# MVC 패턴 – 요청을 구분하여 기능 제공

## 요청의 구분

컨트롤러가 기능에 맞는 로직을 수행하기 위해 클라이언트가 어떤 기능(서비스)을 요청하는 지 구분할 수 있어야 함.

## 요청 기능 구분 방식

* 특정 이름의 파라미터에 명령어 정보를 전달하는 방식(파라미터 입력 방식)
* 요청 URI를 명령어로 사용하는 방식(URI 입력 방식)

## 명령어 정보 기반의 코드

- 사용자가 파라미터로 입력한 명령어를 컨트롤러 서블릿에 전송

- 컨트롤러 서블릿은 사용자가 전송한 명령어에 해당하는 기능을 처리할 핸들러를 생성

- 해당 핸들러는 비즈니스 로직(기능)을 수행하여 결과를 저장

- 뷰를 통해서 결과를 보여주는 방식을 사용

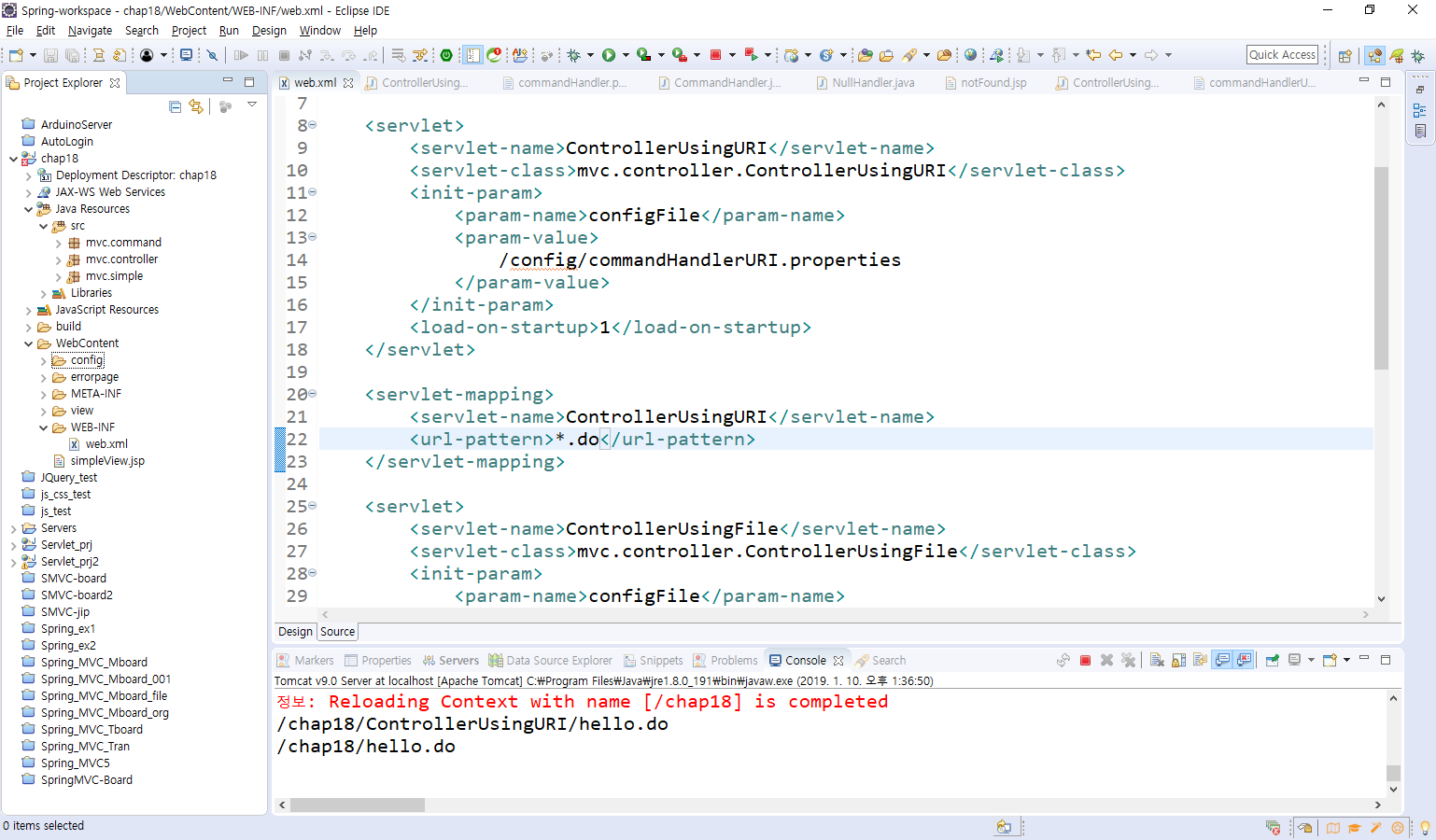
## 커맨드 패턴

각 요청을 처리하는 별도 클래스를 제공하는 구현 패턴.

커맨드(기능)에 따른 핸들러(기능처리) 클래스를 공통 인터페이스로 구현하는 형태.

## 핸들러 클래스의 구성

커맨드를 처리하는 클래스들을 동일한 인터페이스로 구현

 <프로젝트구조>

## 핸들러 인터페이스

command 패키지를 따로 생성하여 부모에 해당하는 인터페이스와 각 커맨드에 대응하는 핸들러 클래스를 구현

예제) CommandHandler interface

|  |
| --- |
| **package** mvc.command;  **public** **interface** CommandHandler {  **public** String process(HttpServletRequest req, HttpServletResponse res)  **throws** Exception;  } |

## 기능별 핸들러 클래스 구현

1. NullHandler

제공하지 않는 기능에 대한 핸들러(예외사항 처리)

|  |
| --- |
| **public** **class** NullHandler **implements** CommandHandler {  @Override  **public** String process(HttpServletRequest req, HttpServletResponse res)  **throws** Exception {  res.sendError(HttpServletResponse.***SC\_NOT\_FOUND***);  **return** **null**;  }  } |

2. HelloHandler

주 기능에 대한 핸들러 : JSP로 데이터 전달

|  |
| --- |
| **public** **class** HelloHandler **implements** CommandHandler {  @Override  **public** String process(HttpServletRequest req, HttpServletResponse res) {  req.setAttribute("hello", "안녕하세요!");  **return** "/view/hello.jsp";  }  } |

## Properties 파일을 활용한 서블릿 초기화(파라미터 입력 방식)

프로퍼티 파일이란 프로그램의 구성에 필요한 파라미터들을 저장하기 위해 사용하는 파일.

매핑 정보를 코드가 아닌 별도 설정 파일에 저장하여 새로운 커맨드의 추가나 클래스 변경 시 편리하게 적용할 수 있음. 여기서는 ‘hello’라는 커맨드를 처리하는 클래스로 ‘HelloHandler’를 지정

지정된 경로에 xxx.properties 파일을 생성. WebContent/config 폴더를 생성하여 그 밑에 저장

예) /config/commandHandler.properties

|  |
| --- |
| hello=mvc.command.HelloHandler |

프로퍼티 파일의 위치 정보는 web.xml에 init-param 속성으로 지정하여 서블릿 초기화 시 불러들임.

|  |
| --- |
| <servlet>  <servlet-name>ControllerUsingFile</servlet-name>  <servlet-class>mvc.controller.ControllerUsingFile</servlet-class>  <init-param>  <param-name>configFile</param-name>  <param-value>/config/commandHandler.properties</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>ControllerUsingFile</servlet-name>  <url-pattern>/controllerUsingFile</url-pattern>  </servlet-mapping> |

<load-on-startup>1</load-on-startup> : 0보다 큰 숫자로 설정되어 있을 경우 서버가 시작할 때 초기화를 진행.

이 요소가 없을 경우 초기화를 매번 서블릿 생성 시 마다 진행. 값이 많을 경우 지연 발생.

숫자 값은 실행 횟수가 아니라 우선 순위. 같은 숫자일 경우 작성된 순서대로 처리

## 커맨드와 클래스의 관계 명시하기(파라미터 입력 방식) - (com.controller.ControllerUsingFile)

컨트롤러의 초기화 메서드(init)에서 프로퍼티 파일을 불러와서 커맨드와 핸들러 클래스를 Map을 사용하여 매핑.

|  |
| --- |
| **private** Map<String, CommandHandler> commandHandlerMap = **new** HashMap<>();  **public** **void** init() **throws** ServletException {  String configFile = getInitParameter("configFile");  Properties prop = **new** Properties();  String configFilePath = getServletContext().getRealPath(configFile);  **try** (FileReader fis = **new** FileReader(configFilePath)) {  prop.load(fis);  } **catch** (IOException e) {  **throw** **new** ServletException(e);  }  Iterator keyIter = prop.keySet().iterator();  **while** (keyIter.hasNext()) {  String command = (String) keyIter.next();  String handlerClassName = prop.getProperty(command);  **try** {  Class<?> handlerClass = Class.*forName*(handlerClassName);  CommandHandler handlerInstance =  (CommandHandler) handlerClass.newInstance();  commandHandlerMap.put(command, handlerInstance);  } **catch** (ClassNotFoundException | InstantiationException  | IllegalAccessException e) {  **throw** **new** ServletException(e);  }  }  } |

Properties 객체 : 프로퍼티 파일로부터 읽어온 설정 정보를 저장/유지하는 객체.

ServletContext로부터 실제 경로(real-path)를 구하여 프로퍼티 파일을 읽어오는데 사용.

Properties 클래스의 load() 메서드는 파일로부터 설정 정보를 읽어오는 메서드.

읽어온 설정 정보는 key, value 형태로 저장되며, Properties 클래스의 keySet() 메서드로 key 목록을 구하여 사용할 수 있음. 본 예제에서 key는 ‘hello’, value는 ‘mvc.command.HelloHandler’가 됨.

Value 값은 getProperty() 메서드로 구함. 즉 패키지와 클래스 이름을 구하게 됨.

커맨드를 처리할 클래스의 인스턴스(객체)를 구하기 위해 Class.forName(클래스이름) 사용.

Class.forName(“패키지.클래스”)은 문자열로 입력된 ‘패키지.클래스’ 이름에 해당하는 클래스를 찾아옴.

찾아온 클래스의 인스턴스 생성은 newInstance() 메서드로 처리. 상위(Class)에서 하위(CommandHandler)에 객체를 넘기기 때문에 형변환 처리.

위의 작업은 커맨드 개수만큼 진행해야 하기 때문에 반복 처리

## 요청 처리의 일원화 - (com.controller.ControllerUsingFile)

요청 방식(get/post)에 상관없이 사용자의 커맨드를 처리

|  |
| --- |
| **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  process(request, response);  }  **protected** **void** doPost(HttpServletRequest request,  HttpServletResponse response) **throws** ServletException, IOException {  process(request, response);  } |

## 커맨드에 따른 핸들러 실행 - (com.controller.ControllerUsingFile)

실질적인 커맨드의 처리는 process() 메서드에서 처리

|  |
| --- |
| **private** **void** process(HttpServletRequest request,  HttpServletResponse response) **throws** ServletException, IOException {  String command = request.getParameter("cmd");    CommandHandler handler = commandHandlerMap.get(command);  **if** (handler == **null**) {  handler = **new** NullHandler();  }  String viewPage = **null**;  **try** {  viewPage = handler.process(request, response);  } **catch** (Throwable e) {  **throw** **new** ServletException(e);  }  **if** (viewPage != **null**) {  RequestDispatcher dispatcher = request.getRequestDispatcher(viewPage);  dispatcher.forward(request, response);  }  } |

커맨드와 핸들러를 매핑한 Map에서 요청 파라미터로 입력받은 커맨드에 해당하는 핸들러를 구하여 실행

실행 결과로 JSP 페이지의 위치정보와 이름을 얻는다.

## 화면 처리(JSP)

핸들러 클래스에 지정한 화면용 페이지 작성. 페이지의 위치(경로)에 주의.(/view/hello.jsp)

|  |
| --- |
| <%@ page contentType = *"text/html; charset=utf-8"* %>  <html>  <head><title>Hello</title></head>  <body>  ${hello}  </body>  </html> |

## 실행



## Not Found 예외 처리

web.xml에 에러 코드 설정

|  |
| --- |
| <error-page>  <error-code>404</error-code>  <location>/errorpage/notFound.jsp</location>  </error-page> |

notFound.jsp

|  |
| --- |
| <%@ page contentType = *"text/html; charset=utf-8"* %>  <html>  <head><title>페이지 오류</title></head>  <body>  해당 페이지가 없습니다.  </body>  </html> |

## Properties 파일을 활용한 서블릿 초기화(URI 입력 방식)

파라미터 입력 방식과 동일. 커맨드 대신 uri에 대한 핸들러 클래스 지정

예) /config/commandHandlerURI.properties

|  |
| --- |
| /hello.do=mvc.command.HelloHandler |

프로퍼티 파일의 위치 정보는 web.xml에 init-param 속성으로 지정하여 서블릿 초기화 시 불러들임.

|  |
| --- |
| <servlet>  <servlet-name>ControllerUsingURI</servlet-name>  <servlet-class>mvc.controller.ControllerUsingURI</servlet-class>  <init-param>  <param-name>configFile</param-name>  <param-value>/config/commandHandlerURI.properties</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>ControllerUsingURI</servlet-name>  <url-pattern>\*.do</url-pattern>  </servlet-mapping> |

<url-pattern>에 ‘\*’ 기호를 사용하면 ‘.do’라고 요청하는 모든 URI에 대해서 서블릿을 연결함.

## 커맨드와 클래스의 관계 명시하기와 요청 처리의 일원화 - (com.controller.ControllerUsingURI)

파라미터 입력 방식과 동일

## 커맨드에 따른 핸들러 실행 - (com.controller.ControllerUsingURI)

커맨드를 획득하는 방식만 다른 뿐 나머지는 파라미터 방식과 동일

|  |
| --- |
| **private** **void** process(HttpServletRequest request,  HttpServletResponse response) **throws** ServletException, IOException {    String command = request.getRequestURI();    **if** (command.indexOf(request.getContextPath()) == 0) {  command = command.substring(request.getContextPath().length());  }  ………//파라미터 방식과 동일  } |

전체 요청 URI에서 ContextPath를 뺀 나머지가 요청 커맨드인 URI가 됨.

## 화면 처리(JSP)

파라미터 입력 방식과 동일

## 실행

