# ChatGPT

https://chatgpt.com/c/66deecc9-2848-800e-b5f9-657d3f945448

Q> Show me a full project source example To write data to a database using a Spring Boot filter

A>

Below is a simple Spring Boot project that demonstrates writing data to a database using a filter. The example uses the H2 in-memory database and JPA for persistence.

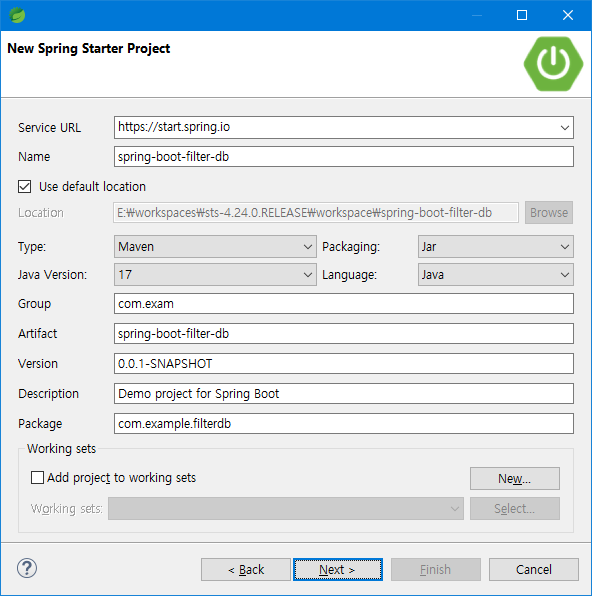
1. pom.xml

The pom.xml file includes dependencies for Spring Boot, JPA, H2, and other necessary components.

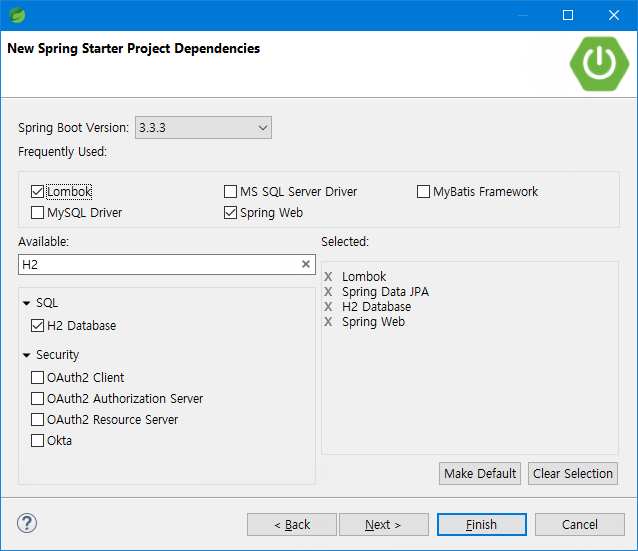
실습

C:\Users\ADMINI~1\AppData\Local\Temp\SNAGHTML15114e86.PNG

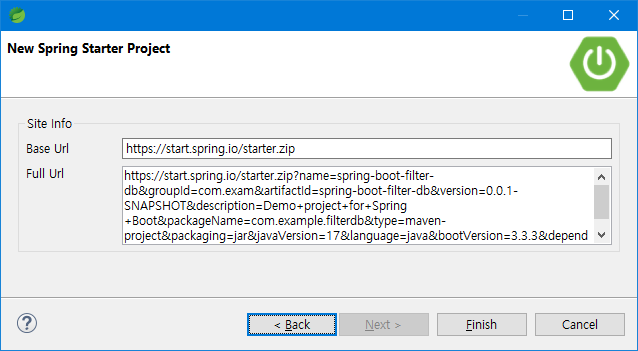
## New Spring Starter Project



## New Spring Starter Project Dependencies



## New Spring Starter Project



https://start.spring.io/starter.zip?name=spring-boot-filter-db&groupId=com.exam&artifactId=spring-boot-filter-db&version=0.0.1-SNAPSHOT&description=Demo+project+for+Spring+Boot&packageName=com.example.filterdb&type=maven-project&packaging=jar&javaVersion=17&language=java&bootVersion=3.3.3&dependencies=lombok&dependencies=h2&dependencies=data-jpa&dependencies=web

Run As > Spring Boot App

에러 발생

|  |
| --- |
| 2024-09-10 05:40:36.811[0;39m [33m WARN[0;39m [35m42224[0;39m [2m---[0;39m [2m[ main][0;39m [36mConfigServletWebServerApplicationContext[0;39m [2m:[0;39m Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.support.BeanDefinitionOverrideException: Invalid bean definition with name 'logFilter' defined in class path resource [com/example/filterdb/config/FilterConfig.class]: Cannot register bean definition [Root bean: class [null]; scope=; abstract=false; lazyInit=null; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=filterConfig; factoryMethodName=logFilter; initMethodName=null; destroyMethodName=(inferred); defined in class path resource [com/example/filterdb/config/FilterConfig.class]] for bean 'logFilter': There is already [Generic bean: class [com.example.filterdb.filter.LogFilter]; scope=singleton; abstract=false; lazyInit=null; autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in file [E:\workspaces\sts-4.24.0.RELEASE\workspace\spring-boot-filter-db\target\classes\com\example\filterdb\filter\LogFilter.class]] bound. |

# 결론 ChatGPT

package com.example.filterdb.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.Ordered;

import org.springframework.stereotype.Component;

import com.example.filterdb.filter.LogFilter;

import com.example.filterdb.filter.MDCLoggingFilter;

import com.example.filterdb.filter.RequestAndResponseLoggingFilter;

import com.example.filterdb.multidb.service.UserMySQLService;

import com.example.filterdb.service.LogEntryService;

import lombok.extern.slf4j.Slf4j;

@Configuration

@Slf4j

public class FilterConfiguration {

/\*

cf. https://www.baeldung.com/spring-autowire-bean-servlet-filter

위의 예에서, 우리의 필터는 기본적으로 우리 애플리케이션의 모든 URL에 대해 등록됩니다. 그러나 때때로 우리는 특정 URL 패턴에만 적용되는 필터를 원할 수 있습니다.

이 경우에는 필터 클래스 정의에서 @Component 주석을 제거하고 FilterRegistrationBean 을 사용하여 필터를 등록해야 합니다 .

필터에 대한 URL 패턴을 설정하려면 addUrlPatterns() 또는 setUrlPatterns() 메서드를 사용할 수 있습니다.

\*/

// 빈생성시 자동주입되어지는 UserMySQLService userMySQLService 이 중요한 개념이었어요!

@Bean

FilterRegistrationBean<LogFilter> loggingFilterRegistration(UserMySQLService userMySQLService

// LogEntryService logEntryService

) {

FilterRegistrationBean<LogFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new LogFilter(

// logEntryService

userMySQLService

));

registrationBean.addUrlPatterns("/api/\*"); // "/api/postgres/\*" // /non-filter/\*

registrationBean.setOrder(100);

return registrationBean;

}

# 참고 자료(좋아요)

## Filter를 등록하는 4가지 방법(스크랩)

https://www.inflearn.com/chats/786858/filter를-등록하는-4가지-방법?srsltid=AfmBOool\_T2egqw3fgmrhOxXG\_0h6hLPm8cQFHhU-OZhkdql3PqEPAcF

###### [김회민](https://www.inflearn.com/users/857748/@shrimp)

작성일 23.02.16 20:34 조회수 4.23k

안녕하세요.

정리 강의를 들으면서 Filter를 스프링 빈으로 등록하는 방법을 따라하다 버전 차이인지, 스프링 부트의 설정 차이인지, 에러가 발생하더군요.

The bean 'logFilter', defined in class path resource [hello/springcoremvc26/config/FilterConfig.class], could not be registered.

A bean with that name has already been defined in file [/project/java/spring/spring-core-mvc2-6/out/production/classes/hello/springcoremvc26/web/filter/LogFilter.class] and overriding is disabled.

이유를 읽어보니, logFilter가 이미 빈으로 등록되어서 중복 등록이 안된다고 써져있었습니다.

그래서 이에 대해 찾아본 결과를 공유하고자 글을 작성합니다.

아래 글은 제 [블로그](https://velog.io/@ksk7584/Filter%EB%A5%BC-%EB%93%B1%EB%A1%9D%ED%95%98%EB%8A%94-4%EA%B0%80%EC%A7%80-%EB%B0%A9%EB%B2%95)에도 정리되어 있습니다. (홍보...ㅎㅎ..)

## @Configuration

### LogFilter - 로그 필터

@Slf4j

public class LogFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LogFilter init()");

}

@Override

public void destroy() {

log.info("LogFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LogFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LogFilter doFilter End", requestURI);

}

}

}

### LoginCheckFilter - 로그인 필터

@Slf4j

public class LoginCheckFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LoginCheckFilter init()");

}

@Override

public void destroy() {

log.info("LoginCheckFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LoginFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LoginFilter doFilter End", requestURI);

}

}

}

### FilterConfig - Filter 등록

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<Filter> logFilter() {

FilterRegistrationBean<Filter> bean = new FilterRegistrationBean<>();

bean.setFilter(new LogFilter());

bean.setOrder(1);

bean.addUrlPatterns("/\*");

return bean;

}

@Bean

public FilterRegistrationBean<Filter> loginCheckFilter() {

FilterRegistrationBean<Filter> bean = new FilterRegistrationBean<>();

bean.setFilter(new LoginCheckFilter());

bean.setOrder(2);

bean.addUrlPatterns("/\*");

return bean;

}

}

### 실행 결과

[/] LogFilter doFilter Start

[/] LoginFilter doFilter Start

[/] LoginFilter doFilter End

[/] LogFilter doFilter End

### 특징

* 강의에서 나온 방법입니다.
* 설정을 위한 별개의 파일(@Configuration 이 붙은 객체)이 필요합니다.
* setOrder() 를 통해 순서를 정할 수 있습니다.
* addUrlPatterns() 을 통해 베이스 URL 및 Whitelist를 설정할 수 있습니다.

## @Component

### LogFilter

@Slf4j

@Component

@Order(1)

public class LogFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LogFilter init()");

}

@Override

public void destroy() {

log.info("LogFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LogFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LogFilter doFilter End", requestURI);

}

}

}

### LoginCheckFilter

@Slf4j

@Component

@Order(2)

public class LoginCheckFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LoginCheckFilter init()");

}

@Override

public void destroy() {

log.info("LoginCheckFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LoginFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LoginFilter doFilter End", requestURI);

}

}

}

### 실행 결과

[/] LogFilter doFilter Start

[/] LoginFilter doFilter Start

[/] LoginFilter doFilter End

[/] LogFilter doFilter End

### 특징

* 컴포넌트 스캔을 이용하기 때문에 설정을 위한 별개의 파일이 필요하지 않습니다.
* @Order 애노테이션을 이용해 순서를 설정할 수 있습니다.
* 기본 URL Pattern이 /\* 이며 설정할 수 없습니다.

## @WebFilter + @ServletComponentScan

### LogFilter

@Slf4j

@WebFilter

public class LogFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LogFilter init()");

}

@Override

public void destroy() {

log.info("LogFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LogFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LogFilter doFilter End", requestURI);

}

}

}

### LoginCheckFilter

@Slf4j

@WebFilter

public class LoginCheckFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LoginCheckFilter init()");

}

@Override

public void destroy() {

log.info("LoginCheckFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LoginFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LoginFilter doFilter End", requestURI);

}

}

}

### MainApplication

@ServletComponentScan

@SpringBootApplication

public class MainApplication {

public static void main(String[] args) {

SpringApplication.run(SpringCoreMvc26Application.class, args);

}

}

### 실행 결과

[/] LogFilter doFilter Start

[/] LoginFilter doFilter Start

[/] LoginFilter doFilter End

[/] LogFilter doFilter End

### 특징

* 설정을 위한 별개의 파일이 필요하지 않습니다.
  + 대신, 애플리케이션 실행되는 메인 객체위에 @ServletComponentScan 을 사용해야 합니다.
* @Order를 이용한 순서 등록을 사용할 수 없습니다.
  + 각 필터에 대한 순서를 보장할 수 없습니다.
* @WebFilter 의 value 나 urlPatterns 파라미터를 이용해 whitelist 방식으로 베이스 URL을 설정할 수 있습니다.
  + @WebFilter("/filter/\*")
  + @WebFilter({"/login", "/items"})
  + @WebFilter(urlPatterns = "/filter/\*")
  + @WebFilter(urlPatterns = {"/login", "/items"})

## @WebFilter + @Component

### LogFilter

@Slf4j

@WebFilter

@Component

@Order(1)

public class LogFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LogFilter init()");

}

@Override

public void destroy() {

log.info("LogFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LogFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LogFilter doFilter End", requestURI);

}

}

}

### LoginCheckFilter

@Slf4j

@WebFilter

@Component

@Order(1)

public class LoginCheckFilter implements Filter {

@Override

public void init(

FilterConfig filterConfig

) throws ServletException {

log.info("LoginCheckFilter init()");

}

@Override

public void destroy() {

log.info("LoginCheckFilter destroy()");

}

@Override

public void doFilter(

ServletRequest request,

ServletResponse response,

FilterChain chain

) throws IOException, ServletException {

HttpServletRequest req = (HttpServletRequest) request;

String requestURI = req.getRequestURI();

log.info("[{}] LoginFilter doFilter Start", requestURI);

try {

chain.doFilter(request, response);

} finally {

log.info("[{}] LoginFilter doFilter End", requestURI);

}

}

}

### 실행 결과

[/] LogFilter doFilter Start

[/] LoginFilter doFilter Start

[/] LoginFilter doFilter End

[/] LogFilter doFilter End

### 특징

* 두 번째 방법과 세 번째 방법을 모두 사용하는 방법입니다.
* 설정을 위한 별개의 파일이 필요하지 않습니다.
  + 컴포넌트 스캔 방식을 사용하기 때문에 @ServletComponentScan 도 필요없습니다.
* @Order 애노테이션을 이용해 순서를 설정할 수 있습니다.
* @WebFilter 의 value 나 urlPatterns 파라미터를 이용해 베이스 URL이나 Whitelist 방식으로 설정할 수 있습니다.
  + @WebFilter("/filter/\*")
  + @WebFilter({"/login", "/items"})
  + @WebFilter(urlPatterns = "/filter/\*")
  + @WebFilter(urlPatterns = {"/login", "/items"})
* 애노테이션이 기본으로 3개가 필요합니다.

## 정리



## 결론

필터를 사용하기 위한 4가지 방식을 알아보았습니다.

첫 번째 방법(@Configuration + FilterRegistrationBean)과 두 번째, 세 번째 방법을 합친 네 번째 방법(@WebFilter + @Component) 중에서 고려하면됩니다.

저의 경우에는 @Configuration + FilterRegistrationBean**을 더 맘에 드는**데, 이유는 Filter Class 위에 애노테이션을 여러개 붙이는 것 보다, 별도의 생성 파일을 하나 만들어서 관리하는게 더 편할 것 같아서였습니다.

하지만, @WebFilter + @Component은 **스프링 빈으로 등록하는 방법**이기에, 다른 스프링 빈을 의존성 주입 받을 수 있습니다. 그런 경우에는 첫 번째 방법으론 할 수 없으니, 네 번째 방법을 사용하면 될 것 같습니다.

# ChatGPT 답변 갈무리

https://chatgpt.com/c/66deecc9-2848-800e-b5f9-657d3f945448

How to write data in db with spring boot filter

###### ChatGPT의 말:

ChatGPT

To write data to a database using a Spring Boot filter, you need to ensure that your filter interacts with your service layer, which in turn communicates with the database. Here's a step-by-step guide on how to implement this:

### 1. ****Create a Filter****

Spring Boot provides the javax.servlet.Filter interface to implement a filter. You can override the doFilter method to handle incoming requests.

java

코드 복사

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import java.io.IOException;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class CustomFilter implements Filter {

@Autowired

private YourService yourService; // Service to interact with the database

@Override

public void init(FilterConfig filterConfig) throws ServletException {

// initialization code

}

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

// You can extract data from the request

String someData = request.getParameter("someData");

// Use the service to write data to the database

yourService.saveDataToDB(someData);

// Continue the request

chain.doFilter(request, response);

}

@Override

public void destroy() {

// cleanup code

}

}

### 2. ****Create a Service Layer****

The service layer contains the business logic and interacts with the repository to perform database operations.

java

코드 복사

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class YourService {

@Autowired

private YourRepository yourRepository;

public void saveDataToDB(String someData) {

YourEntity entity = new YourEntity();

entity.setSomeField(someData);

yourRepository.save(entity);

}

}

### 3. ****Create a Repository Layer****

The repository layer interacts with the database. It extends JpaRepository or CrudRepository to provide basic CRUD operations.

java

코드 복사

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface YourRepository extends JpaRepository<YourEntity, Long> {

}

### 4. ****Create an Entity****

The entity represents a table in your database.

java

코드 복사

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class YourEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String someField;

// Getters and setters

}

### 5. ****Configure the Filter****

By annotating the filter class with @Component, it will be automatically registered in the Spring context. If you need to configure the filter order or URL patterns, you can use the @WebFilter annotation or register the filter as a @Bean in a configuration class.

java

코드 복사

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<CustomFilter> loggingFilter(){

FilterRegistrationBean<CustomFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new CustomFilter());

registrationBean.addUrlPatterns("/api/\*");

return registrationBean;

}

}

### Summary

* The filter intercepts incoming requests.
* Extracts data from the request (e.g., headers, parameters, etc.).
* Passes that data to the service layer.
* The service layer writes the data to the database using the repository layer.

This approach allows your filter to perform database operations in a clean and modular manner.

4o

##### 나의 말:

Show me a full project source example To write data to a database using a Spring Boot filter

###### ChatGPT의 말:

ChatGPT

Below is a simple Spring Boot project that demonstrates writing data to a database using a filter. The example uses the H2 in-memory database and JPA for persistence.

### 1. ****pom.xml****

The pom.xml file includes dependencies for Spring Boot, JPA, H2, and other necessary components.

xml

코드 복사

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>spring-boot-filter-db</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>spring-boot-filter-db</name>

<description>Demo project to write data to DB using filter in Spring Boot</description>

<parent>

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

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

<version>2.7.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<dependencies>

<!-- Spring Boot Starter Web -->

<dependency>

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

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

</dependency>

<!-- Spring Boot Starter Data JPA -->

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- H2 Database -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<!-- Spring Boot Starter Test -->

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

### 2. ****Application Properties****

Configure H2 in-memory database and JPA properties in the application.properties file.

properties

코드 복사

# H2 Database configuration

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.h2.console.enabled=true

# JPA and Hibernate configuration

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

### 3. ****Main Application Class****

The main class to bootstrap the Spring Boot application.

java

코드 복사

package com.example.filterdb;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootFilterDbApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootFilterDbApplication.class, args);

}

}

### 4. ****Entity Class****

An entity representing a simple table to store data.

java

코드 복사

package com.example.filterdb.entity;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class LogEntry {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String data;

// Constructors, getters, and setters

public LogEntry() {}

public LogEntry(String data) {

this.data = data;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getData() {

return data;

}

public void setData(String data) {

this.data = data;

}

}

### 5. ****Repository Interface****

A JPA repository to handle database operations.

java

코드 복사

package com.example.filterdb.repository;

import com.example.filterdb.entity.LogEntry;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface LogEntryRepository extends JpaRepository<LogEntry, Long> {

}

### 6. ****Service Class****

A service layer to interact with the repository.

java

코드 복사

package com.example.filterdb.service;

import com.example.filterdb.entity.LogEntry;

import com.example.filterdb.repository.LogEntryRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class LogEntryService {

@Autowired

private LogEntryRepository logEntryRepository;

public void saveLogEntry(String data) {

LogEntry logEntry = new LogEntry(data);

logEntryRepository.save(logEntry);

}

}

### 7. ****Filter Class****

A custom filter that extracts data from the request and saves it to the database using the service.

java

코드 복사

package com.example.filterdb.filter;

import com.example.filterdb.service.LogEntryService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import javax.servlet.\*;

import javax.servlet.http.HttpServletRequest;

import java.io.IOException;

@Component

public class LogFilter implements Filter {

@Autowired

private LogEntryService logEntryService;

@Override

public void init(FilterConfig filterConfig) throws ServletException {

// Initialization code if necessary

}

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HttpServletRequest httpRequest = (HttpServletRequest) request;

String requestData = httpRequest.getParameter("data");

if (requestData != null) {

logEntryService.saveLogEntry(requestData); // Save to DB

}

// Continue with the request

chain.doFilter(request, response);

}

@Override

public void destroy() {

// Cleanup code if necessary

}

}

### 8. ****Controller Class****

A simple REST controller to handle HTTP requests.

java

코드 복사

package com.example.filterdb.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class LogController {

@GetMapping("/log")

public String logData() {

return "Data logged successfully!";

}

}

### 9. ****Filter Configuration****

If needed, you can manually configure the filter with a specific URL pattern.

java

코드 복사

package com.example.filterdb.config;

import com.example.filterdb.filter.LogFilter;

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<LogFilter> logFilter() {

FilterRegistrationBean<LogFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new LogFilter());

registrationBean.addUrlPatterns("/log/\*"); // Apply filter to specific URLs

return registrationBean;

}

}

### 10. ****Testing the Project****

1. Run the Spring Boot application.
2. Visit http://localhost:8080/h2-console to check the H2 database.
3. Use the URL http://localhost:8080/log?data=YourDataHere to trigger the filter and log the data to the database.
4. You should see the logged data in the LogEntry table in the H2 console.

This project demonstrates how to capture data from HTTP requests using a Spring Boot filter and write that data to a database via the service and repository layers.

4o

##### 나의 말:

how to use Custom FileterConfig and Custom Filter with spring boot 3.3.3

###### ChatGPT의 말:

ChatGPT

In Spring Boot 3.3.3, the approach to configuring filters remains similar to earlier versions. However, with the upgrade to Spring Boot 3.x, it uses the jakarta.servlet package instead of javax.servlet. Below, I'll guide you through how to use a custom filter with a custom FilterConfig in Spring Boot 3.3.3.

### 1. ****pom.xml****

Ensure that your pom.xml includes the necessary dependencies for Spring Boot, JPA, and an embedded database (like H2):

xml

코드 복사

<dependencies>

<!-- Spring Boot Starter Web -->

<dependency>

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

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

</dependency>

<!-- Spring Boot Starter Data JPA -->

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- H2 Database -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<!-- Spring Boot Starter Test -->

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

### 2. ****Custom Filter Implementation****

You can implement a custom filter using the jakarta.servlet.Filter interface. In Spring Boot 3.x, the servlet package has moved from javax.servlet to jakarta.servlet.

java

코드 복사

package com.example.filterdb.filter;

import jakarta.servlet.Filter;

import jakarta.servlet.FilterChain;

import jakarta.servlet.FilterConfig;

import jakarta.servlet.ServletException;

import jakarta.servlet.ServletRequest;

import jakarta.servlet.ServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import java.io.IOException;

@Component

public class CustomFilter implements Filter {

@Autowired

private YourService yourService; // Inject your service to interact with the DB

@Override

public void init(FilterConfig filterConfig) throws ServletException {

// Initialization code if needed

}

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

// Extract data from request

String requestData = request.getParameter("data");

if (requestData != null) {

yourService.saveDataToDB(requestData); // Write to DB

}

// Continue the request

chain.doFilter(request, response);

}

@Override

public void destroy() {

// Cleanup code if needed

}

}

### 3. ****Custom Filter Configuration****

You can register your filter programmatically using FilterRegistrationBean in a configuration class.

java

코드 복사

package com.example.filterdb.config;

import com.example.filterdb.filter.CustomFilter;

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<CustomFilter> customFilterRegistrationBean() {

FilterRegistrationBean<CustomFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new CustomFilter());

registrationBean.addUrlPatterns("/api/\*"); // Apply filter to specific URL patterns

registrationBean.setOrder(1); // Set the order of the filter

return registrationBean;

}

}

In this configuration:

* addUrlPatterns() specifies the URL patterns to which the filter applies. You can change this to fit your requirements.
* setOrder() defines the order in which this filter will be applied relative to other filters.

### 4. ****Service Layer****

Create a service to handle the business logic and database interaction.

java

코드 복사

package com.example.filterdb.service;

import com.example.filterdb.entity.YourEntity;

import com.example.filterdb.repository.YourRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class YourService {

@Autowired

private YourRepository yourRepository;

public void saveDataToDB(String data) {

YourEntity entity = new YourEntity();

entity.setData(data);

yourRepository.save(entity);

}

}

### 5. ****Entity Class****

Define your entity that maps to a database table.

java

코드 복사

package com.example.filterdb.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class YourEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String data;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getData() {

return data;

}

public void setData(String data) {

this.data = data;

}

}

### 6. ****Repository Layer****

Create a repository interface to handle CRUD operations.

java

코드 복사

package com.example.filterdb.repository;

import com.example.filterdb.entity.YourEntity;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface YourRepository extends JpaRepository<YourEntity, Long> {

}

### 7. ****Controller****

Create a controller to expose an endpoint for testing the filter.

java

코드 복사

package com.example.filterdb.controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class YourController {

@GetMapping("/api/log")

public String logData(@RequestParam String data) {

return "Data processed: " + data;

}

}

### 8. ****application.properties****

Configure your in-memory H2 database.

properties

코드 복사

# H2 Database configuration

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.h2.console.enabled=true

# JPA and Hibernate configuration

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

### 9. ****Running the Application****

1. Run the Spring Boot application.
2. Visit http://localhost:8080/h2-console to access the H2 database console.
3. Send a GET request to http://localhost:8080/api/log?data=YourDataHere.
4. Check the H2 console for the YourEntity table to see if the data has been saved.

### Key Points:

* **Filter Registration**: Use FilterRegistrationBean to programmatically register and configure your custom filter.
* **Jakarta Migration**: Be sure to use jakarta.servlet for imports instead of javax.servlet.
* **Dependency Updates**: Spring Boot 3.x uses Jakarta EE 9+, so dependencies should be updated accordingly in pom.xml.

# OncePerRequestFilter 심플 예제

https://techwasti.com/logging-request-and-response-body-in-spring-boot-using-filter

Maheshwar Ligade / Aug 21, 2022

## Logging Request and Response Body In Spring Boot using Filter!!

Table of contents

Introduction:

Tech Stack:

Implementation:

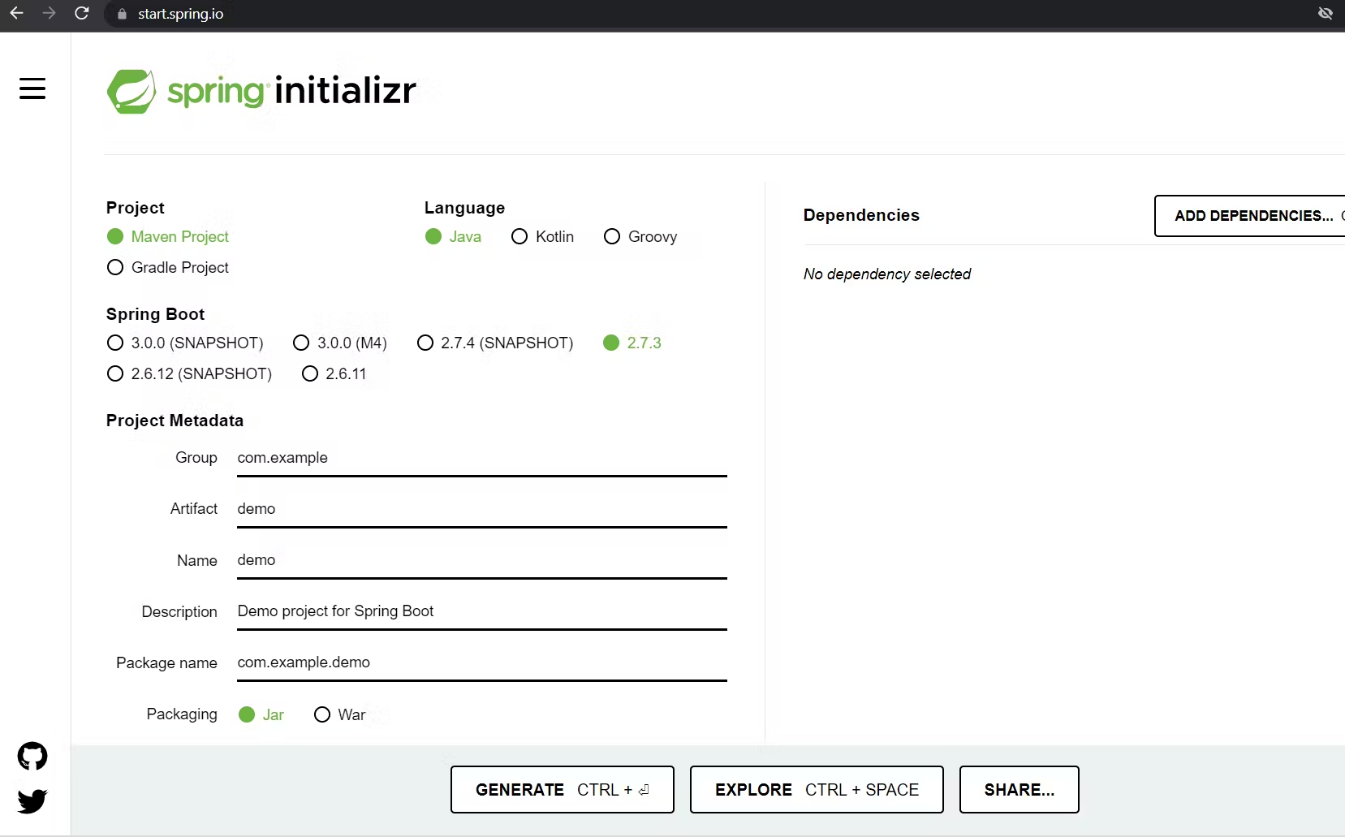
Log request and response body:

Conclusion:

Spring Boot Request and Response logging using Filter. In this article, we are going to learn how to intercept the request, response, and log the body which is helpful for debugging or tracing issue purposes.

### Introduction:

Logging is a crucial part of any application that reflects the data flow and helps to dissolve the complexity of a system. Let us jump and do a coding visit. https://start.spring.io/



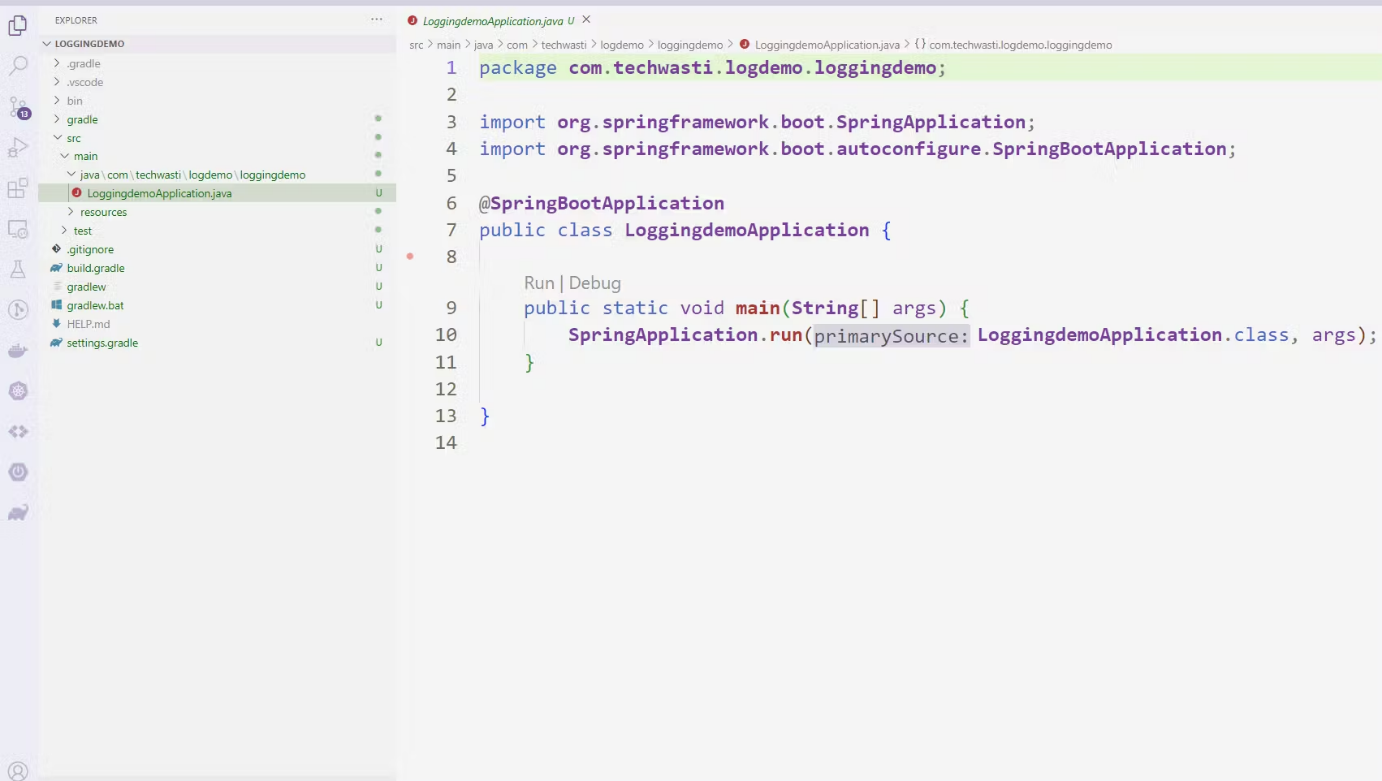
You can generate the project and download the zip or use the VSCode editor to generate the project and import into your favorite editor or IDE.

Tech Stack:

Java Spring Boot VSCode Gradle

Implementation:

Let us create some controller class so that we can have REST API which we will intercept later and logs the response and request body.



Let us create a model class as Customer.java

|  |
| --- |
| public class Customer {  private int id;  private String name;  private String address;  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getAddress() {  return address;  }  public void setAddress(String address) {  this.address = address;  }  } |

CustomerController.java Controller which is having get mapping and which will return the list of customers

|  |
| --- |
| @RestController  @RequestMapping("/customer")  public class CustomerController {  @Autowired  CustomerService customerService;  @GetMapping  public List<Customer> getCustomers(){  return customerService.getAllCust();  }  } |

In the same way, we do need the service layer which will interact with a database and return the result but for our demo, we are using a dummy one.

Service layer.

|  |
| --- |
| @Service  public class CustomerService {  List<Customer> getAllCust(){  return List.of(new Customer(1, "Ram","India"),  new Customer(2, "Richard","India"),  new Customer(3, "Suresh","India"),  new Customer(4, "Mahesh","India"),  new Customer(5, "Ganesh","India"));  }  } |

Now our basic structure is ready let us dive deeper and log the request and response. If you run the above code and hit the endpoint of customer you will get the list of customers.

### Log request and response body:

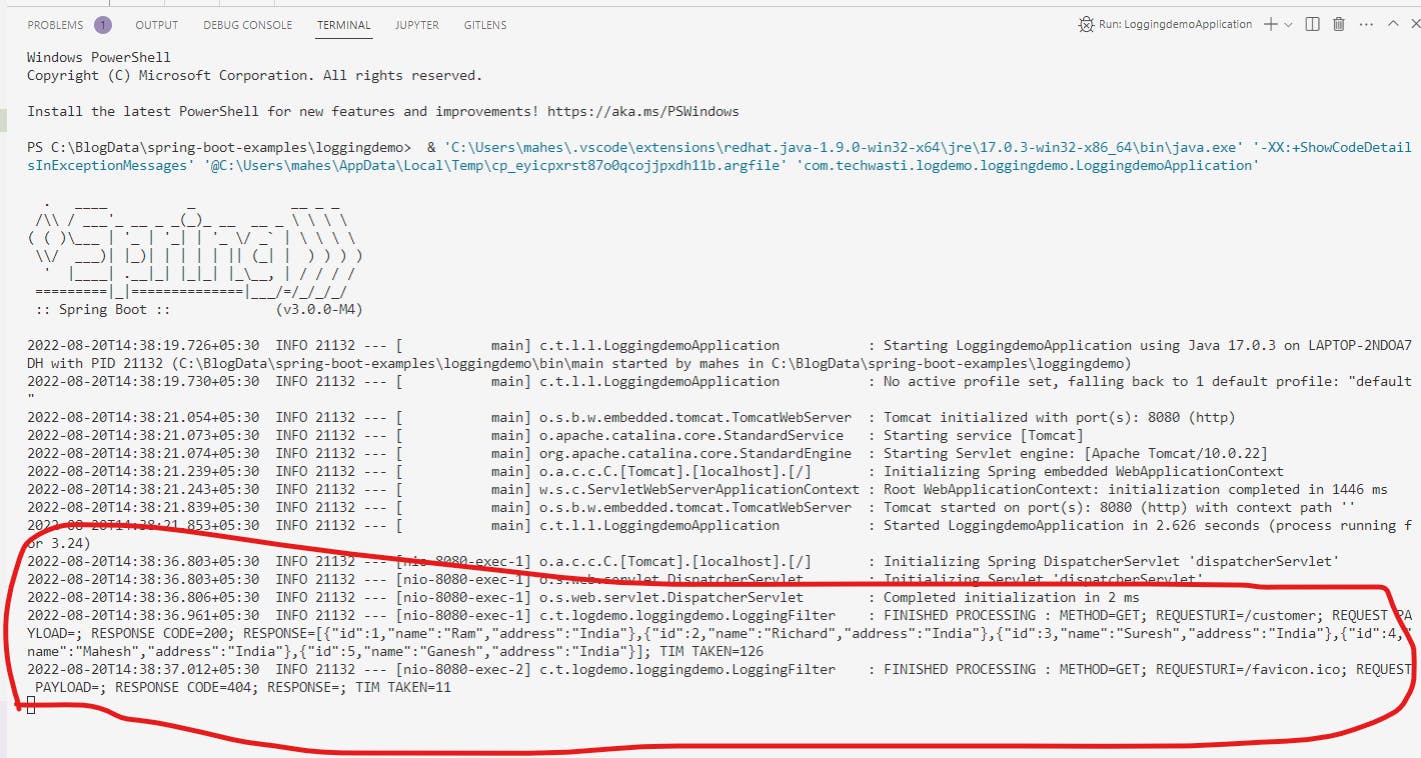
Inside our Rest controller class, we will not log any statement but our filter class will log the request and response body for each API call. So this approach will reduce the lines of code and we don’t need to worry about adding log statements in each API to print the request and response body. Let us see the filter class.

LoggingFilterclass will extend OncePerRequestFilterclass because this is a Filter base class that aims to guarantee a single execution per request dispatch, on any servlet container. we will override the doFilterInternal method with HttpServletRequestand HttpServletResponsearguments.

Filter class

|  |
| --- |
| @Component  public class LoggingFilter extends OncePerRequestFilter {  private static final Logger LOGGER = LoggerFactory.getLogger(LoggingFilter.class);  private String getStringValue(byte[] contentAsByteArray, String characterEncoding) {  try {  return new String(contentAsByteArray, 0, contentAsByteArray.length, characterEncoding);  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  }  return "";  }  @Override  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)  throws ServletException, IOException {  // TODO Auto-generated method stub  ContentCachingRequestWrapper requestWrapper = new ContentCachingRequestWrapper(request);  ContentCachingResponseWrapper responseWrapper = new ContentCachingResponseWrapper(response);  long startTime = System.currentTimeMillis();  filterChain.doFilter(requestWrapper, responseWrapper);  long timeTaken = System.currentTimeMillis() - startTime;  String requestBody = getStringValue(requestWrapper.getContentAsByteArray(),  request.getCharacterEncoding());  String responseBody = getStringValue(responseWrapper.getContentAsByteArray(),  response.getCharacterEncoding());  LOGGER.info(  "FINISHED PROCESSING : METHOD={}; REQUESTURI={}; REQUEST PAYLOAD={}; RESPONSE CODE={}; RESPONSE={}; TIM TAKEN={}",  request.getMethod(), request.getRequestURI(), requestBody, response.getStatus(), responseBody,  timeTaken);  responseWrapper.copyBodyToResponse();  }  } |

Now our structure is ready let us go ahead and run the application and check the logs Without adding extra code into our controller we are able to logging and just extension which is the beauty of spring boot.



Sample logs will be looks like this.

#Source Code:

Copy

Copy

git clone https://github.com/maheshwarLigade/spring-boot-examples.git

cd loggingdemo

gradle build

Conclusion:

This is the one way of intercepting the request, response, and logging the body or payload which helps you to debug and give a better idea without doing any changes in the business logic of the controller. Hope this will help you.

More such articles:

https://medium.com/techwasti

https://www.youtube.com/channel/UCiTaHm1AYqMS4F4L9zyO7qA

https://www.techwasti.com/

==========================\*\*=========================

If this article adds any value for you then please clap and comment.

Let’s connect on Stackoverflow, LinkedIn, & Twitter.

# 기타 참고 URL

[Spring] HTTP Request, Response 내용을 logging 적용해보기

https://beaniejoy.tistory.com/97

How to Autowire a Spring Bean in a Servlet Filter | Baeldung

https://www.baeldung.com/spring-autowire-bean-servlet-filter

[SpringBoot] Controller - Service 리펙토링 :: GOSE\_KOSE

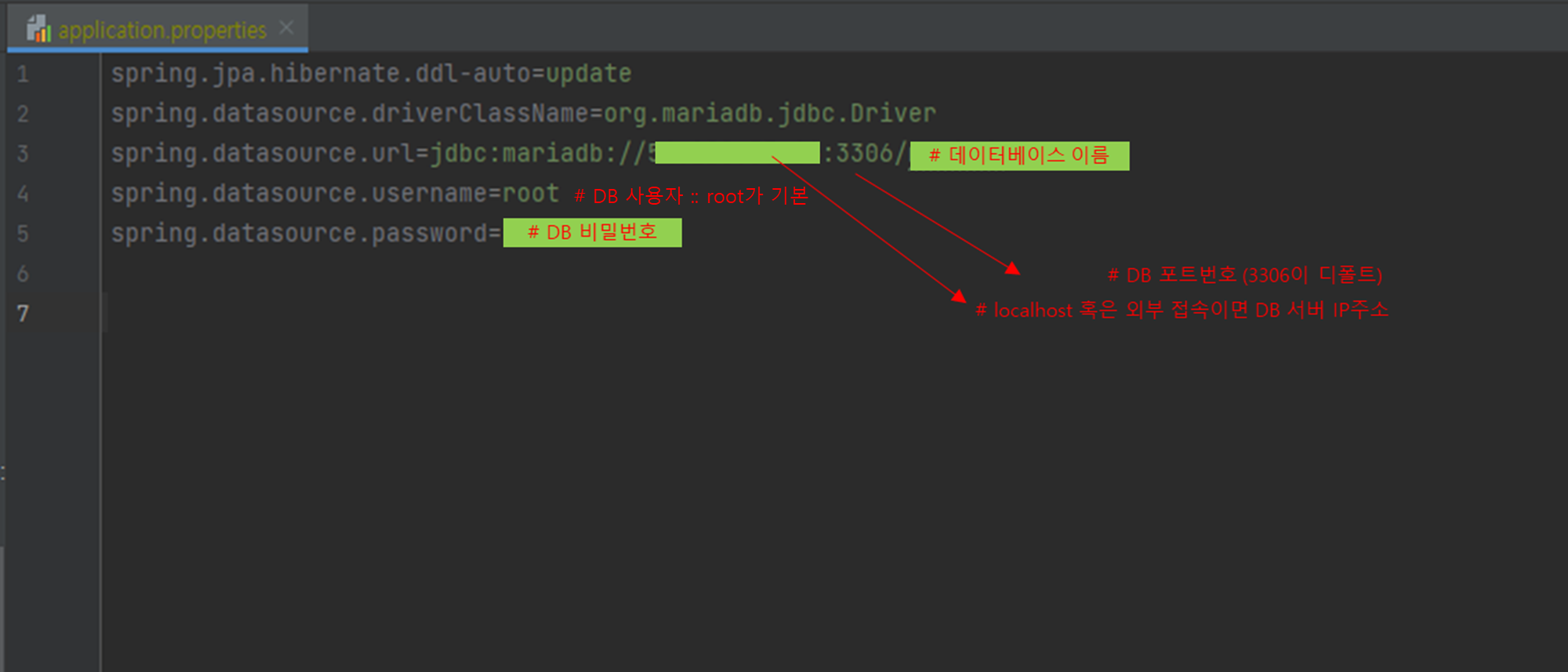
https://gose-kose.tistory.com/29

생성자 주입을 @Autowired를 사용하는 필드 주입보다 권장하는 하는 이유

https://madplay.github.io/post/why-constructor-injection-is-better-than-field-injection

https://velog.io/@sheltonwon/SpringBoot와-MariaDB-연결

properties 추가하기



그리고 /src/main/resources/application.properties파일을 열어 위와 같이 추가해준다.

spring.jpa.hibernate.ddl-auto=update

spring.datasource.driverClassName=org.mariadb.jdbc.Driver

spring.datasource.url=jdbc:mariadb://{DB IP주소}:3306/{데이터베이스이름::본게시물은 mystudy}

spring.datasource.username={DB 사용자명 :: root가 기본}

spring.datasource.password={비밀번호}

여기까지 완료했으면 MariaDB는 끝이다