Chapter2_4

1. Dispatcher-Servlet

정의

- HTTP 프로토콜로 들어오는 모든 요청을 가장 먼저 받아 적합한 컨트롤러에 위임해주는 프론트 Controller

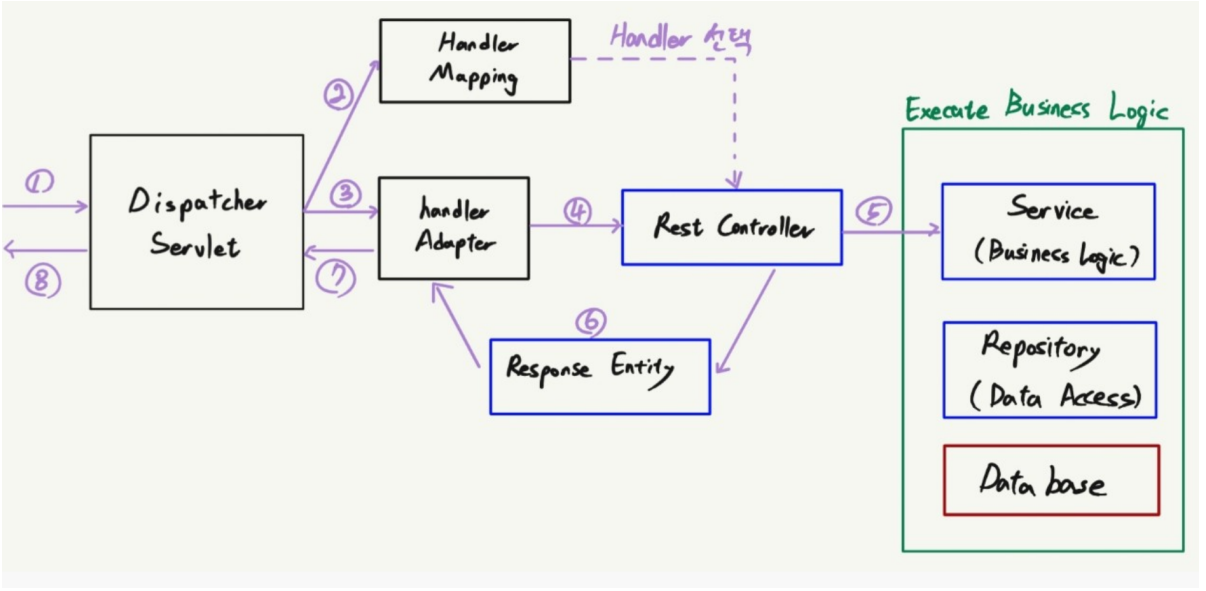
요청을 받는 과정

1. 클라이언트로부터의 요청
2. 서블릿 컨테이너가 요청을 받음
3. 모든 요청을 프론트 컨트롤러인 Dispatcher-Servlet이 먼저 받음
4. Distpatcher-Servlet이 공통적인 작업을 먼저 처리한 후 해당 요청을 처리해야 하는 컨트롤러를 찾아 작업을 위임

특징

- web.xml의 역할을 상당히 축소
  - 과거에는 모든 servlet을 URL 매핑을 위해 web.xml에 모두 등록해주어야 했지만 dispatcher-servlet이 해당 어플리케이션에 들어오는 모든 요청을 관리 & 공통적인 작업을 처리
- 모든 요청을 처리하다보니 이미지나 HTML/CSS/JavaScript 등과 같은 정적 파일에 대한 요청마저 모두 가로챔
  - 정적자원을 불러오지 못하는 상황 발생
    - 해결책
      1. 정적 자원에 대한 요청과 애플리케이션에 대한 요청 분리(코드가 지저분해져서 x)
      2. **애플리케이션에 대한 요청을 탐색하고 정적 자원에 대한 요청으로 처리(컨트롤러 먼저 찾고, 2차적으로 Resource를 탐색)**

Dispatcher-Servlet의 동작 과정



Dispatcher-Servlet 동작 과정

1. 클라이언트의 요청을 디스패처 서블릿이 받음

2. 요청 정보를 통해 요청을 위임할 컨트롤러를 찾음

- Handler Mapping이 요청을 위임할 컨트롤러를 찾음
- Handler Mapping의 구현체 중 하나인 RequestHandlerMapping은 모든 컨트롤러 빈을 파싱하여 HashMap으로 로 요청정보, 요청을 처리할 대상을 관리(요청에 매핑되는 컨트롤러, 메서드 등을 갖는 Handler Method 객체)
- 반환시 HandlerMethodExecutionChain으로 감싸서 반환(컨트롤러로 요청을 넘겨주기전에 처리해야하는 인터셉터등을 포함하기 위해)

3. 요청을 컨트롤러로 위임할 핸들러 어탭터를 찾아서 전달

- Adapter를 사용해서 컨트롤러로 요청을 위임
  - 공통적인 전/후처리 과정이 필요하기 때문
  - @RequestParam, @RequestBody 등을 처리하기 위한 ArgumentResolver들과 응답 시에 ResponseEntity의 Body를 Json으로 직렬화 하는 ReturnValueHandler들이 어탭터를 통해 처리
- HandlerAdapter 구현체인 RequestMappingHandlerAdapter를 찾고 앞서 찾은 HandlerMethodExecutionChain이 갖는 인터셉터들을 모두 실행한 다음 컨트롤러의 메소드를 호출하도록 요청을 위임

4. 핸들러 어댑터가 컨트롤러로 요청을 위임

- 리플렉션의 메서도 객체를 invoke

5. 비즈니스 로직 처리

6. 컨트롤러가 반환값 반환

- ResponseEntity 혹은 View 이름을 반환

7. HandlerAdapter가 반환값을 처리함

- 핸들러 어댑터가 반환값 처리컨트롤러로 요청을 위임한 HandlerAdapter는컨트롤러로부터 받은 응답을 ReturnValueHandler를 통해 후처리한 후에 디스패처 서블릿으로 돌려줌
  - 반환값이 ResponseEntity
    - HttpEntityMethodProcessor가 MessageConverter를 사용해 응답 객체를 직렬화하고 응답 상태(HttpStatus)를 설정
  - 반환값이 View 이름
    - View Resolver를 통해 View를 반환

8. 서버의 응답을 클라이언트로 반환

2. ViewResolver

- 사용자가 요청한 것에 대한 응답 view를 렌더링 하는 역할(view 이름으로부터 사용될 view 객체를 맵핑)
- 컨트롤러는 최종적으로 결과를 출력할 뷰와 뷰에 전달할 객체를 담고 있는 ModelAndView 객체를 리턴
- DispatherServlet은 ViewResolver를 사용하여 결과를 출력할 View 객체를 구하고, 구한 View 객체를 이용하여 내용을 생성
- Template Engine을 멤버로 갖는 어플리케이션이 초기화(생성)될 때, ServletContextTemplateResolver 객체가 생성되며, 이 templateResolver에 prefix(접두사)와 suffix(접미사)를 설정하고, Template Engine 객체를 생성해 엔진에 템플릿 리졸버를 꽂아주는 식이다.

```
public class GTVGApplication {


    ...
```

```
    private final TemplateEngine templateEngine;
    ...


    public GTVGApplication(final ServletContext servletContext) {

        super();

        ServletContextTemplateResolver templateResolver =
                new ServletContextTemplateResolver(servletContext);

        // HTML is the default mode, but we set it anyway for better understanding of
code
        templateResolver.setTemplateMode(TemplateMode.HTML);
        // This will convert "home" to "/WEB-INF/templates/home.html"
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        // Template cache TTL=1h. If not set, entries would be cached until expelled
        templateResolver.setCacheTTLMs(Long.valueOf(3600000L));

        // Cache is set to true by default. Set to false if you want templates to
        // be automatically updated when modified.
        templateResolver.setCacheable(true);

        this.templateEngine = new TemplateEngine();
        this.templateEngine.setTemplateResolver(templateResolver);

        ...

    }

}
```

3. 뷰 컨트롤러 작업

현재까지 Controller 작성 패턴

- 해당 Class 위에 @Controller 사용(해당 Class가 컨트롤러 클래스 임을 나타내기 위함&Bean으로 생성됨)
- 자신이 처리하는 요청 패턴을 정의하기 위해 클래스 수준의 @RequestMapping 사용
- 메서드의 어떤 종류의 요청을 처리해야 하는지 나타내기 위해 @GetMapping/@PostMapping이 지정된 하나 이상의 메서드를 가짐

View Controller(뷰 컨트롤러)

- Model 데이터나 사용자 입력을 처리하지 않는 간단한 Controller의 경우 다른 방법으로 정의가 가능한데 View에 요청을 전달하는 일만 하는 컨트롤러

```
@GetMapping("/design")
public String designTaco() {
    return "design";
}
```

위 코드처럼 특정 요청 URL에 대해 특정 로직 없이 바로 View를 return하는 경우 자주 사용

```
package tacos.web;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer{
        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
                registry.addViewController("/").setViewName("home");
        }
}
```

<코드 분석>

- public class WebConfig implements WebMvcConfigurer

해당 Class가 WebMvcConfigurer 인터페이스 구현

- public void addViewControllers(ViewControllerRegistry registry)

하나 이상의 View Controller를 등록하기 위해 사용할 수 있는 ViewControllerRegistry를 인자로 받음.

- registry.addViewController("/").setViewName("home");

GET 요청을 처리하는 경로인 "/"를 인자로 전달하여 addViewController()를 호출
addViewController()는 ViewControllerRegistration 객체를 반환

"/"경로의 요청이 전달되어야 하는 View로 home을 지정(.setViewName("home"))

즉, "/" url이 요청되면 home이라는 view로 이동

WebMvcConfigurer

[@EnableWebMvc]

- 과거에는 메세지를 변환하는 메세지 컨버터나 뷰를 렌더링 하기 위한 뷰 리졸버 등을 일일이 설정
- 설정을 자동화하는 기능을 제공할 필요성을 느낌
  - @Enable~ 어노테이션 탄생
- 스프링이 제공하는 웹과 관련된 최신 전략 빈들이 등록

[Spring Boot]

- @Enable이라는 어노테이션을 붙여주는 것도 번거롭기에 자동화 해주는 Spring boot라는 프레임워크를 만듦.
- Spring Boot의 AutoConfigure(자동 구성)기능을 통해 많은 설정이 자동화
- @SpringBootApplication 어노테이션 내부에 @EnableAutoConfiguration이 포함(@EnableWebMvc와 동일한 기능)
- 메세지 컨버터(Message Converter)나 뷰 리졸버(View Resolver) 또는 인터셉터(Interceptor) 등을 따로 설정해주지 않아도 되고, 추가로 @EnableWebMvc 기반의 설정도 추가하지 않아도 됨.

[~Configurer 인터페이스을 통한 설정의 변경]

- 스프링이 제공해주는 자동 설정들 외에 추가의 설정이 필요할 수 있음
- @Enable로 적용되는 인프라 빈에 대해 추가적인 설정을 할 수 있도록 ~Configurer로 끝나는 인터페이스(빈 설정자)를 제공
- 이를 구현한 클래스를 만들어 빈으로 등록하면 @Enable 전용 어노테이션을 처리하는 단계에서 설정용 빈을 활용해 인프라 빈의 설정을 마무리
- 대표적으로 @EnableWebMvc의 빈 설정자는 WebMvcConfigurer이며, 이를 구현한 클래스를 만들고 @Configuration을 붙붙여 빈으로 등록

- WebMvcConfigurer에서 필요한 메서드만을 오버라이딩하도록 default 메서드 제공

```
public interface WebMvcConfigurer {
        default void configurePathMatch(PathMatchConfigurer configurer) {
        }
         default void configureContentNegotiation(ContentNegotiationConfigurer
configurer) {
        }
        default void configureAsyncSupport(AsyncSupportConfigurer configurer) {
        }
        default void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        }
        default void addFormatters(FormatterRegistry registry) {
        }
         default void addInterceptors(InterceptorRegistry registry) {
        }
        default void addResourceHandlers(ResourceHandlerRegistry registry) {
        }
        default void addCorsMappings(CorsRegistry registry) {
        }
        default void addViewControllers(ViewControllerRegistry registry) {
        }
        default void configureViewResolvers(ViewResolverRegistry registry) {
        }
        default void addArgumentResolvers(List<HandlerMethodArgumentResolver>
resolvers) {
        }
        default void addReturnValueHandlers(List<HandlerMethodReturnValueHandler>
handlers) {
        }
        default void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
        }
        default void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
        }
        default void configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
        }
        default void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
        }
        @Nullable
        default Validator getValidator() {
                return null;
        }
        @Nullable
        default MessageCodesResolver getMessageCodesResolver() {
                return null;
        }
```

```
}
```

- add~: 기본 설정이 없는 빈들에 대하여 새로운 빈을 추가함
- configure~: 수정자를 통해 기존의 설정을 대신하여 등록함
- extend~: 기존의 설정을 이용하며 추가로 설정을 확장함

※ 스프링에서 제공해주는 웹 기능들에 추가적으로 커스터마이징을 하기를 원한다면 @EnableWebMvc없이 WebMvcConfigurer를 구현한 설정 파일만 등록