**Spring-Boot**

* **Introduction:**

Spring Boot is a Frame work For building applications in the java programming language.

Spring Boot makes it easy to create stand-alone ,production-grade Spring based Application that you can just run.

The core Spring Framework already reduces Boilerplate code and provides a lot of helpful features for Java Application

However, Spring Boot takes this convenience to the next level by focusing specifically on reducing the effort required to set up and

configure a Spring application.

While Spring does simplify many tasks like creating web applications, working with databases, managing transactions and more.

But setting up a Spring project can still involve quite a bit of manual configuration. This is where Spring Boot steps in.

It Provides 2 things :-

**Auto-Configuration.**

**Standalone Applications.**

The @SpringBootApplication annotation alone brings in a lot of pre-configured features, including automatic component scanning and embedded server configuration, which would have required more steps in traditional Spring setup.

* **Core Concepts of Spring-Boot**
* Object ke creation ko Spring-Boot ko sambhala dena
* Giving the control to Spring-Boot for Creating Object for class or Bean
* Spring Provide Inversion of Control for creating objects
* Application-Context is way to Implement IOC Container
* @Component

public class MyComponent {

// Class is automatically registered as a Spring bean.

}

* @Component annotation se who class ko IOC container mai daal deta hai
* @SpringBootApplication Does this 3 things :-
  + @Configuration (Configuration provide karega)
  + @EnableAutoConfiguration
  + @ComponentScan similar to @Component
* @Autowired
* **Rest Api’s**

A **REST API (Representational State Transfer Application Programming Interface)** is a way for different software applications to communicate with each other over the internet. It allows you to access and manipulate data or services in a standardized way using simple HTTP methods like GET, POST, PUT, and DELETE.

**Breaking it Down in Simple Terms:**

1. **What is an API?**  
   An API is like a waiter in a restaurant. You (the client) make a request (order food), and the waiter (API) brings the requested food (data/service) from the kitchen (server).
2. **What is REST?**  
   REST is a set of rules or principles that makes APIs easier to use, flexible, and scalable. It ensures communication is done in a standardized way, typically using HTTP.

**Key Concepts of a REST API:**

1. **Resources**  
   In REST, everything is treated as a resource. For example:
   * A user in a system
   * A product in an e-commerce app
   * A blog post on a website  
     Each resource is identified by a unique URL (Uniform Resource Locator), like:  
     <https://example.com/api/users/123>
2. **HTTP Methods**  
   REST APIs use standard HTTP methods to interact with resources:
   * **GET**: Fetches data (e.g., get a list of users or a specific user).
   * **POST**: Creates new data (e.g., add a new user).
   * **PUT**: Updates existing data (e.g., update user details).
   * **DELETE**: Deletes data (e.g., remove a user).
3. **Statelessness**  
   Every request from the client to the server must contain all the information the server needs to process it. The server does not remember previous requests.
4. **Representation**  
   Resources are usually represented in formats like **JSON** or **XML**, which are easy for both humans and machines to read. Example of JSON data for a user:
5. **Uniform Interface**  
   REST APIs follow a consistent structure, making them predictable and easier to work with.

**Benefits of REST API:**

1. Simple and Standardized: It uses HTTP, which is widely understood.
2. Scalable: Can handle a large number of requests efficiently.
3. Flexible: Supports multiple data formats like JSON and XML.
4. Platform-Independent: Any programming language or device that can make HTTP requests can use it.

**Use Cases:**

* Retrieving data from a database (e.g., fetching a list of products).
* Sending data to a server (e.g., creating a new account).
* Updating existing records (e.g., editing your profile).
* Deleting records (e.g., removing an old blog post).

**Summary:**

A REST API is like a messenger between clients (apps, websites) and servers (databases, services), helping them communicate using standard web protocols. It’s widely used in web development because it’s simple, efficient, and easy to implement.

* **HTTP Verbs**

HTTP verbs (also called HTTP methods) are like actions you can take when communicating with a server over the web. They tell the server **what you want to do** with a resource (like data or files).

Here’s an overview of the main HTTP verbs in simple terms:

**1. GET**

* **What it does**: Fetches data from the server.
* **Example**: Imagine you’re looking at a menu in a restaurant. You’re just asking to see the options (not changing anything).
* **Use case**: Get a list of users, details about a product, or a weather report.

**2. POST**

* **What it does**: Creates a new resource on the server.
* **Example**: Think of filling out a form to create an account—you’re adding new information to the system.
* **Use case**: Create a new user, add a new blog post, or submit an order.

**3. PUT**

* **What it does**: Updates an existing resource on the server or creates it if it doesn’t exist.
* **Example**: Changing your profile picture or updating your address—modifying something that’s already there.
* **Use case**: Edit a user’s information, update a product’s price, or change settings.

**4. DELETE**

* **What it does**: Removes a resource from the server.
* **Example**: Think of deleting an email or removing a photo from your gallery.
* **Use case**: Delete a user account, remove a product, or cancel an order.

**5. PATCH**

* **What it does**: Partially updates an existing resource.
* **Example**: Updating just your email address without touching the rest of your profile.
* **Use case**: Change a single field, like a username or status, without modifying the entire resource.

**6. OPTIONS**

* **What it does**: Checks what actions (HTTP methods) the server allows on a resource.
* **Example**: Asking, “What can I do here?” before interacting with the resource.
* **Use case**: Verify available methods like GET, POST, or DELETE for a specific resource.

**7. HEAD**

* **What it does**: Similar to GET, but only retrieves headers (no body).
* **Example**: Checking if a resource exists or its size before downloading it.
* **Use case**: Validate links or get metadata about a resource.

**Summary Table:**

| **HTTP Verb** | **Purpose** | **Example Action** |
| --- | --- | --- |
| GET | Fetches data | View a user profile |
| POST | Creates new data | Sign up for a new account |
| PUT | Updates or creates a resource | Update account details |
| DELETE | Deletes a resource | Remove a product from the catalog |
| PATCH | Partially updates a resource | Change your email address |
| OPTIONS | Checks supported methods | Ask “What can I do here?” |
| HEAD | Fetches headers only | Check if a file exists |

**In Summary:**

HTTP verbs are like commands you give to the server. They specify what action to take (retrieve, add, update, or delete) when interacting with resources. This simple, standardized system makes it easy for clients (like web browsers or mobile apps) and servers to communicate.

@RestController:-

**@ResquestBody** :- It’s like saying hey Spring please take the data from the request and turn it into a java object that I can use in my code.

The @RequestBody annotation is used to **map the HTTP request body** to a Java object. It tells Spring to take the data sent in the body of an HTTP request (usually in JSON or XML format) and convert it into a Java object that you can work with in your code.

**@PathVariable**

* **What it does**: It extracts a value from the **URL path**.
* **When to use**: Use it when the value is part of the URL path itself.

**@RequestParam**

* **What it does**: It extracts a value from the **query string** in the URL.
* **When to use**: Use it when the value is passed as a **key-value pair** in the URL’s query string.

**Differences in Simple Terms:**

| **Aspect** | **@PathVariable** | **@RequestParam** |
| --- | --- | --- |
| **Where it comes from** | URL path itself (/users/{id}) | Query string (?key=value) |
| **Example URL** | /users/123 | /users?name=John |
| **Use case** | Identifying a specific resource | Filtering or modifying results |

* **Response Entity:-**

The Response Entity class is part of the Spring Framework and is

commonly used in Spring Boot applications to customize the HTTP

response.

It provides methods for setting the response status, headers, and body. You

can use it to return different types of data in your controller methods, such as JSON, XML, or even HTML.

**Lombok:-**

(dependency code from project Lombok site)

* It aims to reduce the boilerplate code that developers have to write,

such as getters, setters, constructors, and more.

* Lombok achieves this by generating this code automatically during compilation based on annotations you add to your Java classes.
* Lombok generates bytecode for methods like getters,setters, constructors, equals(), hashCode(), and toString(), as specified by the annotations used in your code. This generated code is added to the compiled class files (.class files).
* The Java Compiler compiles your classes, including the generated code. This means that the generated methods become part of your compiled class files.
* When you run your application, the generated methods are available for use, just like any other methods in your classes.

Lombok is a Java library that helps reduce boilerplate code, making your code cleaner and easier to read. It automatically generates commonly used methods like getters, setters, constructors, and more at compile time, so you don’t have to write them manually.

**Key Lombok Annotations:**

1. **@Getter and @Setter**
   * Automatically generates getter and setter methods for your class fields.
2. **@ToString**

* Generates a toString() method for your class, including all fields.

1. **@NoArgsConstructor, @AllArgsConstructor and @RequiredArgsConstructor**

Automatically generates constructors:

* + @NoArgsConstructor: No-argument constructor.
  + @AllArgsConstructor: Constructor for all fields.
  + @RequiredArgsConstructor: Constructor for fields marked as final or @NonNull.

1. **@Data**

A shortcut for @Getter, @Setter, @ToString, @EqualsAndHashCode, and @RequiredArgsConstructor.

1. **@Builder**

 @Builder is like a helper tool that lets you create objects easily without writing multiple lines of code.

 Imagine you want to create an object with many properties, like a User with a name, email, and age. Normally, you'd need to call a long constructor or write a lot of set() methods.

**Benefits:**

* Reduces repetitive code.
* Makes the code more readable.
* Useful for large projects to simplify models or DTOs.

Lombok requires adding it to your project (e.g., in Maven or Gradle) and installing the Lombok plugin in your IDE.

**Application Properties in Spring-Boot Resources Folder**

**@Transactional**

**@EnableTransactionManagement :-**  Used on main Class

For Transaction to perform smoothly we need to use MongoDB Atlas.

* **Spring Security**

Spring Security is a powerful and highly customizable security framework that is often used in Spring Boot applications to handle authentication and authorization.

**Authentication**

* The Process Of verifying a user's identity
* (e.g., username and password)

**Authorization**

* The process of granting or denying access to specific resources or actions based on the authenticated user's roles and permissions.
* Once the dependency is added, Spring Boot's auto-configuration feature will automatically apply security to the application.
* By default, Spring Security Uses HTTP Basic Authentication.
* The client sends an Authorization header Authorization: Basic <encoded-string> The server decodes the string, extracts the username and password, and verifies them. If they're correct, access is granted. Otherwise, an "Unauthorized" response is sent back.
* Encoding Credentials are combined into a string like username:password which is then encoded using Base64.
* Spring Security will generate a default user with a random password that is printed in the console logs on startup.
* You can also configure username & password in your application.properties file.

****

**@EnableWebSecurity:-**  This annotation signals Spring to enable its web security support. This is what makes your application secured. It's used in conjunction with @Configuration.

**SecurityFilterChain**

The securityFilterChain method defines the security configuration using the HttpSecurity object. It replaces the old WebSecurityConfigurerAdapter way of configuring security.

**Key Points:**

* **@Bean Annotation:** The method is annotated with @Bean, meaning it will be managed by the Spring container and is used to define a SecurityFilterChain bean for configuring Spring Security.
*  **requestMatchers("/public/\*\*").permitAll():** All requests to URLs under /public/\*\* are allowed without authentication.
* **requestMatchers("/journal/\*\*", "/user/\*\*").authenticated():** Requests to /journal/\*\* and /user/\*\* require the user to be authenticated (logged in).
* **requestMatchers("/admin/\*\*").hasRole("ADMIN"):** Requests to /admin/\*\* require the user to have the **role "ADMIN"**.
* **Note:** The role check internally checks for ROLE\_ADMIN, so your user roles need to be prefixed with ROLE\_.
*  **anyRequest().authenticated():** Any other request (not matched by the above rules) requires authentication.
* **httpBasic(Customizer.withDefaults()):**Enables HTTP Basic Authentication, which allows clients to authenticate using their username and password in the HTTP request headers.
* **csrf(AbstractHttpConfigurer::disable):** Disables **Cross-Site Request Forgery (CSRF) protection**. This is typically done when you are working with APIs or non-browser clients, but it is not recommended for production environments unless you fully understand the risks.
* **build():**This finalizes and builds the security configuration.

**ConfigurGlobal Method**

This method configures authentication by integrating the custom UserDetailsService and password encoder.

**Key Points:**

* **@Autowired Annotation:** Allows Spring to inject dependencies automatically (like AuthenticationManagerBuilder) when the method is called.
* **Authentication Configuration:**

auth.userDetailsService(/\*userDetailsS\*/).passwordEncoder(passwordEncoder());

* + **userDetailsService**: This is used to specify your custom UserDetailsService implementation, which is responsible for loading user details (username, password, roles) from a database or other source. Replace /\*userDetailsS\*/ with an instance of your UserDetailsService bean.
  + **passwordEncoder(passwordEncoder()):** Ensures that passwords are encoded and matched securely. The encoder used is defined in the passwordEncoder() method below.

**PasswordEncoder Bean**

Defines a PasswordEncoder bean for encoding passwords.

* **BCrypt Password Encoding:**

return new BCryptPasswordEncoder();

* + **BCrypt** is a widely used and secure algorithm for hashing passwords.
  + It automatically handles salting (adding randomness to hashes) and is computationally expensive, making it resistant to brute force attacks.
* **How It’s Used:** The BCryptPasswordEncoder will encode raw passwords when saving them (e.g., during user registration) and match encoded passwords during login authentication.

**How It All Works Together**

1. **Request Authorization:**
   * When a request comes in, the SecurityFilterChain evaluates the request against the authorization rules (/public/\*\*, /journal/\*\*, etc.).
   * Depending on the URL, the user is either allowed, required to authenticate, or checked for specific roles.
2. **Authentication:**
   * The configureGlobal method integrates your custom UserDetailsService and the BCryptPasswordEncoder.
   * When a user logs in, their credentials are validated against the data returned by the UserDetailsService.
3. **Password Encoding:**
   * Passwords are stored in an encoded format using BCryptPasswordEncoder.
   * When a user logs in, their raw password is encoded and compared with the stored encoded password.

**Example Flow**

1. **Accessing /public/\*\*:**
   * No authentication is needed. Anyone can access it.
2. **Accessing /journal/\*\* or /user/\*\*:**
   * The user must be authenticated (logged in).
3. **Accessing /admin/\*\*:**
   * The user must have the role ADMIN (encoded as ROLE\_ADMIN).
4. **Making Any Other Request:**
   * The user must be authenticated.

**In Simple Terms**

**SecurityFilterChain :-** This is where you set up the security rules for your application.

1. **Purpose:** It tells your application what kind of access users have for specific URLs.
2. **How It Works:**
   * **@Bean:** This marks the method as a Spring-managed component.
   * **Authorization Rules:**
     + \*\*/public/\*\*: Anyone can access these URLs without logging in.
     + \*\*/journal/\*\* and /user/\*\*: Only logged-in users can access these.
     + \*\*/admin/\*\*: Only logged-in users with the "ADMIN" role can access these.
     + **Other URLs:** If not explicitly mentioned, users must be logged in to access them.
   * **httpBasic:** Enables basic authentication (users provide username and password).
   * **csrf.disable():** Turns off CSRF protection, which is helpful for APIs but risky if used without caution.
   * **build():** Finalizes the configuration.

**configureGlobal Method**

This sets up how users are authenticated (i.e., how the app checks their username, password, and roles).

1. **Purpose:**
   * Connects your app to the user database or service where user details are stored.
   * Ensures passwords are securely stored and compared.
2. **Key Points:**
   * **@Autowired:** Automatically gives you access to an AuthenticationManagerBuilder to configure authentication.
   * **userDetailsService:** A custom service that tells Spring Security where to find user information (e.g., database or memory).
   * **passwordEncoder:** Ensures passwords are stored securely by encoding them (e.g., using hashing).

**PasswordEncoder Bean**

This sets up how passwords are securely stored and checked.

1. **BCrypt:**
   * A strong algorithm for encoding passwords.
   * Adds randomness (called "salting") and is slow enough to resist brute force attacks.
2. **How It Works:**
   * When a user registers, their password is encoded using BCrypt before saving it.
   * When a user logs in, the app compares their entered password (after encoding) with the stored one.

**Summary**

* **SecurityFilterChain:** Defines who can access which URLs and sets up authentication.
* **configureGlobal:** Connects Spring Security to your user database and sets up password encoding.
* **PasswordEncoder:** Ensures passwords are stored securely.

This setup secures your app by controlling access and protecting user credentials. Let me know if you need examples or further clarifications!

When you log in with Spring Security, it manages your authentication across multiple requests, despite HTTP being stateless.

1 **Session Creation:** After successful authentication, an HTTP session is formed. Your authentication details are stored in this session.

2. **Session Cookie:** A JSESSIONID cookie is sent to our browser which ets sent back with subsequent requests, helping the server recognize your session.

3.**SecurityContext:** Using the JSESSIONID, Spring Security fetches your authentication details for each request.

4 **Session Timeout:** Sessions have a limited life. If you're inactive past this limit, you're logged out.

5. **Logout:** When logging out, your session ends, and the related cookie is removed.

6.**Remember-Me:** Spring Security can remember you even after the session ends using a different persistent cookie (typically have a longer lifespan) .

In essence, Spring Security leverages sessions and cookies, mainly JSESSIONID, to ensure you remain authenticated across requests.

**Properties , YAML**

* **Class-Path:-** Class path is the list of jars (Java Archives) and Directories ,that is used by JVM.
* **Configurations Files :**- application.properties , YAML , using Command line Arguments
* **Application.Properties** :- stores in Key = Pairs.
* **YAML** :- extension is yml, yaml is aim;t an Markup Language
* **Priorities**: .properties files have more priority than .yml files (In case we have written the same property in both the files). But the first priority is given to command line arguments.

Command line arguments > application.properties > yml files

* **Testing in Spring Boot**

**Junit {Java Unit}:-**

* **Dependency**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
</dependency>

* **@Test :-**
* **@Disabled**
  + This annotation comes from JUnit 5 (junit.jupiter.api) and is used to disable a test method or class temporarily.
  + In the context shown (// @Disabled), it's commented out, which means it currently has no effect. Removing the comment would prevent the annotated test from being executed.
* **@ParameterizedTest**
  + Used to run the same test multiple times with different input values.
  + This approach allows for better test coverage without redundant test code.
* **@ValueSource(strings = {"admin", "ahmed", "amaan"})**
  + Provides a collection of string values (admin, ahmed, and amaan) for the parameterized test.
  + The test method testFindByUserName(String name) is executed separately for each of these strings.
  + Example: Verifying that userRepository.findByUserName(name) does not return null.
* **@ArgumentsSource(UserArgumentsProvider.class)**
  + Specifies a custom source (UserArgumentsProvider) for supplying arguments to the parameterized test testSaveNewUser(User user).
  + UserArgumentsProvider is typically a class that implements ArgumentsProvider and provides specific user data for the test.
* **assertTrue(userService.saveNewUser(user))**
  + Verifies that the result of the saveNewUser operation is true.
  + This implies the user was successfully saved by the userService.
* **@CsvSource({ "1,1,2", "2,10,12", "3,6,9" })**
  + Provides comma-separated values for multiple test cases.
  + The test method test(int a, int b, int expected) runs with three different sets of values.
    - First test: 1 + 1 is checked against 2.
    - Second test: 2 + 10 is checked against 12.
    - Third test: 3 + 6 is checked against 9.
* **assertEquals(expected, a + b)**
  + Ensures that the sum of a and b equals expected.
  + If not, the test will fail.
* **@SpringBootTest**
  + This annotation is part of the Spring Boot testing framework.
  + It allows tests to load the complete Spring application context, ensuring that all beans are initialized and available for testing.
* **@BeforeAll**: Runs **once before all tests**. Use it for tasks like setting up database connections or loading configurations.
* **@BeforeEach**: Runs **before each test**. Great for initializing or resetting test-specific data.
* **@AfterEach**: Runs **after each test**. Used for cleanup tasks, like closing files or resetting variables.
* **@AfterAll**: Runs **once after all tests**. Ideal for tasks like closing database connections or freeing resources.
* **Summary Table**

| **Annotation** | **Runs When** | **Method Requirements** |
| --- | --- | --- |
| @BeforeAll | Before all tests | static method |
| @BeforeEach | Before each test | Instance method |
| @AfterEach | After each test | Instance method |
| @AfterAll | After all tests | static method |

* These annotations help structure and maintain test cases by organizing setup and teardown processes efficiently.

This setup showcases structured, parameterized testing using JUnit 5 with Spring Boot for backend testing, promoting reusable and maintainable test cases.

* **Mockito**

**Mockito** is a tool that helps you test your Java programs by creating fake versions of the parts your code depends on, like databases, web services, or other classes.

**Imagine This Scenario:**

You have an app that needs to check a bank account balance. Normally, your code would contact a real bank system to get the balance. But during testing, that's slow, unnecessary, and maybe even impossible if the bank server is down.

**What Mockito Does:**

Mockito acts as a stand-in for the bank system. It pretends to be the real thing and gives you whatever answer you tell it to give, like "The balance is $500." Now, you can check if your code handles that balance correctly without ever contacting a real bank.

**Key Benefits:**

* **Isolation:** Test one part of your code without relying on other parts.
* **Speed:** Tests run faster since there are no real database or API calls.
* **Control:** You can simulate different situations like errors or specific data.
* **Verification:** Check if certain functions were called or not during testing.

In simple words, Mockito is like a friendly actor who pretends to be someone else so you can practice without needing the real person.

**Key Annotations**

* @Mock: Creates a fake object.
* @InjectMocks: Injects mocked objects into your class.
* when(...).thenReturn(...): Defines the behavior of the mock.
* verify(...): Ensures methods were called.

Mockito simplifies testing by letting you focus on the logic without worrying about the external environment.

* **Profiles**

In Spring Boot, **profiles** are a way to separate different configurations for different environments, such as **development, testing, staging, or production**. Profiles allow you to switch between configurations easily without manually changing your application properties or logic.

**How Profiles Work:**

Profiles can be used to load specific **beans, configurations, or properties files** based on the active environment.

**Defining Profiles:**

1. **Using Properties Files:**
   * Create separate property files for each environment:
     + application-dev.properties
     + application-test.properties
     + application-prod.properties

In Spring Boot, **profiles** help manage different configurations for various environments, like development, testing, and production. They allow you to create environment-specific settings without manually changing code each time.

**Key Points:**

* **Purpose:** Profiles help load different configurations for different environments, such as connecting to a local database in development and a secure cloud database in production.
* **How It Works:** You define configurations for each environment (like database URLs or server ports) and specify which profile to activate based on your deployment needs.
* **Activation:** Profiles can be enabled through:
  + Property files (application.properties)
  + Command-line arguments
  + Environment variables
* **Benefits:**
  + Simplifies environment management
  + Reduces errors from manual changes
  + Makes applications more flexible and scalable

In essence, Spring Boot profiles are a powerful tool to create adaptable, environment-aware applications with minimal effort.

**Jenkis**

**Jenkins** is an open-source **automation server** used for **Continuous Integration (CI) and Continuous Delivery (CD)** in software development. It helps developers build, test, and deploy applications efficiently by automating repetitive tasks.

**Key Features:**

* **Continuous Integration:** Automatically integrates code changes from multiple developers, builds the application, and runs tests.
* **Continuous Delivery:** Automates the deployment of applications to different environments.
* **Extensibility:** Supports a wide range of plugins for various tools and platforms.
* **Distributed Builds:** Allows tasks to be executed on multiple machines for faster builds.

**How Jenkins Works:**

1. **Code Integration:** Jenkins monitors a version control system (like GitHub).
2. **Trigger:** A build is triggered when code changes are detected.
3. **Build Automation:** Jenkins compiles the code and runs tests.
4. **Deployment:** If tests pass, Jenkins can deploy the application to a staging or production environment.

**Benefits:**

* Saves time by automating build, test, and deployment processes
* Improves code quality through early and frequent testing
* Helps teams maintain a streamlined development workflow

**Typical Use Cases:**

* Automating build pipelines for Java, Python, or Node.js projects
* Deploying applications on cloud services
* Running automated test suites after every code push

Jenkins is widely used in DevOps environments, helping teams deliver software faster and more reliably.

* **Logging**

logging is an essential aspect of application development that allows developers to monitor and troubleshoot their applications.

Spring Boot supports various logging frameworks, such as Logback, Log4j2, and Java Util Logging (JUL)

**Logback**: A popular logging framework that serves as the default in many Spring Boot applications. It offers a flexible configuration and good performance. Genrally we use this

**Log4j2**: Another widely used logging framework with features such as asynchronous logging and support for various output formats.

**Java Util Logging (JUL)**: The default logging framework included in the Java Standard Edition. While it's less feature-rich than some third-party frameworks, it is straightforward and is part of the Java platform.

Spring Boot comes with a default logging configuration that uses Logback as the default logging implementation. It provides a good balance between simplicity and flexibility.

The default configuration is embedded within the Spring Boot libraries, and it may not be visible in your project's source code.

If you want to customize the logging configuration, you can create your own logback- spring.xml or logback.xml file in the src/main/resources directory. When Spring Boot detects this file in your project, it will use it instead of the default configuration.

Logging levels help In categorizing log statements based on their severity. The

common logging levels are

**TRACE**

**DEBUG**

**INFO**

**WARN**

**ERROR**

We can set the desired logging level for specific packages or classes, allowing them to control the amount of information logged at runtime.

By default, logging is enabled for

**INFO**

**WARN**

**ERROR**

**Spring Boot provides annotations like**

**@Slf4j & @Log4j2**

**that you can use to automatically inject**

**logger instances into your classes**

The @Slf4j annotation is part of the **Lombok** library and is used to automatically create a **Logger** instance (private static final Logger) for your class. This helps reduce boilerplate code by eliminating the need to manually declare and initialize a logger. The logger is compatible with the **SLF4J** logging facade.

Spring Boot allows us to configure logging using properties or YAML file



**Key Elements Explained:**

**1. Root Element: <configuration>**

* The top-level element of the Logback configuration file.
* Contains the entire configuration for appenders, encoders, and log levels.

**2. Console Appender (myConsoleAppender)**

* **Purpose**: Outputs logs to the console (standard output).
* **Attributes**:
* name: Identifies the appender as "myConsoleAppender."
* class: Specifies the type of appender (in this case, ConsoleAppender).
* **Encoder**:
* Defines how logs are formatted.
* %d{HH:mm:ss.SSS}: Date format with hours, minutes, seconds, and milliseconds.
* %thread: Name of the thread logging the event.
* %-5level: Log level padded to 5 characters (e.g., INFO, DEBUG).
* %logger{36}: Logger name truncated to 36 characters.
* %msg: Log message.
* %n: New line.

**3. File Appender (myFileAppender)**

* **Purpose**: Writes logs to a file.
* **Attributes**:
  + name: Identifies the appender as "myFileAppender."
  + class: Specifies the appender type (FileAppender).
* **Encoder**: Same formatting pattern as myConsoleAppender.
* **File Path**: Specifies journalApp.log as the output file for logs. You can use an absolute path or place this file in the application's directory.

**4. File Appender (myFileAppender)**

* **Purpose**: Configures the default logging level and appender references.
* **Attributes**:
  + level="INFO": Sets the default logging level to INFO. This means logs of level INFO and above (WARN, ERROR) will be recorded.
* **Appender References (appender-ref)**:
  + Connects the root logger to the specified appenders (myConsoleAppender, myFileAppender).

The rolling.RollingFileAppender in Logback is used to manage logging files by creating new files when certain conditions are met, such as when the file reaches a specific size or at scheduled intervals (like daily or weekly). It helps avoid having a single huge log file by "rolling" or archiving old logs and keeping the log files organized and manageable.

* **Definition**

Logging in Spring Boot refers to the process of capturing and recording information about the application's execution. This information helps in debugging, monitoring, and troubleshooting applications. Spring Boot supports several logging frameworks but uses **SLF4J** with **Logback** as the default logging implementation.

**Key Concepts of Logging in Spring Boot:**

1. **Default Logging Framework**:
   * By default, Spring Boot uses **SLF4J** with **Logback**.
   * It supports popular frameworks like **Log4j2**, **Java Util Logging (JUL)**, and others.
2. **Log Levels**:
   * TRACE: Most detailed information, usually only for debugging purposes.
   * DEBUG: Less detailed information, helpful for developers during debugging.
   * INFO: General information about application execution.
   * WARN: Indicates a potential problem.
   * ERROR: Indicates a significant problem.
3. **Configuration**:
   * Default configuration logs to the console.
   * You can customize logging by adding configuration files:
     + application.properties
     + logback-spring.xml
     + log4j2.properties

* **SonarCube**

Download the SonarCube from the given link :-

<https://www.sonarsource.com/products/sonarqube/downloads/historical-downloads/>

I am using latest version, the below code is for cmd

Steps For Creating Project in Sonar Cube

* go to sonarQube (localhost://9000)
* Create Project locally the give name of your Project and Customize the Project key name
* Generate the token and copy code of Your desired Tools or language , I selected Maven
* Then You will Get the Below Generated Code

.\mvnw clean verify sonar:sonar ^

More? -Dsonar.projectKey=Your-Project-Key ^

More? -Dsonar.projectName="Project Name" ^

More? -Dsonar.host.url="http://localhost:9000" ^

More? -Dsonar.token="Your generated Token "

Use the upper command for installing sonar in ide if you have setup the password of Sonar Qube

* **External Api Integration**

We will use two API’s

* Weather Api: <https://openweathermap.org/api> paid
* Quotes Api: <https://theysaidso.com/> paid
* Weather Stack: <https://weatherstack.com/> freee

Aaaa

<http://api.weatherstack.com/current?access_key=b9f8510c95accc2f821cd12b8e303236&Query=CityName>

**Eleven Labs**:- uses AI to create human-like voices for a variety of purposes, including videos, audiobooks, gaming, and chatbots

* **@Components VS @Service**

@Component is a generic stereotype annotation for any Spring-managed bean. It tells Spring to detect and register the class as a bean.

@Service is a specialization of @Component, used specifically for service layer classes. It makes the code more readable and indicates that the class contains business logic.

Functionally, both are the same, but @Service is preferred for service classes to follow best practices.

* **@Value**

The @Value annotation in Spring Boot is used to inject values into fields from:  
✅ **Application properties (application.properties or application.yml)**  
✅ **Environment variables**  
✅ **Default values**  
✅ **Expressions (SpEL - Spring Expression Language)**

**In Short**

* @Value helps inject property values, system variables, and defaults.
* It makes configurations flexible and avoids hardcoding values in Java code.  
  🚀 **Useful for externalizing configurations in Spring Boot!**
* **@PostConstruct**

@PostConstruct is a **lifecycle callback annotation** in Spring. It runs a method **automatically** after the bean is created and dependencies are injected.

**How It Works?**

* When Spring initializes a bean, it injects dependencies first.
* After that, it runs the method annotated with @PostConstruct.
* This is useful for **initial setup** tasks like loading data, opening connections, or setting default values.

**When to Use @PostConstruct?**

✅ When you need to **execute code after the bean is ready**  
✅ Loading **default data** from a database  
✅ Setting up **caches or configurations**  
✅ Opening **connections**

**Key Points**

* Runs **only once** after bean initialization.
* Works with **Spring-managed beans** (annotated with @Component, @Service, etc.).
* It’s part of **Jakarta EE** (jakarta.annotation.PostConstruct), so ensure you have the dependency.

🚀 **In short:** @PostConstruct helps you run setup logic after a Spring bean is created and ready!

* **Criteria**
* **@Query** → Used when you already know what the query will be. You write it once, and it stays the same.
* **Criteria API** → Used when the query needs to change based on user input, like filtering data dynamically.

✅ **Use @Query** for simple, fixed queries.  
✅ **Use Criteria API** when you need flexible, dynamic queries. 🚀

* **MongoTemplate**

MongoTemplate is a **Spring Data MongoDB** tool used to interact with MongoDB **without using repositories**.

✅ **Why use it?**

* Gives **more control** over MongoDB operations.
* Helps with **custom queries**, **aggregations**, and **complex updates**.

✅ **When to use it?**

* When **Spring Data Repositories** (MongoRepository) are not flexible enough.
* When you need **fine-tuned** queries and **custom operations**.

🚀 **In short:** MongoTemplate is a powerful alternative to MongoRepository for more complex MongoDB interactions.

* **JMS {Java Mail Services}**

Java Mail Services (previously known as **JavaMail API**, now **Jakarta Mail API**) is a **Java library used for sending and receiving emails** via SMTP, POP3, and IMAP protocols.

It allows Java applications to **send emails programmatically** using SMTP (Simple Mail Transfer Protocol) and supports features like:

* Sending plain text & HTML emails
* Adding attachments
* Sending emails with CC and BCC
* Handling authentication and encryption (TLS/SSL)
* Receiving emails via IMAP or POP3

**How to Use Java Mail in Spring Boot**

Spring Boot provides the JavaMailSender interface, which makes it easier to send emails.

1. **Add Java Mail Dependency**

If using Maven, add:

<dependency>

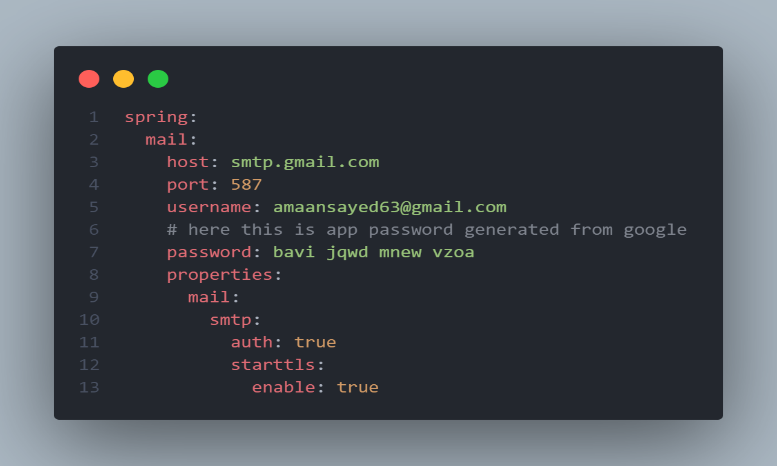
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-mail</artifactId>

</dependency>

1. **Configure SMTP in application.yml**

If using Gmail, add:



* **Cron & Scheduling**

Spring Boot provides a built-in scheduling feature that allows you to automate tasks at specific intervals or times using the @Scheduled annotation. It supports **fixed-rate scheduling, fixed-delay scheduling, and cron expressions**. Use Cron Maker Using Websites

**1. Enabling Scheduling in Spring Boot**

To use scheduling, you must enable it in your Spring Boot application by adding @EnableScheduling. This tells Spring to look for methods annotated with @Scheduled and execute them accordingly.

**2. Scheduling Methods in Spring Boot**

**A. Fixed Rate Scheduling**

* Executes the task at a fixed interval, irrespective of when the previous execution finished.
* Example: If the interval is **5 seconds**, the method will run **every 5 seconds**, even if the previous execution is still running.

**B. Fixed Delay Scheduling**

* Waits for the previous execution to complete and then starts the next execution after a specified delay.
* Example: If the delay is **5 seconds**, the method will execute **5 seconds after the last execution finished**.

**C. Initial Delay Scheduling**

* Starts the first execution after a specified delay, then continues execution at a fixed rate or fixed delay.
* Example: If the initial delay is **2 seconds** and the fixed rate is **5 seconds**, the first execution will start **after 2 seconds**, then continue every **5 seconds**.

**3. Cron-Based Scheduling**

Cron expressions allow you to schedule tasks with precise control over execution times. The cron format consists of six fields:

second minute hour day-of-month month day-of-week

For example:

* "0 0 12 \* \* ?" → Runs **every day at 12 PM**
* "0 \*/5 \* \* \* \*" → Runs **every 5 minutes**
* "0 0 2 \* \* MON" → Runs **every Monday at 2 AM**

You can use cron expressions for **complex scheduling scenarios**, like running a task at specific times or on specific days.

**4. Parallel Execution of Scheduled Tasks**

By default, Spring Boot executes scheduled tasks sequentially in a **single-threaded** mode. If you have multiple scheduled tasks, they will **not run in parallel**.

To enable **parallel execution**, you can:

1. Use @Async to execute tasks asynchronously.
2. Use ThreadPoolTaskScheduler to create a thread pool, allowing multiple tasks to run concurrently.

**5. Dynamic Scheduling**

In some cases, you may need to **start, stop, or update scheduled tasks dynamically** at runtime instead of defining them statically with @Scheduled.  
For this, you can use **ThreadPoolTaskScheduler** to manually control when tasks run and stop.

Example use cases:

* Running a background job **only when a user enables it**.
* Stopping a task **based on some external condition**.
* Dynamically updating the schedule **without restarting the application**.

**Conclusion**

* Spring Boot makes it easy to automate tasks using **@Scheduled**.
* You can schedule tasks using **fixed rate, fixed delay, or cron expressions**.
* By default, tasks run **sequentially**, but you can use **thread pools or async execution** for parallel execution.
* If you need **flexibility**, you can use **dynamic scheduling** to control task execution at runtime.
  + **Redis**

Redis wsl Command Redis Installation:

1. curl -fsSL [https://packages.redis.io/gpg](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqblRxSnp2VldqTFVobVUwT2M3RTgyUHVCMmtud3xBQ3Jtc0tteW1taVJ1eXZ5VFRiRW53bmsyaEhnYm5RbHJ6OGMteDExV2NtN0JJZi1QSFlUV0xEa0c2TVBDZ0FESEtKN1N4MGhPdWF4YmUwZUhLZ1VtSDhSQ1F6aDlscF9mdlJFU0M2d3c5WkRYQmIteGNuR2RsSQ&q=https%3A%2F%2Fpackages.redis.io%2Fgpg&v=2srQ-RiJHps) |sudo gpg --dearmor -o /usr/share/keyrings/redis-archive-keyring.gpg
2. echo "deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg] [https://packages.redis.io/deb](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbW1lN1k4NHExU2ppLUZJbGY5LUlHYzk4TFF2QXxBQ3Jtc0ttSzc2RFlVaHBlXy1BbHdPVjVOOUU4X3NEQkY5RklHZWRSOXBoNENVaHdRYXpEQXJFQlpWaHVTN09KdmswN3BHWUVmUUtrX2tKNi1zWXhqVWJtbEdBb19JNEtXR2IyMlNaSjhMZ09HNnNsUEprS0FTZw&q=https%3A%2F%2Fpackages.redis.io%2Fdeb&v=2srQ-RiJHps) $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/redis.list
3. sudo apt-get update sudo apt-get install redis
4. sudo service redis-server start
5. redis-cli

Execute the Command in Line Wise Order.

* **What is Redis in Simple Terms?**

Redis is like a **super-fast, in-memory database** that stores data temporarily (but can be made persistent). Instead of storing data on a hard drive like traditional databases, Redis keeps everything in **RAM (Random Access Memory)**, making it **extremely fast**.

Think of it like a **notebook** where you can quickly jot down and retrieve information instead of going through a large book (which would be a slower, traditional database like MySQL).

* **How Does Redis Store Data Using Key-Value Pairs?**

Redis follows a **key-value store** model, meaning it stores data as a **pair of a unique key and a value**.

**1. What is a Key?**

A **key** in Redis is like a **label** or a **name tag** that helps identify stored data. It must be unique, just like a username or a product ID.

**2. What is a Value?**

A **value** is the actual data stored in Redis. Unlike some databases that store only simple text or numbers, Redis can store different types of values, including:

* **Strings** (e.g., a user’s name)
* **Lists** (e.g., a list of recent searches)
* **Sets** (e.g., unique visitors on a website)
* **Hashes** (e.g., structured data like a user's profile)
* **Sorted Sets** (e.g., leaderboard rankings)

**Example of Key-Value Pairs**

Imagine a **dictionary** where:

* The **word** is the **key**
* The **definition** is the **value**

For example:

* **Key:** "username:1001" → **Value:** "JohnDoe"
* **Key:** "cart:user123" → **Value:** ["Laptop", "Mouse", "Keyboard"]
* **Key:** "score:game" → **Value:** {"Alice": 100, "Bob": 80}

This makes Redis **very efficient** for fast lookups, caching, and real-time applications.

* **Why Use Redis?**

1. **Speed** – Since it stores data in RAM, fetching data is almost instant.
2. **Scalability** – It handles large-scale applications efficiently.
3. **Session Storage** – Websites use Redis to store user sessions (like login data).
4. **Caching** – Redis speeds up applications by caching frequently accessed data.
5. **Real-Time Applications** – Chat apps, leaderboards, and stock price tracking use Redis for instant updates.

* **Kafka**

Open Source distributed event streaming platform

Kafka is designed to handle data that is constantly being generated and needs to be processed as it comes in, without delays

* **What is Apache Kafka?**

Apache Kafka is a distributed event streaming platform used to handle large amounts of real-time data efficiently. It helps applications communicate with each other by sending and receiving messages in a fast, reliable, and scalable way.

* **Why is Kafka Used?**

Kafka is mainly used for:

* **Real-time Data Processing**: Applications can send and receive data in real time.
* **Message Queuing**: Works as a buffer between applications to prevent overload.
* **Data Integration**: Connects different systems by allowing them to share data.
* **Scalability**: Can handle a growing amount of data without performance issues.
* **How Does Kafka Work?**

Kafka consists of four main components:

1. **Producers**
   * Applications that send (produce) data to Kafka.
   * Example: A stock trading app sending stock price updates.
2. **Topics**
   * A category where messages are stored and organized.
   * Example: A "Stock\_Prices" topic storing real-time stock data.
3. **Brokers**
   * Kafka servers that store and manage messages.
   * Distributes data across multiple servers to ensure reliability.
4. **Consumers**
   * Applications that receive (consume) data from Kafka.
   * Example: A dashboard displaying live stock prices.

* **Example of Kafka in Real Life**

Imagine an **online food delivery app** like Swiggy or Zomato:

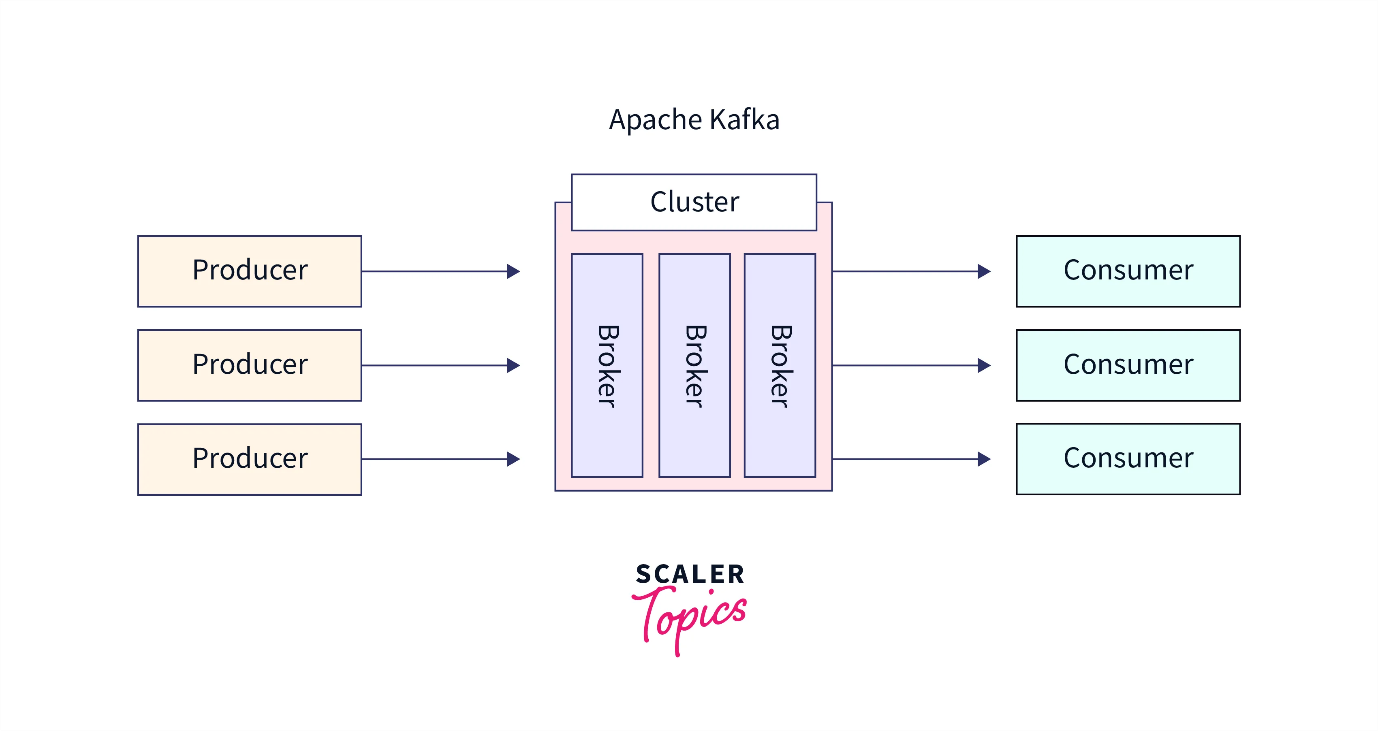
* When you place an order, the **Producer** (restaurant app) sends order details to Kafka.
* Kafka stores this order in a **Topic** (e.g., "Orders\_Topic").
* The **Consumer** (delivery app) picks up the order from Kafka and assigns it to a delivery agent.
* The app updates the order status in real-time, ensuring smooth communication.
* **Benefits of Kafka**

✅ **Fast** – Handles millions of messages per second.  
✅ **Reliable** – Ensures no data loss with backup copies.  
✅ **Scalable** – Can expand as data grows.  
✅ **Fault-Tolerant** – If one server fails, others take over.

* **Where is Kafka Used?**
* **Banking** – Fraud detection and transaction processing.
* **E-commerce** – Order tracking and real-time product availability.
* **Social Media** – Processing live feeds and notifications.
* **IoT Devices** – Collecting data from smart sensors.

Kafka is widely used by companies like LinkedIn, Netflix, Uber, and Twitter for handling massive real-time data.

**Explaination**



Kafka is a distributed event streaming platform. It works like a message broker that helps applications communicate with each other by passing messages in real-time.

Now, let's break down the three main components:

**1. Kafka Cluster 🚀**

A **Kafka Cluster** is a group of Kafka servers (also called **brokers**) working together.

**How it works:**

* Instead of storing all messages on a single server, Kafka distributes them across multiple servers (brokers).
* This improves **scalability, fault tolerance, and performance**.
* If one broker fails, the data is not lost because another broker takes over.

**Key components in a Kafka Cluster:**

✅ **Brokers:** These are the individual servers that store and manage messages.  
✅ **Topics:** A topic is like a folder where messages are categorized.  
✅ **Partitions:** Each topic is divided into smaller sections (partitions) to improve speed and efficiency.  
✅ **Zookeeper:** Keeps track of the brokers, manages leader election, and handles metadata.

📌 **Example:**  
Think of a Kafka Cluster as a **post office**. It has multiple branches (brokers) that handle letters (messages). If one branch closes, another branch delivers the letters.

**2. Kafka Producer 📝**

A **Kafka Producer** is responsible for **sending messages** to Kafka topics.

**How it works:**

* It creates messages and assigns them to cluster.
* The messages are then stored in **partitions** inside the Kafka brokers.
* Producers can choose whether to wait for an acknowledgment from Kafka (to ensure delivery) or send messages without waiting (faster but less reliable).

**Key Features of Kafka Producer:**

✅ **Asynchronous Messaging:** Messages are sent without waiting for responses.  
✅ **Partitioning:** Producers can decide how messages are distributed across partitions.  
✅ **Retries & Acknowledgments:** If a message fails, it can be resent.

📌 **Example:**  
A Kafka Producer is like an **email sender**. You write an email and send it (produce a message). The email service (Kafka) stores and forwards it to the correct recipient (consumer).

**3. Kafka Consumer 📬**

A **Kafka Consumer** is responsible for **reading messages** from Kafka Cluster.

**How it works:**

* Consumers subscribe to a **topic** and read messages from partitions.
* They process the messages and can store the results in a database, display them on a dashboard, etc.
* Kafka ensures that each message is read only **once per consumer group** to prevent duplication.

**Key Features of Kafka Consumer:**

✅ **Consumer Groups:** Multiple consumers can work together by splitting the workload.  
✅ **Offset Management:** Keeps track of which messages have been read.  
✅ **Automatic & Manual Commit:** Consumers can acknowledge messages after processing.

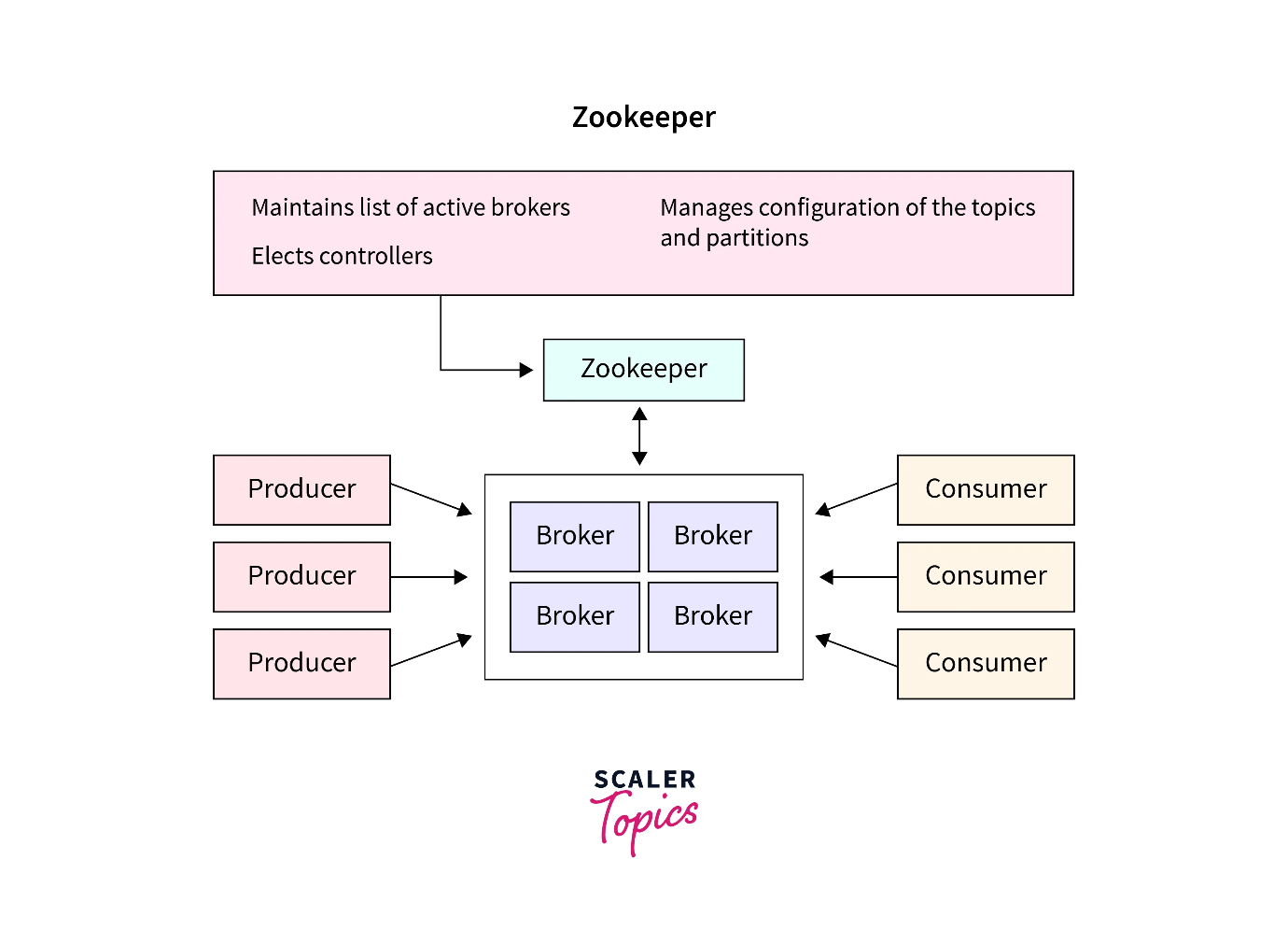
📌 **Example:**  
A Kafka Consumer is like an **email reader**. You open your email inbox (consume messages), read them, and decide what to do next (store, delete, or reply).

**Final Analogy**

🔹 **Kafka Cluster** = A post office managing multiple branches (brokers).  
🔹 **Kafka Producer** = Someone who writes and sends letters (messages).  
🔹 **Kafka Consumer** = Someone who receives and reads the letters.

**ZooKeeper**

Zookeeper keeps track kafka cluster health.



* **What is Zookeeper?**

Zookeeper is a **coordination service** used by Kafka to manage its brokers, topics, partitions, and leader election.

💡 **Think of Zookeeper as the "manager" of Kafka** – it keeps track of everything happening inside the Kafka cluster and ensures everything runs smoothly.

* **Why is Zookeeper Needed in Kafka?**

Kafka is a distributed system, meaning it has multiple servers (brokers) working together.  
But how do these brokers coordinate and keep track of:  
✔️ Which broker is active?  
✔️ Who is the leader of a partition?  
✔️ Where is data stored?  
✔️ What happens if a broker fails?

➡️ **Zookeeper helps answer these questions** by maintaining real-time information about Kafka's health and structure.

* **Main Responsibilities of Zookeeper in Kafka**

1️ **Broker Management** 🖥️

* Keeps track of all Kafka brokers in the cluster.
* If a broker joins or leaves, Zookeeper updates this information.

2️ **Leader Election for Partitions** 👑

* In Kafka, each topic is divided into **partitions**, and each partition has a **leader broker** that handles reads and writes.
* If a leader broker crashes, Zookeeper **elects a new leader** to ensure no data loss.

3️ **Topic and Partition Metadata Management** 📂

* Stores details about Kafka **topics, partitions, and configurations** so brokers know where to send messages.

4️ **Keeping Track of Consumer Offsets** 🔢

* Stores the last message read by consumers so they can resume reading from the correct point if they restart.

5️ **Failure Detection and Recovery** 🚨

* Detects when a broker fails and helps Kafka **reassign partitions** to healthy brokers.
* **How Zookeeper Works in Kafka?**

🔹 Kafka brokers **register** themselves with Zookeeper when they start.  
🔹 Producers and consumers **ask Zookeeper** for broker details before sending/receiving messages.  
🔹 If a broker **crashes**, Zookeeper notifies Kafka, and a new leader is chosen automatically.

* **Analogy – Zookeeper as a Traffic Controller 🚦**

Imagine a **busy city with multiple traffic lights** (Kafka brokers). The **traffic control center** (Zookeeper) manages all traffic lights:  
✅ It ensures traffic flows smoothly.  
✅ If a traffic light (broker) stops working, it assigns another one as the leader.  
✅ It keeps track of road conditions (metadata, partitions).

Without the traffic control center, the city would be chaotic—just like Kafka without Zookeeper!

* **Is Zookeeper Still Required in Kafka?**

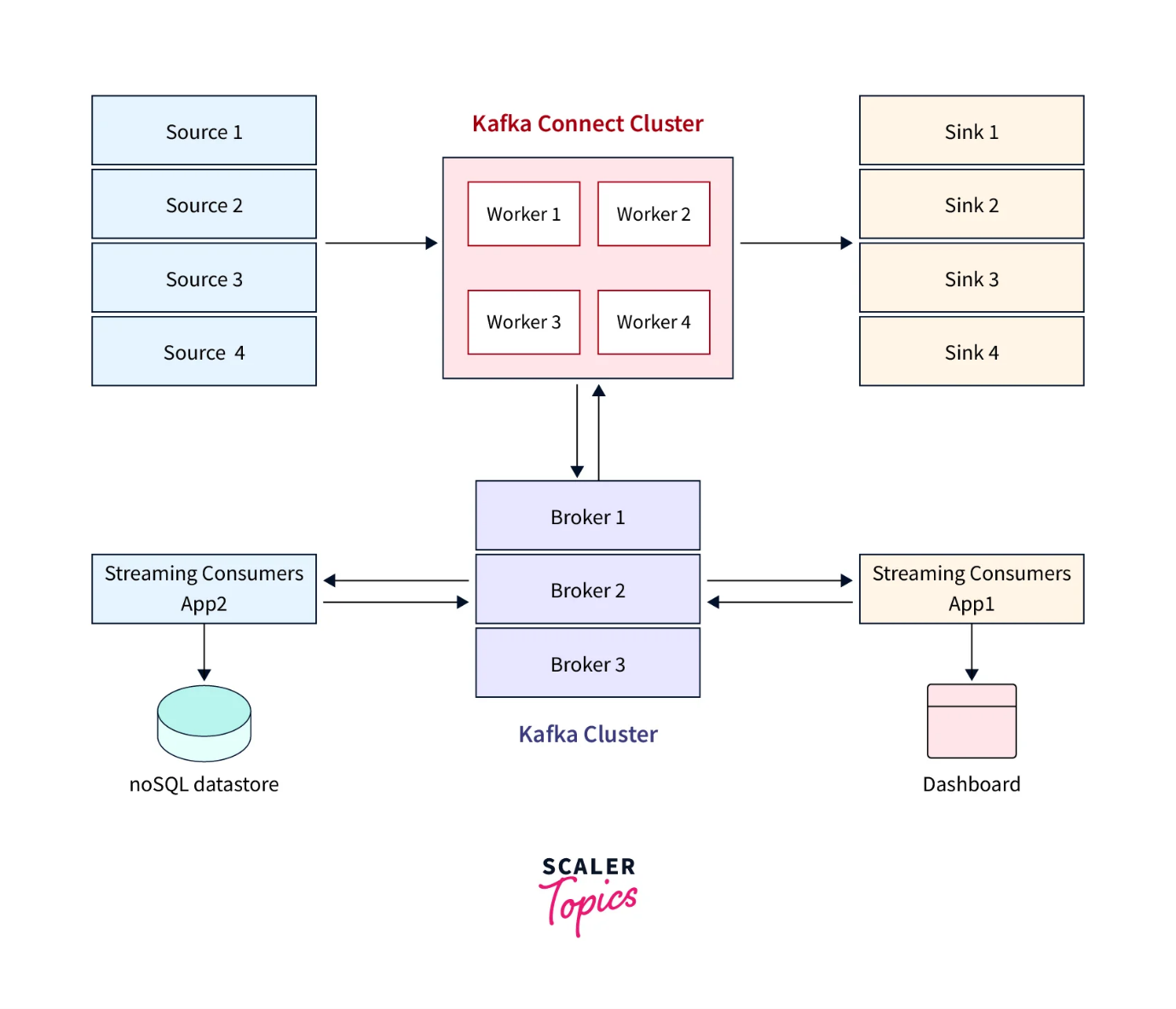
Earlier, Kafka **depended heavily** on Zookeeper. But newer versions of Kafka (starting from **Kafka 3.0**) have introduced **KRaft mode**, which removes the need for Zookeeper and makes Kafka self-sufficient.

* **Summary of Zookeeper in Kafka:**

📌 Zookeeper is the brain of Kafka, managing brokers, partitions, and leader elections.  
📌 It ensures Kafka runs efficiently by tracking metadata and failures.  
📌 Without Zookeeper, Kafka brokers wouldn't know how to coordinate.  
📌 Newer Kafka versions are replacing Zookeeper with KRaft mode for better efficiency.

**Kafka Connect**

It can get any data from external entity without writing any code. Its is called Declarative Integration.



**What is Kafka Connect?**

Kafka Connect is a **data integration tool** that helps **move data in and out of Kafka** easily without writing complex code.

💡**Think of Kafka Connect as a data pipeline builder** that connects Kafka with different data sources (like databases, cloud storage, and applications).

**Why Do We Need Kafka Connect?**

Imagine you have a Kafka system, and you want to:  
✔️ Continuously pull data from a **database** (e.g., MySQL, MongoDB).  
✔️ Send Kafka data to **data warehouses** (e.g., Elasticsearch, Snowflake).  
✔️ Stream data between **different Kafka clusters**.

Instead of manually writing code to **read and write data**, Kafka Connect provides **pre-built connectors** to handle these tasks automatically.

**How Kafka Connect Works?**

Kafka Connect acts as a **bridge** between Kafka and external systems. It has two main components:

1️ **Source Connectors** 🔄

* **Pulls data** from external systems **into Kafka**.
* Example: Fetches data from **MySQL, MongoDB, APIs, or sensors** and stores it in Kafka topics.

2️ **Sink Connectors** 📤

* **Pushes data** from Kafka **to external systems**.
* Example: Reads messages from Kafka and **sends them to PostgreSQL, Elasticsearch, or cloud storage**.

📌 **Example Use Case:**

* A **Source Connector** extracts real-time user data from **MySQL** and puts it in a Kafka topic.
* A **Sink Connector** then sends this data from Kafka to **Elasticsearch** for analytics.

**Key Features of Kafka Connect**

✅ **Pre-Built Connectors** – No need to write custom code, just configure settings.  
✅ **Scalability** – Can handle massive amounts of data.  
✅ **Fault Tolerance** – Automatically retries in case of failures.  
✅ **Distributed & Standalone Modes** – Can run on a single machine or a cluster.  
✅ **Schema Management** – Ensures data consistency across different systems.

**Kafka Connect Analogy – A Shipping Hub 📦**

Think of **Kafka Connect as a shipping hub** that transports packages (data) between different locations (systems):

* **Source Connectors** = Pick up packages from suppliers (databases, APIs).
* **Kafka** = Stores packages temporarily before delivery.
* **Sink Connectors** = Deliver packages to customers (data warehouses, applications).

Without Kafka Connect, you would have to transport every package **manually** (write custom code for each system).

**Kafka Connect in Action (Example Setup)**

**1️ Source Connector Example: MySQL → Kafka**

Imagine a **user registration system** stores data in **MySQL**, and you want to stream this data into Kafka.

* **Use a MySQL Source Connector** to pull new registrations from MySQL.
* Kafka stores this data in a **topic called user\_signups**.
* Now, other applications can process this user data in real time.

**2️ Sink Connector Example: Kafka → PostgreSQL**

Now, let’s say you want to save this user data from Kafka into **PostgreSQL** for reporting.

* **Use a PostgreSQL Sink Connector** to pull messages from the user\_signups topic.
* The connector inserts this data into a **PostgreSQL table**.

📌 **Final Flow: MySQL → Kafka → PostgreSQL** (All done automatically with Kafka Connect!)

**Kafka Connect vs. Kafka Producer & Consumer**

| **Feature** | **Kafka Connect** | **Kafka Producer & Consumer** |
| --- | --- | --- |
| **Purpose** | Moves data between Kafka & external systems | Sends & receives messages inside Kafka |
| **Coding** | Minimal coding required (just config) | Requires writing Java/Python code |
| **Use Case** | Bulk data transfer (DBs, Cloud, APIs) | Real-time processing of messages |
| **Scalability** | Scalable with minimal effort | Needs manual scaling |

**Do You Need Kafka Connect?**

💡 **Use Kafka Connect if:**  
✔️ You need to **move large-scale data** from databases, APIs, cloud storage, etc.  
✔️ You want to integrate Kafka **without writing custom producer/consumer code**.  
✔️ You need a **fault-tolerant and scalable** data pipeline.

❌ **Don’t use Kafka Connect if:**

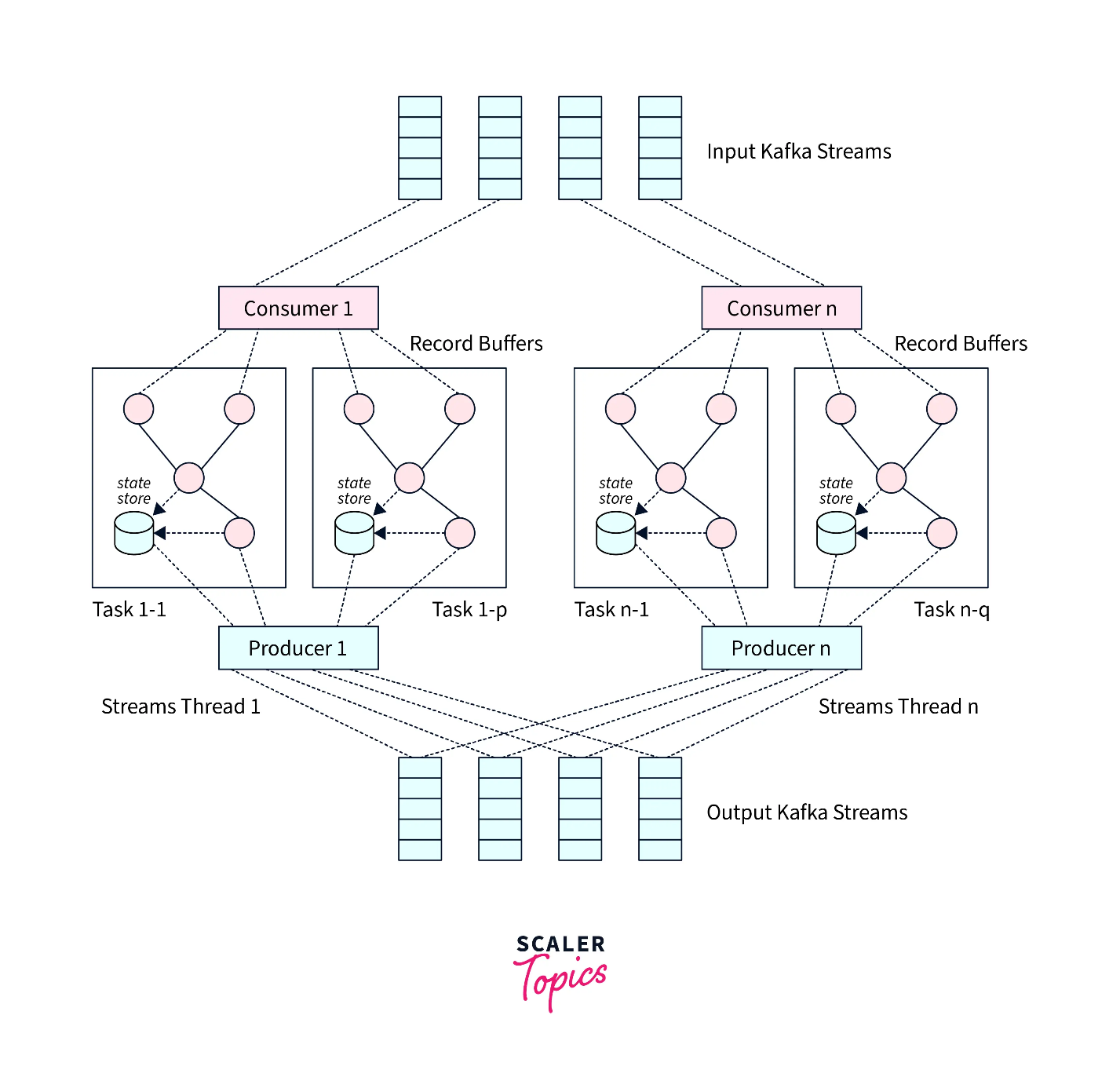
* You only need simple producer-consumer interactions inside Kafka.
* You prefer custom-built solutions for full control.

**Summary of Kafka Connect**

📌 Kafka Connect is a tool for integrating Kafka with external systems.  
📌 It has Source Connectors (import data) and Sink Connectors (export data).  
📌 It automates data movement without custom coding.  
📌 It is scalable, fault-tolerant, and supports real-time streaming.

**Kafka Stream**

Used for Data Transformation



* **What is Kafka Streams?**

Kafka Streams is a **real-time data processing library** that helps you **process, transform, and analyze** data as it flows through **Kafka topics**.

💡 **Think of Kafka Streams as a "mini data factory" inside Kafka** – it continuously processes data from Kafka topics, applies transformations, and writes results back to Kafka or external systems.

* **Why Do We Need Kafka Streams?**

Kafka is great for moving data between systems, but what if you want to:  
✔️ **Filter messages** (e.g., only process orders above ₹1000).  
✔️ **Transform data** (e.g., convert names to uppercase).  
✔️ **Join multiple data sources** (e.g., combine customer and order data).  
✔️ **Count events in real-time** (e.g., number of logins per minute).

➡️ **Kafka Streams does all this in real-time!**

* **How Kafka Streams Works?**

Kafka Streams **reads data** from Kafka topics, **processes** it using a stream processing logic, and **writes the result** to another Kafka topic or external storage.

It follows a simple flow:

1️ **Consume Data** – Reads messages from a Kafka topic.  
2️ **Process Data** – Applies filters, aggregations, joins, etc.  
3️ **Produce Data** – Sends processed data to another topic or database.

📌 **Example Use Case:**

* You have a Kafka topic **orders** where every new order is stored.
* You use Kafka Streams to **filter out** canceled orders and calculate the total order amount.
* The processed data is stored in another topic **valid\_orders**, which is then used for analytics.
* **Core Features of Kafka Streams**

✅ **Real-time Processing** – Processes data as soon as it arrives.  
✅ **Stateful Processing** – Stores intermediate results for aggregation, counting, etc   
✅ **Fault Tolerant** – Can recover from failures automatically.  
✅ **Highly Scalable** – Can run across multiple machines.  
✅ **Works with Java** – Built as a Java library, easy to integrate into Java apps.

* **Kafka Streams vs. Other Processing Tools**

| **Feature** | **Kafka Streams** | **Apache Spark Streaming** | **Flink** |
| --- | --- | --- | --- |
| **Processing Type** | Real-time | Micro-batch | Real-time |
| **Integration** | Kafka-native | External system | External system |
| **Scalability** | Medium | High | Very High |
| **Use Case** | Simple event processing | Complex analytics | Advanced event processing |

📌 **Kafka Streams is best for lightweight, real-time data transformation within Kafka.**

* **Kafka Streams vs. Kafka Consumer**

| **Feature** | **Kafka Streams** | **Kafka Consumer** |
| --- | --- | --- |
| **Purpose** | Process and transform data | Read messages from Kafka |
| **Processing** | Filters, aggregates, joins | Basic message consumption |
| **Code Complexity** | Less | More (manual coding required) |

* **Kafka Streams Analogy – A Juice Factory 🏭**

Think of **Kafka Streams as a juice-making machine**:  
1️ **Fruits (Raw Data in Kafka Topics)** are loaded into the machine.  
2️ **Processing (Kafka Streams)** removes seeds, blends the juice, and adds sugar.  
3️ **Final Juice (Processed Data)** is packaged and sent to another Kafka topic.

Just like the machine **automates juice-making**, Kafka Streams automates **real-time data processing**!

* **When Should You Kafka Streams?**

✅ **Use Kafka Streams if:**  
✔️ You need **real-time** data transformation, filtering, or aggregation.  
✔️ Your data is already in **Kafka topics** and you don’t want to use external tools.  
✔️ You want a lightweight **Java-based solution**.

❌ **Don’t use Kafka Streams if:**

* You need advanced analytics (use **Flink** or **Spark** instead).
* You only need basic message consumption (use a **Kafka Consumer**).
* **Summary of Kafka Streams**

📌 Kafka Streams is a real-time data processing library for Kafka.  
📌 It reads, processes, and writes data within Kafka topics.  
📌 It can filter, transform, and aggregate data in real-time.  
📌 It is lightweight, fault-tolerant, and scalable.

Kafka Download and Configure cmd

🟢 INSTALLATION COMMANDS

zookeeper-server-start.bat ..\..\config\zookeeper.properties

kafka-server-start.bat ..\..\config\server.properties

kafka-topics.bat --create --topic my-topic --bootstrap-server localhost:9092 --replication-factor 1 –partitions 3

kafka-console-producer.bat --broker-list localhost:9092 --topic my-topic

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic my-topic --from-beginning

**Kafka topic**

* Named container for similar events. Unique identifier of a topic is its name.
* **Example**: Student topic will have student related data, Food Topic will have food related data. They are like tables in a database. They live inside a broker.Producer produce a message into the topic ( ultimately to partitions in round robin fashion ) or directly to the partitions. Consumer poll continuously for new messages using the topic name.
* **Partition** - A topic is partitioned and distributed to Kafka brokers in round robin fashion to achieve distributed system.
* **Replication factor** - A partition is replicated by this factor and it is replicated in another broker to prevent fault tolerance.

**Partitions**

* A topic is split into several parts which are known as the partitions of the topic.
* Partitions is where actually the message is located inside the topic.
* Therefore, while creating a topic, we need to specify the number of partitions ( the number is arbitrary and can be changed later ).
* Each partition is an ordered, immutable sequence of records.
* Each partition is independent of each other.
* Each message gets stored into partitions with an incremental id known as its Offset value.
* Ordering is there only at partition level. ( so if data is to be stored in order then do it on same partition )
* Partition continuously grows (offset increases) as new records are produced.
* All the records exist in distributed log file.

**Key And Ordering**

* Same Key =SamePartions
* two things - key and value, key is optional
* We can send message with key or Without key.
* When sending messages with key, ordering will be maintained as they will be in same partition.
* Without key we cannot guarantee the ordering of message as consumer poll messages from all the partition of same times.

**Consumer Offset**

* When a consumer group reads messages from a topic, each member of the group maintains its own offset and updates it as it consumes messages
* Where customer offset is stored

\_\_consumer\_offset

* consumer\_offset is a built-in topic in Apache Kafka that keeps track of the latest offset committed for each partition of each consumer group.
* The topic is internal to the Kafka cluster and not meant to be read or written to directly by clients. Instead, the offset information is stored in the topic and updated by the Kafka broker to reflect the position of each consumer in each partition.
* The information in consumer offset is used by Kafka to maintain the reliability of the consumer groups and to ensure that messages are not lost or duplicated.
* There is a separate consumer\_offsets topic created for each consumer group. So if you have 2 consumer groups containing 4 consumers each, you will have a total of 2 consumer\_offsets topics created.
* The consumer\_offsets topic is used to store the current offset of each consumer in each partition for a given consumer group. Each consumer in the group updates its own offset for the partitions it is assigned in the consumer\_offsets topic, and the group coordinator uses this information to manage the assignment of partitions to consumers and to ensure that each partition is being consumed by exactly one consumer in the group
* When a consumer joins a consumer group, it sends a join request to the group coordinator.
* The group coordinator determines which partitions the consumer should be assigned based on the number of consumers in the group and the current assignment of partitions to consumers.
* The group coordinator then sends a new assignment of partitions to the consumer, which includes the set of partitions that the consumer is responsible for consuming.
* The consumer starts consuming data from the assigned partitions.
* It is important to note that consumers in a consumer group are always assigned partitions in a "sticky" fashion, meaning that a consumer will continue to be assigned the same partitions as long as it remains in the group. This allows consumers to maintain their position in the topic and continue processing where they left off, even after a rebalance.

**What is a Consumer Offset?**

In Apache Kafka, **consumer offset** is a number that keeps track of **how much data a consumer has read from a partition**. It tells Kafka **where the consumer left off** in the last read, so it can resume from the correct spot instead of starting from the beginning.

**Why is Consumer Offset Important?**

1. **Ensures No Data is Re-read or Missed** → If a consumer stops or crashes, it can restart from where it left off.
2. **Supports Multiple Consumers** → Each consumer can track its progress independently.
3. **Helps in Fault Recovery** → After failures, consumers can resume consuming messages from the last saved offset.

**Types of Offset Management**

Kafka provides two ways to manage offsets:

1. **Automatic Offset Management (Auto-Commit)**
   * Kafka **automatically commits offsets** after a set time (default: 5 seconds).
   * Can lead to **data loss** (if a crash happens before processing is complete).
2. **Manual Offset Management**
   * Consumer **manually commits offsets** after successfully processing messages.
   * More control, prevents data loss.

**Offset Retention**

* Kafka **keeps consumer offsets for a set period** (default: **7 days**).
* If a consumer is inactive beyond this period, it loses its last committed offset.

**Offset Reset Policy (If Offsets Are Lost)**

When a consumer starts and no offset is found, Kafka applies **auto.offset.reset**:

* earliest → Read from the **beginning** of the topic.
* latest → Read only **new messages**.
* none → Throw an error if no offset is found.

**Summary**

* **Consumer Offset** → Tracks where a consumer last read messages.
* **Stored in** → \_\_consumer\_offsets topic.
* **Used for** → Resuming consumption after restarts.
* **Managed by** → Auto-commit (Kafka) or Manual-commit (Consumer).
* **If lost** → Can reset to earliest, latest, or throw an error.

**Segments**

**1. What is a Segment in Kafka?**

In Kafka, a **segment** is a **small chunk of a partition's log file** stored on disk. Instead of keeping all messages in one giant file, Kafka **splits the data into multiple smaller files** (segments) to improve performance and manageability.

Each **segment contains a portion of the messages** from a partition and is stored as a file on disk.

**2. Why Does Kafka Use Segments?**

Kafka uses segments for **better performance and easier data management**:

* **Efficient Reads & Writes** → Instead of searching a massive file, Kafka only needs to look at the latest segment.
* **Faster Log Cleanup** → Old segments can be deleted or compacted easily.
* **Avoids Memory Overload** → Large files can slow down processing.

**3. How Do Segments Work in Kafka?**

Each **Kafka partition** is made up of multiple **segments**.

**Step-by-Step Process:**

1. **Messages Are Written to the Active Segment**
   * Every partition has **one active segment** where new messages are appended.
2. **Segment Gets Full Based on Size or Time**
   * A segment closes when it reaches:
     + A certain **size limit** (e.g., 1GB) → log.segment.bytes
     + A certain **time limit** (e.g., 1 hour) → log.roll.ms
3. **A New Segment is Created**
   * Once the active segment is closed, a **new segment** file is created, and Kafka starts writing to it.
4. **Old Segments are Deleted or Compacted**
   * Based on retention policy (log.retention.hours or log.retention.bytes), Kafka **deletes** or **compresses** old segments.

**4 . Example of Segments in Action**

**Imagine a Kafka topic "orders" with a single partition:**

* The first 1,000 messages go into segment 1.
* When full, Kafka creates segment 2 for the next 1,000 messages.
* The process continues…
* After **retention time**, old segments are deleted.

**5. What Happens When a Consumer Reads Data?**

* Kafka **only needs to read relevant segments** instead of scanning the whole partition.
* The **index file** helps find messages quickly.

**6. Summary**

* **Segments** = Small log files that store partition data.
* **New messages go into an "active segment"**; once full, a new one is created.
* **Old segments are deleted or compacted** based on retention settings.
* **Improves Kafka performance** by making reads/writes more efficient.

**JWT**

* Json Web Token..
* JWT is a way to securely transmit information between parties as a JSON object
* JWT is a compact, URL-safe token that can carry information between parties.
* A JWT is a string consisting of three parts, separated by dots.
* **Header**: - The header typically consists of two parts: the type of the token (JWT) and the signing algorithm being used, such as I-IMAC SHA256 or RSA.
* **Payload**:- The payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata.
* **Signature**:- The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way. To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

JWT (JSON Web Token) is a secure way to share information between two parties, typically a client (like a web app) and a server. It is often used for authentication and authorization. JWTs are compact, self-contained, and digitally signed, making them secure and efficient.

**Why JWT Has Three Parts?**

JWT is divided into three parts—**Header, Payload, and Signature**—to ensure security, structure, and integrity. Here’s why each part exists:

1. **Header** – To define the type of token and the signing algorithm.
2. **Payload** – To carry user-related data (claims).
3. **Signature** – To verify the token’s authenticity and prevent tampering

**1. Header (First Part)**

* Contains metadata about the token.
* Includes two main pieces of information:
  + **Type** (typ): Specifies that it is a JWT.
  + **Algorithm** (alg): Defines the encryption method (e.g., HS256, RS256).
* **Why is it needed?**
  + So the system knows how to decode and verify the JWT.
  + It ensures compatibility with different security mechanisms.

**2. Payload (Second Part)**

* Contains actual user-related data, called **claims**.
* There are three types of claims:
  + **Registered claims** (standard fields like exp for expiration).
  + **Public claims** (custom data, e.g., user ID or role).
  + **Private claims** (specific data agreed between sender and receiver).
* **Why is it needed?**
  + It carries the information required for authentication and authorization.
  + Eliminates the need for frequent database lookups.

**3. Signature (Third Part)**

* Created by encrypting the Header and Payload using a secret key.
* Ensures that the token has not been altered.
* **Why is it needed?**
  + Prevents unauthorized modifications.
  + Allows the receiver to verify that the JWT is genuine.

**How JWT Works in Authentication**

1. User logs in → Server verifies credentials → Server generates JWT.
2. The JWT is sent to the user and stored (e.g., in local storage or cookies).
3. When making requests, the user sends the JWT in the headers.
4. The server verifies the JWT (by checking the signature) and grants access if valid.

**Final Thoughts**

JWT is structured with three parts to separate concerns:

* **Header:** Defines how the token should be processed.
* **Payload:** Contains the actual information.
* **Signature:** Ensures security and prevents tampering.

**Jwt Dependency in Maven**

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.12.5</version>  
</dependency>  
  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.12.5</version>  
</dependency>  
  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.12.5</version>  
 <scope>runtime</scope>  
</dependency>

In a Spring Boot project, when using **JSON Web Token (JWT)** for authentication and authorization, the **JJWT (Java JWT)** library requires three dependencies to function properly. Here's why each of these dependencies is needed:

**1️ jjwt-api (jjwt-api):**

* This provides the core **JWT API**, including interfaces and classes to **create, parse, and verify JWT tokens**.
* It defines methods like Jwts.builder(), JwtParser, and Claims.

✅ **Required at compile-time** since you'll be writing code that interacts with the JWT API.

**2️ jjwt-jackson (jjwt-jackson):**

* This provides **JSON processing support** using **Jackson** (a widely used JSON library).
* JWT claims (payload) are stored as JSON, so this dependency ensures correct **serialization and deserialization** of JWT payloads.

✅ **Required for handling JSON claims within JWT tokens**.

**3️ jjwt-impl (jjwt-impl):**

* This provides the **implementation** of the interfaces defined in jjwt-api.
* It contains the actual logic to **sign, verify, encode, and decode JWTs**.

✅ **Marked as runtime because it’s needed only during execution, not at compile-time**.

**🔹 Why do we separate them?**

* jjwt-api defines the structure and methods (compile-time).
* jjwt-impl provides the actual implementation (runtime).
* jjwt-jackson enables JSON support for JWT payloads.

💡 **Without jjwt-impl, your code will compile but fail at runtime because the actual implementation is missing.**  
💡 **Without jjwt-jackson, JWT claims cannot be properly serialized/deserialized.**

**✅ In short, all three are required for a complete JWT implementation in Spring Boot.**

Heroku

Kusho.ai for testing api

OpenAPl Specification It defines a standardized format for describing APIs comprehensively

**Swagger**

* Tools used to implement the OpenAPI specification
* Swagger is an open-source framework used for designing, building, documenting, and consuming RESTful APIs. It provides a standardized way to describe the structure of an API, making it easier for developers to understand, integrate, and consume the API.

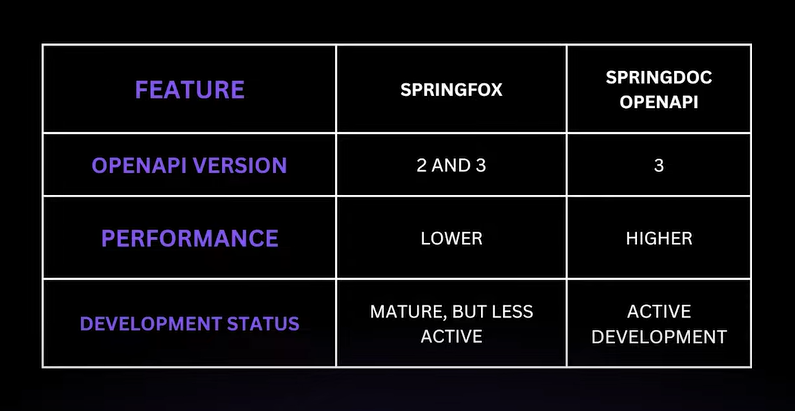
**SpringFox:-**

* Springfox is a Java library used to integrate Swagger with Spring Boot applications. It automatically generates Swagger documentation from your Spring controllers and models.
* springfox-boot-starter: A starter dependency to quickly set

up and integrate Springfox into a Spring Boot project.

**SpringDoc OpenAPI**

* Springdoc OpenAPI is an alternative to Springfox. It is designed to generate API documentation from Spring Boot applications using the OpenAPl 3 specification, which is the latest iteration of the Swagger specification.
* springdoc-openapi-ui: A module that integrates Spring Boot applications with Swagger Ul using the OpenAPl 3 specification.



After adding the dependency just go on this url

localhost:8080/swagger-ui/index.html

**OAuth2**

OAuth2 stands for **Open Authorization 2.0.**

OAuth 2.0 (Open Authorization 2.0) is a widely used security framework that allows applications to access a user’s data from another service (like Google, Facebook, or GitHub) without requiring the user to share their password. It enables secure authorization for applications while keeping user credentials private.

**Why OAuth 2.0?**

Imagine you want to sign up for a new app, and instead of creating a new username and password, the app lets you **"Sign in with Google."** How does this work? The app does not ask for your Google password; instead, it asks Google for permission to access some of your information, like your email address. OAuth 2.0 is the system that makes this process secure and standardized.

**How OAuth 2.0 Works**

OAuth 2.0 involves **four main roles**:

1. **Resource Owner** – The user who owns the data (e.g., you).
2. **Client (Application)** – The app that wants to access the data (e.g., a to-do list app that wants to read your Google Calendar).
3. **Authorization Server** – The server that handles user authentication and issues access tokens (e.g., Google’s authentication system).
4. **Resource Server** – The server that contains the protected data (e.g., Google Calendar API).

**OAuth 2.0 Flow (Step by Step)**

When an app (client) wants to access your data from another service (like Google), the following steps happen:

**Step 1: User Requests Access**

* You try to log in to an app using **"Sign in with Google."**
* The app redirects you to Google’s **Authorization Server**.

**Step 2: User Grants Permission**

* Google asks if you **allow the app to access certain information** (e.g., your email and profile).
* If you approve, Google sends an **Authorization Code** back to the app.

**Step 3: App Requests a Token**

* The app sends the **Authorization Code** to Google’s Authorization Server.
* Google verifies the code and sends back an **Access Token**.

**Step 4: App Accesses Data**

* The app uses the **Access Token** to request data from Google’s **Resource Server** (e.g., Google Calendar).
* If the token is valid, Google provides the requested data (like your email or calendar events).

**Why OAuth 2.0 is Secure**

* **No Password Sharing**: The app never sees your Google password, reducing security risks.
* **Limited Access**: You can control what information the app can access (e.g., only your email, not your contacts).
* **Tokens Expire**: The **Access Token** is temporary and can expire, preventing long-term access.
* **Revocable Access**: You can revoke an app’s access anytime from your Google account settings.

**Common Uses of OAuth 2.0**

1. **Logging in with Social Media** – "Sign in with Google, Facebook, GitHub, etc."
2. **Third-Party App Access** – A fitness app accessing your Fitbit or Google Fit data.
3. **APIs & Cloud Services** – Developers use OAuth 2.0 to securely connect services without storing credentials.

**OAuth 2.0 vs. OAuth 1.0**

* **OAuth 1.0** was more complex, requiring cryptographic signatures for security.
* **OAuth 2.0** is simpler and widely used but relies more on HTTPS for security.

**Conclusion**

OAuth 2.0 is like giving a hotel valet your car key but not your house key. The valet can park your car but cannot enter your home. Similarly, OAuth 2.0 allows an app to access only the data you approve, without giving away your full credentials.

OAuth 2.0, the **Client (Application)** communicates with the **Authorization Server** through different requests and responses. Below is a detailed step-by-step breakdown of what is sent and received, including parameters like **scope, response\_type, redirect\_uri, access\_token, and refresh\_token**.

**Step 1: Client Requests Authorization**

The client (application) redirects the user to the **Authorization Server’s Authorization Endpoint**. This request includes several parameters.

**Request (Sent by Client to Authorization Server)**

GET https://authorization-server.com/auth

?response\_type=code

&client\_id=CLIENT\_ID

&redirect\_uri=https://your-app.com/callback

&scope=email profile

&state=xyz123

**Request Parameters:**

| **Parameter** | **Description** |
| --- | --- |
| response\_type=code | The client requests an **Authorization Code**. |
| client\_id=CLIENT\_ID | The unique identifier of the client (provided by the authorization server). |
| redirect\_uri=https://your-app.com/callback | The URL where the Authorization Server will send the user after approval. |
| scope=email profile | The permissions requested by the app (e.g., access to email and profile). |
| state=xyz123 | A random string to protect against **CSRF attacks**. |

**Summary of Requests & Responses**

| **Step** | **Who Sends It?** | **Endpoint** | **Key Parameters** |
| --- | --- | --- | --- |
| **1. Request Authorization** | Client → Auth Server | /auth (GET) | response\_type, client\_id, redirect\_uri, scope, state |
| **2. Authorization Code Response** | Auth Server → Client | Redirect | code, state |
| **3. Exchange Code for Token** | Client → Auth Server | /token (POST) | grant\_type, code, client\_id, client\_secret, redirect\_uri |
| **4. Access Token Response** | Auth Server → Client | JSON Response | access\_token, token\_type, expires\_in, refresh\_token, scope |
| **5. Request User Data** | Client → Resource Server | API Endpoint (GET) | Authorization: Bearer ACCESS\_TOKEN |
| **6. Refresh Token (Optional)** | Client → Auth Server | /token (POST) | grant\_type=refresh\_token, refresh\_token, client\_id, client\_secret |

**Key Points to Remember**

✅ **OAuth 2.0 uses tokens instead of passwords** for security.  
✅ **Scopes define what data the client can access** (e.g., email, profile).  
✅ **Refresh tokens allow the app to get new access tokens without user interaction**.  
✅ **State parameter prevents CSRF attacks** by ensuring the request wasn't altered.

Sign in with google or github (Waha apne app ko register karwana padega taaki who log ko pata chale ki kon access kar raha hai)

For google we are registering our website via this :

<https://console.cloud.google.com/>