스프링 MVC 설정

Spring Web MVC

스프링 MVC 빈 설정

원래는 Dispatcher servlet의 프로퍼티에 정의되어 있는 기본 전략을 사용하여 기본값들이 적용이 됨

예를 들어 설정을 해주지 않는다면 prefix와 suffix가 없는 상태로 사용이 되는데 밑에서 보는 것과 같이 View Resolver에다가 우리가 직접 설정을 할 수 있음

```
@Configuration
@ComponentScan
public class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        RequestMappingHandlerMapping handlerMapping = new RequestMappingHandlerMapping();
        return handlerMapping;
    }

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }

}
```

스프링 MVC 빈 설정

위에서 본 것과 같이 거의 대부분의 Bean들이 Cutomize할 수 있는 것들이 많음
Ex)handler mapping, handler adapter, etc…

```
@Bean
public HandlerMapping handlerMapping() {
    RequestMappingHandlerMapping handlerMapping = new RequestMappingHandlerMapping()
    handlerMapping.setInterceptors();
    handlerMapping.setOrder(Ordered.HIGHEST_PRECEDENCE);
    return handlerMapping;
}
```

위의 handlermapping.setinterceptors()의 경우 서블릿 필터와 비슷한 개념이며 서블릿보다 조금 더 유연함.

핸들러 매핑 뿐만 아니라 핸들러 어댑터 또한 new로 새로운 객체를 만들어서 설정 할 수 있는 것들이 많아서 이렇게 일일히 다 설정할 수 있음. (그냥 이러한 방법이 있다고만 알아두어도 ㄱㅊ)

스프링 부트가 있으면 알아서 다 설정을 해줌..

@EnableWebMvc

애노테이션(@) 기반의 컨트롤러를 사용할때 편리하도록 이러한 애노테이션을 지원함

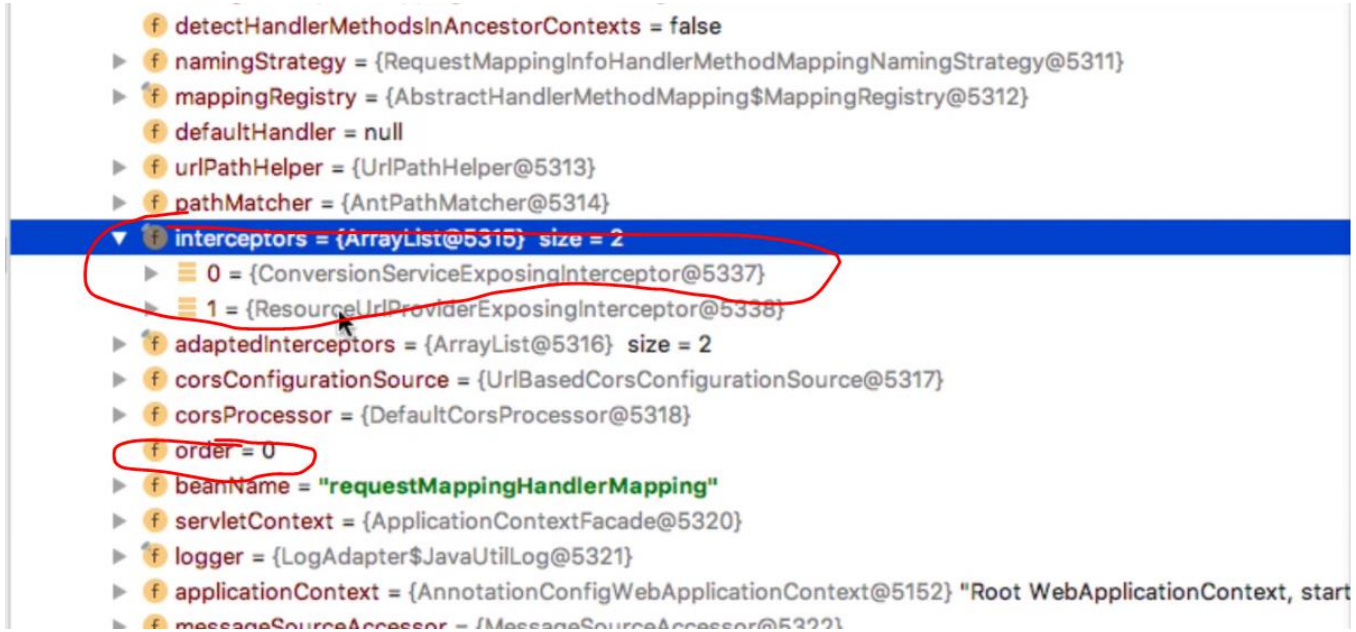
```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

다음과 같이 @Configuration과 같이 사용함

위 애노테이션을 사용하기 위해서는 웹어플리케이션이 실행되는 쪽에다가 setservletcontext를 사용하여
설정을 같이 해주어야 하는데 enablewebmvc가 불러오는 쪽(위임)에서 가끔 servletcontext를 참조하기 때문!

```
@Override
public void onStartup(ServletContext servletContext) throws ServletException {
    AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationC
    context.setServletContext(servletContext);
    context.register(WebConfig.class);
    context.refresh();

    DispatcherServlet dispatcherServlet = new DispatcherServlet(context);
    ServletRegistration.Dynamic app = servletContext.addServlet( servletName: "app", disp
    app.addMapping( ...urlPatterns: "/app/*");
}
```

@EnableWebMvc



디버거를 잡아서 보면 원래는 EnableWebMvc를 사용하지 않았을때 핸들러 매핑에서 Bean~이게 가장 먼저 왔는데 순서가 바뀌었고 request안에 들어가서 인터셉터를 보면 인터셉터가 아무것도 없었는데 enablewebmvc를 통해 배열이 생겼고 order부분도 순서가 기본값에서 0으로 바뀜!

@EnableWebMvc

핸들러 매핑, 핸들러 어댑터 둘다 requestMapping이 위에서 본 것같이 우선순위가 높음을 확인할 수 있고 (Enablewebmvc사용시) 이것을 먼저 사용하면 성능적인 이점이 있다고 함

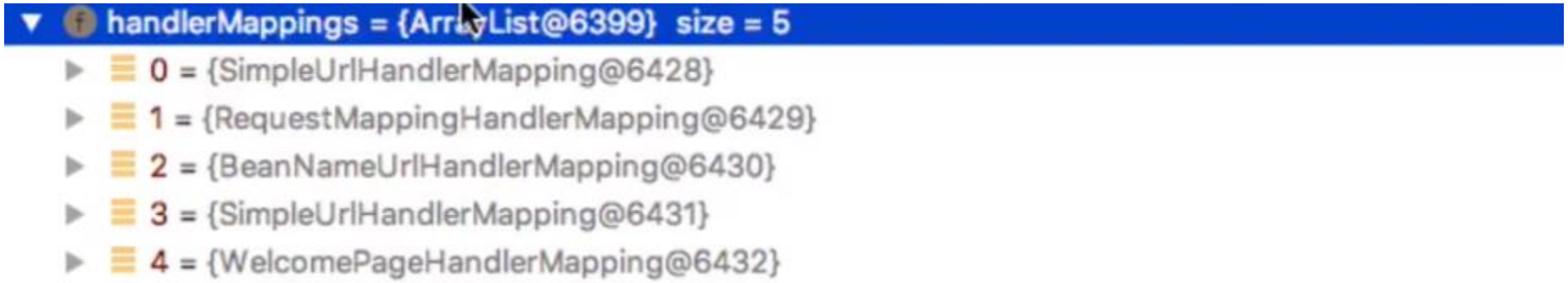
또한 @EnableWebMvc를 사용하게 되면 delegatingwebmvcconfiguration을 import 해오는데 이는 위임 구조라 코드를 손쉽게 기존의 빈에다가 인터셉터나 메시지 컨버터 같은 것들을 조금만 수정해서 사용할 수 있는 장점이 있음.

WebMvcConfigurer

확장 기능을 인터페이스를 통해 지원하고 그전에는 일일히 빈으로 등록하여 사용하였는데
인터페이스에서 제공하는 것들을 통해서 손쉽게 만들 수 있음!

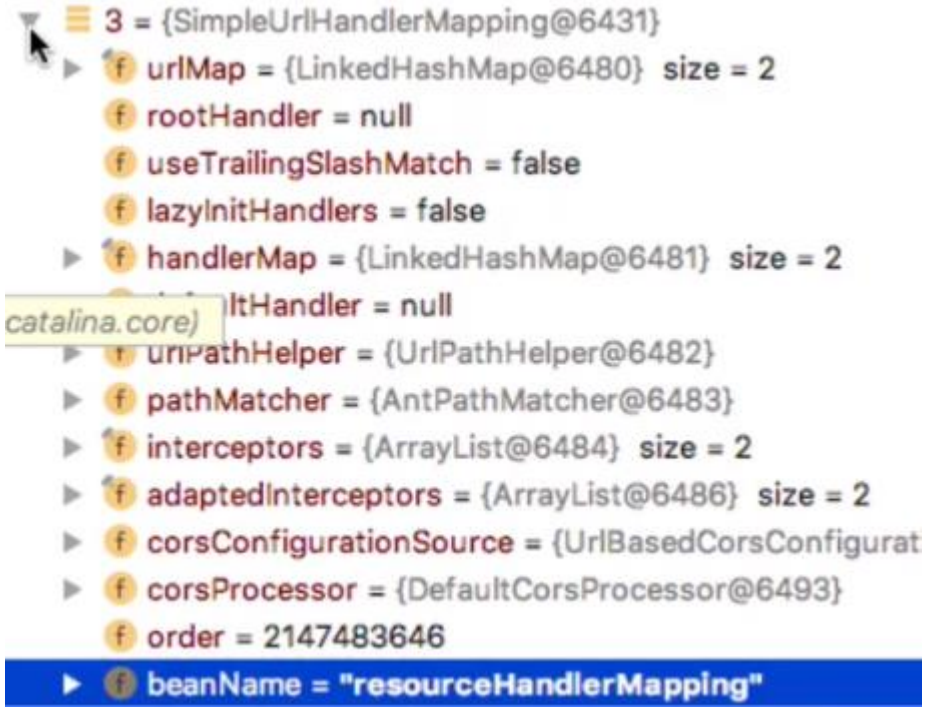

```
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp( prefix: "/WEB-INF/",  suffix: ".jsp");
    }
}
```

스프링 부트 스프링 MVC 설정



스프링 부트의 스프링 MVC 설정을 보면 @Enablewebmvc와 같이 비슷한 형태로 핸들러 매핑이 구성되어 있는 것을 볼 수 있음!(Request우선 그 다음 Bean~)

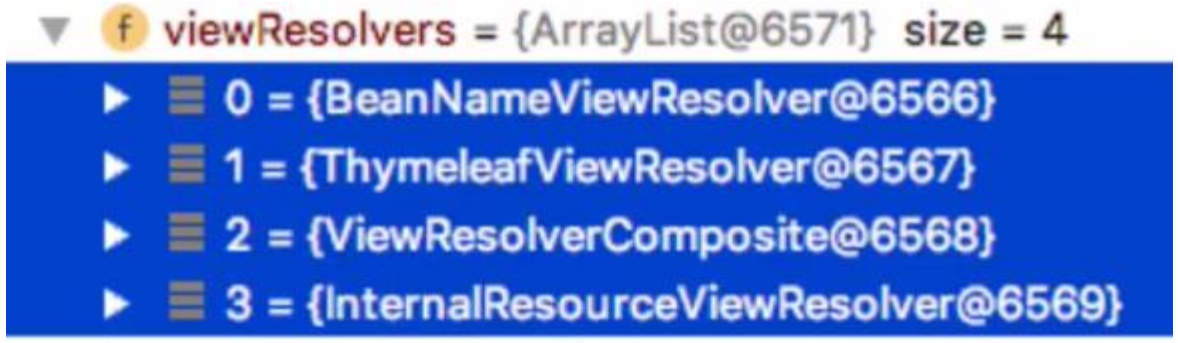
스프링 부트 스프링 MVC 설정



이 부분의 리소스 핸들러 매핑은 브라우저에서 캐싱과 관련된 부분.

스프링 부트 스프링 MVC 설정

뷰 리졸버 부분을 보면 ContentNegotiating 뷰 리졸버가 우선순위가 가장 최 상위로 되어 있으며 나머지 1,2,3,4를 위임하는 구조를 띔.



Content의 안의 view resolver

스프링 부트 스프링 MVC 설정

스프링 MVC 커스터마이징

- application.properties
- **@Configuration + Implements WebMvcConfigurer: 스프링 부트의 스프링 MVC 자동설정 + 추가 설정**
- @Configuration + @EnableWebMvc + Imlements WebMvcConfigurer: 스프링 부트의 스프링 MVC 자동설정 사용하지 않음.

대부분 2번의 경우가 가장 개발하기 쉬움.. <-- 추천!!

스프링 부트 JSP

앞으로 잘 사용하지 않을 기능!

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>
```

Jsp를 쓰기위해 의존성을 넣어줘야함
War패키지로 스프링 부트 프로젝트를 만들면 다음과 같이 이니셜 라이저를 생성해줌

```
public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(DemoJspApplication.class);
    }

}
```

스프링 부트 JSP

스프링 다큐맨테이션에 보면 jsp는 가급적 피하라고 추천하기 때문에 jsp를 잘 사용하지 않는다!

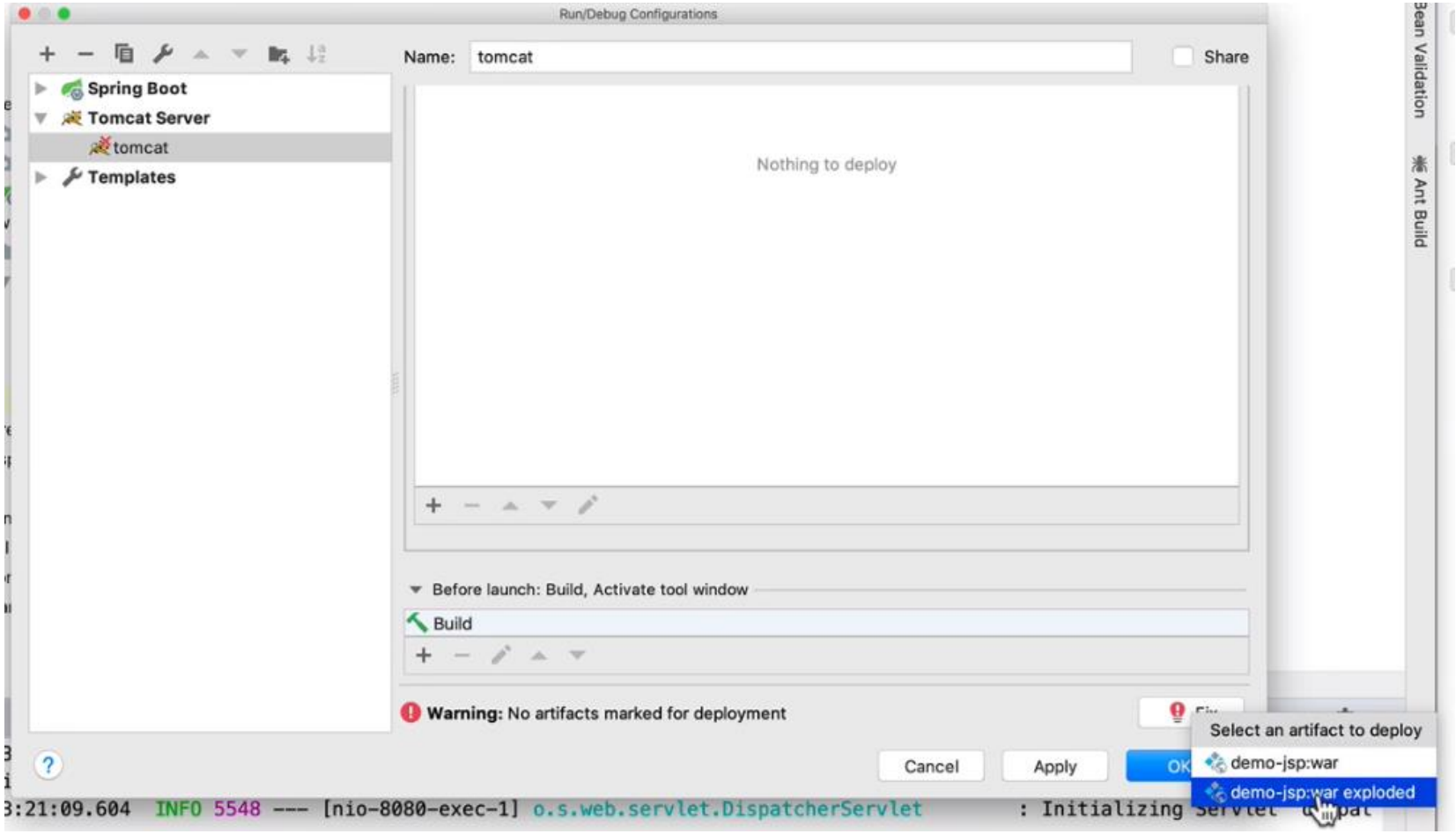
아래와 같은 이유로 잘 사용하지 않음!

제약 사항

- JAR 프로젝트로 만들 수 없음, WAR 프로젝트로 만들어야 함
- Java -JAR로 실행할 수는 있지만 “실행가능한 JAR 파일”은 지원하지 않음
- 언더토우(JBoss에서 만든 서블릿 컨테이너)는 JSP를 지원하지 않음
- Whitelabel 에러 페이지를 error.jsp로 오버라이딩 할 수 없음.

WAR 파일 배포하기

톰 캣을 다운받고 실행 가능한 파일을 만들어 주고 톰캣으로 실행 like below



WebMvcConfigurer Formatter

```java
@RestController
public class SampleController {

    @GetMapping("/hello/{name}")
    public String hello(@PathVariable("name") Person person) {
        return "hello " + person.getName();
    }

}
```

다음 상황에서 formatter가 없으면 person 객체에서 name을 바로 받을 수 없음!

```java
public class PersonFormatter implements Formatter<Person> {

    @Override
    public Person parse(String text, Locale locale) throws ParseException {
        Person person = new Person();
        person.setName(text);
        return person;
    }

    @Override
    public String print(Person object, Locale locale) {
        return object.toString();
    }
}
```

Printer와 parser를 합친것!

Formatter

- Printer: 해당 객체를 (Locale 정보를 참고하여) **문자열**로 어떻게 출력할 것인가
- Parser: 어떤 **문자열**을 (Locale 정보를 참고하여) 객체로 어떻게 변환할 것인가

WebMvcConfigurer Formatter

등록하는 방법 2가지

포매터 추가하는 방법 1

- WebMvcConfigurer의 addFormatters(FormatterRegistry) 메소드 정의

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatter(new PersonFormatter());
    }
}
```

포매터 추가하는 방법 2 (스프링 부트 사용시에만 가능 함)

- 해당 포매터를 빈으로 등록

```
@Component
public class PersonFormatter implements Formatter<Person> {
```

컴포넌트 애노테이션을 통해! 빈으로 등록!

WebMvcConfigurer Formatter

```java
@RunWith(SpringRunner.class)
@WebMvcTest
public class SampleControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void hello() throws Exception {
        this.mockMvc.perform(get( urlTemplate: "/hello")
                .param( name: "name",  ...values: "keesun"))
                .andDo(print())
                .andExpect(content().string( expectedContent: "hello keesun"));
    }

}
```

테스트에서도 코드를 정상적으로 실행시키기 위해서는
@SpringBootTest와 같이 통합 테스트로 변경 시켜주고
@AutoConfigureMockMvc를 통해 자동으로 빈으로 등록이 안된 MockMvc도 빈으로 등록 시켜주어야함

```java
@SpringBootTest
@AutoConfigureMockMvc
public class SampleControllerTest {
```

도메인 클래스 컨버터

의존성 설정

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

엔티티 맵핑

```
@Entity
public class Person {

   @Id @GeneratedValue
   private Integer id;
...
```

리파지토리 추가

```
public interface PersonRepository extends JpaRepository<Person, Integer> {
}
```

스프링 데이터 JPA가 제공하는 Repository를 사용해서 ID에 해당하는 엔티티를 읽어올 수 있음

이를 사용하여 id값을 가지고 그에 해당하는 value를 찾아서 값을 전해줌

핸들러 인터셉터

핸들러 매핑 -> 어떠한 요청을 처리해주는 것을 찾아주는 것

핸들러 인터셉터들을 핸들러 매핑에 설정함

```
// preHandle 1
// preHandle 2
// 요청 처리
// postHandler 2
// postHandler 1
// 뷰 랜더링
// afterCompletion 2
// afterCompletion 1
```

다음과 같은 순서대로 실행이 되고 posthandler와 afterCompletion같은 경우 역순으로 호출이 됨

핸들러 인터셉터

HandlerInterceptor

- 핸들러 맵핑에 설정할 수 있는 인터셉터
- 핸들러를 실행하기 전, 후(아직 랜더링 전) 그리고 완료(랜더링까지 끝난 이후) 시점에 부가 작업을 하고 싶은 경우에 사용할 수 있다.
- 여러 핸들러에서 반복적으로 사용하는 코드를 줄이고 싶을 때 사용할 수 있다.
    - 로깅, 인증 체크, Locale 변경 등...

boolean preHandle(request, response, **handler**)

- 핸들러 실행하기 전에 호출 됨
- "핸들러"에 대한 정보를 사용할 수 있기 때문에 서블릿 필터에 비해 보다 세밀한 로직을 구현할 수 있다.
- 리턴값으로 계속 다음 인터셉터 또는 핸들러로 요청,응답을 전달할지(true) 응답 처리가 이곳에서 끝났는지(false) 알린다.

void postHandle(request, response, **modelAndView**)

- 핸들러 실행이 끝나고 아직 뷰를 랜더링 하기 이전에 호출 됨
- "뷰"에 전달할 추가적이거나 여러 핸들러에 공통적인 모델 정보를 담는데 사용할 수도 있다.
- 이 메소드는 인터셉터 역순으로 호출된다.
- 비동기적인 요청 처리 시에는 호출되지 않는다.

void afterCompletion(request, response, handler, ex)

- 요청 처리가 완전히 끝난 뒤(뷰 랜더링 끝난 뒤)에 호출 됨
- preHandler에서 true를 리턴한 경우에만 호출 됨
- 이 메소드는 인터셉터 역순으로 호출된다.
- 비동기적인 요청 처리 시에는 호출되지 않는다.

vs 서블릿 필터

- 서블릿 보다 구체적인 처리가 가능하다.
- 서블릿은 보다 일반적인 용도의 기능을 구현하는데 사용하는게 좋다.

일반적인 기능을 구현할 시에는 서블릿 필터를 주로 쓰지만

스프링 MVC에 특화되어있게 만들려면 handler interceptor로 구현해야함!

Ex) xxs attack의 경우 서블릿 필터로 구현을 하여서 막아야 함!

핸들러 인터셉터

```
public class GreetingInterceptor implements HandlerInterceptor {

  @Override
  public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
      System.out.println("preHandle 1");
      return true;
  }

  @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
      System.out.println("postHandle 1");
  }

  @Override
  public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
      System.out.println("afterCompletion 1");
  }

}
```

옆의 그림과 같이 핸들러 인터셉터를 만들어주고 다른 핸들러 인터셉터를 만들어주어서 같이 찍어보면 위에서 설명한 것과 같이 posthandle부터는 역순으로 찍히는 것을 볼 수 있음!

핸들러 인터셉터

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new GreetingInterceptor()).order(0);
    registry.addInterceptor(new AnotherInterceptor())
        .addPathPatterns("/keeun")
        .order(-1);
  }
}
```

- 특정 패턴에 해당하는 요청에만 적용할 수도 있다.
- 순서를 지정할 수 있다

다음과 같은 특정 패턴이 들어올때만 핸들러가 작동할 수 있도록 설정할 수 있음!

리소스 핸들러

원래 서블릿 컨테이너가 기보능로 제공하는 디폴트 서블릿이 적용이 되어 있지만
이 위에다가 스프링 MVC 리소스 핸들러를 등록해서 정적 리소스를 처리할 수 있음

스프링 부트의 경우 기본 정적 리소스 핸들러를 제공!

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler( ...pathPatterns: "/mobile/**")
            .addResourceLocations("classpath:/mobile/")
            .setCacheControl(CacheControl.maxAge( maxAge: 10, TimeUnit.MINUTES));
}
```

리소스 핸들러를 등록하는 부분

Http 메시지 컨버터

HTTP 메시지 컨버터

- 요청 본문에서 메시지를 읽어들이거나(@RequestBody), 응답 본문에 메시지를 작성할 때(@ResponseBody) 사용한다.

RestController가 붙어있는경우 자동적으로 Responsebody가 붙어있는것임

@RequestBody를 통해서 객체를 받고, 그것을 @Responsebody를 통해 본문에 메시지를 작성!

이 메시지 컨버터도 다른 것과 마찬가지로 WebMvcConfigurer에서 설정이 가능함

HTTP 메시지 컨버터

보통의 설정 방법 – 의존성 추가로 컨버터 등록하기 때문에 직접 구현할 일이 많이 없음

설정 방법

- 기본으로 등록해주는 컨버터에 새로운 컨버터 추가하기: extendMessageConverters
- 기본으로 등록해주는 컨버터는 다 무시하고 새로 컨버터 설정하기: configureMessageConverters
- **의존성 추가로 컨버터 등록하기 (추천)**
  - **메이븐 또는 그래들 설정에 의존성을 추가하면 그에 따른 컨버터가 자동으로 등록 된다.**
  - **WebMvcConfigurationSupport**
  - (이 기능 자체는 스프링 프레임워크의 기능임, 스프링 부트 아님.)

HTTP 메시지 컨버터 JSON

기본적으로 JacksonJSON 2가 의존성에 들어있기 때문에 아무런 설정을
하지 않아도 빈으로 설정되어 있는 objectMapper를 사용해서 json 타입을 사용함

```
@Test
public void jsonMessage() throws Exception {
    Person person = new Person();
    person.setId(2019l);
    person.setName("keesun");

    String jsonString = objectMapper.writeValueAsString(person);

    this.mockMvc.perform(get( urlTemplate: "/jsonMessage")
                .contentType(MediaType.APPLICATION_JSON_UTF8)
                .accept(MediaType.APPLICATION_JSON_UTF8)
                .content(jsonString))
            .andDo(print())
            .andExpect(status().isOk())
    ;
}
```

HTTP 메시지 컨버터 XML

XML도 json과 마찬가지로 의존성을 추가해주면 됨
스프링 부트를 사용하는 경우 기본으로 xml 의존성을 추가해주지 않음

Marshaller 등록

```
@Bean
public Jaxb2Marshaller marshaller() {
   Jaxb2Marshaller jaxb2Marshaller = new Jaxb2Marshaller();
   jaxb2Marshaller.setPackagesToScan(Event.class.getPackageName());
   return jaxb2Marshaller;
}
```

다음과 같이 등록을 해준후 도메인 클래스 부분에 XmlRootElement 애노테이션을 추가해 주면 사용할 수 있음