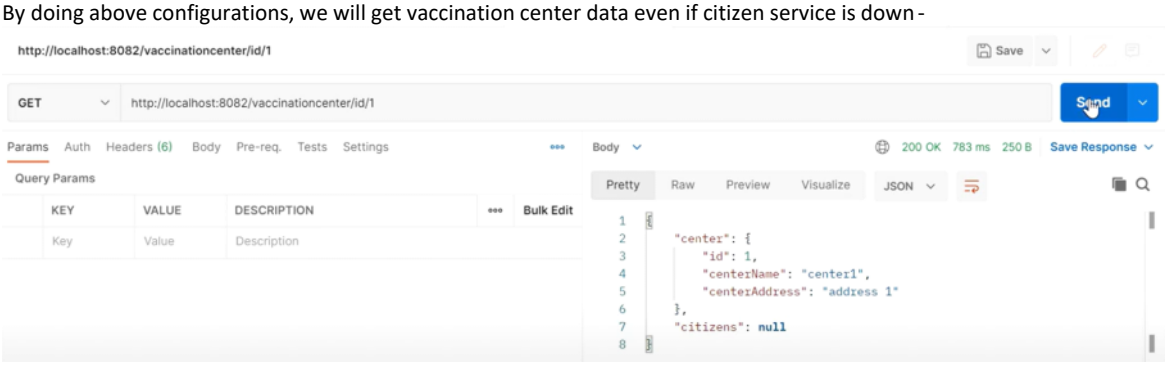
In our case, let's say if citizen service is failed then we can't get data for even vaccination service. We need to configure fault tolerance to get at least vaccination center data even if citizen service is failed.




Firstly we need to add Hystrix dependency in pom.xml (in vaccination service). Then we need to mark our SpringBootApplication with annotation **@EnableCircuitBreaker** and then add **@HystrixCommand** on the top of critical method.

```java
@GetMapping(path = "/id/{id}")
@HystrixCommand(fallbackMethod = "handleCitizenDownTime")
public ResponseEntity<RequiredResponse> getAllDadaBasedonCenterId(@PathVariable Integer id){
    RequiredResponse requiredResponse =  new RequiredResponse();
    //1st get vaccination center detail
    VaccinationCenter center  = centerRepo.findById(id).get();
    requiredResponse.setCenter(center);

    // then get all citizen registerd to vaccination center

    java.util.List<Citizen> listOfCitizens = restTemplate.getForObject("http://CITIZEN-SERVICE/citizen/id/"+id, List.class);
    requiredResponse.setCitizens(listOfCitizens);
    return new ResponseEntity<RequiredResponse>(requiredResponse, HttpStatus.OK);
}

public ResponseEntity<RequiredResponse> handleCitizenDownTime(@PathVariable Integer id){
    RequiredResponse requiredResponse =  new RequiredResponse();
    VaccinationCenter center  = centerRepo.findById(id).get();
    requiredResponse.setCenter(center);
    return new ResponseEntity<RequiredResponse>(requiredResponse, HttpStatus.OK);
}
```

By doing above configurations, we will get vaccination center data even if citizen service is down -




Internally it is achieved using proxy class. It will not directly call citizen service but calls a proxy. If we get response then only it will call actual citizen service. We can set the retry limit to ensure that at least those many attempts are done before calling fallback method.

## @HystrixCommand Elements

1) fallbackMethod : Specifies a method to process fallback logic
2) threadPoolKey : The thread-pool key is used to represent a HystrixThreadPool for monitoring, metrics publishing, caching and other such uses.
3) threadPoolProperties: Specifies thread pool properties.
4) groupKey: The command group key is used for grouping together commands such as for reporting, alerting, dashboards or team/library ownership

## @HystrixCommand Elements

5) commandProperties: Specifies command properties like

a) circuitBreaker.requestVolumeThreshold : number of requests reties before which circuit breaks
b) circuitBreaker.errorThresholdPrecentage eg 60%: If 60 % request fails out of total requests made then open the circuit
c) timeout s: Its like wait for milliseconds and if no response, break circuit

## API Gateway:

# Why API Gateway

1) We had eureka server, hystrix implemented in Previous videos, Is that not enough ? Why do we need API Gateway?
2) The reasons are many :
    a) When we scale up our microservices, clients needs to now exactly which microservice they need. Client/ FE Dont need such information that backend developer is developing which microservice for which functionality. To over come this we have API Gateway to create an abstraction layer between FE and Backend developers . Thus FE developers need not depend on Backend people anymore. This is as good as implementing Abstraction principle in Java OOps.
    b) It becomes harder to change microservice without affecting the client. In reality, the client doesn't need to know microservice and its implementation behind.

    a) With API gateway we can centralise common functionalities like authentication, logging, and authorization. Now API will authenticate all requests hence each MS need not implement the same. Thus reduces code redundancy.
    b) Multiple Versions of Service : Our MS is updated with the new changes, but the updated service is not compatible with the mobile client. Our mobile client will continue using version-V1 while other clients will move to the version-V2. This can be done using API Gateway Routing Technique
2) To address these issues, the architecture now contains another layer between the client and the microservices. This is API Gateway. API Gateway acts like a proxy that routes the request to the appropriate microservices and returns a response to the client. Microservices can also interact with each other through this Gateway.

# Different Implementations of API Gateways Available?

1) There are a number of API Gateways available and one can use any of these based on the needs.
    a) Netflix API Gateway (Zuul)
    b) Amazon API Gateway
    c) Mulesoft
    d) Kong API Gateway
    e) Azure API Gateway

In Real world you will see most of the projects using Netflix API Gateway (Zuul), But we will be implementing API Gateway using Spring Cloud Gateway. Why??

# Why Netflix Era came to an end and we all have to move on to Spring Cloud technologies?

Whenever if speak about MS, the first thing came to our mind was NETFLIX OSS support tools like Eureka, Zuul, Hystrix etc. But recently we see that many of these are now in maintenance mode. This means that there won't be any new features added to these modules, and the Spring Cloud team will perform only some bug fixes and fix security issues. The maintenance mode does not include the Eureka module, which is still supported.

Hystrix has been already superseded by the new solution for telemetry called Atlas.

The successor of Spring Cloud Netflix Zuul is Spring Cloud Gateway.

# Why Not Zuul? Why Spring Cloud Gateway.

Zuul is a blocking API. A blocking gateway api makes use of as many threads as the number of incoming requests. So this approach is more resource intensive. If no threads are available to process incoming request then the request has to wait in queue.

Spring Cloud Gateway is a non blocking API. When using non blocking API, a thread is always available to process the incoming request. These request are then processed asynchronously in the background and once completed the response is returned. So no incoming request never gets blocked when using Spring Cloud Gateway.

# Getting Started with Spring Cloud Gateway.

To include Spring Cloud Gateway in your project, use the starter with a group ID of org.springframework.cloud and an artifact ID of spring-cloud-starter-gateway.

Also you need to make it eureka client

If you include the starter, but you do not want the gateway to be enabled, set spring.cloud.gateway.enabled=false.

Api-gateway works on routing principle. Create a new speing boot application with spring-cloud-gateway dependency.
application.yaml/application.properties ->

```yaml
server:
  port: 8083

spring:
  application:
    name: API_GATEWAY


  cloud:
    gateway:
      routes:
      - id: CITIZEN-SERVICE
        uri:
          lb://CITIZEN-SERVICE
        predicates:
        - Path=/citizen/**

      - id: VACCINATION-CENTER-SERVICE
        uri:
          lb://VACCINATION-CENTER-SERVICE
        predicates:
        - Path=/vaccinationcenter/**

```

After configuring above routes, start the application and hit any endpoint on 8083 port it will redirect to respective service. As we have eureka service in place and also it is lodbalanced, we can directly use uri with **lb: //<service_name>**

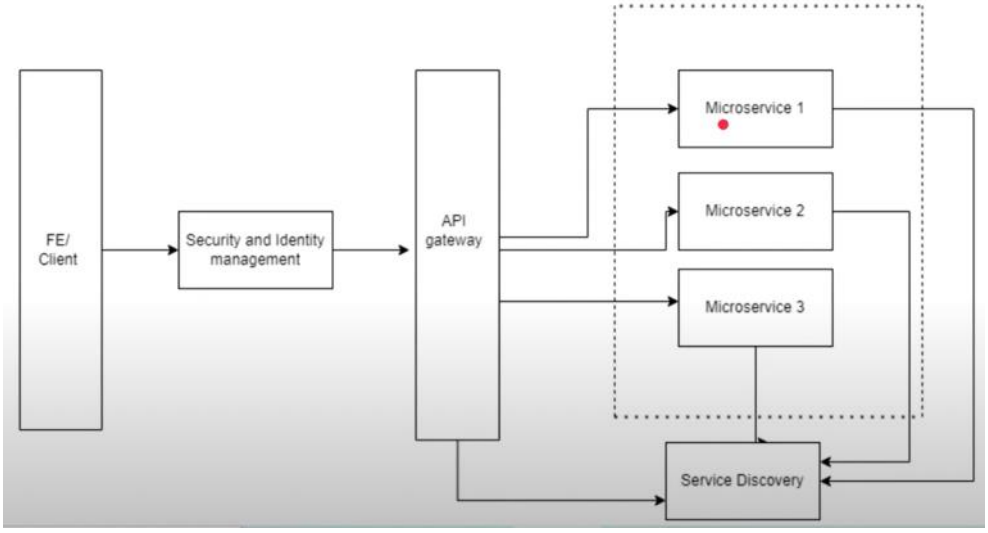
# Terms and terminologies Spring Cloud Gateway.

Route: The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.

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.

Filter: These are instances of Spring Framework GatewayFilter that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.

# How does Microservice Architecture work?



# What is the difference between Monolithic, SOA and Microservices Architecture?

- Monolithic Architecture is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
- A Service-Oriented Architecture is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
- Microservice Architecture is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.
- Main diff b/w SOA and MS architecture is Based on sharing of data and info. SOA shares and reuses as much as possible while MS focuses on sharing as little as possible

## Messaging queue:

# Ways to communicate between Microservices

- We have seen Synchronous communications through -
  - Rest APIs
  - GraphQl
  - Feign using Eureka discoveries
  - GRPC ( 10 times faster than REST APIs ) - developed by Google as substitute of REST with many more features.
- A synchronous call means that a service waits for the response after performing a request.
- Today we will look at ways to do asynchronous communication in java. This communication usually involves some kind of messaging system like
  - Active Mqs
  - Rabbit MQs
  - Kafka

**Asynchronous Communication:**

In the case of Asynchronous Communication, the client service doesn't wait for the response from another service. When the client microservice calls another microservice, the thread is not blocked till a response comes from the server. The message producer service generates a message and sends the message to a message broker on a defined topic. The message producer waits for only the acknowledgment from the message broker to know that message is received by the broker.

The consuming service subscribes to a topic in the messaging queue. All the messages belonging to that topic will be fed to the consuming system(s). The message producer service and consuming services don't even know each other. The response is received in the same methodology through a message broker via defined message topics.

Different messaging tools are based on the AMQP (Advanced Message Queuing Protocol). Some examples are given below.

a. Apache Kafka
b. RabbitMQ
c. Apache ActiveMQ

## What is Async communication

- In Async communication , To initiate such type of communication, a Microservice who wants to send some data to another Microservice publishes a message to a separate component known as a message broker. It is responsible for handling the message sent by the producer service and it will guarantee message delivery.
- After the message is received by the broker, it's now its job to pass the message to the target service. If the recipient is down at the moment, the broker might be configured to retry as long as necessary for successful delivery.
- These messages can be persisted if required or stored only in memory. In the latter case, they will be lost when the broker is restarted and they are not yet sent to the consumer.
- Since the broker is responsible for delivering the message, it's no longer necessary for both services to be up for successful communication. Thus async messaging mitigates the biggest problem of synchronous communication - coupling.

## What if the message broker is down?

- A message broker is a vital part of the asynchronous architecture and hence must be fault tolerant
- This can be achieved by setting up additional standby replicas that can do failover. Still, even with auxiliary replicas, failures of the messaging system might happen from time to time.
- If it's essential to ensure the message arrives at its destination, a broker might be configured to work in at-least-once mode. After the message reaches the consumer, it needs to send back ACK to the broker. If no acknowledgement gets to the broker, it will retry the delivery after some time.

**What is a Messaging Queue?**

A **Messaging Queue (MQ)** is a communication mechanism used in distributed systems, where messages are sent between **producers** (senders) and **consumers** (receivers) in a way that decouples the sender from the receiver. It is a type of **asynchronous communication** where the producer sends messages to a queue, and the consumer processes these messages later. The queue temporarily stores messages until they can be processed.

The MQ ensures that the messages are reliably delivered from the sender to the receiver, even if the receiver is temporarily unavailable or there is a delay in processing.

**Key Components of a Messaging Queue:**

1. **Producer**: The component that sends the message to the queue.
2. **Queue**: The temporary storage where messages are placed.
3. **Consumer**: The component that retrieves and processes the messages from the queue.

**Why is a Messaging Queue Needed?**

In modern distributed applications, especially with **microservices**, there is a need to handle **asynchronous communication** between services to achieve scalability, reliability, and fault tolerance. Messaging queues are essential in the following scenarios:

1. **Decoupling**: Microservices and components are decoupled from one another. The producer doesn't need to know when or how the consumer will process the message. The messaging queue ensures that the message is delivered, even if the consumer is down or unavailable at the moment.
2. **Reliability**: If the consumer is temporarily unavailable, the message is still stored in the queue and will be processed later. This makes the system more reliable.
3. **Load Balancing**: A messaging queue helps distribute the load across multiple consumers, which can process messages in parallel, ensuring high throughput and scalability.
4. **Asynchronous Processing**: It allows services to process requests asynchronously, meaning the producer does not have to wait for the consumer to process the request. This results in better responsiveness and performance.
5. **Fault Tolerance**: Messaging queues offer durability (messages are persisted), ensuring that even in case of system failure, messages will not be lost and can be retried.

**Popular Messaging Queue Solutions:**

1. **RabbitMQ**: A widely used, open-source message broker that implements the **AMQP (Advanced Message Queuing Protocol)**. It is known for its ease of use, flexibility, and reliability.
2. **Apache Kafka**: A distributed streaming platform that is commonly used for real-time event streaming. Kafka is optimized for high throughput and scalability, making it ideal for big data and event-driven systems.

**Point-to-point and Pub-sub pattern** -

In the **Point-to-Point model**, the message is sent from a producer to a single consumer. This is a traditional queue system where each message is delivered to only one consumer.

## What is PTP Async communication

- PTP - A queue will be used for this type of messaging-based communication.
- The service that produces the message, which is called as producer (sender), will send the message to a queue in one message broker and the service that has an interest in that message, which is called a consumer (receiver), will consume the message from that queue and carry out further processes for that message
- One message sent by a producer can be consumed by only one receiver and the message will be deleted after consumed.
- If the receiver or an interested service is down, the message will remain persistent in that queue until the receiver is up and consumes the message.
- For this reason, messaging-based communication is one of the best choices to make our microservices resilient.
- A popular choice for the queueing system is RabbitMQ, ActiveMQ

In the **Publish-Subscribe** model, the message is sent from a **producer** to multiple **subscribers** (consumers). The **publisher** doesn't care about who receives the message; it's broadcast to **all subscribers** who are interested in that message or event.

## What is Publisher-Subscriber Async communication

- In publisher-subscriber messaging-based communication, the topic in the message broker will be used to store the message sent by the publisher and then subscribers that subscribe to that topic will consume that message
- Unlike point to point pattern, the message will be ready to consume for all subscribers and the topic can have one or more subscribers. The message remains persistent in a topic until we delete it.
- In messaging-based communication, the services that consume messages, either from queue or topic, must know the common message structure that is produced or published by producer or publisher.
- examples are Kafka, Amazon SNS etc

## What is Event Based Async communication

- Unlike messaging-based communication, in event-based communication, especially in event-driven pattern , the services that consume the message do not need to know the details of the message.
- In event-driven pattern, the services just push the event to the topic in the message broker and then the services that subscribe to that topic will react for each occurrence event in that topic Each event in the topic will be related to a specific business logic execution.

## How does synchronous and asynchronous communication work?

- Applications generate messages in the form of calls to functions, services and APIs. The way an architect designs these communications affects the application's performance, resource consumption and ability to execute tasks.
- When microservices communicates synchronously, it sits idle until it receives a call, response, value or other data transfer. For example, synchronous execution occurs during vaccination slot booking, A user decides to book a slot, and the system generates a query to determine if slot is available. The app waits for a response before starting the slot booking process. This synchronou design prevents mismatches between slot booking and slot availability.
- Conversely, asynchronous communication allows code to continue to run after it has generated a call or response. Asynchronous communication is particularly valuable for reporting and alerts, such as sending emails and text messages of slot booked to citizens

## What did u use in your project and why? Pros and cons of each communication method

- Both have potential pros and cons, but the method you choose depends on an application's purpose.
- Pros of Synchronous Communication
  - Synchronous communication is simpler in design
- Cons -
  - carries the risk of spreading failures across services
- Way to mitigate failures in Synchronous communication -
  - The architect must implement sophisticated service discovery and application load balancing among microservices.

## What did u use in your project and why? Pros and cons of each communication method

- Cons -
  - Asynchronous communication has lesser architectural simplicity and data consistency
- Pros of Asynchronous Communication
  - Resilience and scalability
  - Provide better control over failures than synchronous setups
  - Loosely coupled Micro services.

## When to use which communication method

- When we start creating the application from scratch go with a synchronous system to optimize for speed of evolution
- And then once your microservices architecture grows and starts becoming complex and multifunctional then focus on switching to asynchronous communications. Identify all possible communications your microservice do to interact with order Microservices. Figure out if it strictly needs to be synchronous. If the response is really not necessary to proceed with other functionalities then convert that communication channel to Asynchronous communication channel ( like with ActiveMQ, Rabbit Mq, kafka etc)

## Conclusion for Sync and Asyn communication

One of the traditional approaches for communicating between microservices is through their REST APIs. However, as your system evolves and the number of microservices grows, communication becomes more complex, with services depending on each other or tightly coupled, slowing down development teams. This model can exhibit low latency but only works if services are made highly available.

To overcome this design disadvantage, new architectures aim to decouple senders from receivers, with asynchronous messaging. In a Kafka-centric architecture, low latency is preserved, with additional advantages like message balancing among available consumers and centralized management.

When dealing with a legacy platform , a recommended way to de-couple a monolith and ready it for a move to microservices is to implement asynchronous messaging.

**Apache Kafka -**

# What is Kafka

- Apache Kafka is publish-subscribe based fault tolerant messaging system. It is fast, scalable and distributed by design.
- It was initially thought of as a message queue and open-sourced by LinkedIn in 2011. Its community evolved Kafka to provide key capabilities:
  - Publish and Subscribe to streams of records, like a message queue.
  - Storage system so messages can be consumed asynchronously. Kafka writes data to a scalable disk structure and replicates for fault-tolerance. Producers can wait for write acknowledgments.
  - Stream processing with Kafka Streams API, enables complex aggregations or joins of input streams onto an output stream of processed data.

Traditional messaging models are queue and publish-subscribe. In a queue, each record goes to one consumer. In publish-subscribe, the record is received by all consumers.

# Pros of Kafka

- Loose coupling — Neither service knows about each other regarding data update matters.
- Durability — Guarantees that the message will be delivered even if the consumer service is down. Whenever the consumer gets up again, all messages will be there.
- Scalability — Since the messages get stored in a bucket, there is no need to wait for responses. We create asynchronous communication between all services.
- Flexibility — The sender of a message has no idea who is going to consume it. Meaning you can easily add new consumers (new functionality) with less work.

# Cons of Kafka

- Semantics — The developer needs to have a deep understanding of the message flow as its strict requirements. Complex fallback approaches may take place.
- Message Visibility — You must track all those messages to allow you to debug whenever a problem occurs. Correlation IDs may be an option.

# Active MQ Vs RABBIT MQ VS Kafka communication

| Features | Active MQ/Rabbit MQ | Kafka |
|---|---|---|
| Developed By | Active mq - Created by founders from LogicBlaze, as an open-source message broker, later donated to Apache Software Foundation, where founders continue to develop the code base<br>Rabbit MQ- RabbitMQ is now owned by Pivotal Software Inc | Initially developed by LinkedIn. Later in In 2011 Kafka was open sourced and developed via the Apache Software Foundation hence named Apache Kafka. |
| Written in | Active mq - Java<br>Rabbit mq - Erlang | Java and Scala |
| Type PTP/ Pub-sub Messaging System | PTP Messaging System | Publish - Subscriber. Apache Kafka allows publishing and subscribing to streams of records. |

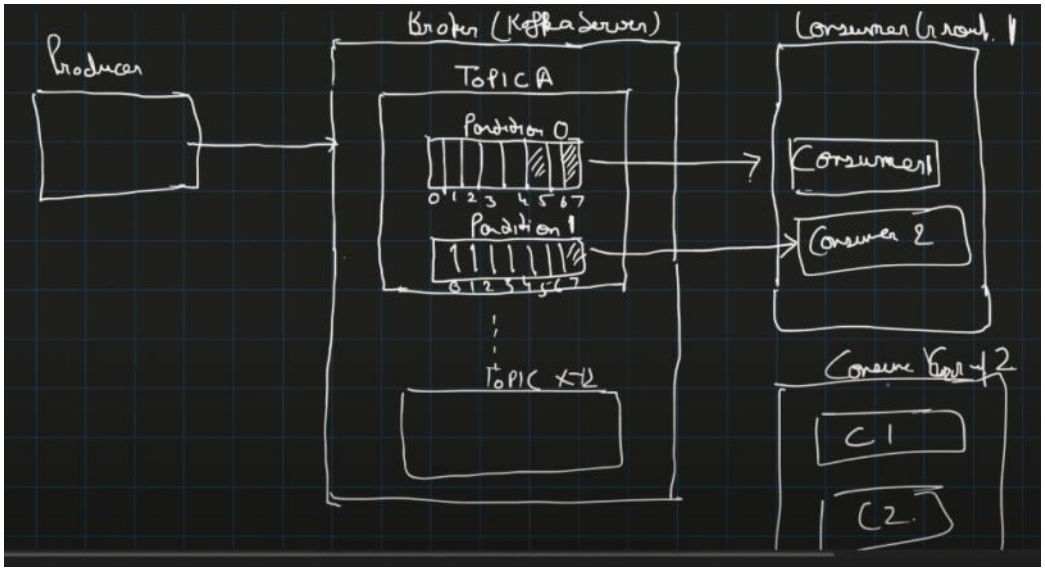
| Features | Active MQ/Rabbit MQ | Kafka |
|---|---|---|
| Overview | It is a traditional messaging system that deals with a small amount of data.<br><br>With point to point messaging, one or more consumers are connected to the queue, while the broker uses the 'round robin' approach to direct messages to specific consumers. | Kafka is a distributed publish-subscribe message delivery and logging system that follows a publisher/subscriber model with message persistence capability. Producers push event streams to the brokers, and consumers pull the data from brokers.<br><br>In subscription-based messaging, brokers broadcast messages to all consumers 'subscribed' to the topic. |
| Recommendation | Use them for exactly once delivery and for valuing messages | Use Kafka for high-performance monitoring and where losing messages is not important |

| Features | Active MQ/Rabbit MQ | Kafka |
|---|---|---|
| Throughput | Lower throughput since each delivery message state is maintained. | Higher throughput since producers dont wait for acknowledgements from brokers. So brokers can write messages at a very high rate. |
| Order of Messages | Cannot assure the order of messages to remain the same | Can assure that the messages retain the order in which they are sent |
| Filtering | Work of filtering is done by the JMS API message selector and not by the applications | No fundamental filters at the brokers. Hence it must be done by consumers or apps |
| Messaging Type | Push type messaging platform wherein the providers push the messages to consumers | Pull type messaging platform wherein consumers pull the message from the brokers |

| Features | Active MQ/Rabbit MQ | Kafka |
|---|---|---|
| Responsible Stakeholders | Responsibility of producers to ensure delivery of messages | Responsibility of consumers to consume messages |
| Scalability | No chances of horizontal scalability and replication | Scalable and available because of replication of partitions |
| Performance | Performance slows down as the number of consumers starts increasing. 4K-10K messages per second | Performance remains effective and good even if there are newer consumers being added. 1 million messages per second |
| Support for synchronous or asynchronous | Supports both | Supports asynchronous |

| Features | Active MQ/Rabbit MQ | Kafka |
|---|---|---|
| Message Retention | Acknowledgment based. These messages are removed from the queue once they are processed and acknowledged. | Policy-based (e.g retain only for 30 days). Kafka messages are always forever saved until the retention time expires, the developers have no other option. Kafka is a log. |
| Consumer Mode | Smart broker/dumb consumer. The broker consistently delivers messages to consumers and keeps track of their status | Dumb broker/smart consumer. Kafka doesn't monitor the messages each subscriber has read. Rather, it retains unread messages only, preserving all messages for a set amount of time. |
| Payload Size | No constraints | Default 1MB limit |
| Usage Cases | Simple use cases | Massive data/high throughput cases |

**Architecture -**




**1. Producer**

- **Definition**: A producer is any application or service that **publishes** data (messages) to Kafka.
- **Role**: Producers push data (messages) to topics in Kafka brokers. They decide to which topic and partition the messages should be sent.
- **Working**: Producers can choose to send data to a specific partition or allow Kafka to do it automatically based on some strategy (like round-robin or partition key).

**2. Consumer**

- **Definition**: A consumer is an application or service that **reads** data (messages) from Kafka topics.
- **Role**: Consumers subscribe to Kafka topics and read messages. They process the data for various tasks such as storing, analyzing, or forwarding it.
- **Working**: Consumers can pull data from Kafka brokers at their own pace. They are typically designed to be scalable and allow for parallel consumption.

**3. Consumer Group**

- **Definition**: A consumer group is a **group of consumers** that work together to consume messages from Kafka topics.
- **Role**: Kafka ensures that each message in a topic is consumed by only one consumer in a consumer group. This is how Kafka achieves **parallel processing** and **load balancing** for message consumption.
- **Working**: Each consumer in a group is responsible for consuming messages from specific partitions of the topic. If a new consumer joins the group, Kafka reassigns the partitions to balance the load. If a consumer leaves the group, Kafka redistributes the partitions among remaining consumers.

**4. Topic**

- **Definition**: A topic is a **category or feed name** to which messages are sent by producers and from which messages are consumed by consumers.
- **Role**: Topics help organize and structure the messages. A topic can have multiple partitions, and producers and consumers can interact with them based on the topic name.
- **Working**: Topics are logical channels that group similar messages. Kafka allows messages to be sent to multiple topics and read from them by different consumer groups.

**5. Partition**

- **Definition**: A partition is a **subdivision** of a Kafka topic. A topic can have one or more partitions to distribute data across multiple brokers.
- **Role**: Partitions allow Kafka to scale horizontally by enabling parallel processing and distributed storage. Each partition is an ordered, immutable sequence of messages.
- **Working**: Kafka divides each topic into multiple partitions to distribute the load and allow parallel consumption. Each partition is replicated across multiple brokers for fault tolerance and availability.

**6. Offset**

- **Definition**: The **offset** is a unique identifier for each message within a partition.
- **Role**: It is used to keep track of which message has been consumed by a consumer in a partition. Consumers use the offset to know the last message they've read.
- **Working**: Each message within a partition is assigned an offset. Kafka consumers can read messages starting from a specific offset or continue from their last read offset. Offsets are typically stored in Kafka or an external storage system.

**7. Broker**

- **Definition**: A Kafka **broker** is a server or node that is part of the Kafka cluster. It is responsible for storing, receiving, and sending messages.
- **Role**: Brokers manage the partitioned data and handle the storage, retrieval, and delivery of messages. A Kafka cluster consists of multiple brokers that collectively manage the storage and handling of Kafka topics.
- **Working**: Kafka brokers work together to provide scalability and fault tolerance. Each broker manages one or more partitions and their corresponding messages. Brokers are responsible for data replication and ensuring data availability.

**8. Cluster**

- **Definition**: A Kafka **cluster** is a group of Kafka brokers that work together to manage the distributed storage and delivery of messages.
- **Role**: A Kafka cluster allows for horizontal scaling, data redundancy, and fault tolerance. The cluster ensures that data is evenly distributed across brokers, and each broker can handle a portion of the total load.
- **Working**: A Kafka cluster typically consists of several brokers. Each broker manages partitions of topics and replicates partitions to ensure data availability in case of broker failures. The cluster coordination is managed by **Zookeeper**.

**9. Zookeeper**

- **Definition**: **Zookeeper** is a distributed coordination service used by Kafka to manage the metadata and state of the Kafka cluster.
- **Role**: Zookeeper coordinates brokers, tracks the health of the cluster, and helps in leader election for partitions. It is also responsible for managing consumer group offsets and metadata about topics, partitions, and brokers. **If consumer goes down** then it will hand over the remaining message offset to another consumer. It uses **committed offset** property value for the same.
- **Working**: Zookeeper helps ensure that the Kafka cluster is working as expected. It stores the cluster metadata and elects a partition leader (among brokers) for handling read and write operations. Although Kafka is moving towards a **KRaft** mode (Kafka Raft), Zookeeper was traditionally used for this coordination.

Concept of leaders and followers (replicas) -
A **replica** is a copy of a **partition** in Kafka. Each partition of a topic can have multiple replicas, which are stored on different **Kafka brokers**. These replicas are primarily used for **fault tolerance** and ensuring data availability in the event of broker failure. If the broker storing the leader replica fails, one of the follower replicas (non-leader replicas) can become the new leader. If a partition has a replication factor of 3, then three copies of the data exist on different brokers: one is the leader, and the other two are followers (replicas). All brokers storing replicas are called **replica brokers**. The **leader** is the replica of a partition that is responsible for **handling all read and write operations** for that partition.

**Key Points:**

- **Single Leader per Partition**: For each partition, only one replica is elected as the leader at any time, and this leader handles all the reads and writes for that partition.
- **Write Operations**: Producers send their messages to the leader replica of a partition. The leader then replicates the data to the follower replicas.
- **Read Operations**: Consumers read from the leader replica by default. This ensures they get the most up-to-date data.
- **Partition Leader Election**: If a leader replica fails (e.g., the broker hosting the leader goes down), one of the follower replicas is elected as the new leader, ensuring that the partition remains available.

**Leader Election:**

- If the current leader of a partition fails, Kafka triggers a **leader election** process to choose a new leader from the available follower replicas.
- The **ZooKeeper** (or **KRaft** in the newer versions of Kafka) helps manage the leader election process and ensures that only one leader exists for each partition at any given time.

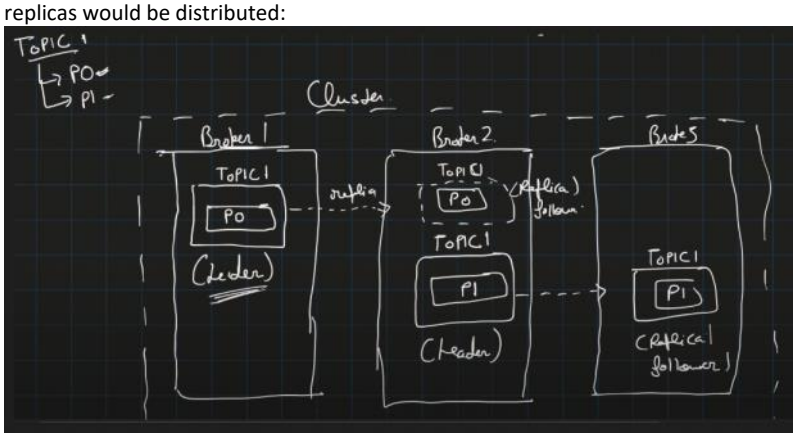
**How Replicas and Leaders Work Together**

In Kafka, the combination of **replicas** and **leader/follower** architecture ensures **high availability** and **fault tolerance**:

- Each Kafka **partition** has exactly one **leader** and potentially multiple **replicas** (followers).
- **Producers** send data to the **leader** of a partition.
- The **leader** then replicates the data to its **follower replicas**.
- If the **leader** fails, one of the **follower replicas** is promoted to be the new **leader**, and Kafka continues to operate without data loss (assuming the replicas are in sync and the replication factor is greater than 1).

**Example of Kafka's Leader and Replica Architecture**

Let's say we have a Kafka cluster with three brokers (B1, B2, and B3) and a topic T1 with a partition and a replication factor of 3. Here's how the leader and replicas would be distributed:

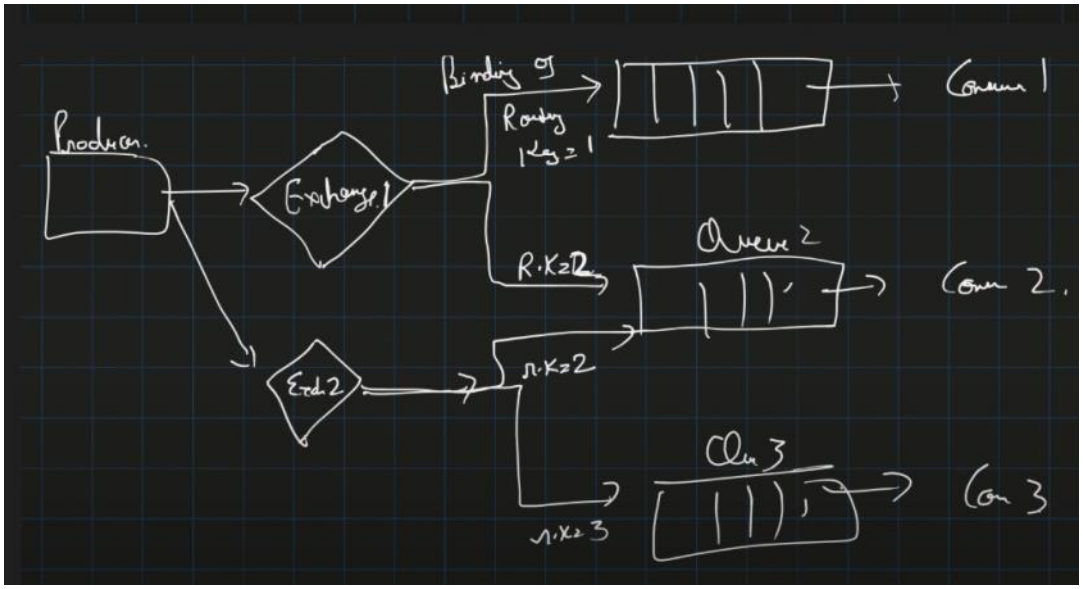



1. Partition P1 of Topic T1:
    - **Leader**: B1
    - **Replica 1**: B2 (Follower)
    - **Replica 2**: B3 (Follower)
2. If Broker B1 (which is the leader) fails, one of the followers (B2 or B3) would be promoted to be the new leader.
    - **New Leader**: B2 (or B3 depending on the leader election)
    - **Replica 1**: B3 (or B2, now the new follower)

This ensures that if a broker or partition leader fails, Kafka can still maintain data availability through the replicas.

What if consumer not able to process the message? (buggy message)
-> In this case consumer will retry till the retry limit reached. In this process committed offset will not move further. After retry limit is reached, consumer puts the buggy message to a **failure queue (dead letter queue or DLQ ).** Once, the buggy message has been put into DLQ, then committed offset will move forward and kafka sends next message to consume.

**RabbitMQ -**




Producer
Exchange
Routing key
Queue
Consumer

**How RabbitMQ Routing Works -**

1. The **Producer** sends a message to an **Exchange**.
2. The **Exchange** uses its routing rules (based on the exchange type and the routing key) to decide which **Queue(s)** should receive the message.
3. The **Queue(s)** holds the message until a **Consumer** retrieves it.
4. The **Consumer** processes the message and can acknowledge or reject it.

**Types of exchanges -**

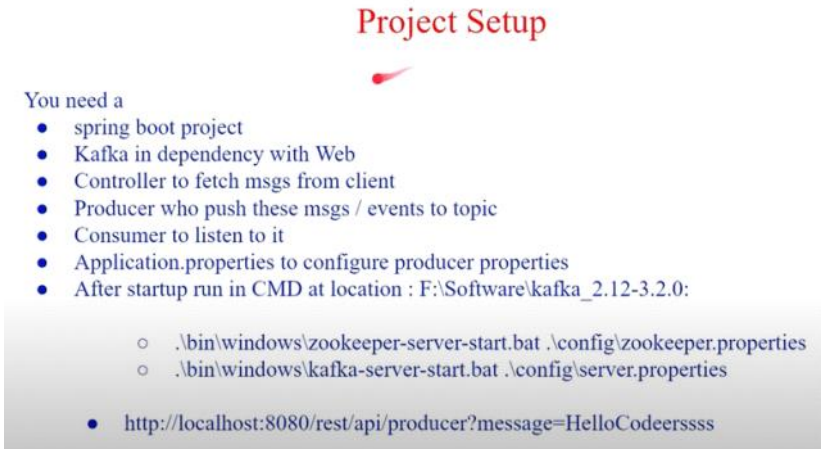
**1. Direct Exchange**

- **Routing mechanism**: A **direct exchange** routes messages to queues based on an exact match between the routing key and the queue binding key.
- **Use case**: Direct exchanges are used when messages need to be sent to specific queues based on a routing key.

**2. Fanout Exchange**

- **Routing mechanism**: A **fanout exchange** broadcasts messages to all queues bound to the exchange, ignoring the routing key. This means the message is delivered to all queues regardless of the routing key.
- **Use case**: Fanout exchanges are useful for scenarios where you want to broadcast messages to multiple consumers, such as logging, notifications, or event broadcasting.

**Kafka implementation-**




Controller -

```java
@RestController
@RequestMapping("/rest/api")
public class FetchMessageFromClient {

        @Autowired
    KafkaProducer kafkaProducer;


    @GetMapping(value = "/producer")
    public String sendMessage(@RequestParam("message") String message)
    {
        kafkaProducer.sendMessageToTopic(message);
        return "Message sent Successfully to the your code decode topic ";
    }
```

Producer -

```java
@Service
public class KafkaProducer {

        @Autowired
        private KafkaTemplate<String, String> kafkaTemplate;

        public void sendMessageToTopic(String message) {
                kafkaTemplate.send("CodeDecodeTopic", message);
        }
```

Consumer (KafkaListener) -

```java
@Service
public class KafkaListner {

        @KafkaListener(topics = "CodeDecodeTopic", groupId = "codedecode-group")
        public void listenToCodeDecodeKafkaTopic(String messageReceived) {
                System.out.println("Message received is " + messageReceived);
        }
```

Use @KafkaListener to listen message by passing topic and topic_name -

Kafka configurations -

```yaml
spring:
  kafka:
    producer:
      bootstrap-servers: localhost:9092
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

After the project setup, download and run kafka locally. So our machine acts as kafka server (broker). Then start the application and hit the api to send the message. Once message is send, consumer will consume it. -

Start kafka service -

```
F:\Software\kafka_2.12-3.2.0>.\bin\windows\kafka-server-start.bat .\config\server.properties
```

Start zookeeper service -

```
F:\Software\kafka_2.12-3.2.0>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
[2022-07-27 19:00:41,473] INFO Reading configuration from: .\config\zookeeper.properties (org.apache.zo
```

Console before hitting api -

```
de_group-1, groupId=codeDecode_group] Successfully joined group with generation Generation{generationId=1, memberId='consum
de_group-1, groupId=codeDecode_group] Resetting the last seen epoch of partition codeDecode_Topic-0 to 0 since the associat
de_group-1, groupId=codeDecode_group] Finished assignment for group at generation 1: {consumer-codeDecode_group-1-e3e967d6-
de_group-1, groupId=codeDecode_group] Successfully synced group in generation Generation{generationId=1, memberId='consumer
de_group-1, groupId=codeDecode_group] Notifying assignor about the new Assignment(partitions=[codeDecode_Topic-0])
de_group-1, groupId=codeDecode_group] Adding newly assigned partitions: codeDecode_Topic-0
de_group-1, groupId=codeDecode_group] Found no committed offset for partition codeDecode_Topic-0
de_group-1, groupId=codeDecode_group] Found no committed offset for partition codeDecode_Topic-0
de_group-1, groupId=codeDecode_group] Resetting offset for partition codeDecode_Topic-0 to position FetchPosition{offs
```

Hit the api -

localhost:8080/rest/api/producerMsg?message="CongratsForFirstKafkaImpl"

Console after hitting the api -

```
8
9       @KafkaListener(topics = "codeDecode_Topic" , groupId = "codeDecode_group")
10      public void listenToTopic(String receivedMessage) {
11          System.out.println("The message received is " + receivedMessage);
12      }
13
14  }
15
```

Problems Javadoc Declaration Search Console Progress Debug Call Hierarchy

KafkaDemo - KafkaDemoApplication (1) [Spring Boot App]

```
        transaction.timeout.ms = 60000
        transactional.id = null
        value.serializer = class org.apache.kafka.common.serialization.StringSerial

2022-07-27 19:03:49.341  INFO 13088 --- [nio-8080-exec-2] o.a.k.clients.producer.Ka
2022-07-27 19:03:49.360  INFO 13088 --- [nio-8080-exec-2] o.a.kafka.common.utils.Ap
2022-07-27 19:03:49.360  INFO 13088 --- [nio-8080-exec-2] o.a.kafka.common.utils.Ap
2022-07-27 19:03:49.360  INFO 13088 --- [nio-8080-exec-2] o.a.kafka.common.utils.Ap
2022-07-27 19:03:49.368  INFO 13088 --- [ad | producer-1] org.apache.kafka.clients.
2022-07-27 19:03:49.369  INFO 13088 --- [ad | producer-1] org.apache.kafka.clients.
2022-07-27 19:03:49.402  INFO 13088 --- [ad | producer-1] o.a.k.c.p.internals.Trans
The message received is "CongratsForFirstKafkaImpl"
```