Spring official website:

[www.spring.io](http://www.spring.io)

Spring Boot and Dispatcher Servlet

1. SpringBoot Application does not have web.xml to configure the Servlet and filters.
2. DispatcherServlet is configured automatically by Spring Boot's auto-configuration mechanism. DispatcherServlet is configured within the ServletWebServerFactoryAutoConfiguration
3. DispatcherServlet is typically mapped to the root URL ("/"), meaning it handles all incoming requests by default.

Dispatcher servlet is responsible for deserializing the request and serializing the response.

Dispatcher servlet is responsible for calling appropriate controller to handle request based on request mapping.

Sprint Boot and Filters

The filter chain is a sequence of filters applied to an incoming request before it reaches the servlet. Spring Boot automatically registers filters as beans, adding them to the servlet processing chain. When a request arrives, the container invokes the doFilter() method of each filter in the chain, in the order they are configured. Each filter can choose to pass the request to the next filter in the chain by calling chain.doFilter() or terminate the chain. After all filters have been processed, the request reaches the DispatcherServlet for further handling.

Spring Boot provides several ways to define and configure filters, including using @Component annotation, FilterRegistrationBean, or @WebFilter annotation. The order of filter execution can be controlled using Ordered interface or FilterRegistrationBean. Spring Security also relies heavily on filters to implement security features, such as authentication and authorization.

**Adding a filter in spring boot**

1. **Create a filter class implementing Filer**
2. **Using @Component**
3. To control the order of filter execution, use the @Order annotation.

|  |
| --- |
| @Component  @Order(1)  public class MyFilter implements Filter {  @Override  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  throws IOException, ServletException {  // Filter logic here  chain.doFilter(request, response);  }  } |

1. Using FilterRegistrationBean

Create a bean of type FilterRegistrationBean in a configuration class. This provides more control over the filter's configuration, such as specifying URL patterns and the order of execution.

|  |
| --- |
| @Configuration  public class FilterConfig {  @Bean  public FilterRegistrationBean<MyFilter> myFilterRegistration() {  FilterRegistrationBean<MyFilter> registration = new FilterRegistrationBean<>(new MyFilter());  registration.addUrlPatterns("/api/\*"); // Apply filter to specific URL patterns  registration.setOrder(2); // Set filter order  return registration;  }  } |

**Interceptor Handler**

1. **To create an interceptor, we should implement the HandlerInterceptor interface directly.**
2. **The HandlerInterceptor interface contains three main methods:**
3. **prehandle() – called before the execution of the actual handler**
4. **postHandle() – called after the handler is executed**
5. **afterCompletion() – called after the complete request is finished and the view is generated**

|  |
| --- |
| **@Override**  **public boolean preHandle(**  **HttpServletRequest request,**  **HttpServletResponse response,**  **Object handler) throws Exception {**  **// your code**  **return true;**  **}** |

**Notice that the method returns a boolean value. It tells Spring to further process the request (true) or not (false).**

**We need to register the interceptor**

|  |
| --- |
| **@Override**  **public void addInterceptors(InterceptorRegistry registry) {**  **registry.addInterceptor(new LoggerInterceptor());**  **}** |

**These three methods provide flexibility to do all kinds of pre- and post-processing.**

How to intercept request in Bootstrap applications.

1. Filter
2. Interceptor Handler
3. ControllerAdvice

Spring annotations

A screenshot of a computer

AI-generated content may be incorrect.

A diagram of a system

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A diagram of a component

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Why we need different Annotations based on layers**

Any web application can be divided into three parts

1. Controller level
2. Service level
3. DB(Repository level)

Each has its own requirements or purposes, DB layer can deal with db related tasks where as service layer can deal with business or any third party intergrations, so its better to have there own annotations which gives them their own purpose along with registering then as beans in applicationContext.

* Web Annotations

1. SpringBootApplication
   1. Configuration
   2. EnableAutoConfiguration
   3. ComponentScan
2. @ComponentScan
3. RestController
4. Autowire
5. Qualifier
6. Primary
7. Component
8. Scope
9. PostCreate
10. PreDestroy
11. Bean
12. ControllerAdvice🡪 used to handle Exception logic and can be used by all controllers.
13. ExceptionHandler🡪 Marks the method as handling particular type of execption.

* JPA annotation

1. Entity🡪 marks the bean as bean mapped to DB table
2. Table
3. Column
4. Id
5. GeneratedValue(Strategy=GenerationType.IDENTITY)
6. Transactional

* Sprint rest Annotation

1. PathVariable🡺 Binds method parameters with request parameters.

When we query database using entity manager we don’t query to DB, we query to Entity class which is in Java, so all queries should refer to Entity class not DB table

PUT vs Patch request

if a table has 3 columns but we sent only 2 then 3rd column value becomes null or default, it overrides existing value with default value

PUT request updates the whole resource, if no value passed for any particular property then it will be set to default value as null or whatever is the default value in db. So PUT is used for complete update

If we want to do partial update Patch is the better option. Other fields are not touched.