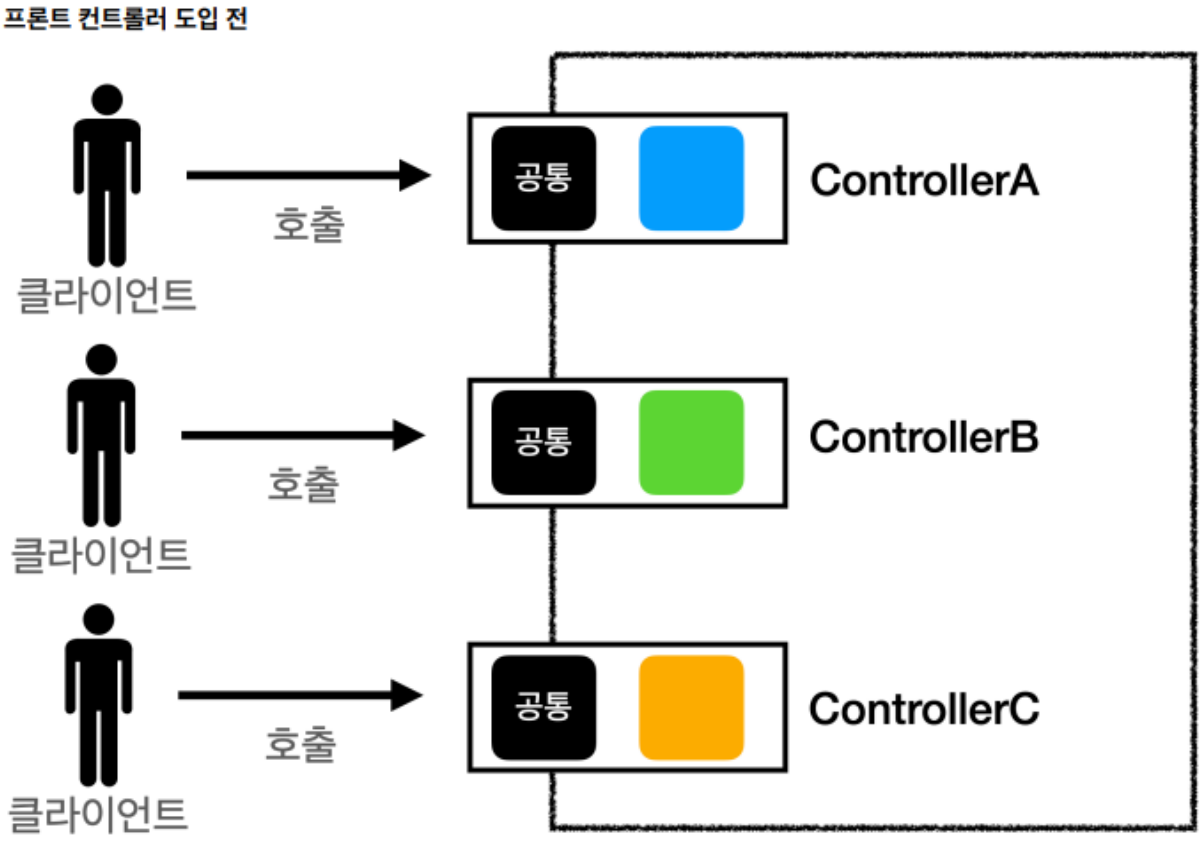
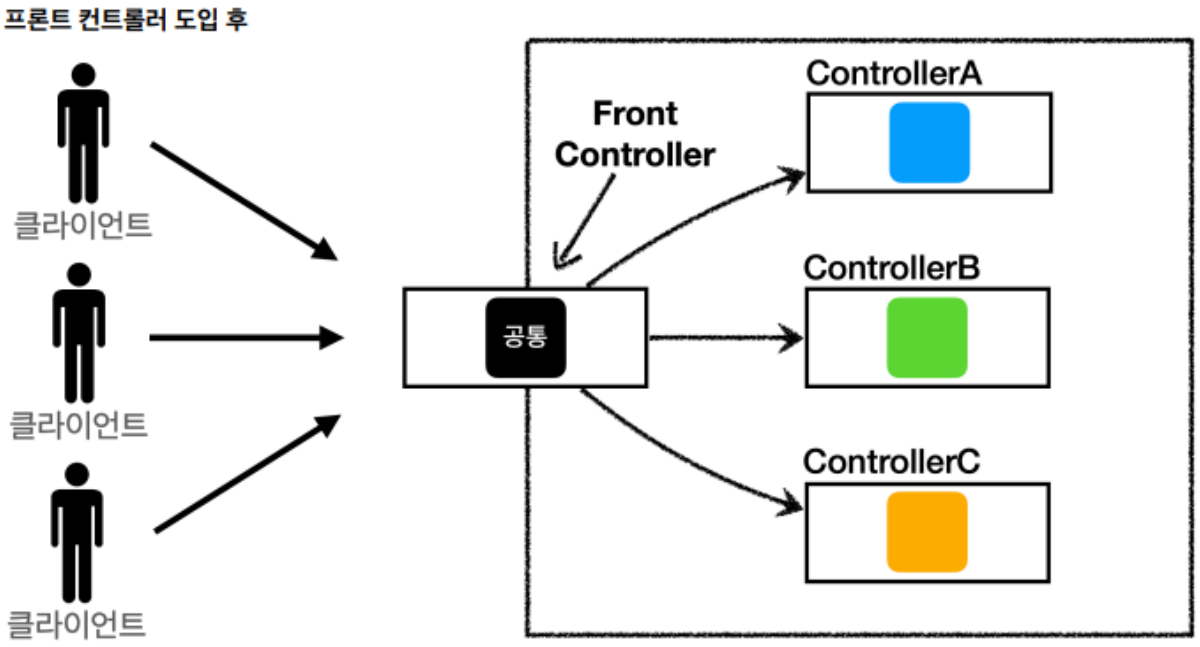
Section4. MVC 프레임워크 만들기

프론트 컨트롤러 패턴 소개





특징

- 프론트 컨트롤러 서블릿 하나로 클라이언트의 요청을 받는다.
- 요청에 맞는 컨트롤러를 찾아서 호출(입구를 하나로!)

  → **공통 처리 가능**
- 프론트 컨트롤러를 제외한 나머지 컨트롤러는 서블릿을 사용 X

스프링 웹 MVC와 프론트 컨트롤러

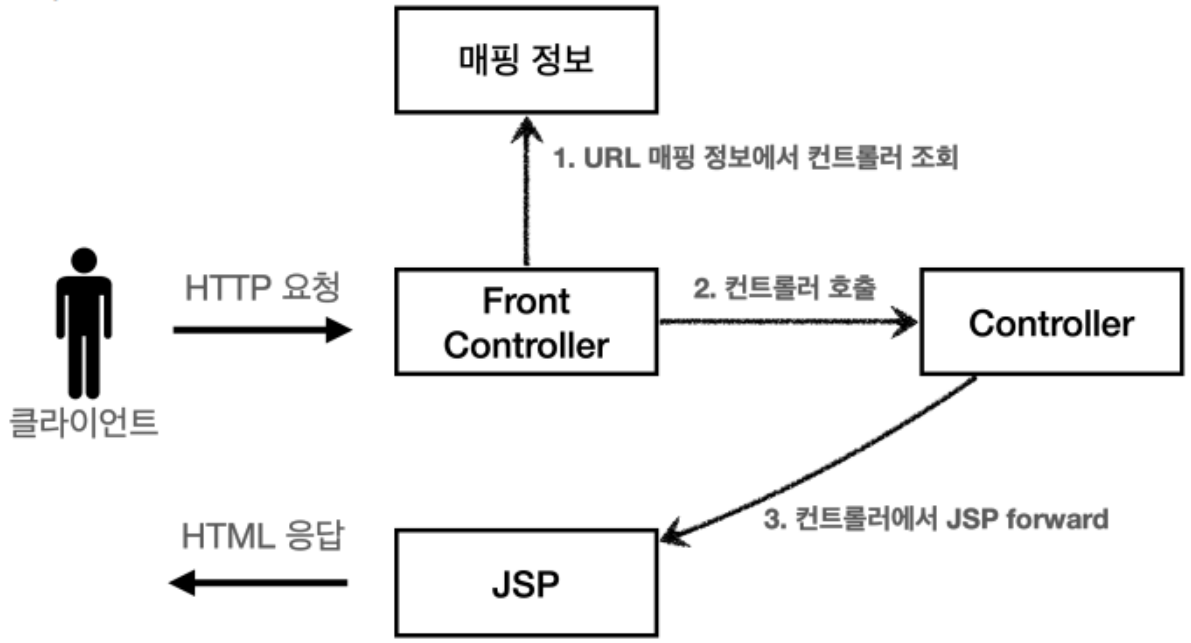
스프링 웹 MVC의 핵심도 바로 `FrontController`

스프링 웹 MVC의 `` `DispatcherServlet` `` 이 FrontController 패턴으로 구현되어 있다.

프론트 컨트롤러 도입 - v1

목표는 기존 코드를 최대한 유지하면서, 프론트 컨트롤러를 도입하는 것이다.

V1 구조




```
public interface ControllerV1 {
    void prcoess(HttpServletRequest request, HttpServletRespo
}
```

- 서블릿과 비슷한 모양의 컨트롤러 인터페이스를 도입한다.
  - 각 컨트롤러는 이 인터페이스를 구현하면 된다.
  - 프론트 컨트롤러는 이 인터페이스를 호출해서 구현과 관계없이 로직의 일관성을 가져갈 수 있다.

```
@WebServlet(name = "frontControllerServletV1", urlPatterns =
public class FrontControllerServletV1 extends HttpServlet {
     private Map<String, ControllerV1> controllerMap = new Ha
     public FrontControllerServletV1() {
     controllerMap.put("/front-controller/v1/members/new-form
     controllerMap.put("/front-controller/v1/members/save", n
     controllerMap.put("/front-controller/v1/members", new Me

    @Override
    protected void service(HttpServletRequest request, HttpSe
        System.out.println("FrontControllerServletV1.service"
```

```
        String requestURI = request.getRequestURI();
        ControllerV1 controller = controllerMap.get(requestUR
        if (controller == null) {
        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
        return;
        }
        controller.process(request, response);
        }
    }
}
```

urlPatterns

- urlPatterns = "/front-controller/v1/*" : /front-controller/v1 를 포함한 하위 모든 요청은 이 서블릿에서 받아들인다.
    - 예) /front-controller/v1 , /front-controller/v1/a , /front-controller/v1/a/b

controllerMap

- key: 매핑 URL
- value: 호출될 컨트롤러

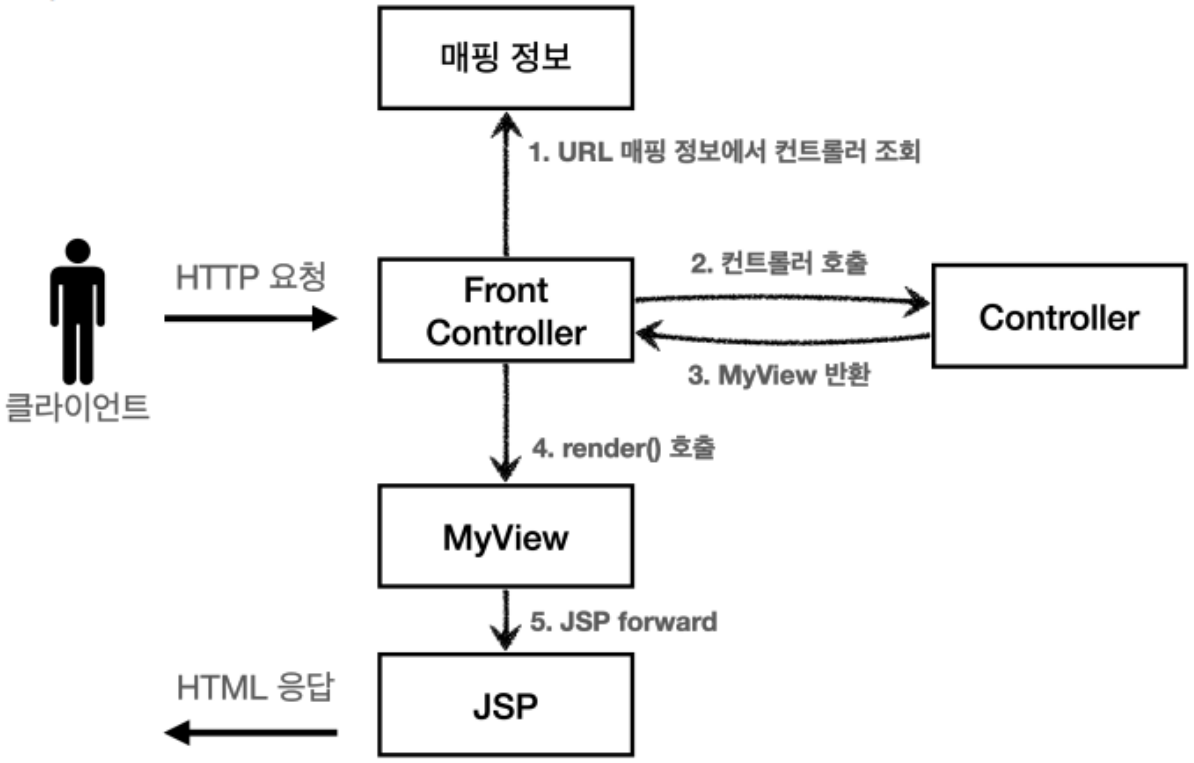
View 분리 - v2

모든 컨트롤러에서 뷰로 이동하는 부분에 중복이 있고, 깔끔하지 않다.

```
String viewPath = "/WEB-INF/views/new-form.jsp";
RequestDispatcher dispatcher = request.getRequestDispatcher(v
dispatcher.forward(request, response);
```

이 부분을 깔끔하게 분리하기 위해 별도로 뷰를 처리하는 객체를 만들자.

V2 구조



MyView 객체 생성

```
public class MyView {
         private String viewPath;
         public MyView(String viewPath) {
         this.viewPath = viewPath;
    }
     public void render(HttpServletRequest request, HttpServl
            RequestDispatcher dispatcher = request.getRequest
            dispatcher.forward(request, response);
    }
}
```

ControllerV2

```
public class MemberFormControllerV2 implements ControllerV2 {
     @Override
     public MyView process(HttpServletRequest request, HttpSe
     return new MyView("/WEB-INF/views/new-form.jsp");
```

```
    }
}
```

각 Member[Form | Save | List]ControllerV2

```
return new MyView("/WEB-INF/views/new-form.jsp");
return new MyView("/WEB-INF/views/save-result.jsp");
return new MyView("/WEB-INF/views/members.jsp");
```

프론트 컨트롤러

```
@WebServlet(name = "frontControllerServletV2", urlPatterns =
public class FrontControllerServletV2 extends HttpServlet {
    private Map<String, ControllerV2> controllerMap = new Has
    public FrontControllerServletV2() {
    controllerMap.put("/front-controller/v2/members/new-form"
    controllerMap.put("/front-controller/v2/members/save", ne
    controllerMap.put("/front-controller/v2/members", new Mem
    }
    @Override
    protected void service(HttpServletRequest request, HttpSe
    String requestURI = request.getRequestURI();

    ControllerV2 controller = controllerV1Map.get(requestURI)
    if(controller == null) {
        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
        return;
    }
  MyView view = controller.prcoess(request, response);
  view.render(request, response);
}
```

ControllerV2의 반환 타입이 `MyView` 이므로 프론트 컨트롤러는 컨트롤러의 호출 결과로 `MyView` 를 반환 받는다. 그리고 `view.render()` 를 호출하면 forward 로직을 수행해서 JSP가 실행된다.

```
// MyView.render()
public void render(HttpServletRequest request, HttpServletRes
```

```
        RequestDispatcher dispatcher = request.getRequestDispatch
        dispatcher.forward(request, response);
    }
```

프론트 컨트롤러의 도입으로 `MyView` 객체의 `render()` 를 호출하는 부분을 모두 일관되게 처리할 수 있다. 각각의 컨트롤러는 `MyView`객체를 생성만 해서 반환하면 된다.

Model 추가 - v3

서블릿 종속성 제거

컨트롤러 입장에서 HttpServletRequest, HttpServletResponse이 꼭 필요할까?

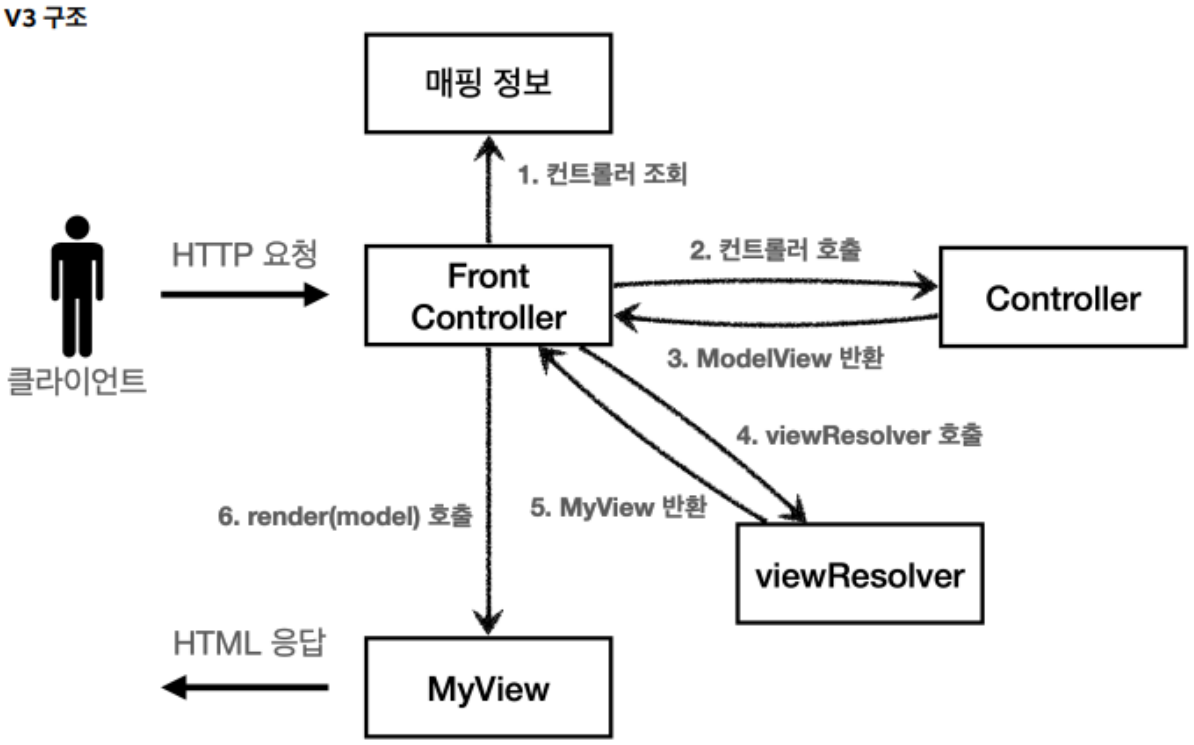
`request` 객체를 `Model`로 사용하는 대신에 **별도의 Model** 객체를 만들어서 반환하면 된다.

우리가 구현하는 컨트롤러가 **서블릿 기술을 전혀 사용하지 않도록 변경**해보자.

뷰 이름 중복 제거

컨트롤러는 **뷰의 논리 이름**을 반환하고, 실제 물리 위치의 이름은 프론트 컨트롤러에서 처리하도록 단순화 하자.

- `/WEB-INF/views/new-form.jsp` → **new-form**
- `/WEB-INF/views/save-result.jsp` → **save-result**
- `/WEB-INF/views/members.jsp` → **members**



ModelView

컨트롤러에서 서블릿에 종속적인 HttpServletRequest를 사용했다. 그리고 Model도 `request.setAttribute()` 를 통해 **데이터를 저장하고 뷰에 전달**했다.

서블릿의 종속성을 제거하기 위해 `Model`을 직접 만들고, 추가로 `View` 이름까지 전달하는 객체를 만들어보자

```
// 뷰의 이름과 뷰를 렌더링할 때 필요한 model 객체를 가지고 있다.
public class ModelView {
     private String viewName;
     private Map<String, Object> model = new HashMap<>();

     public ModelView(String viewName) {
         this.viewName = viewName;
     }

     public String getViewName() {
         return viewName;
     }

     public void setViewName(String viewName) {
         this.viewName = viewName;
```

```
    }

    public Map<String, Object> getModel() {
        return model;
    }

    public void setModel(Map<String, Object> model) {
        this.model = model;
    }
}
```

ControllerV3

```
// 이 컨트롤러는 서블릿 기술을 전혀 사용하지 않는다.
public interface ControllerV3 {
  ModelView process(Map<String, String> paramMap);
}
```

FrontControllerServletV3

```
@WebServlet(name = "frontControllerServletV3", urlPatterns =
public class FrontControllerServletV3 extends HttpServlet {
    private Map<String, ControllerV3> controllerV3Map = new H
    public FrontControllerServletV3(){
        controllerV3Map.put("/front-controller/v3/members/new
        controllerV3Map.put("/front-controller/v3/members/sav
        controllerV3Map.put("/front-controller/v3/members", n
    }
    @Override
    protected void service(HttpServletRequest request, HttpSe
        String requestURI = request.getRequestURI();

        ControllerV3 controller = controllerV3Map.get(request
        if(controller == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOU
            return;
        }
```

```
            HashMap<String, String> paramMap = createParamMap(req
            ModelView mv = controller.process(paramMap);

            String viewName = mv.getViewName();// 논리이름 new-form
            MyView view = viewResolver(viewName);

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

        private static MyView viewResolver(String viewName) {
            return new MyView("/WEB-INF/views/" + viewName + ".js
        }

        private static HashMap<String, String> createParamMap(Htt
            HashMap<String, String> paramMap = new HashMap<>();
            request.getParameterNames().asIterator()
                    .forEachRemaining(paramName -> paramMap.put(p
            return paramMap;
        }
    }
```

뷰 리졸버

컨트롤러가 반환한 논리 뷰 이름을 실제 물리 뷰 경로로 변경한다. 그리고 실제 물리 경로가 있는 `MyView` 객체를 반환한다.

- 논리 뷰 이름: `members`
- 물리 뷰 경로: `/WEB-INF/views/members.jsp`

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

- 뷰 객체를 통해서 HTML 화면을 렌더링 한다.
- 뷰 객체의 render() 는 모델 정보도 함께 받는다.
- JSP는 `request.getAttribute()` 로 데이터를 조회하기 때문에, 모델의 데이터를 꺼내서 `request.setAttribute()` 로 담아둔다.
- JSP로 포워드 해서 JSP를 렌더링 한다.

MyView

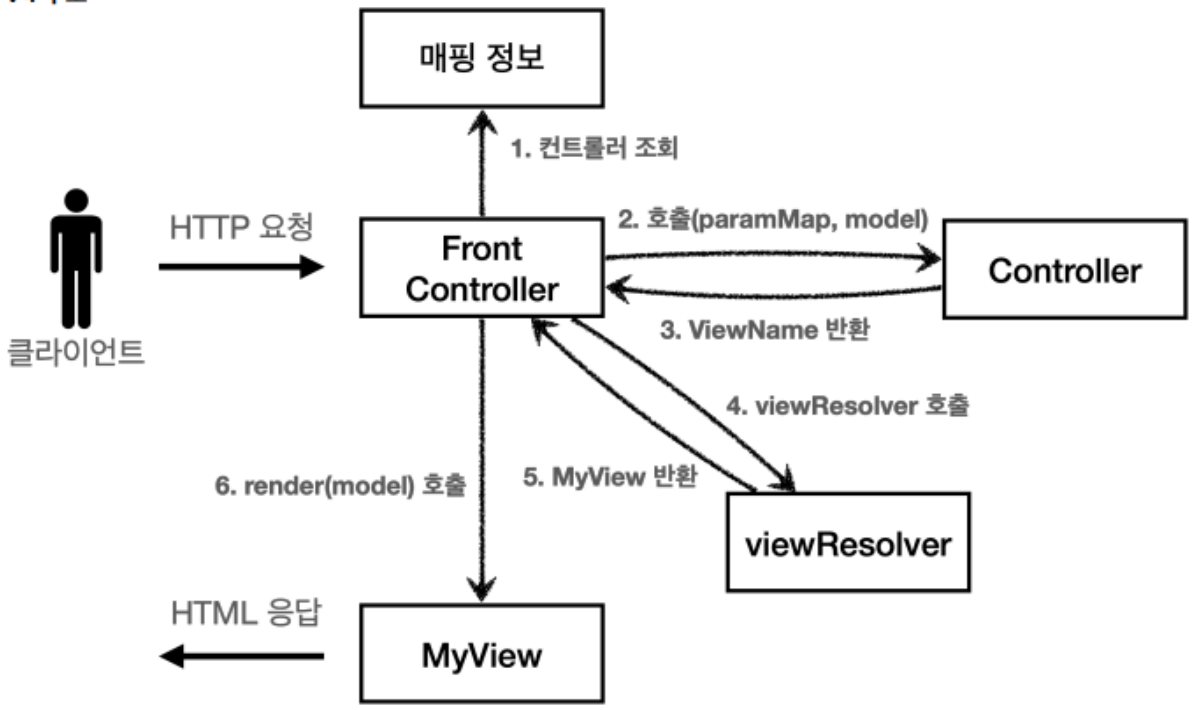
```
public class MyView {
    private String viewPath;

    public MyView(String viewPath) {
        this.viewPath = viewPath;
    }

    public void render(Map<String, Object> model, HttpServlet
        modelToRequestAttribute(model, request);
        RequestDispatcher dispatcher = request.getRequestDisp
        dispatcher.forward(request, response);
    }

    private static void modelToRequestAttribute(Map<String, O
        model.forEach((key, value) -> request.setAttribute(ke
    }
}
```

단순하고 실용적인 컨트롤러 - v4

실제 컨트롤러 인터페이스를 구현하는 개발자 입장에서 보면, 항상 `ModelView` 객체를 생성하고 반환해야 하는 부분이 조금은 번거롭다.

좋은 프레임워크는 아키텍처도 중요하지만, 그와 더불어 실제 개발하는 개발자가 단순하고 편리하게 사용할 수 있어야 한다.

V4 구조



- 기본적인 구조는 V3와 같다. 대신 컨트롤러가 `ModelView`를 반환하지 않고, `ViewName`만 반환한다.

```
public interface ControllerV4 {
 /**
 * @param paramMap
 * @param model
 * @return viewName
 */
 String process(Map<String, String> paramMap, Map<String, Obj
}
```

이번 버전은 인터페이스에 `ModelView`가 없다. `model` 객체는 파라미터로 전달되기 때문에 그냥 사용하면 되고, 결과로 뷰의 이름만 반환해주면 된다.

```
public class MemberFormControllerV4 implements ControllerV4 {
    @Override
    public String process(Map<String, String> paramMap, Map<S
        return "new-form";
    }
}
```

정말 단순하게 new-form 이라는 뷰의 논리 이름만 반환하면 된다.

```
import java.util.Map;

public class MemberSaveControllerV4 implements ControllerV4 {
    private MemberRepository memberRepository = MemberReposit
    @Override
    public String process(Map<String, String> paramMap, Map<S
        String username = paramMap.get("username");
        int age = Integer.parseInt(paramMap.get("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);
                // 모델이 파라미터로 전달되기 때문에, 모델을 직접 생성ㅎ
        model.put("member", member);
        return "save-result";
    }
}
```

모델 객체 전달

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

- 모델 객체를 프론트 컨트롤러에서 생성해서 넘겨준다.
- 컨트롤러에서 모델 객체에 값을 담으면 여기에 그대로 담겨있게 된다.

뷰의 논리 이름을 직접 반환

```
String viewName = controller.process(paramMap, model);
MyView view = viewResolver(viewName);
```

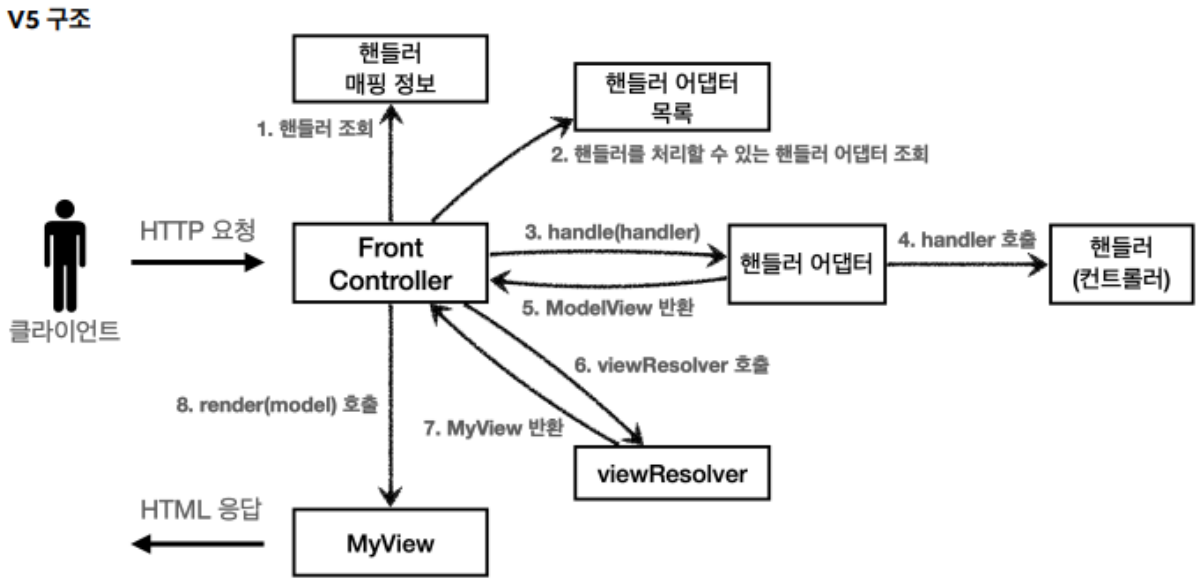
- 컨트롤러가 직접 뷰의 논리 이름을 반환하므로 이 값을 사용해서 실제 물리 뷰를 찾을 수 있다.

유연한 컨트롤러1 - v5

어떤 개발자는 v3 방식으로, 어떤 개발자는 v4 방식으로 개발하고 싶다면 `Map` 에 매칭 된 `interface` 를 매번 바꾸어야 한다.

어댑터 패턴

프론트 컨트롤러가 다양한 방식의 컨트롤러를 처리할 수 있도록 변경해보자.



핸들러 어댑터

중간에 어댑터 역할을 하는 어댑터가 추가되었는데 이름이 핸들러 어댑터이다.
여기서 어댑터 역할을 해주는 덕분에 다양한 종류의 컨트롤러를 호출할 수 있다.

핸들러

컨트롤러의 이름을 더 넓은 범위인 핸들러로 변경했다.
그 이유는 이제 어댑터가 있기 때문에 꼭 컨트롤러의 개념 뿐만 아니라 어떠한 것이든 해당하는 종류의 어댑터만 있으면 다 처리할 수 있기 때문이다.

MyHandlerAdapter

```
public interface MyHandlerAdapter {
     boolean supports(Object handler);
     ModelView handle(HttpServletRequest request, HttpServlet
                                                    Object h
}
```

boolean supports(Object handler)

- handler는 컨트롤러를 말한다.
- 어댑터가 해당 컨트롤러를 처리할 수 있는지 판단하는 메서드다.

ModelView handle(HttpServletRequest request, HttpServletResponse response, Object handler)

- 어댑터는 실제 컨트롤러를 호출하고, 그 결과로 `ModelView`를 반환해야 한다.
- 실제 컨트롤러가 `ModelView`를 반환하지 못하면, 어댑터가 `ModelView`를 직접 생성해서라도 반환해야 한다.
- 이전에는 프론트 컨트롤러가 실제 컨트롤러를 호출했지만 이제는 이 어댑터를 통해서 실제 컨트롤러가 호출된다.

구현

ControllerV3HandlerAdapter

```
public class ControllerV3HandlerAdapter implements MyHandlerA
    @Override
    public boolean supports(Object handler) {
            // ControllerV3 을 처리할 수 있는 어댑터를 뜻한다.
        return (handler instanceof ControllerV3);
    }

    @Override
    public ModelView handle(HttpServletRequest request, HttpS
                // handler를 컨트롤러 V3로 변환한 다음에 V3 형식에 맞
                // supports() 를 통해 ControllerV3 만 지원하기 때
                // ControllerV3는 ModelView를 반환하므로 그대로 Mo
        ControllerV3 controller = (ControllerV3) handler;
        HashMap<String, String> paramMap = createParamMap(req
        ModelView mv = controller.process(paramMap);

        return mv;
    }
    private static HashMap<String, String> createParamMap(Htt
        HashMap<String, String> paramMap = new HashMap<>();
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(pa
        return paramMap;
    }
}
```

컨트롤러(Controller) —> 핸들러(Handler)

이전에는 컨트롤러를 직접 매핑해서 사용했다. 이제는 어댑터를 사용하기 때문에, 컨트롤러 뿐만 아니라 어댑터가 지원하기만 하면, **어떤 것이라도 URL에 매핑해서 사용**할 수 있다.

```
@WebServlet(name = "frontControllerServletV5", urlPatterns =
public class FrontControllerServletV5 extends HttpServlet {
    private final Map<String, Object> handlerMappingMap = new
    private final List<MyHandlerAdapter> handlerAdapters = ne

    public FrontControllerServletV5() {
        initHandlerMappingMap();
        initHandlerAdapters();
    }

    private void initHandlerMappingMap() {
        handlerMappingMap.put("/front-controller/v5/v3/member
        handlerMappingMap.put("/front-controller/v5/v3/member
        handlerMappingMap.put("/front-controller/v5/v3/member
    }
    private void initHandlerAdapters() {
        handlerAdapters.add(new ControllerV3HandlerAdapter())
    }

    @Override
    protected void service(HttpServletRequest request, HttpSe
        Object handler = getHandler(request);

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

        MyHandlerAdapter adapter = getHandlerAdapter(handler)
        ModelView mv = adapter.handle(request, response, hand

        String viewName = mv.getViewName();// 논리이름 new-form
        MyView view = viewResolver(viewName);
```

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

        private static MyView viewResolver(String viewName) {
            return new MyView("/WEB-INF/views/" + viewName + ".js
        }
        private MyHandlerAdapter getHandlerAdapter(Object handler
            for (MyHandlerAdapter adapter : handlerAdapters) {
                if(adapter.supports(handler)){
                    return adapter;
                }
            }
            throw new IllegalArgumentException("handler adapter를
        }

        private Object getHandler(HttpServletRequest request) {
            String requestURI = request.getRequestURI();
            return handlerMappingMap.get(requestURI);
        }
    }
```

유연한 컨트롤러2 - v5

기존 `FrontControllerServletV5` 를 다음과 같이 변경(v4 추가)해주자.

```
private void initHandlerMappingMap() {
        handlerMappingMap.put("/front-controller/v5/v3/member
        handlerMappingMap.put("/front-controller/v5/v3/member
        handlerMappingMap.put("/front-controller/v5/v3/member

        handlerMappingMap.put("/front-controller/v5/v4/member
        handlerMappingMap.put("/front-controller/v5/v4/member
        handlerMappingMap.put("/front-controller/v5/v4/member
}
private void initHandlerAdapters() {
    handlerAdapters.add(new ControllerV3HandlerAdapter());
```

```
        handlerAdapters.add(new ControllerV4HandlerAdapter());
    }
```

- 단 4줄의 코드만 추가했을 뿐인데 이제 우리가 만든 controller는 v3, v4를 사용할 수 있게 됐다.

물론 아래와 같은 Adapter를 추가해주어야 한다.(주입)

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

    @Override
    public ModelView handle(HttpServletRequest request, HttpS
        ControllerV4 controller = (ControllerV4) handler;

        Map<String, String> paramMap = createParamMap(request
        HashMap<String, Object> model = new HashMap<>();

        String viewName = controller.process(paramMap, model)

        ModelView mv = new ModelView(viewName);
        mv.setModel(model);

        return mv;
    }

    private static HashMap<String, String> createParamMap(Htt
        HashMap<String, String> paramMap = new HashMap<>();
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(pa
        return paramMap;
    }
}
```

위 코드에서 중요하게 봐야 할 부분은 아래와 같다.

```
ModelView mv = new ModelView(viewName);
mv.setModel(model);
return mv;
```

어댑터가 호출하는 ControllerV4 는 뷰의 이름을 반환한다. 그런데 어댑터는 뷰의 이름이 아니라 `ModelView`를 만들어서 반환해야 한다. 여기서 어댑터가 꼭 필요한 이유가 나온다.

ControllerV4 는 뷰의 이름을 반환했지만, 어댑터는 이것을 `ModelView`로 만들어서 형식을 맞추어 반환한다

```
public interface ControllerV4 {
    String process(Map<String, String> paramMap, Map<String, 
}
public interface MyHandlerAdapter {
    ModelView handle(HttpServletRequest request, HttpServletR
}
```

정리

지금까지 v1 ~ v5로 점진적으로 프레임워크를 발전시켜 왔다.

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

여기에 애노테이션을 사용해서 컨트롤러를 더 편리하게 발전시킬 수도 있다. 만약 애노테이션을 사용해서 컨트롤러를 편리하게 사용할 수 있게 하려면 어떻게 해야할까? 바로 애노테이션을 지원하는 어댑터를 추가하면 된다!

> → 다형성과 어댑터 덕분에 기존 구조를 유지하면서, 프레임워크의 기능을 확장할 수 있다.

스프링 MVC
스프링 MVC의 핵심 구조를 파악하는데 필요한 부분은 모두 만들어보았다.
지금까지 작성한 코드는 스프링 MVC 프레임워크의 핵심 코드의 축약 버전이고, 구조도 거의 같다.