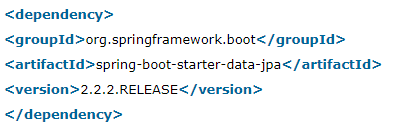
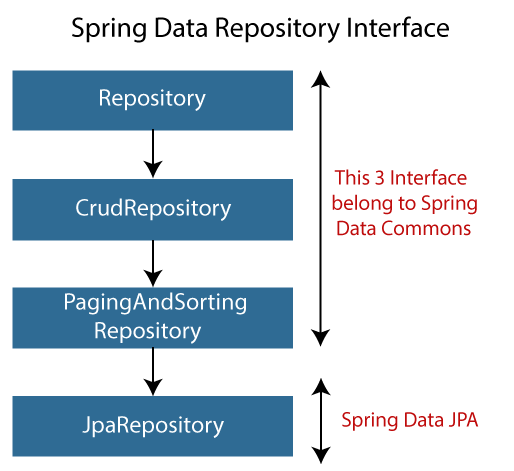
Spring Data JPA

Spring Data JPA handles most of the complexity of JDBC-based database access and ORM (Object Relational Mapping). It reduces the boilerplate code required by JPA. It makes the implementation of your persistence layer easier and faster.

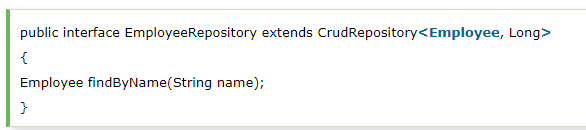
Spring Data JPA adds a layer on the top of JPA. It means, Spring Data JPA uses all features defined by JPA specification, especially the entity, association mappings, and JPA's query capabilities. Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.  
  
Spring Boot provides starter dependency **spring-boot-starter-data-jpa** to connect Spring Boot application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.  


## Spring Data Repository

Spring Data JPA provides **three** repositories are as follows:

* **CrudRepository:** It offers standard **create, read, update,** and **delete** It contains method like **findOne(), findAll(), save(), delete(),**etc.
* **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the findAll methods. It allows us to **sort** and **retrieve** the data in a paginated way.
* **JpaRepository:** It is a **JPA specific repository** It is defined in **Spring Data Jpa**. It extends the both repository **CrudRepository** and **PagingAndSortingRepository.** It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context. JpaRepository provides JPA related methods such as flushing, persistence context, and deletes a record in a batch. 

Below, we extend the **CrudRepository** that uses two generics: **Employee** and **Long**. The Employee is the **entity** that is to be managed, and **Long**is the data type of primary key  
Spring internally generates **JPQL** (Java Persistence Query Language) query based on method name.



# **Spring Boot CRUD Operations**

## What is the CRUD operation?

The **CRUD** stands for **Create, Read/Retrieve, Update,** and **Delete**. These are the four basic functions of the persistence storage.CRUD is data-oriented and the standardized use of **HTTP action verbs**. HTTP has a few important verbs.

* **POST:** Creates a new resource
* **GET:** Reads a resource
* **PUT:** Updates an existing resource
* **DELETE:** Deletes a resource

Within a database,

* **CREATE Operation:** It performs the INSERT statement to create a new record.
* **READ Operation:** It reads table records based on the input parameter.
* **UPDATE Operation:** It executes an update statement on the table. It is based on the input parameter.
* **DELETE Operation:** It deletes a specified row in the table. It is also based on the input parameter.

## How CRUD Operations Works

CRUD operations are at the foundation of the most dynamic websites. Therefore, we should differentiate **CRUD** from the **HTTP** **action verbs**.

Suppose, if we want to **create** a new record, we should use HTTP action verb **POST**. To **update** a record, we should use the **PUT** verb. Similarly, if we want to **delete** a record, we should use the **DELETE**verb. Through CRUD operations, users and administrators have the right to retrieve, create, edit, and delete records online.

# Spring Boot Caching

Spring Framework provides caching in a Spring Application, transparently. In Spring, the **cache abstraction** is a mechanism that allows consistent use of various caching methods with minimal impact on the code.

Caching  
Caching is a part of temporary memory ([RAM](https://www.javatpoint.com/ram-full-form)). It lies between the application and persistence database. It stores the recently used data that reduces the number of database hits as much as possible. In other words, caching is to store data for future reference.

Why should we use the cache?  
The primary reason for using cache is to make data access faster and less expensive. When the highly requested resource is requested multiple times, it is often beneficial for the developer to cache resources so that it can give responses quickly. Using cache in an application enhances the performance of the application. Data access from memory is always faster in comparison to fetching data from the database.

## Types of Caching There are ****four**** types of caching are as follows:

* In-memory Caching
* Database Caching
* Web server Caching
* CDN Caching

### **In-memory Caching**

In-memory caching increases the performance of the application. It is the area that is frequently used. **[Memcached](https://www.javatpoint.com/memcached-tutorial)** and **Redis** are examples of in-memory caching. It stores key-value between application and database.

### **Database Caching**

Database caching is a mechanism that generates web pages on-demand (dynamically) by fetching the data from the database. It is used in a **multi-tier** environment that involved clients, web-application server, and database. It improves **scalability** and **performance** by distributing a query workload. The most popular database caching is the first level cache of [Hibernate](https://www.javatpoint.com/hibernate-tutorial).

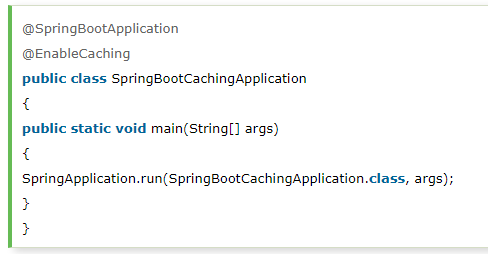
### **Web Server Caching**

Web server caching is a mechanism that stores data for **reuse**. For example, a copy of a web page served by a web server. It is cached for the first time when a user visits the page. If the user requests the same next time, the cache serves a copy of the page. It avoids server form getting overloaded. Web server caching enhances the page delivery speed and reduces work to be done by backend server.  
 **CDN Caching**

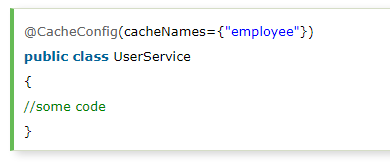
The **CDN** stands for **Content Delivery Network**. It is a component used in modern web applications. It improves the delivery of the content by **replicating** commonly requested files (such as [HTML](https://www.javatpoint.com/html-tutorial) Pages, stylesheet, [JavaScript](https://www.javatpoint.com/javascript-tutorial), images, videos, etc.) across a globally distributed set of **caching servers.**

## Spring Boot Cache Annotations

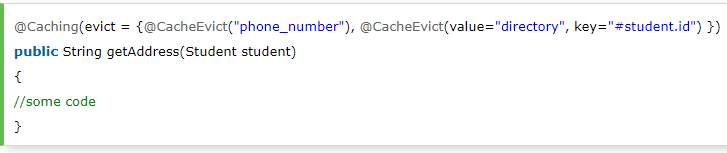
### **@EnableCaching**

It is a class-level annotation. We can enable caching in the Spring Boot application by using the annotation **@EnableCaching.** It is defined in **org.springframework.cache.annotation**package. It is used together with **@Configuration**class.  


### **@CacheConfig**

It is a class-level annotation that provides a common cache-related setting. It tells the Spring where to store cache for the class. When we annotate a class with the annotation, it provides a set of default settings for any cache operation defined in that class. Using the annotation, we need not to declare things multiple times.   
In the following example, **employee** is the name of the cache.  


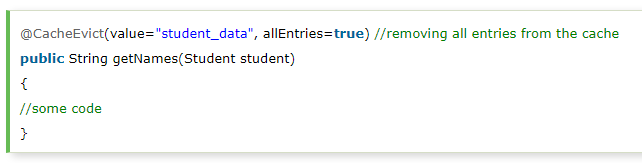
### **@Caching**

It is used when we need both annotations- **@CachePut**or **@CacheEvict**at the same time on the same method. In other words, it is used when we want to use multiple annotations of the same type.But **Java does not allow multiple annotations** of the same type to be declared for a given method. To avoid this problem, we use**@Caching** annotation.  
We have used the annotation **@Caching** and grouped all the **@CacheEvict** annotations.  


### **@CacheEvict**

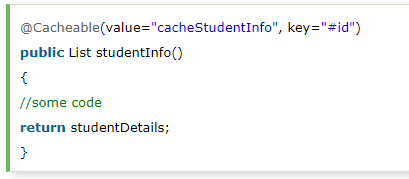
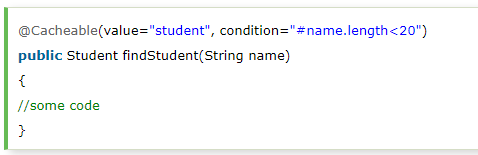
It is a method level annotation. It is used when we want to remove stale or unused data from the cache. It requires one or multiple caches that are affected by the action. We can also specify a key or condition into it. If we want wide cache eviction, the **@CacheEvict** annotation provides a parameter called **allEntries**. It evicts all entries rather than one entry based on the key.

One important point about **@CacheEvict** annotation is that it can be used with void methods because the method acts as a trigger. It avoids return values. On the other hand, the annotation @Cacheable requires a return value that adds/updates data in the cache.

The following annotated method evicts all the data from the cache **student\_data**.  


### **@Cacheable**

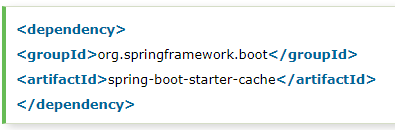
It is a method level annotation. It defines a cache for a method's return value. For example, we can provide a **cache name** by using the **value** or **cacheNames** attribute.  
We can also specify the **key** attribute of the annotation that uniquely identifies each entry in the cache. If we do not specify the key, Spring uses the default mechanism to create the key.

In the following example, we have cached the **return value** of the method **studentInfo()** in **cacheStudentInfo,** and **id** is the unique key that identifies each entry in the cache.  
  
  
We can also apply a condition in the annotation by using the condition attribute. When we apply the condition in the annotation, it is called **conditional caching**.  
For example, the following method will be cached if the argument name has a length shorter than 20.  


### **@CachePut**

It is a method level annotation. It is used when we want to **update** the cache without interfering the method execution. It means the method will always execute, and its result will be placed into the cache. It supports the attributes of @Cacheable annotation.

## A point to be noticed that the annotations @Cacheable and @CachePut are not the same because they have different behavior. There is a slight difference between @Cacheable and @CachePut annotation is that the @**Cacheable** annotation **skips the method execution** while the **@CachePut** annotation **runs the method** and put the result into the cache. The following method will update the cache itself. Spring Boot Cache Dependency

If we want to enable cache mechanism in a Spring Boot application, we need to add cache dependency in the pom.xml file. It enables caching and configures a CacheManager.  


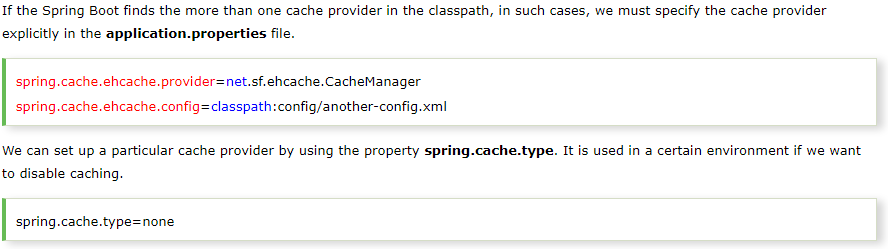
# **Spring Boot Cache Provider**

The Spring Boot framework allows the integration of various **cache providers,** such as **EhCache, Redis, Hazelcast, Infinispan, Caffeine,** etc. The cache provider allows the developer to configure cache transparently and explicitly in an application. In [Spring Boot](https://www.javatpoint.com/spring-boot-tutorial), the cache abstraction does not provide the actual space for the cache. It depends on the abstraction that occurred by the **org.springframework.cache.Cache** or **org.springframework.cache.CacheManager** interfaces.

## Caching Auto-configuration

The Spring Boot Framework simplifies the implementation of caching by auto-configuration support. It searches for the libraries and configuration-files in the classpath and initializes the required dependency beans at the time of application startup. The auto-configuration of caching includes the following steps:

* Add the annotation **@EnableCaching** in the configuration file.
* Add the required **caching libraries** in the classpath.
* In the root of the classpath, add the **configuration file**for the cache provider.

For example, if we want to implement **EhCache** in an application, first we enable the cache in the configuration file.  
  


# **Spring Boot Auto Configuration and Dispatcher Servlet**

* Spring Boot automatically configures a spring application based on dependencies present or not present in the classpath as a jar, beans, properties, etc.
* It makes development easier and faster as there is no need to define certain beans that are included in the auto-configuration classes.
* A typical MVC database driven Spring MVC application requires a lot of configuration such as **dispatcher servlet, a view resolver, Jackson, data source, transaction manager,** among many others.
  + Spring Boot auto-configures a **Dispatcher Servlet** if **Spring MVC jar** is on the classpath.
  + Auto-configures the **Jackson** if **Jackson jar** is on the classpath.
  + Auto-configures a **Data Source** if **Hibernate jar** is on the classpath.
* Auto-configuration can be enabled by adding **@SpringBootApplication** or **@EnableAutoConfiguration** annotation in startup class. It indicates that it is a spring context file.
* It enables something called **auto-configuration**.
* It enable something called **components scan**. It is the features of Spring where it will start automatically scanning classes in the package and sub package for any bean file.
* Classes can be **excluded** from auto-configuration by adding:  
  @SpringBootApplication (exclude={JacksonAutoConfiguration.**class**, JmxAutoConfiguration.**class**})    
    
  Or add the following statement in the **application.properties** file.  
  spring.autoconfiguration.exclude=org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration

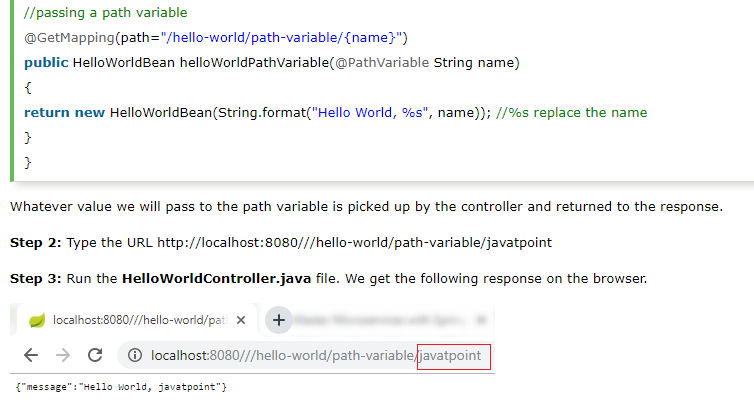
We exclude classes form the auto-configuration for **faster startup** and **better performance** of the application.

## Dispatcher Servlet

In Spring MVC all incoming requests go through a single servlet is called **Dispatcher Servlet (front controller)**. The front controller is a design pattern in web application development. A single servlet receives all the request and transfers them to all other components of the application.  
The job of DispatcherServlet is to take an incoming URI and find the right combination of **handlers** (Controller classes) and **views** (usually JSPs). When the DispatcherServlet determines the view, it renders it as the response. Finally, the DispatcherServlet returns the Response Object to back to the client.

# Path Variable

The **@PathVariable** annotation is used to extract the value from the URI. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple @PathVariable annotations in the same method. A path variable is a critical part of creating rest resources.   
This is also used where we need to fetch the particular entry based on the PathVariable ‘id’ passed in url.



# **Implementing the POST Method to create Resource**

Here we are using two annotations, **@RequestBody** and **@PathMapping**.

## @RequestBody

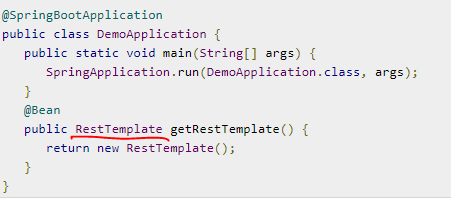
The @RequestBody annotation maps body of the web request to the method parameter. The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request.

## @PathMapping

The @PathMapping annotation is the specialized version of the **@RequestMapping** annotation which acts as a shortcut for **@RequestMapping(method=RequestMethod=POST)**. @PostMapping method handles the Http POST requests matched with the specified URI.



**Rest Template**

Rest Template is used to create applications that consume RESTful Web Services. You can use the **exchange()** method to consume the web services for all HTTP methods. The code given below shows how to create Bean for Rest Template to auto wiring the Rest Template object.  


**Steps to consume the GET/ POST/ PUT/ DELETE API’s using the Rest Template is follows:**

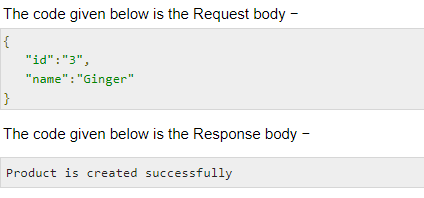
* Autowired the Rest Template Object.
* Use HttpHeaders to set the Request Headers.
* Use HttpEntity to wrap the request object.
* Provide the URL, HttpMethod, and Return type for Exchange() method.

## GET Assume this URL http://localhost:8080/products returns the following JSON and we are going to consume this API response by using Rest Template using the following code −

## POST

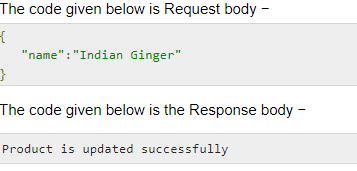
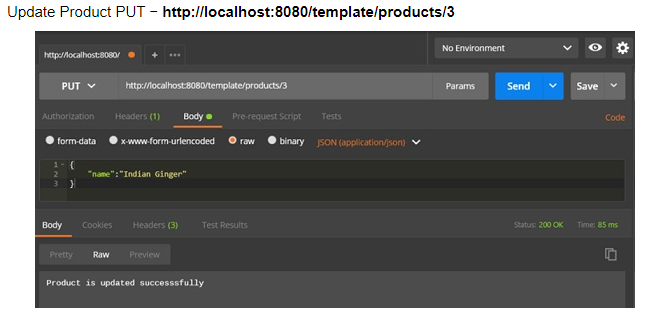
**Consuming POST API by using RestTemplate - exchange() method**

Assume this URL **http://localhost:8080/products** returns the response shown below, we are going to consume this API response by using the Rest Template. **So, Request Body inputs in form JSON format will be consumed by the REST Template and then response will be generated inside Response Body**

  
  
In above, we are creating the new product with id “3” and name “Ginger” so this is passed in Request Body of POSTMAN tool and getting output as “Product is created successfully” i.e why below we are only sending the Product as argument only because product which is being passed in request body is having both the id and name.  

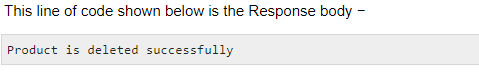
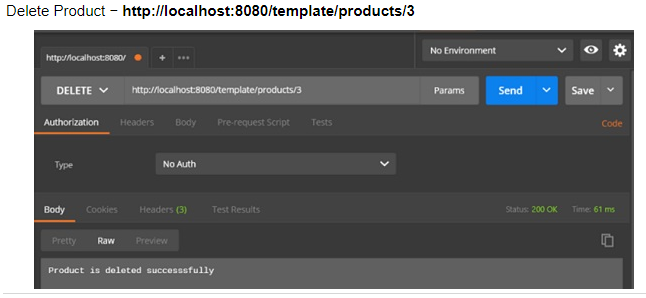

## PUT

**Consuming PUT API by using RestTemplate - exchange() method**

Assume this URL **http://localhost:8080/products/3** returns the below response and we are going to consume this API response by using Rest Template.  
  
  
In above, we are updating the exiting product with id “3” from old name “Ginger” to new name “Indian Ginger” so “Indian Ginger” is passed in Request Body of POSTMAN tool as product parameter and we are also passing the id “3” in the URL as path variable also. So here we are passing the product in request body and id in path variable and getting output as “Product is updated successfully.  
i.e. there are two values being passed in HttpEntity(product ,headers)  


## DELETE

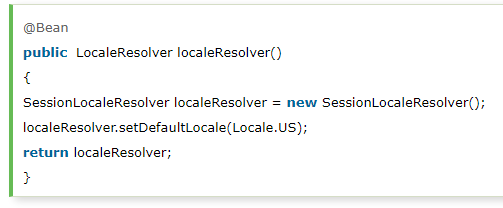
**Consuming DELETE API by using RestTemplate - exchange() method**

Assume this URL **http://localhost:8080/products/3** returns the response given below and we are going to consume this API response by using Rest Template.  
  
  
In above, we are deleting the exiting product with id “3” so we are only passing the path variable only.  
  


**Internationalization**

Internationalization is a process that makes your application adaptable to different languages and regions without engineering changes on the source code. We need to configure two things to make the service internationalized.

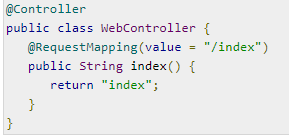
* LocaleResolver
* ResourceBundleMessageSource

Default Locale is Locale.**US**. If somebody does not specify the location, it returns the default locale. We also need to customize the ResourceBundle. It has a list of properties that are to be internationalized. We will store the properties in ResourceBundle.  


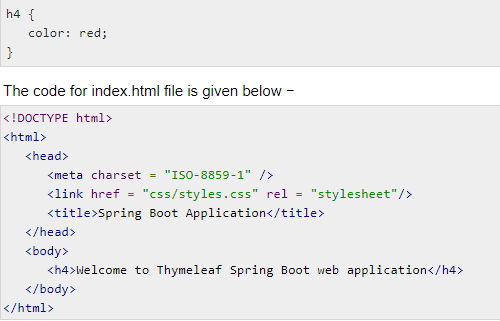
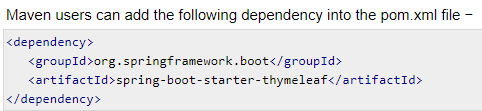
**Thymeleaf**Thymeleaf is a Java-based library used to create a web application. It provides a good support for serving a XHTML/HTML5 in web applications. Thymeleaf converts your files into well-formed XML files. It contains 6 types of templates as given below −

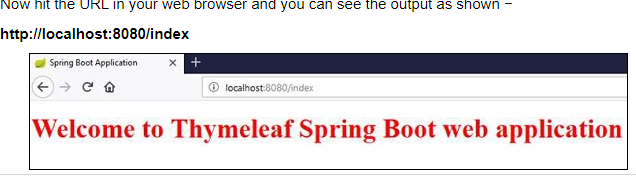
* XML
* Valid XML
* XHTML
* Valid XHTML
* HTML5

Use Thymeleaf templates to create a web application in Spring Boot. You will have to follow the below steps to create a web application in Spring Boot by using Thymeleaf.

Use the following code to create a @Controller class file to redirect the Request URI to HTML file −  


In the above example, the request URI is /index, and the control is redirected into the index.html file. Note that the index.html file should be placed under the templates directory and all JS and CSS files should be placed under the static directory in classpath. In the example shown, we used CSS file to change the color of the text.

You can use the following code and created a CSS file in separate folder css and name the file as styles.css.  
  
  
  
Below, the text is coming in red color as it is picking the css style and printing the output in index.html



**Spring Boot DataSource configuration**

A datasource is a factory for connections to any physical data source. An object that implements the [javax.sql.DataSource](https://docs.oracle.com/javase/10/docs/api/javax/sql/DataSource.html) interface will typically be registered with JNDI service and can be discovered using it’s JNDI name. A datasource may be used to obtain :

* standard Connection object
* connection which can be used in connection pooling

Spring boot allows defining datasource configuration in both ways i.e. Java config and properties config. DataSourceAutoConfiguration checks for DataSource.class on the classpath and few other things before configuring a DataSource bean for us.

#### **Using : application.properties** DataSource configuration is provided by external configuration properties ( spring.datasource.\* ) in application.properties file. For eg- to configure the MYSQL DB

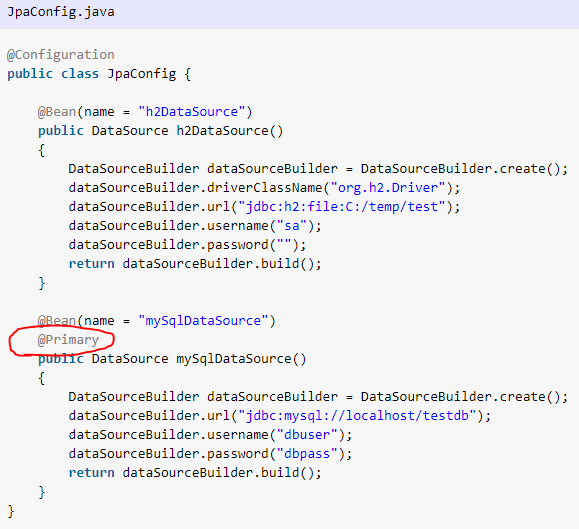
#### **Using: DataSource Bean** Recommended way to create DataSource bean is using **DataSourceBuilder class** within a class annotated with the @Configuration annotation. The datasource uses the underlying connection pool as well.

## Connection Pooling

For a pooling DataSource to be created, Spring boot verifies that a valid Driver class is available. If we set **spring.datasource.driver-class-name** property then that mentioned driver class has to be loadable.

The auto-configuration first tries to find and configure **HikariCP**. If HikariCP is available, it always choose it. Otherwise, if the **Tomcat pooling** is found, it is configured. If neither HikariCP nor the Tomcat pooling datasource are available and if **Commons DBCP2** is available, it is used.  
  
HikariCP comes inbuilt with spring-boot-starter-jdbc or spring-boot-starter-data-jpa starters.

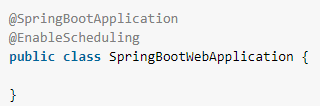
## Multiple Datasources with Spring boot

To configure multiple data sources, create as many bean definitions you want but mark one of the DataSource instances as **@Primary**, because various auto-configurations down the road expect to be able to get one by type.  


We can configure multiple datasources and one of them must be marked as @Primary. Primary datasource is autowired by default, and other datasources need to be autowired along with @Qualifier annotation.

# **Spring Boot – Scheduling**

Scheduling is a process of executing the tasks for the specific time period. Spring Boot provides a good support to write a scheduler on the Spring applications. To schedule job in spring boot application to run periodically, spring boot provides [**@EnableScheduling**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableScheduling.html) and [**@Scheduled**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html) annotations.

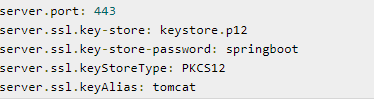
**1. Add @EnableScheduling to Spring Boot Application class**Add **@EnableScheduling** annotation to your spring boot application class. @EnableScheduling is a Spring Context module annotation. It internally imports the [SchedulingConfiguration](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/SchedulingConfiguration.html)  


## 2. Add Spring boot @Scheduled annotations to methods Now you can add @Scheduled annotations on methods which you want to schedule. Only condition is that methods should be without arguments. So the time will be printed after every 10 miliseconds of gap because fixedRate =10,000 i.e 10 miliseconds. Fixed Rate scheduler is used to execute the tasks at the specific time. It does not wait for the completion of previous task. The values should be in milliseconds. Here fixedRate =1000 ie. 1 miliseconds of gap

# **Spring Boot - Enabling HTTPS**

By default, Spring Boot application uses HTTP 8080 port when the application starts up.  
The steps given below to configure the HTTPS and the port 443 in Spring Boot application −

* Obtain the SSL certificate – Create a self-signed certificate or get one from a Certificate Authority
* Enable HTTPS and 443 port

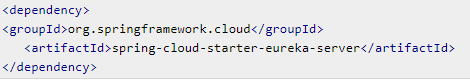
We need to provide the server port as 443, key-store file path, key-store-password, key-store-type and key alias name into the application.properties file.  


To create a self-signed certificate (keystore.p12), Java Run Time environment comes bundled with certificate management utility key tool. This utility tool is used to create a Self-Signed certificate.  
  
**Redirect HTTP requests to HTTPS**This is an optional step in case you want to redirect your HTTP traffic to HTTPS, so that the full site becomes secured. To do that in spring boot, we need to add HTTP connector at 8080 port and then we need to set redirect port 8443. So that any request in 8080 through http, it would be automatically redirected to 8443 and https.

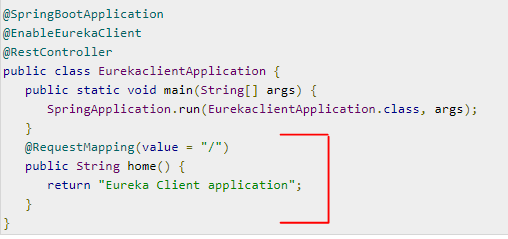
# **Spring Boot - Eureka Server**

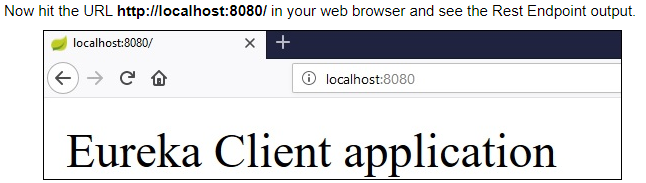
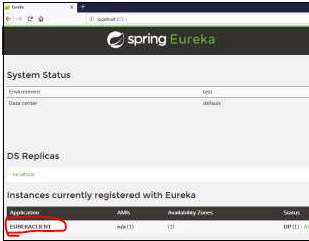
Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

Eureka Server comes with the bundle of Spring Cloud. For this, we need to develop the Eureka server and run it on the default port 8761.

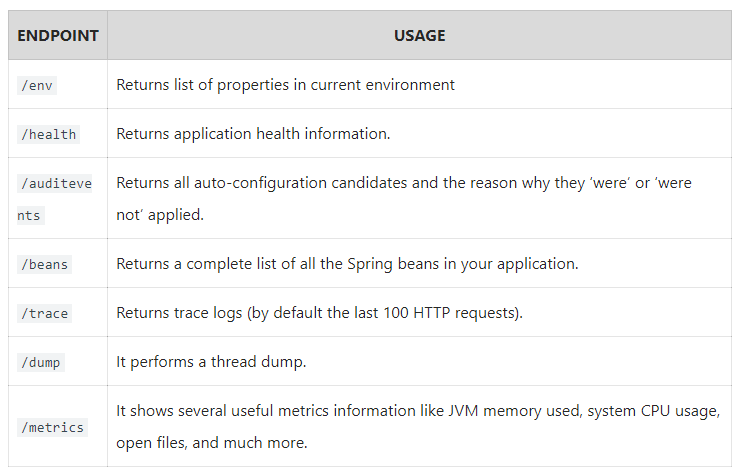
The **@EnableEurekaServer** annotation is used to make your Spring Boot application acts as Eureka Server.  
  
  
Make sure Spring cloud Eureka server dependency is added in your build configuration file.  
  
After application is build and started, You can find that the application has started on the Tomcat port 8761.  
Now, hit the URL **http://localhost:8761/** in your web browser and you can find the Eureka Server running on the port 8761

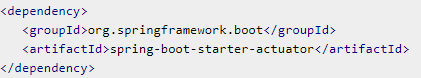
# **Service Registration with Eureka**

How to register the Spring Boot Micro service application into the Eureka Server. Before registering the application, please make sure Eureka Server is running on the port 8761 or first build the Eureka Server and run it. Steps are as follows:  
1. Add the maven dependency of eureka server in (same as above) and also spring boot starter web.  
  
2. Add the @EnableEurekaClient annotation in the main Spring Boot application class file. The @EnableEurekaClient annotation makes your Spring Boot application act as a Eureka client.  
  
  
3. Register the Spring Boot application into Eureka Server by configuration in our application.properties file  
  
  
4. Add the Rest Endpoint to return String in the main Spring Boot application  


Now, the application has started on the Tomcat port 8080 and Eureka Client application is registered with the Eureka Server.  
Hit the URL **http://localhost:8761/** in your web browser and you can see the Eureka Client application is registered with Eureka Server.  
 

# **Spring Boot – Actuator**

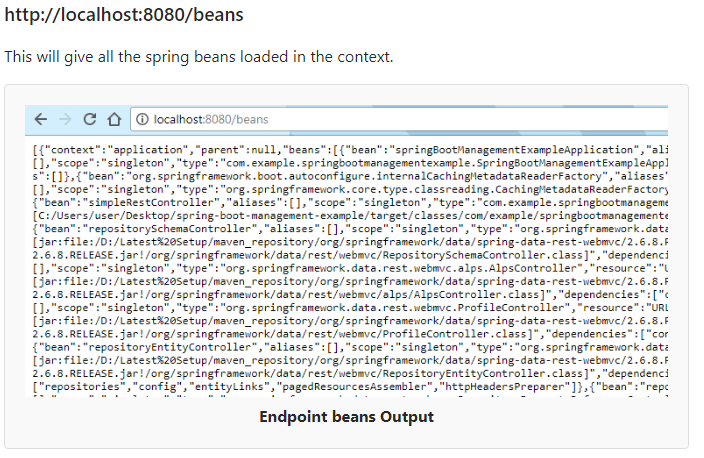
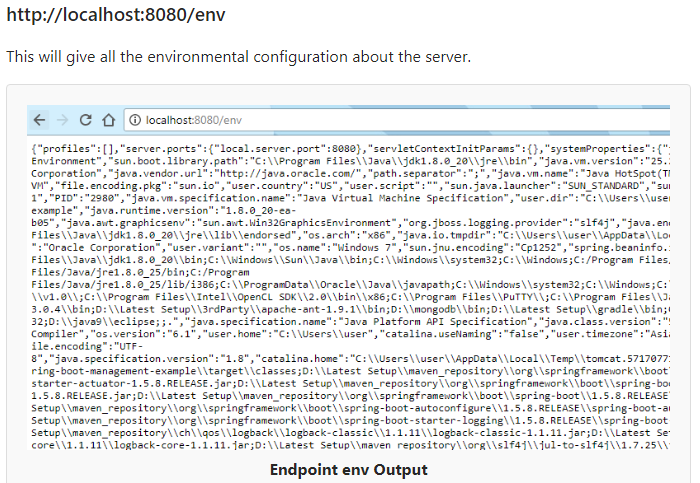
Spring Boot Actuator provides secured endpoints for monitoring and managing your Spring Boot application. By default, all actuator endpoints are secured. These monitoring and management information is exposed via [REST](https://restfulapi.net/) like endpoint URLs.  


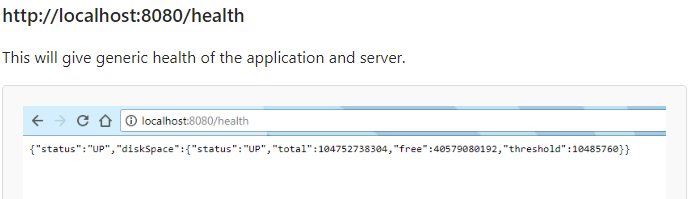
1. To enable Spring Boot actuator endpoints to your Spring Boot application, we need to add the Spring Boot Starter actuator dependency in our build configuration file.  


2. In the application.properties file, we need to disable the security for actuator endpoints. By default, [spring security](https://howtodoinjava.com/spring-security-tutorial/) is enabled for all actuator endpoints. It is inbuilt [form-based authentication](https://howtodoinjava.com/spring/spring-boot/role-based-security-jaxrs-annotations/) with the userid as the user and a randomly generated password.   


3. Now add one simple Rest endpoint /example to the application  


4. Run the application and access **/example** API in browser to generate monitoring information on server.





# **Spring Boot – Get all loaded beans**

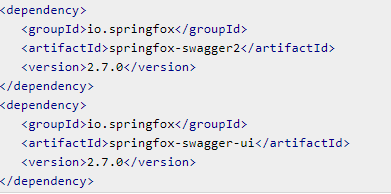
Spring boot loads lots of beans internally to run your application with minimal configuration

[CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html) is used to indicate that a bean should run when it is contained within a Spring Application.

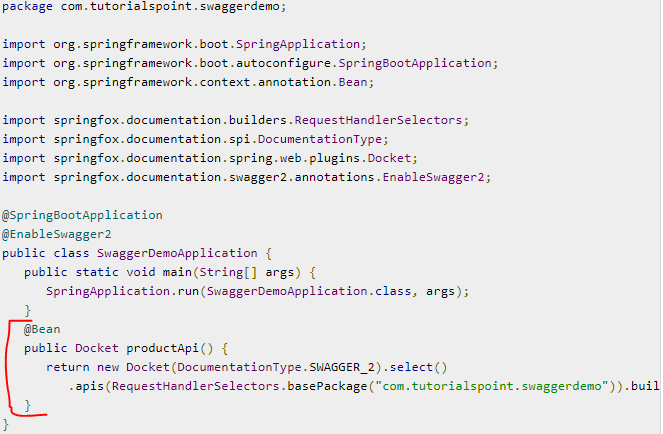
1) Use **ApplicationContext.getBeanDefinitionNames()** to find the name of all loaded beans  
2) Use **ApplicationContext.getBean(beanName)** to get bean including its runtime type information.  
  
Running above application will print bean names and type information in console

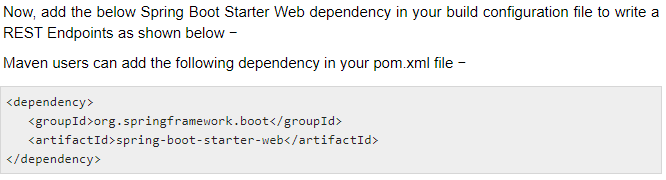
# **Spring Boot - Enabling Swagger2**

Swagger2 is an open source project used to generate the REST API documents for RESTful web services. It provides a user interface to access our RESTful web services via the web browser.

1. To enable the Swagger2 in Spring Boot application, you need to add the following dependencies in our build configurations file.  


2. Now, add the @EnableSwagger2 annotation in your main Spring Boot application. The @EnableSwagger2 annotation is used to enable the Swagger2 for your Spring Boot application.

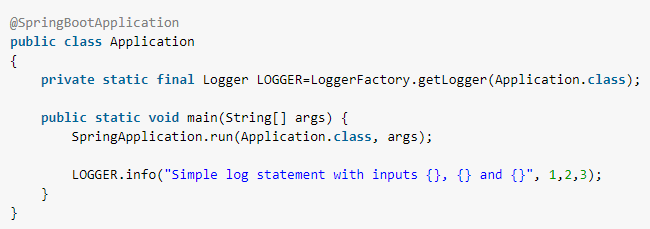
3. Next, create Docket Bean to configure Swagger2 for your Spring Boot application. We need to define the base package to configure REST API(s) for Swagger2.  




4. Now, the code to build two simple RESTful web services GET and POST in Rest Controller fileka  


5. Now, run the main java file and then hit the URL in your web browser and see the Swagger API screen  
[**http://localhost:8080/swagger-ui.html**](http://localhost:8080/swagger-ui.html)  


# **Logging in Spring Boot**

**Logging in spring boot** is very flexible and easy to configure. Spring boot supports various logging providers through some simple configuration. Spring boot active enabled logging is determined by [spring-boot-starter-logging](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-logging/pom.xml) artifact.   
If we do not provide any logging specific configuration, we will still see logs printed in “console”. These are because of **default logging support** provided in spring boot which uses **Logback**.  


Logback supports **ERROR, WARN, INFO, DEBUG**, or **TRACE** as logging level. By default, logging level is set to **INFO**. It means that code>DEBUG and TRACE messages are not visible. To enable debug or trace logging, we can set the logging level in application.properties file.  


#### **Exclude logback and include log4j2** As mentioned earlier, spring boot uses logback as default. So if we have to use any other logging framework e.g. log4j2, we must **exclude logback** from classpath of the application. Also, add **spring-boot-starter-log4j2** to pom.

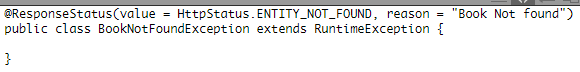


## Spring Boot - Exception Handling Controller Advice The @ControllerAdvice is an annotation, to handle the exceptions globally. A controller advice allows you to use exactly the same exception handling techniques but apply them across the whole application, not just to an individual controller.

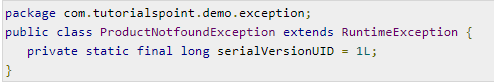
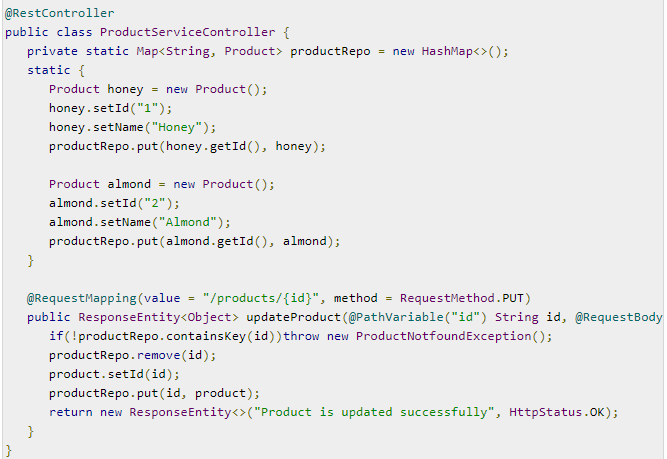
## 

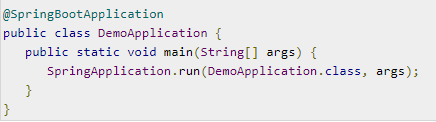
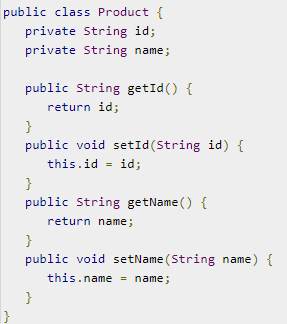
## Exception Handler The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client. Common approaches of Exception handling provided by Spring

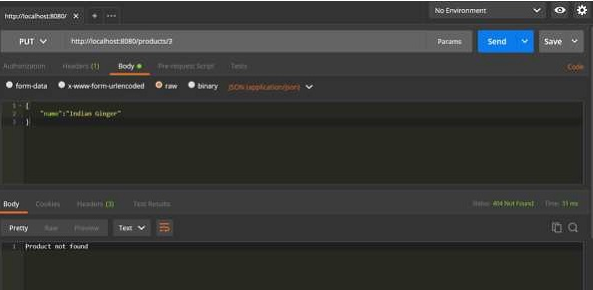
Having a proper exception handling approach is a essential for any application. The Spring MVC Framework provides the below smart approaches to provide exception handling. We will go through some of the commonly used Spring annotations used for exception handling.

* **@ResponseStaus annotation** – This is the straight forward way provided by Spring to respond the status of HTTP response by providing a status code. This annotation can be used on methods and classes.  
  
* **Controller Based** – Spring provides many annotations to handle exceptions. In case of defining exception handler methods of controller classes, we can annotate these methods with @ExceptionHandler annotation. By doing this, Spring provides mechanism to handle exceptions that are thrown while executing controller classes. This will be used as an entry point for handling exceptions that are thrown within the controller class.
* **Global Exception Handler** – The handler methods in Global Controller Advice is similar to Controller based exception handler methods and are used when the controller class is not able to handle an exception. It is essential to provide exception handling for all pointcuts in an application, globally. Spring provides @ControllerAdvice annotation to define global exception handler to any class. The @ControllerAdvice annotation takes Exception class as it’s argument. This annotation was introduced in Spring 3.2. In accordance with it’s name, the @ControllerAdvice annotation is used to provide a single ExceptionHandler for multiple controllers.
* **Custom Error Handler** – The Spring Framework provides ResponseEntityExceptionHandler interface that can be implemented to create custom exception handler. The reason of having this additional custom exception handler is that Spring provides default classes that can be implemented in our spring bean configuration to get the benefits of exception handling. We can also override it to create our custom handler.

# [**@RestControllerAdvice**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestControllerAdvice.html) is a new annotation used to provide a common **ExceptionHandler** code for multiple controllers. It is normally used along with [**@ExceptionHandler**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html)**.** **Example:**

The complete code to handle the exception is given below. In this example, we used the PUT API to update the product. Here, while updating the product, if the product is not found, then return the response error message as “Product not found”. Note that the **ProductNotFoundException** exception class should extend the **RuntimeException**.  
  
  
The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.  
  
The Product Service API controller file is given below to update the Product. If the Product is not found, then it throws the **ProductNotFoundException** class.  


The code for main Spring Boot application class file is given below −  
  
  
The code for **POJO class** for Product is given below −  
  
Now hit the below URL in POSTMAN application and you can see the output as shown below −

Update URL: [**http://localhost:8080/products/3**](http://localhost:8080/products/3)

# Spring Boot – Interceptor

You can use the Interceptor in Spring Boot to perform operations under the following situations −

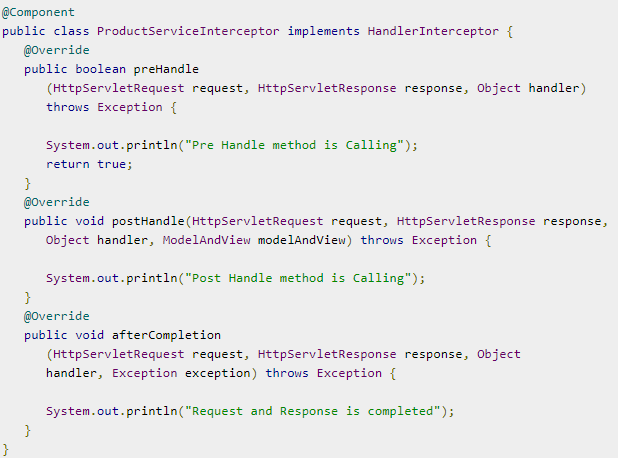
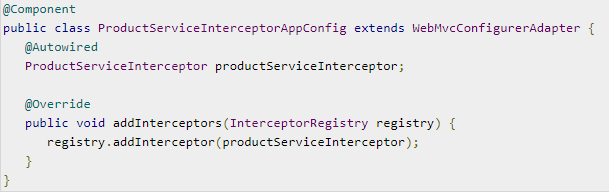
* Before sending the request to the controller
* Before sending the response to the client

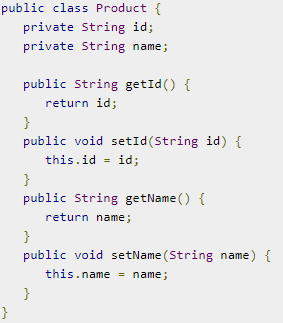
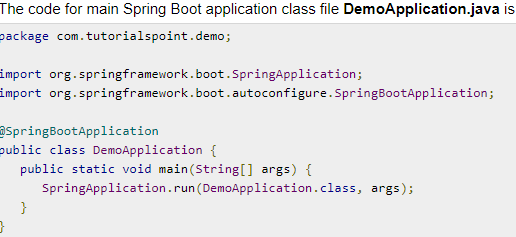
For example, you can use an interceptor to add the request header before sending the request to the controller and add the response header before sending the response to the client.

To work with interceptor, you need to create **@Component** class that supports it and it should implement the**HandlerInterceptor** interface.

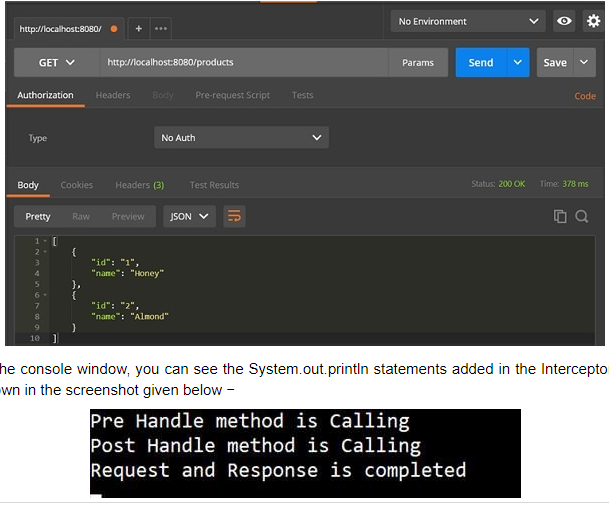
The following are the three methods you should know about while working on Interceptors −

* **preHandle()** method − This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
* **postHandle()** method − This is used to perform operations before sending the response to the client.
* **afterCompletion()** method − This is used to perform operations after completing the request and response.

  
You will have to register this Interceptor with **InterceptorRegistry** by using **WebMvcConfigurerAdapter** as shown below −  


The code for Controller class file ProductServiceController.java is given below −  
  
The code for POJO class for Product.java is given below −  
  


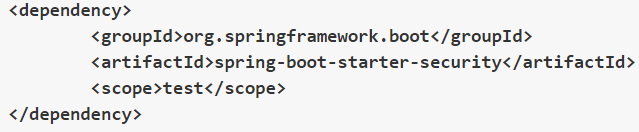
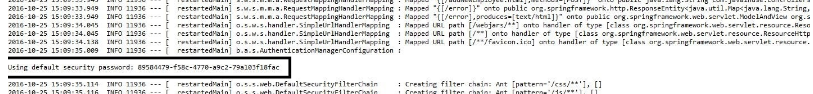
Now hit the below URL in POSTMAN application and you can see the output as shown under −

GET API: [**http://localhost:8080/products**](http://localhost:8080/products)

Spring Boot – FreeMarker Template  
  
[**Apache FreeMarker™**](https://freemarker.apache.org/) is a free Java-based template engine. It's is a general purpose template engine, with no dependency on servlets or HTTP or HTML. It used for generating HTML web pages (dynamic web page generation), e-mails, configuration files, source code, etc

Templates are written in the FreeMarker Template Language (FTL), which is a simple and specialized language. Java is used to prepare the data (issue database queries, do business calculations). Then, FreeMarker displays that prepared data using templates. In the template you are focusing on how to present the data, and outside the template you are focusing on what data to present. Similar like most of MVC approach.

1. Add **spring-boot-starter-web** and **spring-boot-starter-freemarker** into your project as a dependency on pom.xml   
2. Create the freemarker template under the resources/templates folder with <filename>.ftl  
3. Spring controller which handle require and return String which is name of freemarker template name which is available in resources/templates folder with <filename>.ftl

Spring Boot – Security Form Login Authentication  
  
1. In the Maven we only need the **spring-boot-starter-security** dependency  
  
  
2. So without adding nay security code when you Go to-**http://localhost:8080/**We can see that no security configuration is added still it asks for username password. This is expected behaviour. We will see that the password is system generated when we run the boot application.  
  
  
  
3.  The security configuration by writing our own authorization and authentication. For this create a new class SecurityConfig that extends the **WebSecurityConfigurerAdapter** and overrides its methods.  


# Spring Boot – Runners

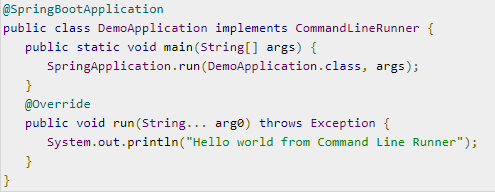
**Application Runner** and **Command Line Runner** interfaces lets you to execute the code after the Spring Boot application is started. You can use these interfaces to perform any actions immediately after the application has started.

## Application Runner

Itwill execute the run() method, just after applicationcontext is created and before springboot application startup. ApplicationRunner has **run() method** with argument **ApplicationArgument**. It has convenient methods like getOptionNames(), getOptionValues() and getSourceArgs().



## Command Line Runner

CommandLineRunner will execute run() method, just after applicationcontext is created and before springboot application starts up.  


Both provide the same functionality and the only difference between CommandLineRunner and ApplicationRunner is CommandLineRunner.run() accepts **String array[]** whereas ApplicationRunner.run() accepts **ApplicationArguments as argument**.

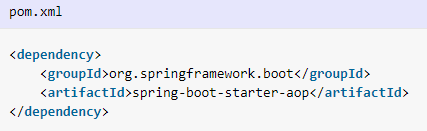
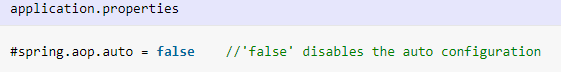
# Spring-boot-starter-parent and Templates The **spring-boot-starter-parent** dependency is the parent POM providing dependency and plugin management for Spring Boot-based applications. It contains the default versions of Java to use, the default versions of dependencies that Spring Boot uses, and the default configuration of the Maven plugins.

The spring-boot-starter-parent dependency further inherits from spring-boot-dependencies.  
  
**How starter template work?**

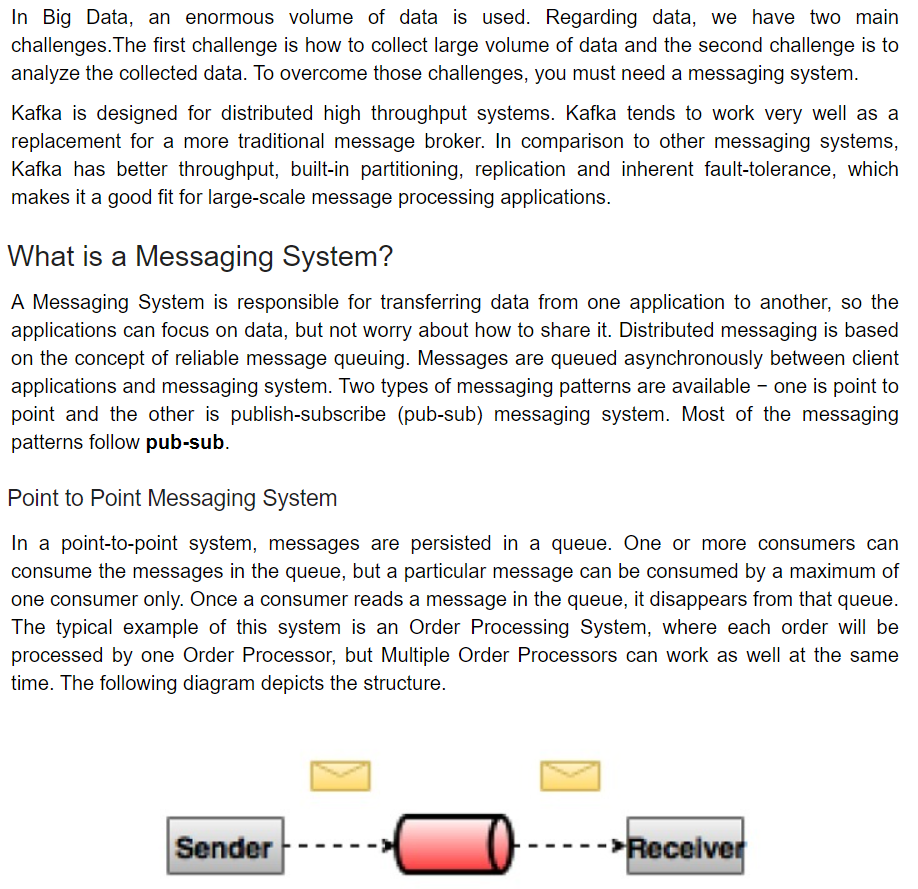
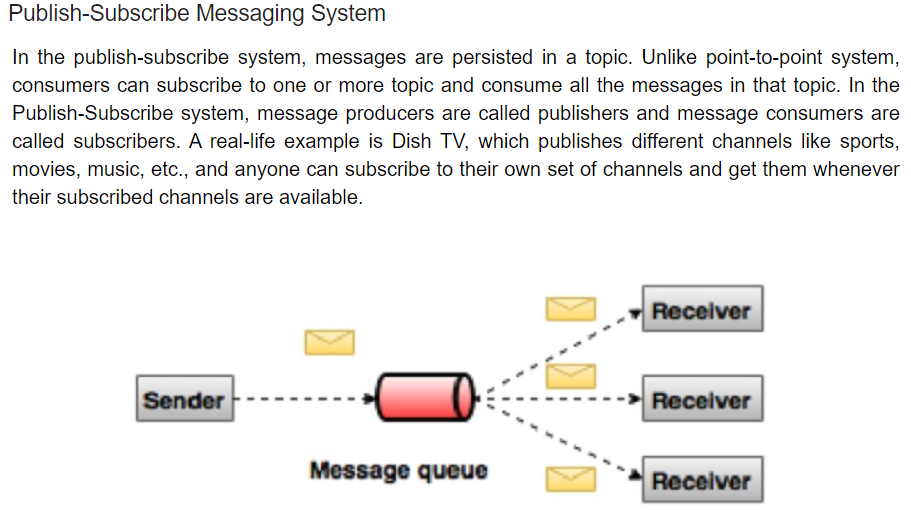
Spring Boot starters are templates that contain a collection of all the relevant transitive dependencies that are needed to start a particular functionality. Each starter has a special file, which contains the list of all the provided dependencies Spring provides.

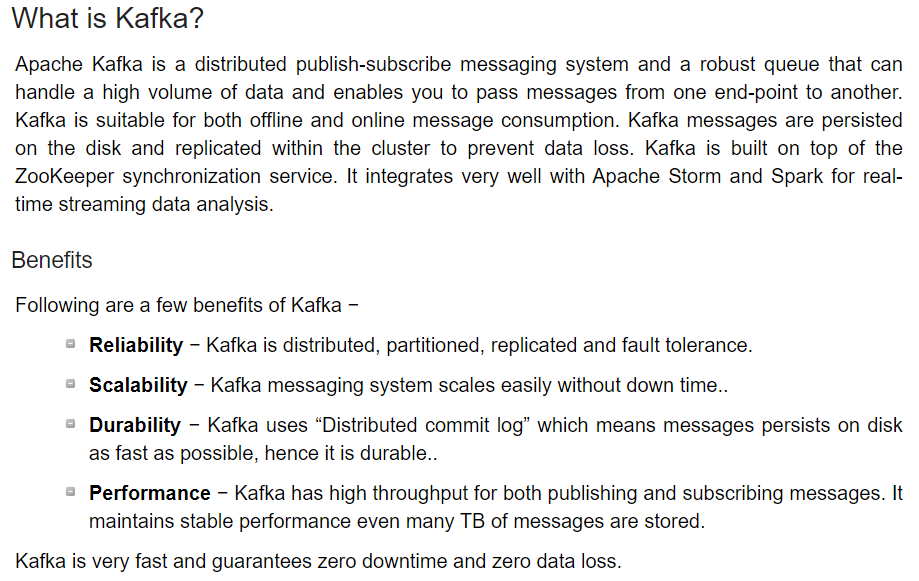
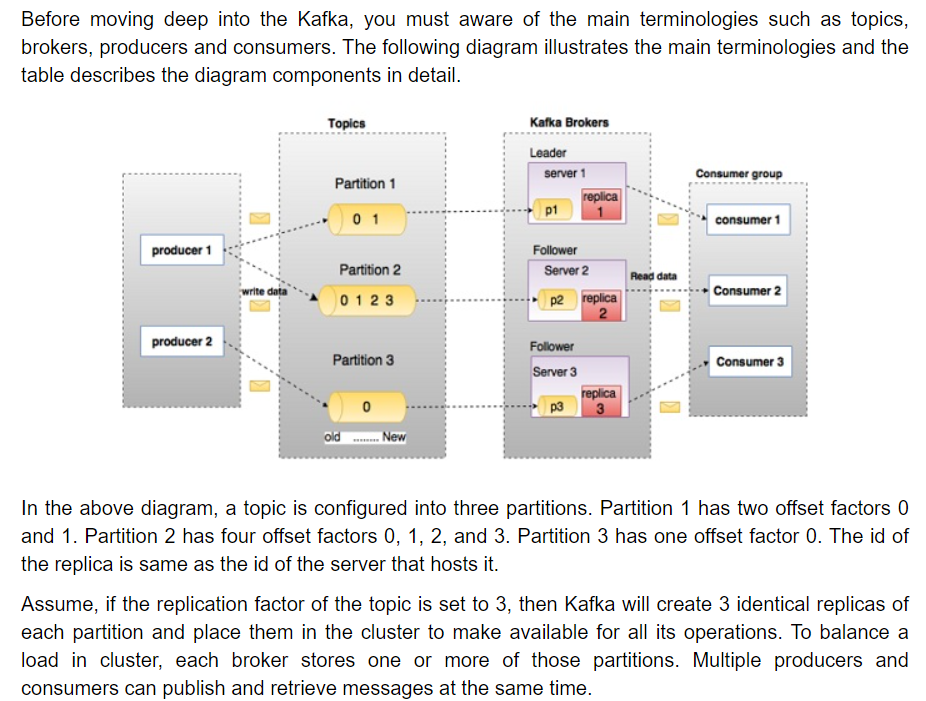
These files can be found inside pom.xml files in respective starter module.

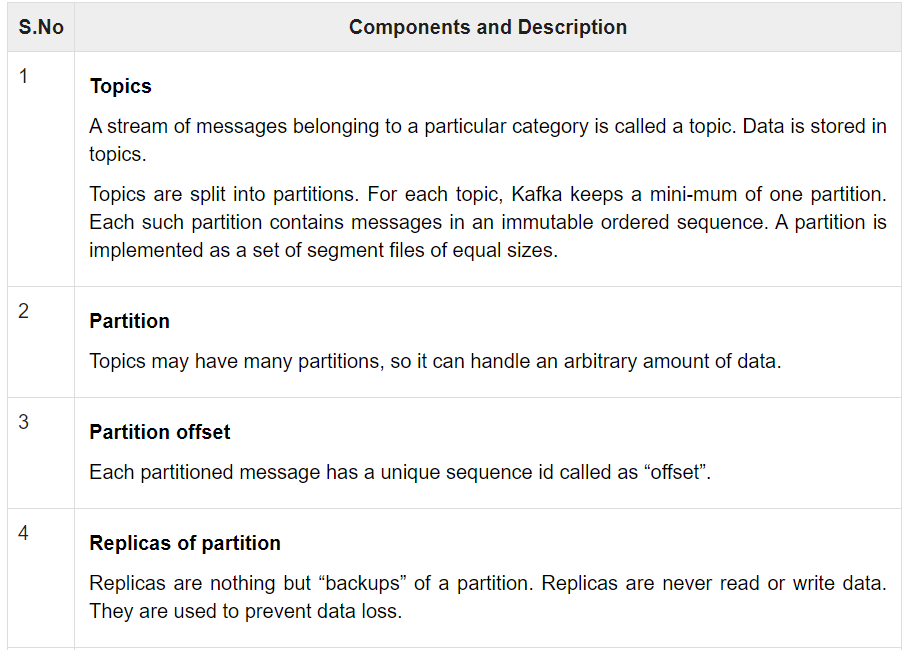
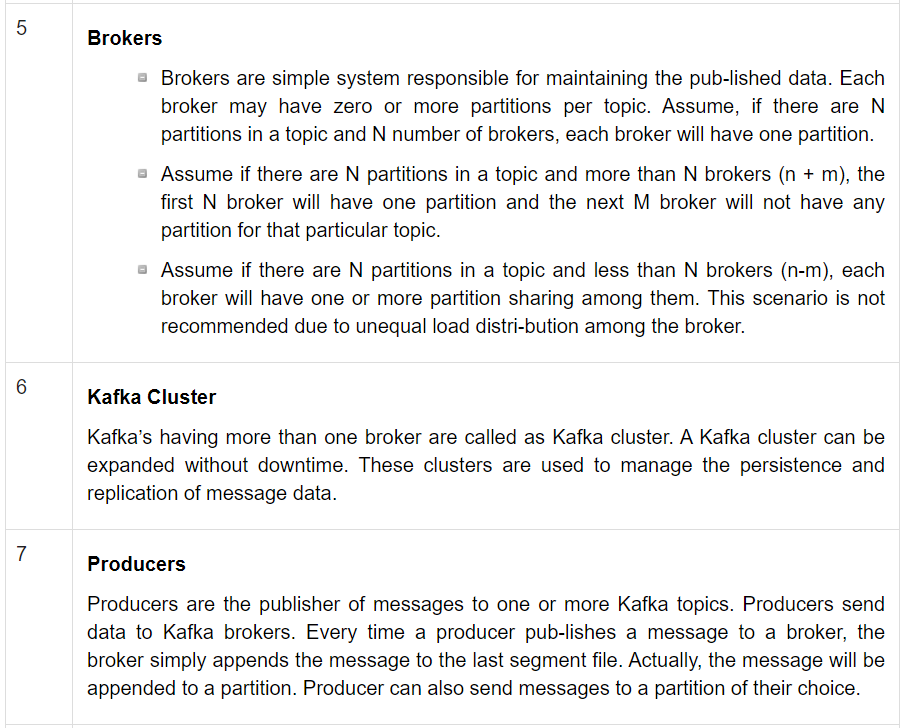
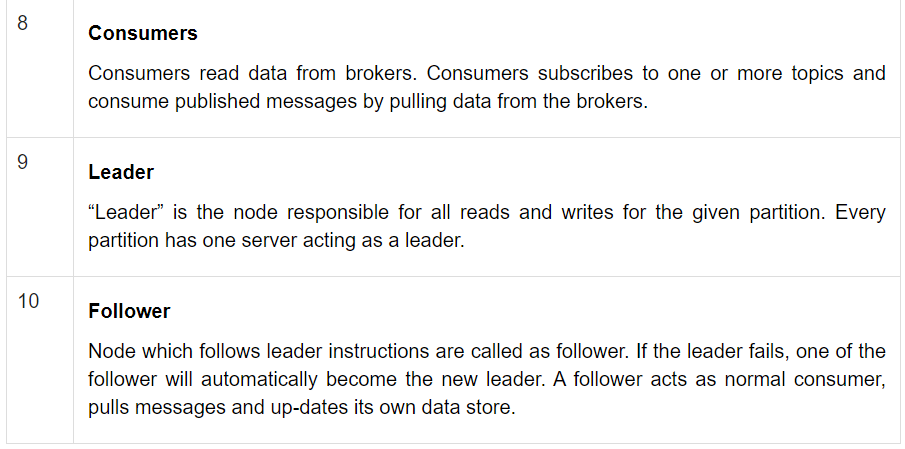
This tells us that by including spring-boot-starter-data-jpa in our build as a dependency, we will automatically get spring-orm, hibernate-entity-manager and spring-data-jpa. These libraries will provide us all basic things to start writing JPA/DAO code.  

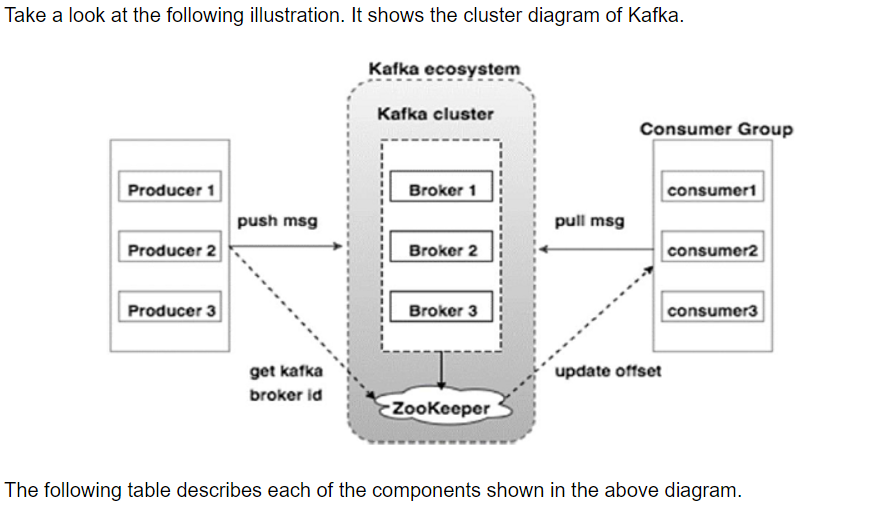

Spring boot- AOP  
  
Implement AOP in Spring Boot applications and add different aop advices using [AspectJ](https://www.eclipse.org/aspectj/" \t "_blank) to support cross-cutting concerns, such as logging, profiling, caching, and transaction management.  
Steps to configure:  
  
1. Setting up AOP in spring boot requires including **spring-boot-starter-aop** dependency. It imports spring-aop and aspectjweaver dependencies into the application.   
  
  
2. **Enable/disable auto configuration**Importing above dependencies triggers **[AopAutoConfiguration](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/aop/AopAutoConfiguration.html)** which enables AOP using @EnableAspectJAutoProxy annotation.  
The auto configuration will NOT be activated if spring.aop.auto = false in application.properties.  


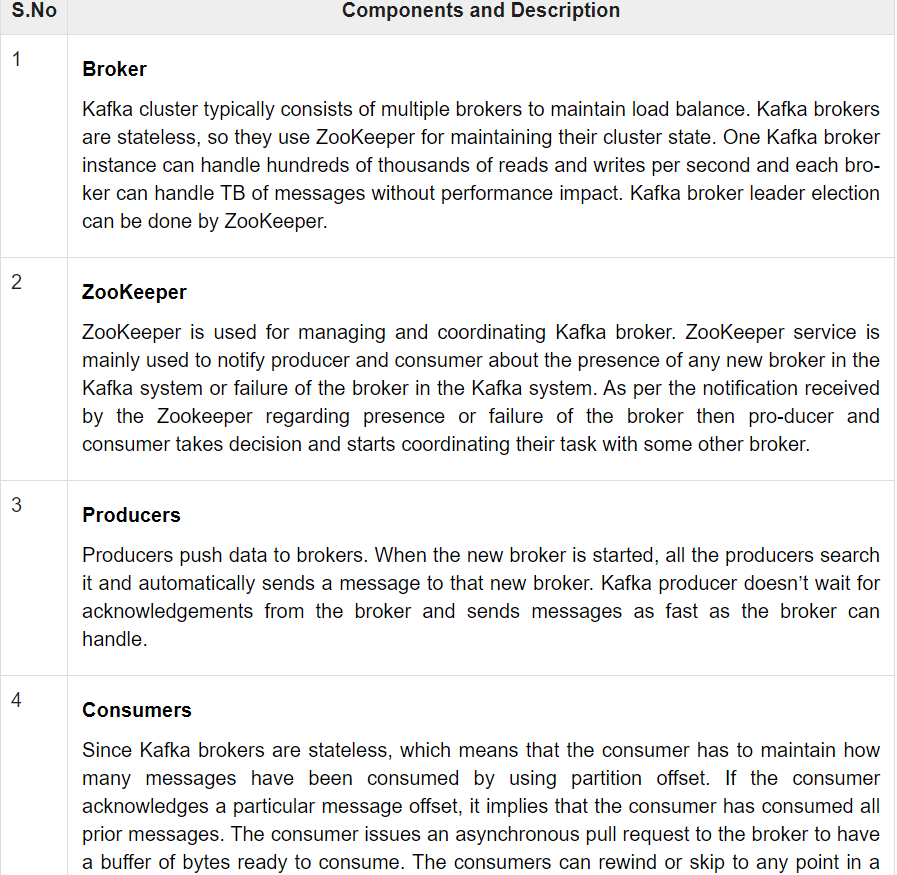
*3.***Creating aspects with @Aspect**An aspect can be created in spring boot with help of annotations @Aspect annotation and registering with bean container with @Component annotation. Inside aspect class, we can create advices as required.

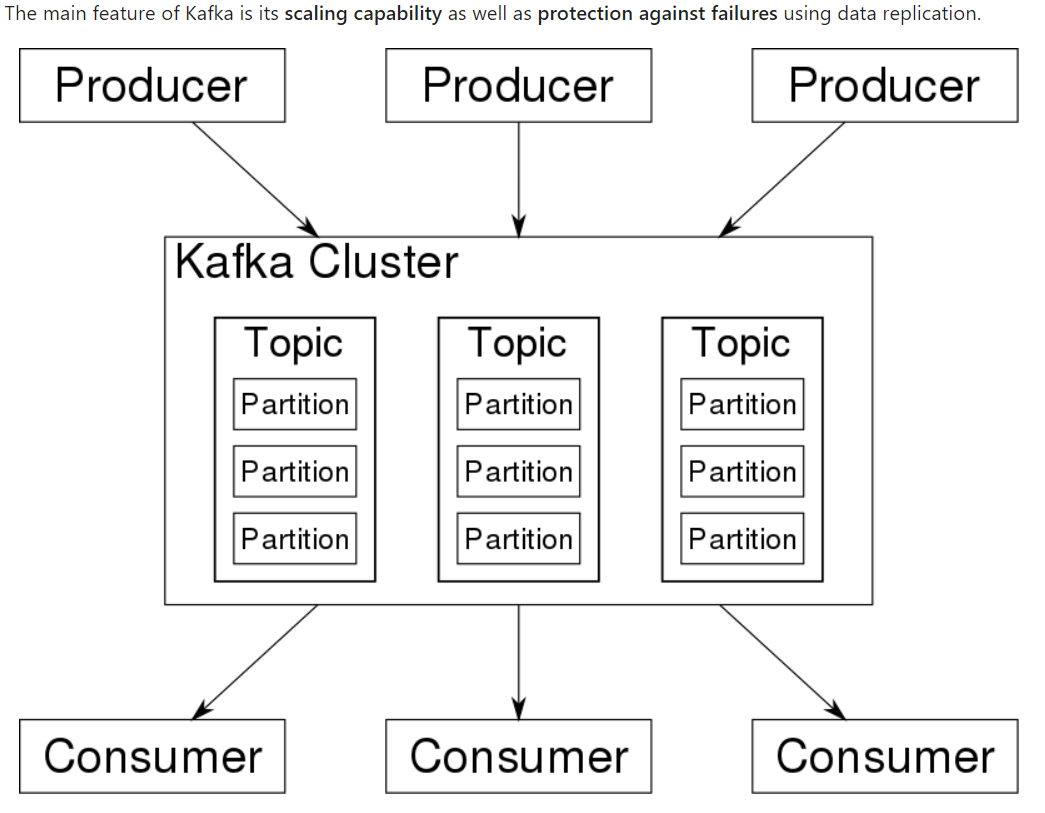
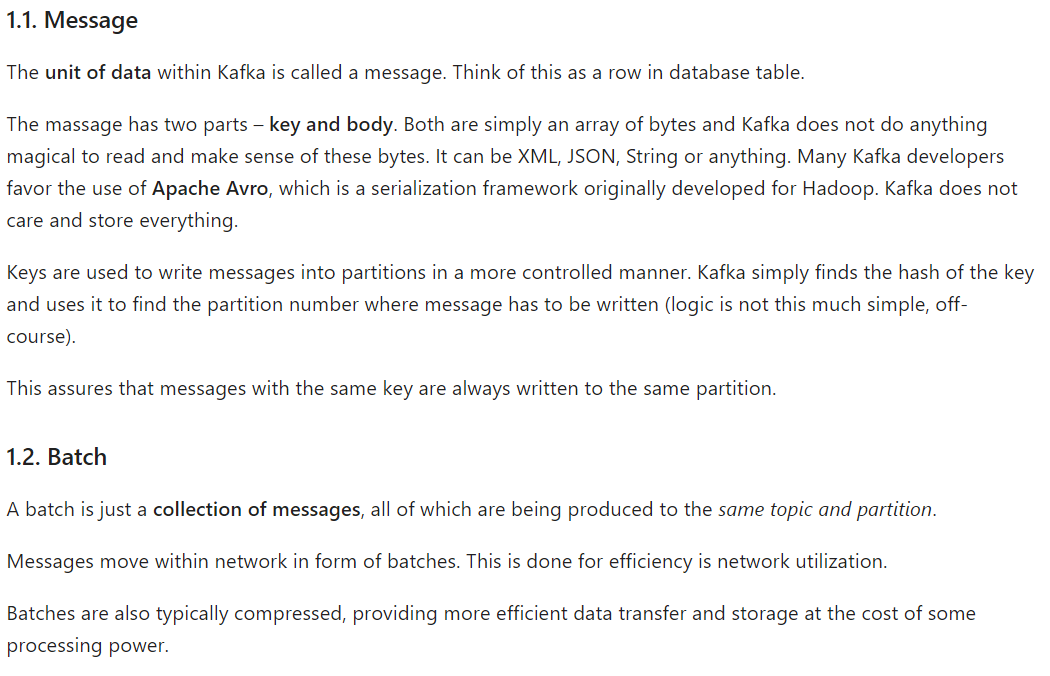
Apache Kafka:  
  
  
  


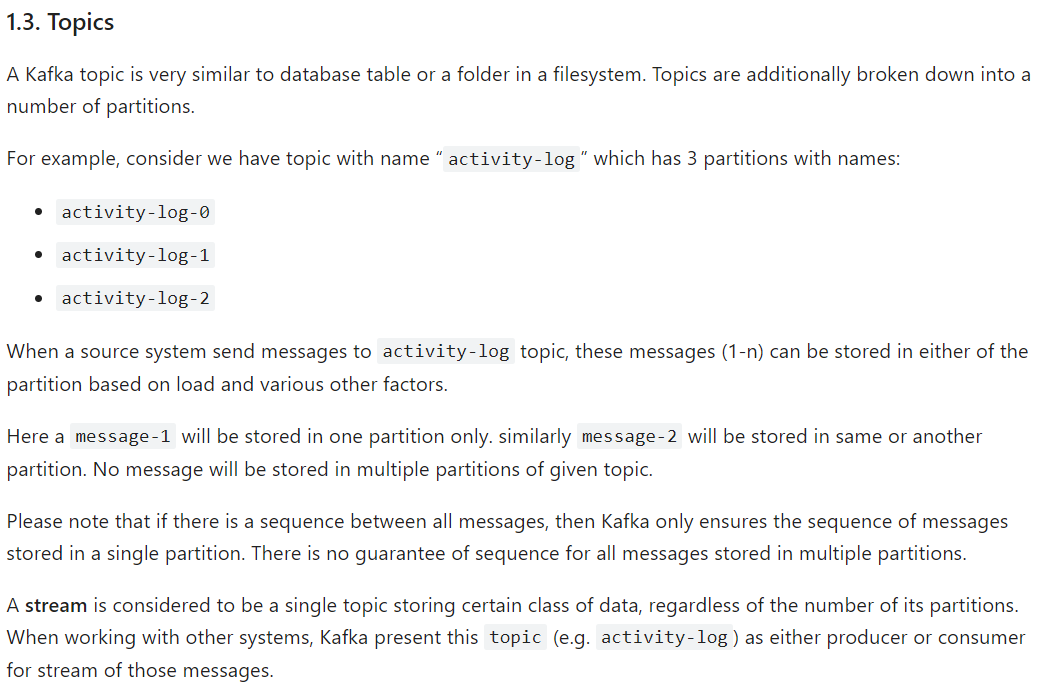
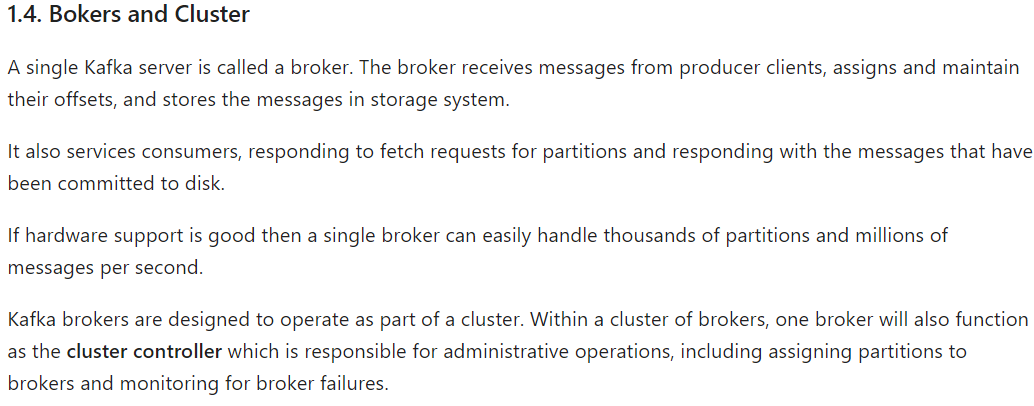
  
  








ZUUL Edge Server and Routing

