

**Spring IOC Container:**

* Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.
* **If we want to add our own behaviour, we need to extend the classes of the framework or plugin our own classes.**
* We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).
* The interface ApplicationContext represents the IoC container.
* The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles.
* **Why Inversion of control:**

1. **Loose Coupling:** IoC helps in achieving loose coupling between components in a system.
2. **Testability:** you can mock or substitute dependencies when testing a particular component, making unit testing more straightforward.
3. **Flexibility & Maintainability:** easier to change, replace, or upgrade individual components without affecting the entire system.
4. **Separation of Concerns:** IoC helps in separating the concerns of a system. Components focus on their specific functionality, and the responsibility of managing and injecting dependencies is delegated to the IoC container.
5. **Easier Configuration:** In Spring, this configuration is typically done using XML, Java annotations, or Java configuration classes. This makes it easier to modify the configuration without changing the actual code, promoting a cleaner separation of configuration and implementation.
6. **AOP (Aspect-Oriented Programming) Integration:** IoC is closely related to AOP. AOP allows for the modularization of cross-cutting concerns such as logging, transaction management, security, etc. Spring combines IoC with AOP to provide a comprehensive solution for building enterprise-level applications.

* The Spring framework provides several implementations of *the*ApplicationContext*interface:*ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications, *and*WebApplicationContext*for web applications.*

**EX:**

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

**Dependency Injection:**

* Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.
* Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.
* **Dependency Injection in Spring can be done through constructors, setters or fields.**

## ****Constructor-Based Dependency Injection:****

## the container will invoke a constructor with arguments each representing a dependency we want to set.

## Spring resolves each argument primarily by type, followed by name of the attribute, and index for disambiguation.

* We use the @Bean annotation on a method to define a bean. If we don't specify a custom name, then the bean name will default to the method name.
* For a bean with the default singleton scope, Spring first checks if a cached instance of the bean already exists, and only creates a new one if it doesn't. If we're using the prototype scope, the container returns a new bean instance for each method call.

## ****Setter-Based Dependency Injection:****

## For setter-based DI, the container will call setter methods of our class after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

## The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones because:

## Constructor-based injection enforces that all mandatory dependencies are provided at object creation time, ensuring that the object is in a valid and usable state from the moment it's created.

## Setter-based injection allows for more flexible initialization, but it also increases the risk of forgetting to set essential dependencies before using the object.

## ****Field-Based Dependency Injection:****

## In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:

## public class Store {

## @Autowired

## private Item item; }

## While constructing the Store object, if there's no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.

## ****Autowiring Dependencies****:

* [Wiring](https://www.baeldung.com/spring-annotations-resource-inject-autowire) allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.
* We can use the @Autowired to **mark a dependency which Spring is going to resolve and inject**.
* We can use this annotation with a constructor, setter, or field injection.
* Instead of using @Required, you can use constructor injection or @Autowired(required = true) to achieve similar functionality with more modern Spring features.

**by declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans**. This is called **Spring bean autowiring**.

Constructor injection:

**class** **Car** {

Engine engine;

@Autowired

Car(Engine engine) {

this.engine = engine;

}

}

Setter injection:

**class** **Car** {

Engine engine;

@Autowired

**void** **setEngine**(Engine engine) {

this.engine = engine;

}

}

Field injection:

**class** **Car** {

@Autowired

Engine engine;

}

## @Autowired has a boolean argument called required with a default value of true.

## If we use constructor injection, all constructor arguments are mandatory.

### **@Bean:**

## @Bean marks a factory method which instantiates a Spring bean:

## @Bean

## Engine engine() {

## return new Engine(); }

## **Spring calls these methods** when a new instance of the return type is required.

## All methods annotated with @Bean must be in @Configuration classes.

## @Qualifier :

* when multiple beans of the same type exist in the application context, and autowiring is used to inject dependencies, Spring may encounter ambiguity in selecting which bean to inject.
* We use*@Qualifier* along with *@Autowired* to provide the bean id or bean name we want to use in ambiguous situations.

The following two beans implement the same interface:

class Bike implements Vehicle {}

class Car implements Vehicle

If Spring needs to inject a Vehicle bean, it ends up with multiple matching definitions. In such cases, we can provide a bean's name explicitly using the @Qualifier annotation.

**Using constructor injection:**

@Autowired

Biker(@Qualifier("bike") Vehicle vehicle) {

this.vehicle = vehicle;

}

**Using setter injection:**

@Autowired

**void** **setVehicle**(@Qualifier("bike") Vehicle vehicle) {

this.vehicle = vehicle;

}

**Alternatively:**

@Autowired

@Qualifier("bike")

**void** **setVehicle**(Vehicle vehicle) {

this.vehicle = vehicle;

}

**Using field injection:**

@Autowired

@Qualifier("bike")

Vehicle vehicle;

## @Value:

## We can use @Value for injecting property values into beans. It's compatible with constructor, setter, and field injection.

## Constructor injection:

## Engine(@Value("8") int cylinderCount) {

## this.cylinderCount = cylinderCount;

## }

## Setter injection:

## @Autowired

## void setCylinderCount(@Value("8") int cylinderCount) {

## this.cylinderCount = cylinderCount;

## }

## Alternatively:

## @Value("8")

## void setCylinderCount(int cylinderCount) {

## this.cylinderCount = cylinderCount;

## }

## Field injection:

## @Value("8")

## int cylinderCount;

### **@DependsOn:**

We can use this annotation to make Spring **initialize other beans before the annotated one**.

### **@Lookup:**

A method annotated with @Lookup tells Spring to return an instance of the method’s return type when we invoke it.

**@Primary:**

if **we mark the most frequently used bean with @Primary** it will be chosen on unqualified injection points.

### **@Scope:**

We use @Scope to define the scope of a @Component class or a @Bean definition. It can be either singleton, prototype, request, session, globalSession or some custom scope.

@Component

@Scope(“prototype”)

Class Engine { }

### **@Import:**

We can use **specific @Configuration classes without component scanning** with this annotation.

Ex:

@Import(VehiclePartSupplier.class)

class VehicleFactoryConfig {}

**@ModelAttribute :**

an annotation that binds a method parameter or method return value to a named model attribute, and then exposes it to a web view.

**@ModelAttribute methods are invoked before the controller methods annotated with @RequestMapping are invoked.**

### **@ImportResource:**

We can **import XML configurations** with this annotation. We can specify the XML file locations with the locations argument, or with its alias, the value argument:

@Configuration

@ImportResource("classpath:/annotations.xml")

**class** **VehicleFactoryConfig** {}

**common Spring bean annotations:**

**@ComponentScan:**

* It configures which packages to scan for configuration classes. We can specify the base package names directly with one of the *basePackages* or *value* arguments (*value* is an alias for *basePackages*):

@Configuration

@ComponentScan(basePackages = "com.baeldung.annotations")

**class** **VehicleFactoryConfig** {}

* Both arguments are arrays so that we can provide multiple packages for each.
* If no argument is specified, the scanning happens from the same package where the *@ComponentScan* annotated class is present.

@Configuration:

* to indicate that it contains bean definitions.
* When the Spring container initializes, it scans for classes annotated with @Configuration.
* It then processes the @Bean methods within those classes, registering the beans defined by those methods in the application context.

## @Component:

@Component is a class level annotation. During the component scan, **Spring Framework automatically detects classes annotated with**@Component:

@Component

**class** **CarUtility** {

// ...

}

By default, the bean instances of this class have the same name as the class name with a lowercase initial.

**Stereotype annotations**, such as @Component, @Service, @Repository, and @Controller, are used to indicate the roles of classes in the Spring application context.

Since @Repository, @Service, @Configuration, and @Controller are all meta-annotations of @Component, they share the same bean naming behavior. Spring also automatically picks them up during the component scanning process.

**If we annotate a repository class with @Service or any other component annotations or vice versa, It won’t show any error because all these are annotations of @Component.**

## @Repository:

DAO or Repository classes usually represent the **database access layer** in an application, and should be annotated with @Repository.

## @Service:

The **business logic** **of an application** usually resides within the service layer, so we’ll use the @Service annotation to indicate that a class belongs to that layer.

## @Controller:

*@Controller* is a class level annotation, which tells the Spring Framework that this class serves as a **controller in Spring MVC.**

## @Configuration:

Configuration classes can **contain bean definition methods** annotated with @Bean.

Ex:

@Configuration

class VehicleFactoryConfig {

@Bean

Engine engine() {

return new Engine();

}

}

* Constructor injection has a few advantages compared to field injection.
* The first benefit is testability.
* **with field injection, we can't enforce class-level invariants.**
* with constructor injection, it's easier to build immutable components.
* **using constructors to create object instances is more natural from the OOP standpoint.**

**@SpringBootApplication =** @Configuration, @EnableAutoConfiguration, and @ComponentScan.

**It will automatically scan the components in the current package and its sub-packages.**

Thus, it will register them in Spring's Application Context, and allow us to inject beans using @Autowired.

When there are multiple beans of the same type, it's a good idea to**use @Qualifier to avoid ambiguity.**

**@ConfigurationProperties:**

* It supports conversion for multiple types of binding the properties to their corresponding beans.
* works best with hierarchical properties that all have the same prefix; therefore, we add a prefix of mail.
* As of Spring Boot 2.2, Spring finds and registers @ConfigurationProperties classes via classpath scanning.
* Scanning of @ConfigurationProperties needs to be explicitly opted into by adding the @ConfigurationPropertiesScan annotation.
* Therefore, we don’t have to annotate such classes with @Component (and other meta-annotations like @Configuration), or even use the @EnableConfigurationProperties:



**Controller:**

* We can annotate classic controllers with the @Controller annotation.
* This is simply a specialization of the @Component class, which allows us to auto-detect implementation classes through the classpath scanning.
* We typically use @Controller in combination with a @RequestMapping annotation for request handling methods.
* Spring 4.0 introduced the **@RestController** annotation in order to simplify the creation of RESTful web services.
* **It's a convenient annotation that combines @Controller*and*@ResponseBody**, which eliminates the need to annotate every request handling method of the controller class with the @ResponseBody annotation.
* **@ResponseBody annotation is used to indicate that the return value of a method should be serialized directly into the HTTP response body.**
* It converts the return value of a controller method to JSON, XML, or another format based on the content negotiation strategy configured in the application.
* Typically used in methods of @Controller or @RestController classes to send data back to the client as the body of the HTTP response.
* **@RequestBody annotation is used to bind the body of an HTTP request to a method parameter.**
* **It converts the request body, which can be in JSON, XML, or another format, into a Java object of the specified parameter type.**
* **By default,** **the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side controller.**
* **Typically used in methods of @Controller or @RestController classes to receive data from the client as part of the HTTP request body.**

**Ex:**

**@PostMapping("/echo")**

**public String echo(@RequestBody String message) {**

**return "You said: " + message;**

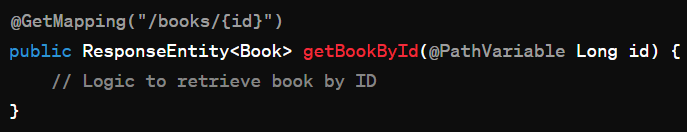
**}**

* **Here's why GET is idempotent:**
* **The HTTP method that is idempotent is the GET method.**
* **An operation is considered idempotent if performing it multiple times has the same effect as performing it once. For the GET method, this means that multiple identical requests will result in the same response, and the state of the server will remain unchanged.**
* **Safe Operation: GET requests are considered safe, meaning they should not modify the server's state. They are used for retrieving data from the server, such as fetching a resource or querying information. Since GET requests don't modify data on the server, repeating the same GET request multiple times will not alter the server's state.**
* **No Side Effects: GET requests do not have any side effects on the server. They only retrieve data without altering it. Therefore, making multiple GET requests to the same resource will not cause any unexpected changes in the server's state.**
* **For example, if you make a GET request to retrieve a user's profile information, it doesn't matter how many times you make the same request; the user's profile data remains the same, and the server's state remains unchanged. This property makes GET requests idempotent.**

**@PathVariable:**

* **@PathVariable is used to extract values from URI templates in the request URL.**
* **It binds the value of a URI template variable to a method parameter in a controller method.**
* **Typically used to extract values from path segments in the request URL.**
* **Use @PathVariable to extract values from path segments in the request URL.**

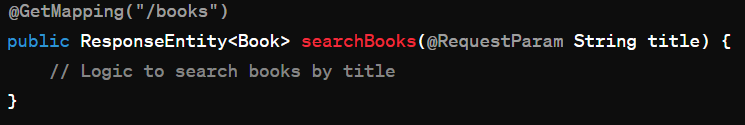
**Ex:**

****

**@RequestParam:**

* **@RequestParam is used to extract query parameters from the request URL or form data submitted with the request.**
* **It binds the value of a request parameter to a method parameter in a controller method.**
* **Typically used to extract query parameters or form parameters from the request URL or form data.**
* **Use @RequestParam to extract query parameters or form parameters from the request URL or form data.**

**Ex:**

****

**In this example, title is a query parameter in the request URL “/books?title=Spring”, and @RequestParam String title in the method parameter binds the value of the title query parameter to the title parameter in the method.**

**Exceptional Handling:**

* T**he @ExceptionHandler annotated method is only active for that particular Controller**, not globally for the entire application. Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.
* We can work around this limitation by having **all Controllers extend a Base Controller class.**

The actual mechanism is extremely simple but also very flexible:

* It gives us full control over the body of the response as well as the status code.
* It provides mapping of several exceptions to the same method, to be handled together.
* It makes good use of the newer RESTful *ResposeEntity* response.

One thing to keep in mind here is to **match the exceptions declared with *@ExceptionHandler* to the exception used as the argument of the method.**

If these don’t match, the compiler will not complain — no reason it should — and Spring will not complain either.

* If we want to specify the **response status of a controller method**, we can mark that method with @ResponseStatus. It has two interchangeable arguments for the desired response status: code, and value.
* When we want to signal an error, we can provide an error message via the reason argument:

@ResponseStatus(HttpStatus.BAD\_REQUEST, reason = "Some parameters are invalid")

void onIllegalArgumentException(IllegalArgumentException exception) {}

* when we set reason, Spring calls HttpServletResponse.sendError(). Therefore, it will send an **HTML error page to the client, which makes it a bad fit for REST endpoints**.
* Spring only uses @ResponseStatus, when **the marked method completes successfully** (without throwing an Exception).
* When we don’t need dynamic error responses, the most straightforward solution is the third one: marking the Exception class with @ResponseStatus:

@ResponseStatus(code = HttpStatus.BAD\_REQUEST)

class CustomException extends RuntimeException {}

* when we mark an Exception class with @ResponseStatus, Spring always calls HttpServletResponse.sendError(), whether we set reason or not.

## ResponseEntity:

* ResponseEntity **represents the whole HTTP response: status code, headers, and body**. As a result, we can use it to fully configure the HTTP response.
* ResponseEntity is a generic type. Consequently, we can use any type as the response body:

@GetMapping("/hello")

ResponseEntity<String> hello() {

return new ResponseEntity<>("Hello World!", HttpStatus.OK);

}

**REST API:**

**HTTP Status Codes:**

**For every HTTP request, the server returns a status code indicating the processing status of the request. Let's see the frequently used HTTP status codes:**

**1xx: Information :**

**Communicates transfer protocol-level information**

**100: Continue**

**2xx: Success:**

**This indicates that the client’s request was accepted successfully.**

**200: OK**

**201: Created**

**202: Accepted**

**204: No Content**

**3xx: Redirect:**

**This indicates that the client must take some additional action in order to complete their request.**

**301: Moved Permanently**

**307: Temporary Redirect**

**4xx: Client Error:**

**This category of error status codes points the finger at clients.**

**400: Bad Request**

**401: Unauthorized**

**403: Forbidden**

**404: Not found**

**5xx Server Error:**

**The server takes responsibility for these error status codes.**

**500: Internal Server Error**

**501: Not Implemented**

**502: Bad Gateway**

**503: Service Unavailable**

**504: Gateway Timeout**

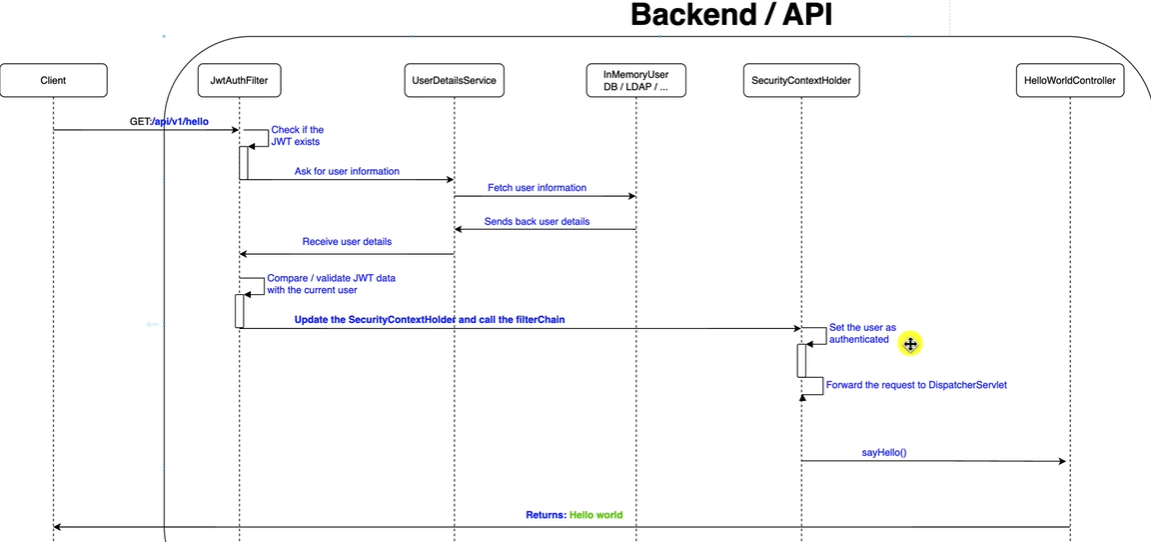
**Spring MVC:**

* **It's a module of the Spring framework dealing with the Model-View-Controller or MVC pattern.**
* Spring implements MVC with the front controller pattern using its DispatcherServlet.
* the DispatcherServlet acts as the main controller to route requests to their intended destination. Model is nothing but the data of our application, and the view is represented by any of the various template engines.
* **we can register view controllers that create a direct mapping between the URL and the view name** using the ViewControllerRegistry.

Internal working:

1. Request Processing: user Http request is intercepted by Dispatcher Servlet which acts as a front controller for the application.
2. Dispatcher Servlet: based on the request URL, it determines which controller method should handle that request.
3. Handler Mapping: It helps Dispatcher Servlet to find appropriate controller for the request based on the URL patterns.
4. Controller: the appropriate controller processes the request, interacts with service and model and prepares data to send to view.
5. Model: it represents the logic and data of the application. The data is then stored in data structure known as ModelMap or ModelAndView and sent to view.
6. View Resolver: It resolves logical view name returned by the controller to actual view like JSP or Thymeleaf. It combines the prefix and suffix given in web.xml file with view name to get the right view file path.
7. View: the rendered data output is sent as a HTTP response to the client.
8. Response: Dispatcher Servlet sends the response to the client.

**Spring Security:**

****

**Annotations:**

**@EnableGlobalMethodSecurity(prePostEnabled = true):**

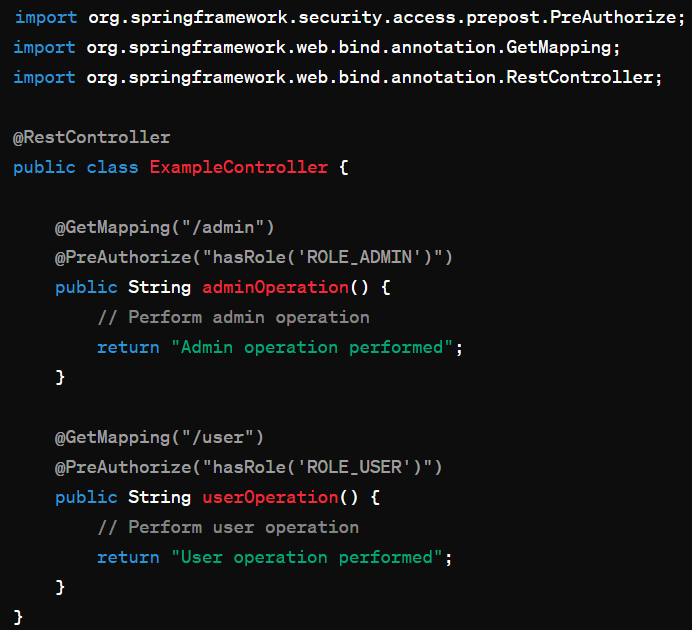
* It is used to enable method-level security in a Spring application.
* you can use method-level security annotations throughout your application to control access to specific methods based on user roles, permissions, or custom expressions.
* It allows you to use annotations such as @PreAuthorize, @PostAuthorize, @Secured, and @RolesAllowed to enforce access control rules directly on individual methods.

**@PreAuthorize, @PostAuthorize, @PreFilter, and @PostFilter**

**@PreAuthorize:**

* It is an annotation in Spring Security that allows you to define access control rules directly in your Spring MVC controllers or service methods.
* It is typically used to restrict access to specific methods based on user roles, permissions, or custom expressions.
* The @PreAuthorize annotation accepts a SpEL (Spring Expression Language) expression as its value.
* This expression defines the access control rule that must be satisfied for the method to be executed.
* The expression can reference method parameters, Spring Security's authentication object, and other context variables.
* Before executing the annotated method, Spring Security evaluates the security expression defined in @PreAuthorize.
* If the expression evaluates to true, the method is executed as usual.
* If the expression evaluates to false, Spring Security denies access to the method and may return an HTTP 403 Forbidden response or handle the access denial according to the configured access control strategy.

Ex:



**Important concepts:**

[**https://chat.openai.com/share/8ec8e3c0-1dfb-4db7-b24b-3606bdf25d05**](https://chat.openai.com/share/8ec8e3c0-1dfb-4db7-b24b-3606bdf25d05)

* The @EnableWebFlux annotation enables the standard Spring Web Reactive configuration for the application.
* Cross-Origin Resource Sharing (CORS)

The *@CrossOrigin* annotation has the following default configuration:

* Allows all origins (that explains the ‘\*' value in the response header)
* Allows all headers
* All HTTP methods mapped by the handler method are allowed
* Credentials are not enabled
* The ‘max-age' value is of 1800 seconds (30 minutes)
* This annotation is also supported at a class level, and it will affect all its methods.

**The best way to enable CORS on functional endpoints is by using a WebFilter.**

**JWT Token:**

* JWT, or JSON Web Token, is a compact, URL-safe means of representing claims between two parties.
* It is often used for authentication and information exchange between a client and a server.
* The structure of a JWT is a JSON object that is compactly encoded as a string, and it may include various claims (pieces of information) about the user.

**A JWT typically consists of three parts(Header, Payload, Signature):**

**Header:**

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

Example Header:

{

"alg": "HS256",

"typ": "JWT"

}

**Payload:**

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata.

Example Payload:

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022

}

Common claims include iss (issuer), sub (subject), exp (expiration time), iat (issued at), aud (audience), and custom claims.

**Signature:**

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.

Example Signature:

HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

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.

**The resulting JWT is a string of the form header.payload.signature.** It can be sent in an HTTP header (commonly Authorization) or as a query parameter in a URL.

**How JWT Works in Authentication:**

**1.User Authentication:**

* A user logs in with their credentials on the authentication server.
* The server validates the credentials and creates a JWT containing user claims.

**2.Token Issuance:**

* The server sends the JWT to the client.
* The client stores the JWT, typically in a cookie or local storage.

**3.Subsequent Requests:**

* The client includes the JWT in the Authorization header or as a parameter when making requests to protected resources.

**4.Server Validation:**

* The server receiving the request validates the JWT's signature and checks the claims.
* If valid, the server processes the request.
* JWT is stateless, meaning the server doesn't need to store the user's state.
* The claims within the token provide all the necessary information for authentication and authorization.
* However, JWT should be used carefully, and sensitive information should be encrypted within the token if necessary.
* Additionally, proper security measures, such as token expiration and secure key management, should be implemented.

**Spring Data JPA:**

* **By implementing one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.**

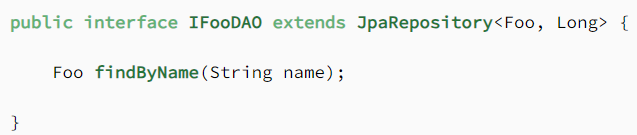
To define more specific access methods, Spring JPA supports quite a few options:

* simply **define a new method** in the interface
* provide the actual **JPQL query** by using the @Query annotation

**Automatic custom queries:**

* When Spring Data creates a new Repository implementation, it analyses all the methods defined by the interfaces and tries to automatically generate queries from the method names.

Ex: If the entity has a name field (and the Java Bean standard getName and setName methods), we’ll define the findByName method in the DAO interface. This will automatically generate the correct query:



Hibernate supports the three basic inheritance mapping strategies:

* table per class hierarchy
* table per subclass
* table per concrete class

@Entity : Specifies that the class is an entity. This annotation can be applied on Class, Interface of Enums.

@Table: It specifies the table in the database with which this entity is mapped. In the example below the data will be stores in the “employee” table. Name attribute of  @Table annotation is used to specify the table name.

@Id: This annotation specifies the primary key of the entity.

@GeneratedValue: This annotation specifies the generation strategies for the values of primary keys.

@OrderBy: Sort your data using [@OrderBy](https://www.digitalocean.com/community/users/orderby) annotation. In example below, it will sort all employees\_address by their id in ascending order.

@Transient: Every non static and non-transient property of an entity is considered persistent, unless you annotate it as [@Transient](https://www.digitalocean.com/community/users/transient).

@Lob: Large objects are declared with [@Lob](https://www.digitalocean.com/community/users/lob).

* [@PrimaryKeyJoinColumn](https://www.digitalocean.com/community/users/primarykeyjoincolumn) should be used for associated entities sharing the same primary key.
* [@JoinColumn](https://www.digitalocean.com/community/users/joincolumn) & [@OneToOne](https://www.digitalocean.com/community/users/onetoone) should be mappedBy attribute when foreign key is held by one of the entities.

the id genereated for Communication will be mapped to ‘communication\_id’ column of CommunicationDetail table. [@MapsId](https://www.digitalocean.com/community/users/mapsid) is used for the same.

@ManyToOne Many employees can share the same status. So, employee to employeeStatus is a many to one relation.

@OneToMany Employee to Communication will be a one-to-many relationship. The owner of this relationship is Communication so, we will use ‘mappedBy’ attribute in Employee to make it bi-directional relationship.

@Entity

@Table(name = "employee")

public class Employee implements Serializable {

@OneToMany(mappedBy = "employee", fetch = FetchType.EAGER)

@OrderBy("firstName asc")

private Set communications;

}

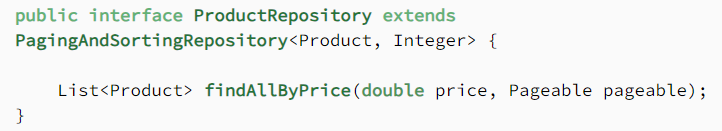
@JoinColumn [@JoinColumn](https://www.digitalocean.com/community/users/joincolumn) annotation is used for one-to-one or many-to-one associations when foreign key is held by one of the entities.

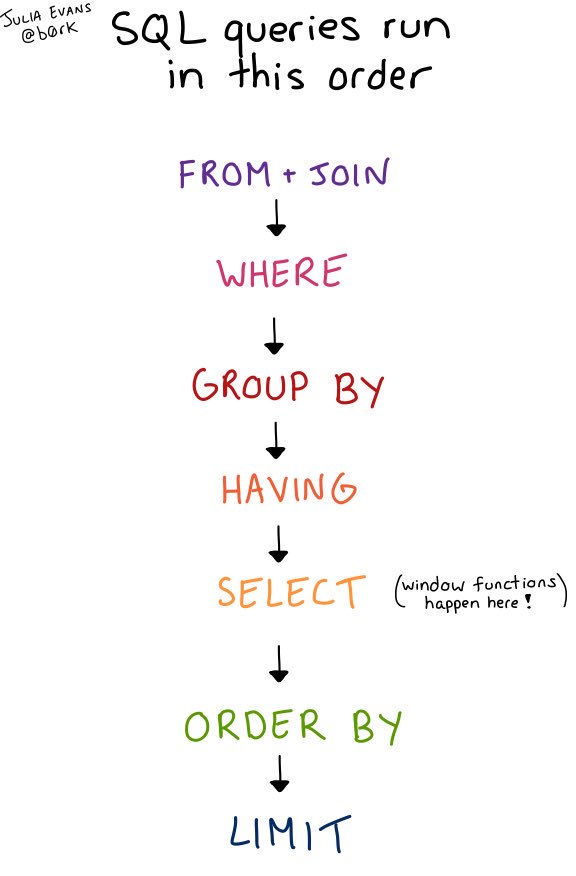
@JoinTable: [@JoinTable](https://www.digitalocean.com/community/users/jointable) and mappedBy should be used for entities linked through an association table.

 @MapsId: Two entities with shared key can be persisted using [@MapsId](https://www.digitalocean.com/community/users/mapsid) annotation.

**Pagination and Sorting using Spring Data JPA:**

By having it extend PagingAndSortingRepository, we get findAll(Pageable pageable) and findAll(Sort sort) methods for paging and sorting.





**Spring Boot:**

Here are just a few of the features in Spring Boot:

* Opinionated ‘starter' dependencies to simplify the build and application configuration
* Embedded server to avoid complexity in application deployment
* Metrics, Health check, and externalized configuration
* Automatic config for Spring functionality – whenever possible

**Spring requires both the standard spring-security-web and spring-security-config dependencies** to set up Security in an application.

Some of the advantages of Spring Boot over Spring in the context of deployment include:

* Provides embedded container support
* Provision to run the jars independently using the command *java -jar*
* Option to exclude dependencies to avoid potential jar conflicts when deploying in an external container
* Option to specify active profiles when deploying
* Random port generation for integration tests

The benefits of using Spring Boot starters:

* increase pom manageability
* production-ready, tested & supported dependency configurations
* decrease the overall configuration time for the project

### **Spring Boot Layers**

The spring boot consists of the following four layers:

1. **Presentation Layer** – Authentication & Json Translation
2. **Business Layer** – Business Logic, Validation & Authorization
3. **Persistence Layer** – Storage Logic
4. **Database Layer** – Actual Database

**Recommended structure of Spring Boot Project:**

* it's recommended that we avoid using the default package — that is, **we should always declare the package in our classes.**
* **The main class of the project reside in the base package.**
* **One popular design strategy is package-by-feature,** which enhances modularity and enables package-private visibility inside sub-packages.
* There are 2 ways to organize codebase:
  1. Package by Layer
  2. Package by Feature

**Spring Boot Actuator:**

* In essence, Actuator brings production-ready features to our application.
* Monitoring our app, gathering metrics, understanding traffic, or the state of our database become trivial with this dependency.
* Can be used to stop tomcat server.
* Spring Boot Actuator ensures that the shutdown process is graceful, allowing the application to complete any in-flight requests before shutting down.
* It's important to note that only the embedded Tomcat server is stopped, and the application context remains active.
* Actuator is mainly used to **expose operational information about the running application —** health, metrics, info, dump, env, etc.
* It uses HTTP endpoints or JMX beans to enable us to interact with it.
* It helps monitor and manage spring application in production.
* It provides a number of endpoints:
  1. Beans – Complete list of Spring beans in your application
  2. Health – Application health info using health indicators
  3. Metric – Application metrics
  4. Auditing – track events in application
  5. Environment- environment info using “actuator/env”.
  6. Shutdown- graceful shutdown using “actuator/shutdown”.
  7. Mappings – Details around request mappings

**Spring Batch:**

[https://chat.openai.com/c/f24ecef5-ca34-4dfc-93de-c df86c7eafe1](https://chat.openai.com/c/f24ecef5-ca34-4dfc-93de-cdf86c7eafe1)

**Microservice Communication:**

There are several ways to establish communication between Spring microservices:

**1.RESTful APIs:** Using HTTP and RESTful principles, microservices can communicate by sending and receiving JSON or XML payloads over HTTP.

**2.Feign Client:** As mentioned earlier, Spring Cloud Feign provides a declarative way to interact with microservices using interfaces.

**3.Spring Cloud Gateway:** A powerful API gateway that routes requests to appropriate microservices, enabling load balancing and other cross-cutting concerns.

**4.RabbitMQ or Apache Kafka:** Messaging systems that allow asynchronous communication between microservices through message queues or topics.

**5.Spring Cloud Bus:** An event bus that helps in broadcasting configuration changes across multiple microservices.

**6.Eureka or Consul:**

•Service discovery mechanisms that enable microservices to find and communicate with each other dynamically.

•Eureka is a service registry that allows microservices to register themselves and discover other services in the system dynamically.

•This enables a more resilient and flexible architecture, as services can locate and communicate with each other without hardcoding IP addresses or relying on static configurations.

**7.gRPC:** A modern RPC framework that allows high-performance communication using Protocol Buffers.

**Service Discovery:**

* In a microservices-based system, where multiple independent services communicate with each other, service discovery plays a key role in enabling dynamic and resilient interactions between these services.
* The main goal of service discovery is to facilitate the location and communication with the various microservices within the system.

Here are the key aspects of service discovery in microservices:

**1.Dynamic Service Registration:**

* Microservices register themselves with a service registry when they start up.
* This registration includes essential information about the service, such as its network location, IP address, port, and any other metadata that might be useful for other services to know.

**2.Service Registry:**

* A centralized service registry (also known as a service registry server or service registry and discovery server) is responsible for maintaining an up-to-date list of all available services and their respective locations.
* This registry serves as a centralized source of truth for the current state of the microservices architecture.

**3.Service Discovery:**

* Clients (other microservices or components) that need to interact with a specific service query the service registry to discover the location and details of that service.
* Service discovery allows services to locate and communicate with each other dynamically.

**4.Load Balancing:**

* Service discovery often involves load balancing mechanisms.
* When there are multiple instances of a service registered in the registry, a load balancer can distribute incoming requests among these instances to achieve better resource utilization and scalability.

**5.Dynamic Updates:**

* Service registry and discovery systems support dynamic updates.
* When a microservice starts, stops, or scales (increases or decreases the number of instances), the service registry is updated in real-time.
* This ensures that the system is aware of the current state of the services.

**6.Fault Tolerance and Resilience:**

* Service discovery contributes to fault tolerance and resilience in a microservices architecture.
* If a service instance becomes unavailable, clients can use the service registry to locate an alternative instance or take appropriate actions, such as circuit breaking or retrying.
* Service discovery is a critical component in achieving the benefits of microservices architecture, such as flexibility, scalability, and resilience, by enabling seamless communication and interaction between microservices.

**Service registry and Discovery tools:**

They play a crucial role in enabling service discovery in microservices architectures. Here are a few examples of popular service registry tools:

**Netflix Eureka:**

**Description:** Eureka is an open-source service registry developed by Netflix. It is designed for high availability and can be used with various programming languages.

**Key Features:**

* Service registration and discovery.
* Support for client-side load balancing with Ribbon.
* Health checking of services.
* Integration with Spring Cloud for seamless use with Spring-based applications.

**Usage Example:**

* Netflix Eureka is often used in conjunction with other Netflix components, such as Ribbon for load balancing. It is widely used in Spring Cloud applications.

**Consul:**

**Description:** It provides a flexible and feature-rich platform for service registry, health checking, and key-value storage.

**Key Features:**

* Service registration and discovery.
* Health checking with support for complex health checks.
* Distributed key-value store.
* Multi-datacenter support.

**Usage Example:**

* Consul is commonly used in modern microservices architectures and can be integrated with various platforms and frameworks.

**etcd:**

**Description:** etcd is a distributed key-value store that can be used for service discovery and coordination. It is often used in Kubernetes environments for managing configuration and state.

**Key Features:**

* Distributed key-value store.
* Consistent and highly available.
* Strong consistency guarantees.
* Integration with Kubernetes for service discovery.

**Usage Example:**

* etcd is used as the primary key-value store in Kubernetes for storing configuration data and managing coordination among cluster nodes.

**ZooKeeper:**

**Description:** Apache ZooKeeper is a distributed coordination service that can be used for various tasks, including service discovery. It provides a hierarchical namespace and coordination primitives.

**Key Features:**

* Distributed coordination service.
* Hierarchical namespace.
* Consistent and highly available.
* Integration with various programming languages.

**Usage Example:**

* ZooKeeper is used in various distributed systems for tasks such as distributed locking and configuration management.

**API Gateway:**

* In a microservices architecture, an API Gateway is a centralized entry point that serves as a single point of contact for clients to interact with various microservices.
* It acts as a reverse proxy that handles the routing, composition, and management of requests from clients to the underlying microservices.
* The API Gateway plays a crucial role in simplifying the client experience, improving security, and providing various cross-cutting concerns.

**Key features and responsibilities of an API Gateway in a microservices architecture include:**

**1.Request Routing:**

* The API Gateway routes incoming requests from clients to the appropriate microservices based on the request's path, headers, or other criteria.
* It abstracts the complexity of the underlying microservices architecture, allowing clients to interact with a unified API.

**2.Protocol Translation:**

* It can handle protocol translation, converting requests and responses between different protocols (e.g., translating between HTTP and WebSocket).

**3.Aggregation and Composition:**

* The API Gateway can aggregate and compose responses from multiple microservices into a single response to fulfill a client's request.
* This helps reduce the number of round-trips between the client and the microservices.

**4.Authentication and Authorization:**

* Centralized authentication and authorization can be enforced at the API Gateway.
* It can handle authentication mechanisms, such as API keys, OAuth tokens, or JWTs, and enforce access control policies before forwarding requests to microservices.

**5.Rate Limiting and Throttling:**

* The API Gateway can implement rate limiting and throttling to control the rate of incoming requests from clients, preventing abuse and ensuring fair usage of resources across microservices.

**6.Logging and Monitoring:**

* It can collect and centralize logs, metrics, and traces from microservices, providing observability into the system's health and performance.

**7.Caching:**

* API Gateways can implement caching strategies to cache responses from microservices, reducing latency and improving overall system performance.

**8.Static Content Delivery:**

* Serve static assets or content directly, reducing the load on microservices for handling non-dynamic content.

**9.Fault Tolerance and Circuit Breaking:**

* Implement fault tolerance mechanisms, such as circuit breaking, to handle errors gracefully and prevent cascading failures in the system.

**10.Transformation and Normalization:**

* Transform or normalize data between clients and microservices, ensuring that the data exchanged is in a format that is most suitable for the client or microservice.

**11.Versioning:**

* Manage API versioning, allowing clients to specify the version of the API they want to use.
* By consolidating these responsibilities in a central component, the API Gateway simplifies the client-facing interface, provides a level of abstraction over the microservices, and enables the implementation of various cross-cutting concerns.
* It acts as a powerful tool for managing the complexities associated with communication in a microservices-based system.

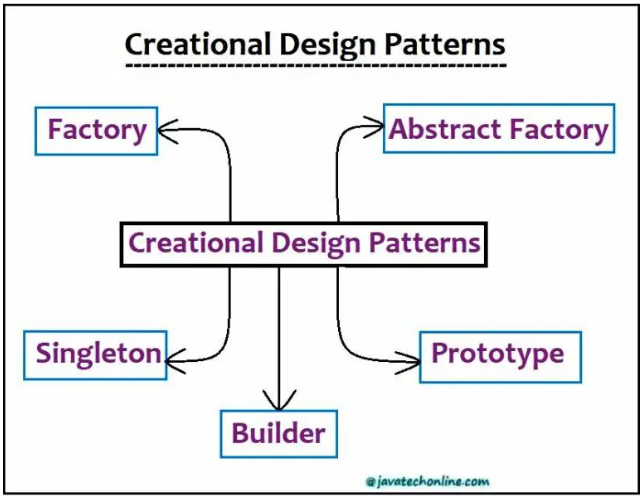
**What is axon in microservices?**

* Axon leverages the separation of components and uses explicit messaging between them, which makes these components Location Transparent.
* Axon Framework is an open-source framework for building microservice systems that are event-driven and evolutionary. It is based on the principles of Domain-Driven Design (DDD), Command-Query Responsibility Separation (CQRS), and Event Sourcing.
* Unlike the use of service discovery, the approach Axon takes for messaging doesn't require a component to know the destination of a message at all.

**DESIGN PATTERNS:**

**1.CREATIONAL DESIGN PATTERN:**

It provides solution to instantiate an object in the best possible way for specific situations.



**A. Singleton Pattern:**

**Concept:**

* There are only two points in the definition of a singleton design pattern:

1) There should be only one instance allowed for a class.

2) We should allow global point of access to that single instance.

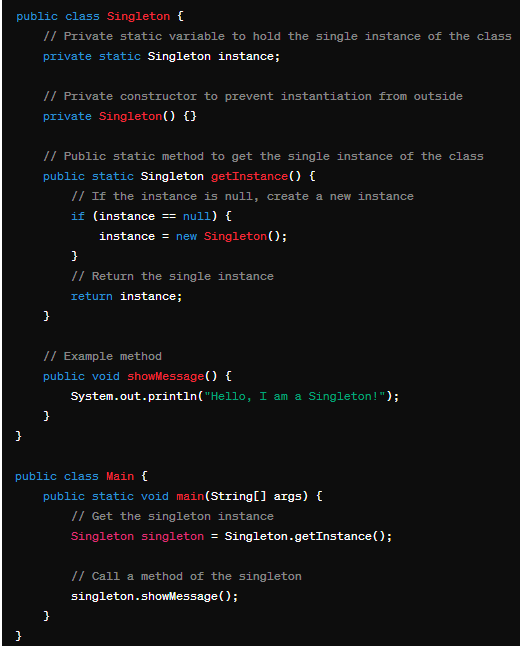
* Only one instance is created and ensures that only one instance of the class exists in the java virtual machine.

Ex: Runtime environment, Logger, Spring Beans, thread pool, caching, driver objects.

* We can even see the usage of Singleton design pattern in core Java API classes also. For example, java.lang.Runtime, java.awt.Desktop.

**Design:**

* Class is responsible for lifecycle of singleton object
* Static in nature
* Needs to be thread safe
* Contains private instance,private static variable, private constructor to restrict instantiation of class from other classes.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.
* No parameters required for construction



**B. Builder Pattern:**

* The Builder pattern in Java specifies an abstract interface for creating parts of a Product object, effectively separating the construction logic from the object class into a separate builder class.
* This design approach allows for multiple builder classes, each with different implementations for the series of steps needed to construct the object.
* Consequently, each builder implementation results in a different representation of the object.
* The intent of the Builder Pattern is to further separate the construction of a complex object from its representation, facilitating the creation of different representations using the same construction process.
* This separation enhances modularity, reducing the object's size and making it more modular, with each implementation contained within a different builder object.
* As a result, adding a new implementation (i.e., adding a new builder) becomes easier, and the object construction process becomes independent of the components having the object, providing greater control over the construction process.
* Handles complex constructors
* Large number of parameters

Ex: StringBuilder, DocumentBuilder, Locale.Builder

* Flexibility over telescoping constructors(avoids confusion of which constructor to use like empty or parameterized constructor)
* Builder is build using Static class
* Calls appropriate constructor based on state
* Negates the need for exposed setters

**ConcreteBuilder:**

* Constructs and assembles parts of the product by implementing the Builder interface.
* Defines and keeps track of the representation it creates.
* Provides an interface for retrieving the product.

**Director:**

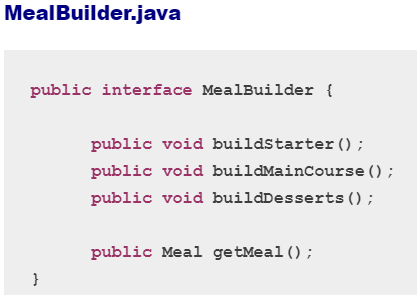
* It constructs an object using the Builder interface.
* The Builder pattern suggests using a dedicated object referred to as a Director, which is responsible for invoking different builder methods required for the construction of the final object.
* Moreover, different client objects can make use of the Director object to create the required object.
* Once the object is constructed, the client object can directly request from the builder the fully constructed object.
* In order to facilitate this process, a new method getObject () can be declared in the common Builder interface to be implemented by different concrete builders.

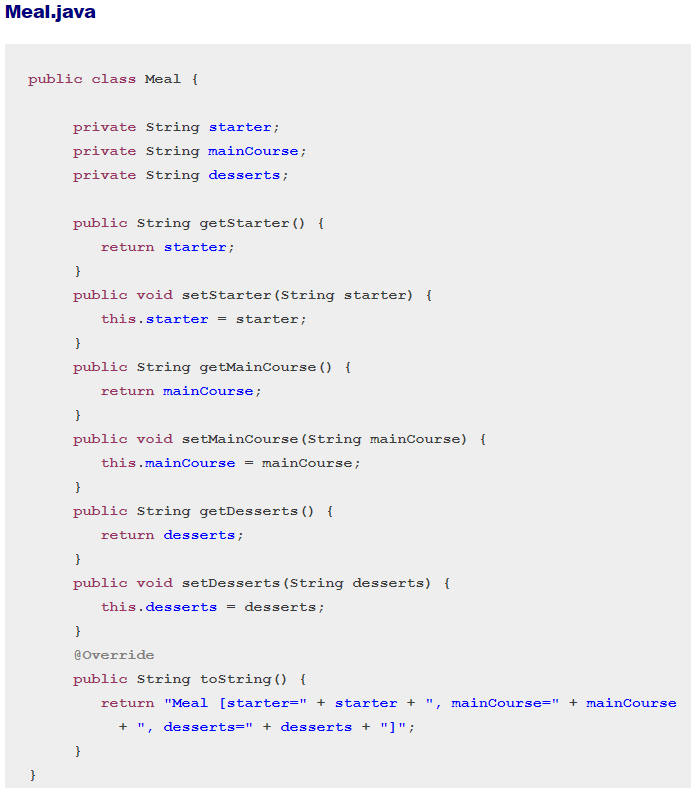
**Product:**

* Represents the complex object under construction. Further, ConcreteBuilder builds the product’s internal representation and defines the process by which it’s assembled.
* Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

**Example:**

Let’s assume an example of building a meal as below:













**Another form/version of the Builder Pattern:**

* Sometimes, there is an object with a long list of properties. Most of these properties are optional.
* Writing a constructor with long list of parameters is not a good choice, this could frustrate the user especially if the important fields are only a few.
* Also, this could increase the scope of error. The user may provide a value accidentally to a wrong field. Long sequences of identically typed parameters can cause subtle bugs.
* If the user accidentally reverses two such parameters, the compiler won’t complain, but the program will misbehave at runtime.
* Instead of making the desired object directly, the user calls a constructor with all of the required parameters and gets a builder object.
* Then, the user calls setter-like methods on the builder object to set each optional parameter of interest.
* Finally, the user calls a parameter-less build method to generate the object.

**Steps to create the Pattern:**

* First of all, we need to create a static nested class and then copy all the arguments from the outer class to the Builder class. Additionally, we should follow the naming convention. For example, if the class name is Cake, then builder class should be named as CakeBuilder.
* Next, the Builder class should have a public constructor with all the required attributes as parameters.
* Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
* Finally, we need to provide a build () method in the builder class that will return the Object needed by client program. In order to accomplish this, we need to have a private constructor in the Class with Builder class as argument.

**Example :**

Let’s assume an example of building a cake programmatically, follow the above steps accordingly as below:







**Use the Builder pattern:**

* When the algorithm for creating a complex object should be independent of the parts that make up the object and how they’re assembled.
* When the construction process must allow different representations for the object that’s constructed.

**Drawbacks:**

* Immutable
* Inner static class
* Design first
* Complexity

**C. Prototype Pattern:**

* This pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Moreover, this pattern uses java cloning to copy the object.
* Avoids costly creation by cloning existing object.
* Avoids subclassing(using new keyword) instead uses cloning.
* Often utilizes an interface defining the clone() method.
* Usually implemented with a registry to manage and retrieve prototype instances.
* Although a copy, each instance is unique
* Can still utilize parameters for construction by setting the initial state of the cloned object after cloning.
* Useful when we need to modify an object data multiple times
* To use cloning, you can utilize the clone () method and the Cloneable interface. By default, clone () performs a shallow copy. Moreover, Serializable can be used to simplify deep copying.

**Ex:**

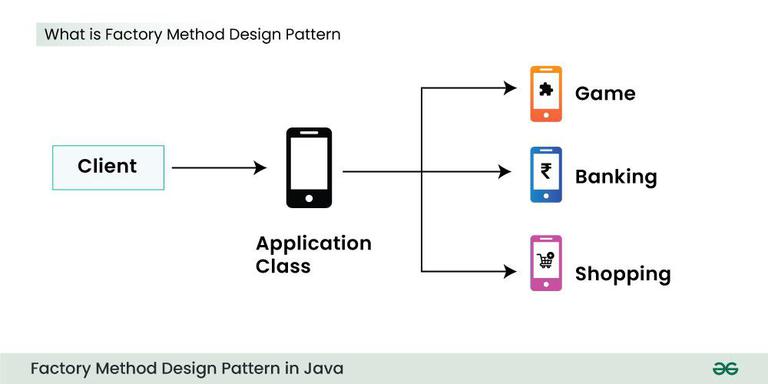
below code describes how to design an advanced bike after cloning the basic bike object:

****

****

**D. Factory Pattern:**

* Factory Method Design Pattern define an interface for creating an object, but let subclass decide which class to instantiate. Factory Method lets a class defer instantiation to subclass.

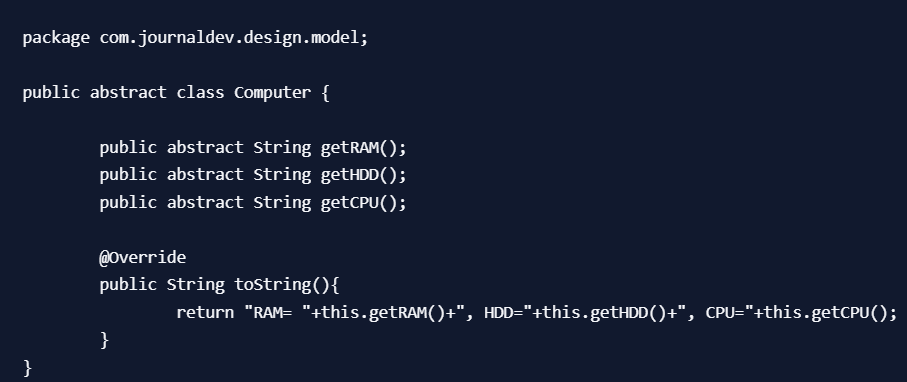


* The Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-classes.
* This pattern takes out the responsibility of instantiation of a class from a client program to the factory class.
* Super class in factory pattern can either be an interface, or an abstract class, or a normal Java class.

**Example:**

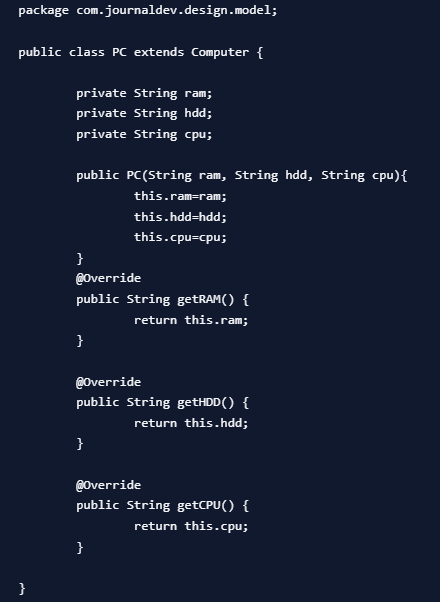
we have abstract super class with overridden toString() method for testing purpose.

Computer.Java:



Let’s say we have two sub-classes PC and Server with below implementation:

Pc.Java:

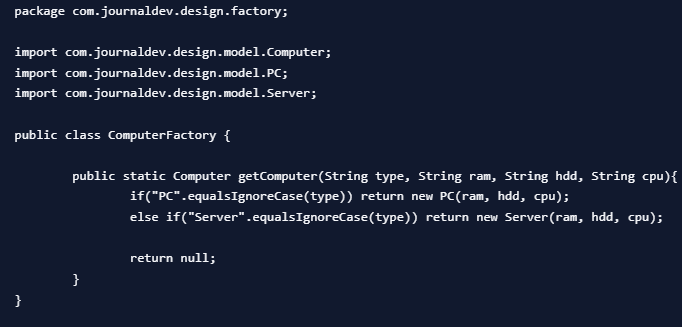


Server.Java:

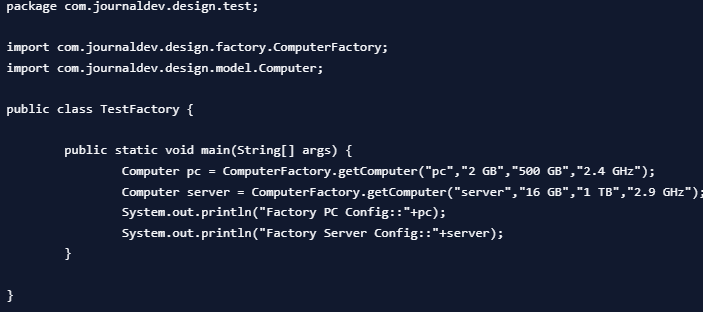


Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation:

ComputerFactory.Java:



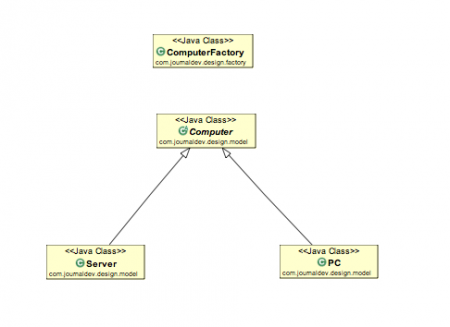
TestFactory.Java:



Output:



* We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
* Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.



**Factory Design Pattern Advantages:**

* Factory design pattern provides approach to code for interface rather than implementation.
* Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
* Factory pattern provides abstraction between implementation and client classes through inheritance.

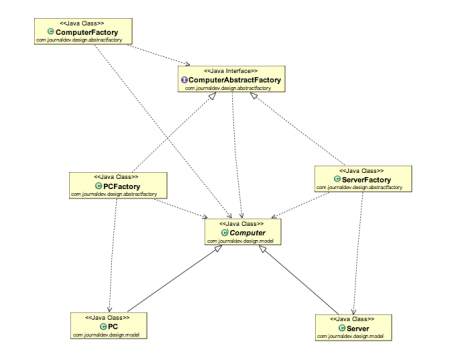
**Factory Design Pattern Examples in JDK:**

* java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
* valueOf() method in wrapper classes like Boolean, Integer etc.

**E.Abstract Factory Pattern(**factory of factories**):**

* In the Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class.
* Then an Abstract Factory class that will return the sub-class based on the input factory class.

**Example:**

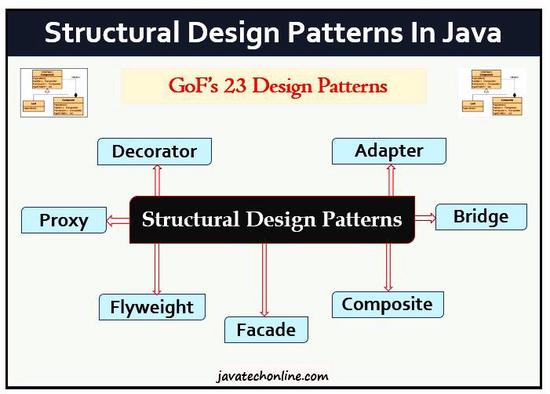


<https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java>

**Abstract Factory Design Pattern Benefits:**

* Abstract Factory design pattern provides approach to code for interface rather than implementation.
* Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
* Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

**2.STRUCTURAL DESIGN PATTERNS:**



* They deal with how classes and objects can be composed to form larger structures, while keeping these structures flexible and efficient.( deal with different ways to create a class structure)
* They provide different ways to create a class structure using inheritance and composition to create a large object from small objects.
* Composite lets clients treat individual objects and compositions of objects uniformly, that’s the intent of the Composite Pattern.

**A. Adapter Pattern:**

* It makes two unrelated interfaces work together. Moreover, the object that joins these unrelated interfaces is called an Adapter just like a mediator.

Ex:

util.Arrays#asList()

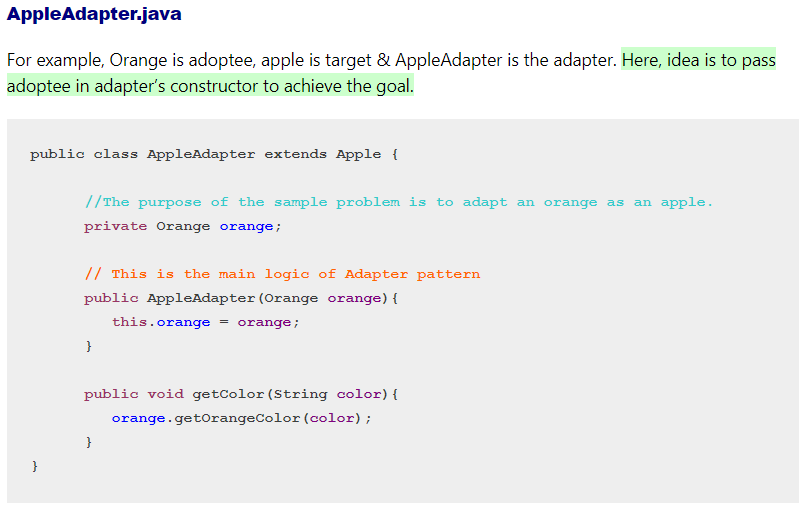
io.InputStreamReader(InputStream) (returns a Reader)

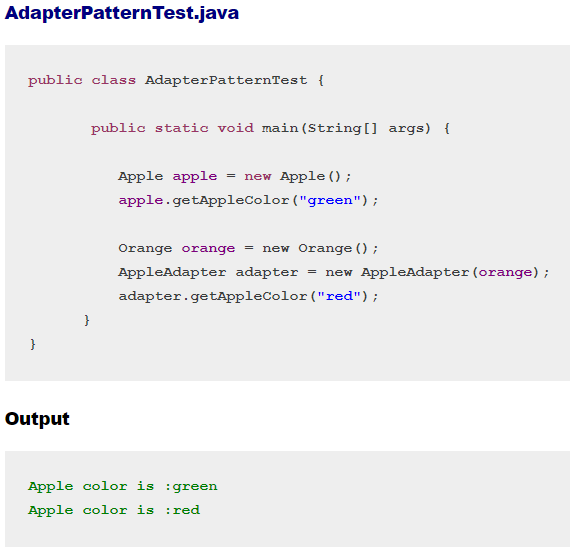
io.OutputStreamWriter(OutputStream) (returns a Writer)

* As a real-life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge, but the normal socket produces either 120V (in US) or 240V (in India). Therefore, the mobile charger works as an adapter between mobile charging socket and the wall socket.
* In the adapter pattern, a wrapper class (i.e., the adapter) is used to translate requests from it to another class (i.e., the adoptee).
* In effect, an adapter provides particular interactions with an adoptee that are not offered directly by the adoptee.
* The adapter pattern can take two forms : Inheritance or Composition form. In the first form, a “class adapter” utilizes inheritance.
* The class adapter extends the adoptee class and adds the desired methods to the adapter.
* These methods can be declared in an interface (i.e., the “target” interface).
* However, in the second form; an “object adapter” utilizes composition.
* The object adapter contains an adoptee and implements the target interface to interact with the adoptee.

**Example :**

****

****

****

**When to use Adapter Pattern?**

The Adapter pattern should be used when:

1. There is an existing class, and its interface does not match the one you need.
2. You want to create a reusable class that co-operates with unrelated or unforeseen classes, that is, classes that don’t necessarily have compatible interfaces.
3. There are several existing subclasses to be used, but it’s impractical to adapt their interface by sub-classing each one. An object adapter can adapt the interface of its parent class.

MICROSERVICES MAIN CHARACTERISTICS

A 'good' microservice have 7 main characteristics

1. Fine-grained capabilities that are stateless\_Eg:A&A

2. Well defined interfaces which 'hide' how the service

is executed. Eg: Forex rate & Collect payment from different bank

3. A 'very loosely-coupled' approach (can change one

service without impacting another)

4. Autonomous, explicitly versioned & completely

independent

5. stupid - "do one thing, and do it well"

6. cost and value fully defined

7. Disposable

SERVICE INFRASTRUCTURE PRINCIPLES

A 'good' microservice have 7 main characteristics

1. Utility based

2. Open standards based

3. Multi-Tenant support

4. Technology independent & virtualized

5. Service oriented

6. Scalable

7. Fit-for-purpose

8. Sustainable

9. Automated

10. Modularity

