Spring MVC 05

[프론트 컨트롤러 패턴 - 2]

스프링 MVC 1편 - 백엔드 웹 개발 핵심 기술

김영한

2022.08.21

프론트 컨트롤러 V4 - 단순하고 실용적

V3는 잘 설계 되었지만 실제 컨트롤러 인터페이스 구현이 복잡하기 때문에 편리하게 개편해야 한다. - 실용성을 올려야한다.

```
Map<String, Object> model = new HashMap<>();
```

Model를 사용해서 넘겨주고 mv 대신 Controller에서 viewPath 논리 경로만 리턴해준다.

```
public interface ControllerV4{

   String process(Map<String, String> paramMap, Map<String, Object> model);
}
```

단순하고 실용적이다.

model Map을 넘겨주면 Controller에서 채우기만 하면 된다.

⇒ 여러가지 방식으로 사용하고 싶으면 어뎁터를 사용하면 된다. (V5 버전)

어댑터 패턴

여러가지 컨트롤러 방식을 적용하고 싶어지기 때문에 생긴 패턴

컨트롤러V3, V4를 동시에 사용하고 싶다면? 즉 ModelAndView 객체를 쓰고 싶거나 Model만 쓰고싶거나 하는 경우가 생기는 것.

어댑터를 통해서 컨트롤러를 호출하게 되면 컨트롤러가 사용되는 메소드 방식을 정해서 사용할수 있다는 컨셉.

핸들러 어댑터

어댑터가 컨트롤러를 처리할 수 있는지 판단하는 메소드.

모델엔 뷰를 반환하는 메소드

핸들러 어댑터

```
public interface MyHandlerAdapter {

   boolean supports(Object handler);

   ModelView handle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws ServletException, IOException;
}
```

핸들러 어댑터 구현 - **ModelView** 객체 사용

```
public class ControllerV3HandlerAdapter implements MyHandlerAdapter {
   @Override
   public boolean supports(Object handler) {
       return (handler instanceof ControllerV3);
   } V3 컨트롤러인 경우를 확인

   @Override
   public ModelView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws ServletException, IOException {
       ControllerV3 controller = (ControllerV3) handler;

       Map<String, String> paramMap = createParamMap(request);
       ModelView mv = controller.process(paramMap);

       return mv;
   }

   private Map<String, String> createParamMap(HttpServletRequest request) {
       //paramMap에 request에 있는 파라미터를 전부 PUT
       Map<String, String> paramMap = new HashMap<>();
       request.getParameterNames().asIterator()
               .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
       return paramMap;
   }
```

```
}
```

핸들러 어댑터 구현 **V4 - Model** 객체만 사용

-ModelView 객체를 생성해서 보내준다 (어댑터의 역할)

```
public class ControllerV4HandlerAdapter implements MyHandlerAdapter {
   @Override
   public boolean supports(Object handler) {
       return (handler instanceof ControllerV4);
   }

   @Override
   public ModelView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws ServletException, IOException {
       ControllerV4 controllerV4 = (ControllerV4) handler;

       Map<String, String> paramMap = createParamMap(request);
       Map<String, Object> model = new HashMap<>();
       String viewName = controllerV4.process(paramMap, model);
       String만 리턴해주는 controllerV4의 경우를 mv로 확장시켜서 리턴해준다.
       ModelView mv = new ModelView(viewName);
       mv.setModel(model);

       return mv;
   }

   private Map<String, String> createParamMap(HttpServletRequest request) {
       Map<String, String> paramMap = new HashMap<>();
       request.getParameterNames().asIterator()
               .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
       return paramMap;
   }
}
```

프론트 컨트롤러 **V3**구현

```
@WebServlet(name = "frontControllerServletV5", urlPatterns =
```

```
"/front-controller/v5/*")
public class FrontControllerServletV5 extends HttpServlet {

   private final Map<String, Object> handlerMappingMap = new HashMap<>();
   private final List<MyHandlerAdapter> handlerAdapters = new ArrayList<>();
   각 핸들러 어댑터구현체 들을 담아둘 리스트
   public FrontControllerServletV5() {
       initHandlerMappingMap();
       핸들러( 컨트롤러 ) 들을 매핑해준다
       initHandlerAdapters();
       리스트에 구현체를 초기화 해준다.
   }

   private void initHandlerMappingMap() { 핸들러( 컨트롤러 ) 들을 매핑해준다
       handlerMappingMap.put("/front-controller/v5/v3/members/new-form", new
MemberFormControllerV3());
       handlerMappingMap.put("/front-controller/v5/v3/members/save", new
MemberSaveControllerV3());
       handlerMappingMap.put("/front-controller/v5/v3/members", new
MemberListControllerV3());
       . . . + v5/v4/ url, Controller 추가
   }

   private void initHandlerAdapters() {리스트에 구현체를 초기화 해준다.
       handlerAdapters.add(new ControllerV3HandlerAdapter());
       handlerAdapters.add(new ControllerV4HandlerAdapter());
   }

   @Override
   protected void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

       Object handler = getHandler(request);

       if (handler == null){
           response.setStatus(HttpServletResponse.SC_NOT_FOUND);
           return;
       }

       MyHandlerAdapter adapter = getHandlerAdapter(handler);

       ModelView mv = adapter.handle(request, response, handler);
       각 핸들러 어댑터에서 hadle매소드에서 구현된 ModelView객체를 반환한다.

       String viewName = mv.getViewName(); // 논리 이름 밖에 못 얻는다.
       MyView view = viewResolver(viewName);
```

```
        view.render(mv.getModel(), request, response);
    }

    private MyHandlerAdapter getHandlerAdapter(Object handler) {
        for (MyHandlerAdapter adapter : handlerAdapters) {
            if(adapter.supports(handler)){
                return adapter;
            }
        }
        throw new IllegalArgumentException("handler adapter를 찾을 수 없습니다." +
handler);
    }

    private Object getHandler(HttpServletRequest request) {
        String requestURI = request.getRequestURI();
        return handlerMappingMap.get(requestURI);
    }
    //논리 이름 --> 물리 이름
    private MyView viewResolver(String viewName) {
        return new MyView("/WEB-INF/views/" + viewName + ".jsp");
    }
}
```

정리

프론트 컨트롤러에서 OCP를 지킬 수 있다. 기능을 확장하더라도 어댑터만 추가해주면 된다.

기능을 추가해도 로직 돌아가는 것은 똑같다. init 부분을 다른 곳으로 이동시킨다면 OCP를 더 확실하게 구현 가능할 것.

요약

v1: 프론트 컨트롤러를 도입 기존 구조를 최대한 유지하면서 프론트 컨트롤러를 도입

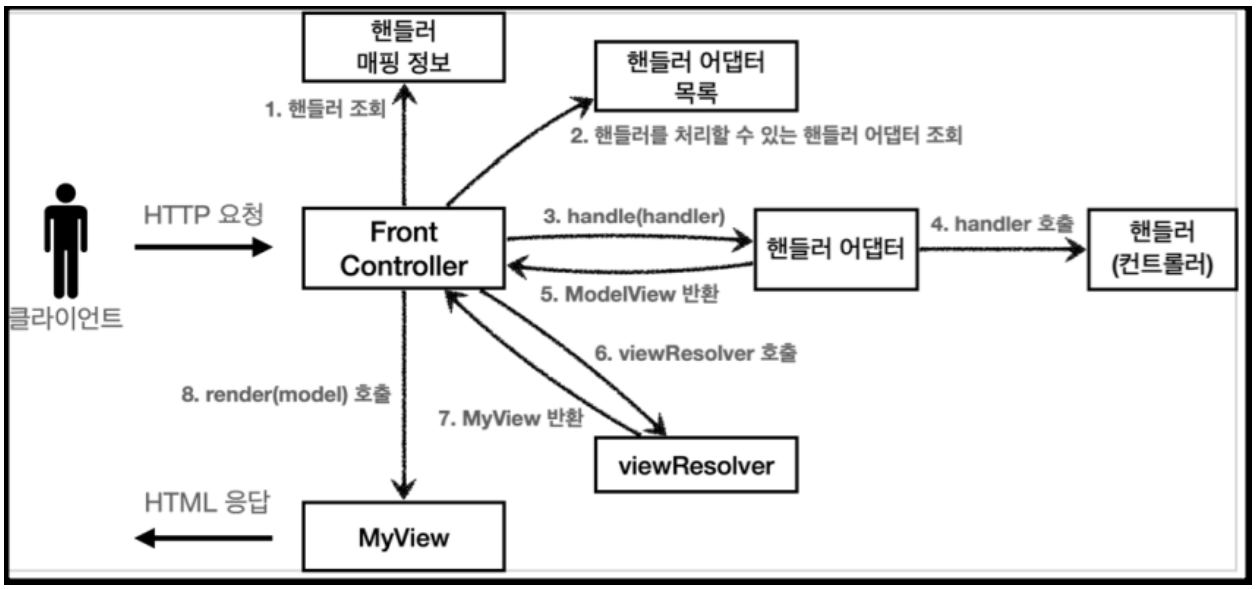
v2: View 분류 단순 반복 되는 뷰 로직 분리

v3: Model 추가 서블릿 종속성 제거 뷰 이름 중복 제거

v4: 단순하고 실용적인 컨트롤러 v3와 거의 비슷 구현 입장에서 ModelView를 직접 생성해서 반환하지 않도록 편리한 인터페이스 제공

v5: 유연한 컨트롤러 어댑터 도입 어댑터를 추가해서 프레임워크를 유연하고 확장성 있게 설계

최종 구현. 스프링도 같은 구조를 가지고 있다.



확장이 필요하다면 핸들러 어댑터만 추가해주면 여러 컨트롤러를 생성시킬수 있다.