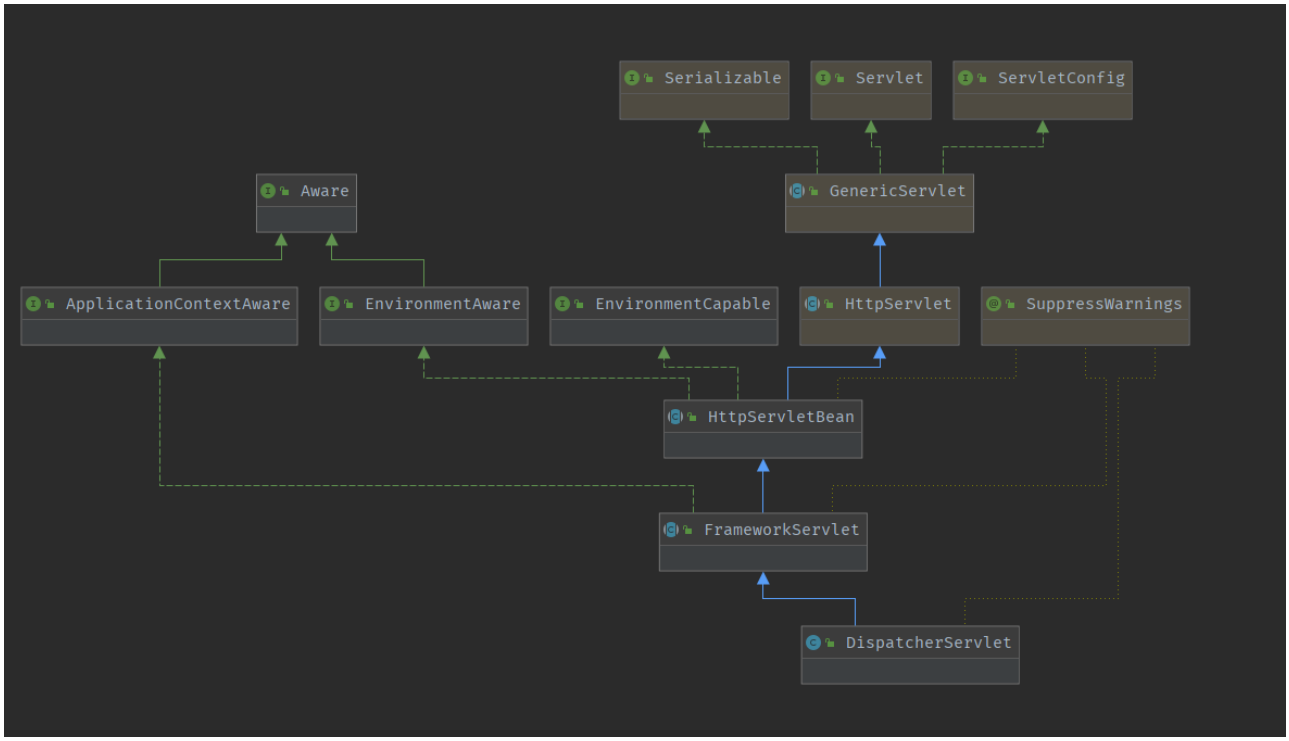
소스코드

DispatcherServlet의 동작과정

- 계층 구조



- HttpServlet
    - HttpServlet을 구현하기위한 J2EE 스펙의 추상 클래스
    - 특정 HTTP 메소드를 지원하기 위해선 doX메소드를 오버라이딩 해야함(템플릿 메소드패턴)
    - doPatch는 지원하지 않음
- HttpServletBean
    - HttpServlet을 스프링이 구현한 추상 클래스
    - 스프링이 모든 유형의 서블릿 구현을 위해 정의한 공통 클래스
- FrameWorkServlet
    - 스프링 웹 프레임워크의 기반이 되는 서블릿
    - doX메소드를 오버라이딩, doX요청들을 공통된 요청 처리메소드인 processRequest로 전달
    - process Request에서 실제 요청 핸들링은 추상 메소드 doService로 위임(템플릿 메소드 패턴)
- DispatcherServlet
    - 컨트롤러로 요청을 전달하는 중앙 집중형 컨트롤러(서블릿 구현체)
    - 실제 요청을 처리하는 doService를 구현

디스패처 서블릿이 요청을 받아 컨트롤러로 위임하는 과정

1. 서블릿 요청/응답을 HTTP 서블릿 요청/응답으로 변환

2. Http Method에 따른 처리 작업 진행
3. 요청에 대한 공통 처리 작업 진행
4. 컨트롤러로 요청을 위임
    1. 요청에 매핑되는 HandelrExecutionChain 조회
    2. 요청을 처리할 HandlerAdapter 조회
    3. Handler Adapter를 통해 컨트롤러 메소드 호출(HandlerExecutionChain 처리)

1. 서블릿 요청/응답을 HTTP 서블릿 요청/응답으로 변환

- Http 요청은 등록된 필터들을 거쳐 디스패처 서블릿이 처리하게 함
- 가장 먼저 요청을 받는 부분은 부모클래스인 HttpServlet에 구현된 service 메소드

```
public abstract class HttpServlet extends GenericServlet{
...
@Override
public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException{
    HttpServletRequest request;
    HttpServletResponse response;
    try{
        request=(HttpServletRequest)req;
        response=(HttpServletResponse)res;

    }catch(ClassCastException e){
        throw new ServletException(Strings.getString("http.non_http"));
    }
    service(request,response)
}
protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        ...
    }

}
```

- service에서 먼저 Servlet 관련 Request, Response 객체를 Http관련 요청/응답으로 캐스팅, 캐스팅시 에러가 발생하면 Http요청이 아니므로 에러를 던짐

2. Http Method에 따른 처리 작업 진행

- HttpServletRequest 객체를 파라미터로 갖는 service 메소드를 호출하는데, HttpServlet에도 service가 있지만 자식 클래스인 FrameworkServlet에 service가 오버라이딩되어 있어 자식의 메소드가 호출됨.

```
public abstract class FrameworkServlet extends HttpServletBean implements
ApplicationContextAware{
    ...
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException{
```

```
        HttpMethod httpMethod = HttpMethod.resolve(request.getMethod());
        if(httpMethod==HttpMethod.PATCH || httpMethod==null){
            processRequest(request,response);
        }
        else super.service(request,response);
    }
    ...
}
```

- FrameworkServlet의 service
  - method = PATCH인 경우 processRequest를 호출
  - method = PATCH가 아닌 경우 : 부모 클래스의 service 호출
- Patch 메소드만 이런 흐름을 거치는 이유
  - 과거 HTTP 표준엔 Patch 메소드가 없었다.
  - 탄생 : 2010년도에 Ruby on Rails가 부분 수정의 필요를 주장
    - Spring 개발팀은 Patch요처을 처리하고자 자체 개발
- PatchMethod가 아닌 경우 처리되는 로직

```
public abstract class HttpServlet extends GenericServlet {

    ...

    protected void service(HttpServletRequest req, HttpServletResponse
resp)
        throws ServletException, IOException {

        String method = req.getMethod();

        if (method.equals(METHOD_GET)) {
            ... // lastModifed에 따른 doGet 처리(요약함)
            doGet(req, resp);

        } else if (method.equals(METHOD_HEAD)) {
            long lastModified = getLastModified(req);
            maybeSetLastModified(resp, lastModified);
            doHead(req, resp);

        } else if (method.equals(METHOD_POST)) {
            doPost(req, resp);

        } else if (method.equals(METHOD_PUT)) {
            doPut(req, resp);

        } else if (method.equals(METHOD_DELETE)) {
            doDelete(req, resp);

        } else if (method.equals(METHOD_OPTIONS)) {
            doOptions(req,resp);

        } else if (method.equals(METHOD_TRACE)) {
```

```
            doTrace(req,resp);

        } else {
            ... // 에러 처리
        }
    }

    ...
}
```

- HttpServlet에서는 요청 메소드에 따라 필요한 doX메소드를 호출해줌.
- doX메소드를 오버라이딩한 자식클래스인 FrameworkServlet으로 다시 요청이 이어짐, Patch메소드를 제외한 메소드들에 대해 템플릿 메소드 패턴이 적용 된 것
- doX메소드의 구현

```
public abstract class Framework extends HttpServletBean implements
ApplicationCotnextAware{
    ...
    @Override
    protected final void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }
    @Override
    protected final void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }
    ...
}
```

- 각각의 doX메소드는 HttpMetho에 맞는 작업을 하는데, doGet에서 lastModifed 관련 처리해주는 것 말고는 거의 차이 가 없다.
- doGet메소드들은 processRequest를 거치게 됨

3. 요청에 대한 공통 처리 작업 진행

- 오버 라이딩 된 doX메소드는 모두 request를 처리하는 processRequest메소드를 호출

```
protected final void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

    ... // LocaleContextHolder 처리 등 생략

    try {
        doService(request, response);
    } catch (ServletException | IOException ex) {
```

```
            failureCause = ex;
            throw ex;
        } catch (Throwable ex) {
            failureCause = ex;
            throw new NestedServletException("Request processing failed", ex);
        } finally {
            ... // 후처리 진행
        }
    }

    protected abstract void doService(HttpServletRequest request,
    HttpServletResponse response)
            throws Exception;
```

- processRequest는 request에 대한 공통 작업을 한 후, doService를 호출
- doService는 템플릿 메소드 패턴이 적용된 추상 메소드이므로 자식 클래스인 DispatcherServlet을 가야 볼 수 있음

```
public class DispatcherServlet extends FrameworkServlet {
    ...
    @Override
    protected void doService(HttpServlet request, HttpServletResponse
response) throws ServletException, IOException {
        logRequest(request);
        ... //flash map 등의 처리 진행
        try{
            doDispatch(request, response);
        } finally{
            ... //후처리 진행
        }

    }
    ...
}
```

- processRequest, doService에서는 LocaleContextHolder와 flashMap등에 대한 공통 처리 작업이 진행
- doDispatch는 Http요청을 컨트롤러로 위임

4. 컨트롤러로 요청을 위임

- doDispatch는 크게 3단계로 나뉘어 진행
    1. 요청에 패핑되는 HandlerMapping (HandlerExecutionChain) 조회
    2. 요청을 처리할 HandlerAdapter 조회
    3. HandlerAdapter를 통해 컨트롤러 메소드 호출(HandlerExecutionChain 처리)

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    HttpServletRequest processedRequest= request;
    HandlerExecutionChain mappedHandler=null;
```

```
    boolean multipartRequestParsed=false;

    WebAsyncManager asyncManager =
WebAsyncManager.getAsyncManager(request);

    try{
        ModelAndView mv=null;
        Exception dispatchException=null;
        try{
            processedRequest=checkMultipart(request);
            multipartRequestParsed=(processRequest!=request);
            //1.요청에 패핑되는 HandlerMapping (HandlerExecutionChain) 조회
            mappedHandler=getHandler(processedRequest);
            if(mappedHandler==null){
                noHandlerFond(processRequest,response);
                return;
            }
            //2.요청을 처리할 HandlerAdapter 조회
            HandlerAdapter
adapter=getHandlerAdapter(mappedHandler.getHandler());

            ...

            //3.HandlerAdapter를 통해 컨트롤러 메소드 호출(HandlerExecutionChain
처리)

mv=adapter.handle(processedRequest,response,mappedHandler.getHandler());
            ...// 후처리 진행(인터셉터 등)
        }catch (Exception ex) {
            dispatchException = ex;
        } catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler
methods as well,
            // making them available for @ExceptionHandler methods and
other scenarios.
            dispatchException = new NestedServletException("Handler
dispatch failed", err);
        }
        processDispatchResult(processedRequest, response,
mappedHandler,mv, dispatchException);
    }catch(Exception ex){
        triggerAfterCompletion(processRequest,response,mappedHandler,ex);
    }catch(Throwable err){
        triggerAfterCompletion(processRequest,response,mappedHandler,new
NestedServletException("Handler dispatch failed", err));
    }finally{
        ...//후처리 진행
    }
}
```

1. 요청에 패핑되는 HandlerMapping(HandlerExecutionChain) 조회

- HandlerExecutionChain은 HandlerMethod와 Interceptor들로 구성
- HandlerMethod에는 매핑되는 컨트롤러의 매소드와 컨트롤러 빈 이름(또는 컨트롤러 빈) 및 빈 팩토리 등이 저장되어 있음
    - 매핑되는 컨트롤러의 메소드
        - Reflection 패키지의 Method객체
        - Method를 호출(invoke)하기 위해서는 메소드의 주인인 빈 객체를 필요로 함
    - 빈 정보(컨트롤러 빈 이름 또는 컨트롤러 빈)
        - Object 타입으로써 컨트롤러 빈 이름(기본) 또는 컨트롤러 빈이 될 수 있음
        - 기본적으로 컨트롤러 빈 이름을 갖고 있으며, 빈 객체 조회를 위해 사용 됨
        - HandlerMethod의 createWithResolveBean을 호출하면 빈 이름이 아닌 실제 빈을 갖는 HandlerMethod 객체를 생성함
    - 빈 팩토리
        - 스프링 부트가 관리하는 빈들을 가지고 있음
        - 컨트로럴의 메소드를 호출하기 위한 빈 객체 조회를 위해 사용됨
- DispatcherServlet이 HandlerMethod가 아닌 HandlerExecutionChain를 얻는 이유는 공통 처리를 위한 인터셉터가 존재하기 때문

```
@Nullable
protected HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception {
    if (this.handlerMappings != null) {
        for (HandlerMapping mapping : this.handlerMappings) {
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```

- getHandler는 HandlerMapping목록을 순회하여 HandlerExecutionChain을 찾는다.
- Controller, @ReuqestMapping => RequestMappingHandlerMapping가 HandlerExecutionChain를 생성해 반환

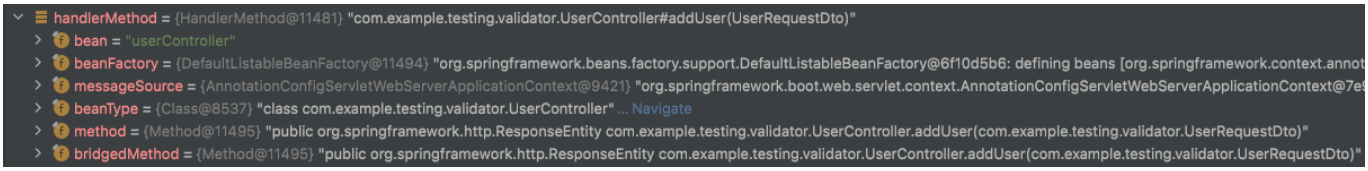
```
@Override
@Nullable
public final HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception {
    Object handler = getHandlerInternal(request);
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    ...
```

```
    HandlerExecutionChain executionChain =
getHandlerExecutionChain(handler, request);

    ... // CORS 및 기타 처리

    }

    return executionChain;
}
```

- 핸들러 메소드를 찾는 RequestMappingHandlerMapping의 getHandlerInternal는 내부적으로 다시 추상 부모 클래스인 AbstractHandlerMethodMapping의 getHandlerInternal로 위임

```
@Override
@Nullable
protected HandlerMethod getHandlerInternal(HttpServletRequest request)
throws Exception {
    String lookupPath = initLookupPath(request);
    this.mappingRegistry.acquireReadLock();
    try {
        HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath,
request);
        return (handlerMethod != null ?
handlerMethod.createWithResolvedBean() : null);
    } finally {
        this.mappingRegistry.releaseReadLock();
    }
}
```

- 먼저 API URI인 lookupPath를 찾고
- mappingRegistry에 대해 readLock을 얻고
- HandlerMethod를 찾고 있다.
- HandlerMethod
    - 빈 이름, 빈 팩토리, 메소드 객체



- lookupHandlerMethod는 핸들러메소드를 찾는다.
    - 스프링은 애플리케이션 초기화 시 모든 컨트롤러를 파싱해서 (요청 정보, 요청 정보를 처리할 대상)을 관리해 둔다.
    - 요청이 들어오면 URI를 기준으로 매핑되는 후보군을 찾는다.
    - 만약 동일한 URI에 대해 POST,PUT 메소드가 있으면 2개의 후보군이 찾아진다.
    - HTTP METHOD는 다른 조건들을 통해 완전 매핑되는지 검사한다.

- 매핑 정보는 RequestMappingInfo 클래스인데. 관련 자세한 내용은 다른 포스팅


```
@Override
@Nullable
public final HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception {
    Object handler = getHandlerInternal(request);
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    ...

    HandlerExecutionChain executionChain =
getHandlerExecutionChain(handler, request);

    ... // CORS 및 기타 처리

    }

    return executionChain;
}
```

- HandlerMethod는 최종적으로 HandlerExecutionChain으로 반환
    - 컨트롤러에 위임하기전 인터셉터를 처리하기 위함임. (HandlerMethod와 인터셉터를 가지고 있음)

2. 요청을 처리할 HandlerAdapter 조회

- 디스패처 서블릿은 HandlerExecutionChain을 HandlerAdapter를 통해 실행
    - 컨트롤러 구현 방식에 상관없이 요청을 위임할수 있음.
        - 컨트롤러 인터페이스
        - 어노테이션 방식
- doDispatch는 getHandlerAdepter를 통해 HandlerAdapter를 조회한다.

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // 1. 요청에 패핑되는 HandlerExecutionChain 조회
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // 2. 요청을 처리할 HandlerAdapter 조회
            HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());

            ...

            // 3. HandlerAdapter를 통해 컨트롤러 메소드 호출
(HandlerExecutionChain 처리)
            mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());

            ... // 후처리 진행(인터셉터 등)
        } catch (Exception ex) {
            dispatchException = ex;
        } catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler
methods as well,
            // making them available for @ExceptionHandler methods and
other scenarios.
            dispatchException = new NestedServletException("Handler
dispatch failed", err);
        }
        processDispatchResult(processedRequest, response, mappedHandler,
mv, dispatchException);
    } catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
ex);
    } catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
            new NestedServletException("Handler processing failed", err));
```

```
    } finally {
        ... // 후처리 진행
    }
}
```

- 어노테이션으로 구현된 컨트롤러에 대한 API 요청인 경우 RequestMappingHandlerAdapter가 찾아진다.
- 다양한 방식으로 구현되므로 HandlerAdapter를 생성

3. HandlerAdapter를 통해 컨트롤러 메소드 호출(HandlerExecutionChain 처리)

- HandlerAdapter를 통해 HandelrExecutionChain을 처리하는데, 내부에 인터셉터를 가져, 공통적인 전/후 처리 과정이 처리됨.
- 대표적으로 컨트롤러 메소드 호출 전에는 적합한 파라미터를 만들어 넣어주어야 하며(ArgumentResolver), 호출 후에는 메세지 컨버터를 통해 ResponseEntity의 Body를 찾아 Json직렬화 하는 등(ReturnValueHandler)이 필요하다.
- 적합한 HandlerAdpater가 HandlerExecutionChain를 모두 찾았으면 이제 핸들러 어댑터가 요청을 처리할 차례이다.
- AbstractHandlerMethodAdapter

```
@Override
@Nullable
public final ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler)
      throws Exception {

   return handleInternal(request, response, (HandlerMethod) handler);
}

@Nullable
   protected abstract ModelAndView handleInternal(HttpServletRequest
request,
      HttpServletResponse response, HandlerMethod handlerMethod) throws
Exception;
```

- 요청의 종류에 따라 HandlerAdapter 구현체가 달라지며, 그에 따라 전/후처리 달라지므로 세부 구현을 구체 클래스로 위임하는 탬프릿 메소드 패턴이 또 사용 된 것
- @Controller로 작성된 컨트롤러를 처리하는 RequestMappingHandlerAdapter를 보면 RequestMappingHandlerAdapter의 handlerInternal은 실제로 요청을 위임하는 InvokeHandlerMethod를 호출함

```
@Override
protected ModelAndView handleInternal(HttpServletRequest request,
      HttpServletResponse response, HandlerMethod handlerMethod) throws
Exception {

   ModelAndView mav;
   checkRequest(request);
```

```
    // Execute invokeHandlerMethod in synchronized block if required.
    if (this.synchronizeOnSession) {
        ... // 생략
    } else {
        // No synchronization on session demanded at all...
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }

    ...

    return mav;
}
```

- InvokeHandlerMethod가 컨트롤러로 요청을 위임하는 곳

```
@Nullable
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
HttpServletResponse response, HandlerMethod handlerMethod) throws
Exception {
    ServletWebRequest webRequest = new
ServletWebRequest(request,response);
    try{
        WebDataBinderFactory
binderFactory=getDataBinderFactory(handlerMethod);
        ModelFactory modelFacotry =
getModelFactory(handlerMethod,binderFactory);

        ServletInvocableHandlerMethod
invocableHandlerMethod=createInvocableHandlerMethod(handlerMethod);
        if(this.argumentResolvers!=null){

invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }
        if(this.returnValueHandlers!=null){

invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandle
rs);
        }
        ...
        invocableMethod.invokeAndHandle(webRequest,mavContainer);
        ...
        return getModelAndView(mavContainer,modelFacotry,webRequest);
    }
    finally{
        webRequest.requestCompleted();
    }
}
```

여기서 먼저 주목할 부분은 HandlerMethod가 ServletInvocableHandlerMethod로 재탄생한다는 것이다. HandlerExecutionChain에는 공통적인 전/후처리가 진행된다고 하였는데, 이러한 작업에는 대표적으로 컨트롤러의 파라미터를 처리하는 ArgumentResolver와 반환값을 처리하는 ReturnValueHandler가 있다.

즉, ServletInvocableHandlerMethod로 다시 만드는 이유는 HandlerMethod와 함께 argumentResolver나 returnValueHandlers 등을 추가해 공통 전/후 처리를 하기 위함이다. 세팅이 끝나면 ServletInvocableHandlerMethod의 invokeAndHandle로 이어진다.

public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception {

Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs); setResponseStatus(webRequest);

...

try { this.returnValueHandlers.handleReturnValue(returnValue, getReturnValueType(returnValue), mavContainer, webRequest); } catch (Exception ex) { ... throw ex; } }

그리고는 바로 부모 클래스인 InvocableHandlerMethod의 invokeForRequest로 이어진다.

@Nullable public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception {

Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs); if (logger.isTraceEnabled()) { logger.trace("Arguments: " + Arrays.toString(args)); } return doInvoke(args); }

invokeForRequest에서는 먼저 메소드 호출을 위해 필요한 인자값을 처리한다. @RequestHeader, @CookieValue 및 @PathVariable 등도 모두 스프링이 만들어둔 ArgumentResolver에 의해 처리가 되는데, 이러한 인자값을 만드는 작업이 getMethodArgumentValues 내에서 처리가 된다. 그리고 doInvoke에서 만들어진 인자값을 통해 컨트롤러의 메소드를 호출한다.

(getMethodArgumentValues에서 ArgumentResolver를 이용해 인자값을 처리하는 과정에는 컴포지트 패턴이 적용되어 있는데, 후처리하는 과정에서도 동일하게 사용되므로 이따가 살펴보도록 하자.)

doInvoke는 부모 클래스인 InvocableHandlerMethod에 다음과 같이 구현되어 있다.

@Nullable protected Object doInvoke(Object... args) throws Exception { Method method = getBridgedMethod(); try { if (KotlinDetector.isSuspendingFunction(method)) { return CoroutinesUtils.invokeSuspendingFunction(method, getBean(), args); } return method.invoke(getBean(), args); } catch (IllegalArgumentException ex) { ... // IllegalArgumentException 처리 } catch (InvocationTargetException ex) { ... // Unwrap for HandlerExceptionResolvers ... } }

가장 먼저 요청을 처리할 컨트롤러의 메소드 객체(Java의 리플렉션 Method)를 꺼내온다. 그리고 Method 객체의 invoke를 통해서(Reflection을 사용해서) 실제 컨트롤러로 위임을 해준다.

컨트롤러에서 성공적으로 작업을 처리한 후에 ResponseEntity를 반환했다면 invokeAndHandle의 returnValue로 해당 객체가 온다.

private HandlerMethodReturnValueHandlerComposite returnValueHandlers;

public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception {

```
Object returnValue = invokeForRequest(webRequest, mavContainer,
providedArgs);

...

try {
    this.returnValueHandlers.handleReturnValue(
        returnValue, getReturnValueType(returnValue), mavContainer,
webRequest);
    } catch (Exception ex) {
        ...
    }
}
```

}

그 다음에는 응답에 대한 후처리를 할 차례인데, 후처리는 returnValueHandlers를 통해 처리된다.

ArgumentResolver로 요청을 전처리하는 과정과 ReturnValueHandler로 후처리하는 과정 모두에는 컴포지트 패턴이 적용되어 있다. 요청을 처리하기 위한 인터페이스로 HandlerMethodArgumentResolver가 있다면, 응답을 처리하기 위한 인터페이스로는 HandlerMethodReturnValueHandler가 있다.

public interface HandlerMethodReturnValueHandler {

boolean supportsReturnType(MethodParameter returnType);

void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception;

}

응답에 따라 다양한 형태로 처리하기 위해서 이를 리스트로 갖고 있으며 supportsReturnType으로 처리 가능한 구현체인지를 판별해야 한다. 스프링은 HandlerMethodReturnValueHandler 인터페이스 목록을 갖고 있는 컴포지트 객체인 HandlerMethodReturnValueHandlerComposite를 만들어두고 HandlerMethodReturnValueHandler를 구현받도록 하여 컴포지트 패턴을 적용하고 있다.

public class HandlerMethodReturnValueHandlerComposite implements HandlerMethodReturnValueHandler {

private final List returnValueHandlers = new ArrayList<>();

@Override public boolean supportsReturnType(MethodParameter returnType) { return getReturnValueHandler(returnType) != null; }

@Nullable private HandlerMethodReturnValueHandler getReturnValueHandler(MethodParameter returnType) { for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) { if (handler.supportsReturnType(returnType)) { return handler; } } return null; }

@Override public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {

```
  HandlerMethodReturnValueHandler handler = selectHandler(returnValue,
returnType);
  if (handler == null) {
     throw new IllegalArgumentException("Unknown return value type: " +
returnType.getParameterType().getName());
  }
  handler.handleReturnValue(returnValue, returnType, mavContainer,
webRequest);
```

}

...

}

오버라이딩된 supportsReturnType 메소드의 경우에는 리스트를 순회하여 처리 가능한 핸들러가 있을 경우에 true를 반환하게 하였으며, handleReturnValue의 경우에는 처리 가능한 핸들러를 찾아서 해당 핸들러의 handleReturnValue 호출을 해주고 있다. 아주 적절하게 컴포지트 패턴을 적용해서 문제를 해결함을 확인할 수 있다.

ResponseEntity 객체를 반환한 경우에는 컴포지트 객체가 갖는 HandlerMethodReturnValueHandler 구현체 중에서 HttpEntityMethodProcessor가 사용된다. HttpEntityMethodProcessor 내부에서는 Response를 set해주고, 응답 가능한 MediaType인지 검사한 후에 적절한 MessageConverter를 선택해 응답을 처리하고 결과를 반환한다.

2. DispatcherServlet(디스패처 서블릿)의 초기화 과정 마지막으로 디스패처 서블릿의 초기화 과정 중 일부만 살펴보도록 하자.

[DispatcherServlet(디스패처 서블릿)의 초기화 과정] 앞서 설명하였듯 디스패처 서블릿은 J2EE 스펙의 HttpServlet 클래스를 확장한 서블릿 기반의 기술이다. 그러므로 디스패처 서블릿 역시 일반적인 서블릿의 라이프사이클을 따르게 되는데 서블릿의 라이프사이클은 다음과 같다.

초기화: 요청이 들어오면 서블릿이 웹 컨테이너에 등록되어 있는지 확인하고, 없으면 초기화를 진행함 요청 처리: 요청이 들어오면 각각의 HTTP 메소드에 맞게 요청을 처리함 소멸: 웹 컨테이너가 서블릿에 종료 요청을 하여 종료 시에 처리해야하는 작업들을 처리함

클라이언트로부터 요청이 오면 웹 컨테이너는 먼저 서블릿이 초기화 되었는지를 확인하고, 초기화되지 않았다면 init() 메소드를 호출해 초기화를 진행한다. init() 메소드는 첫 요청이 왔을 때 한번만 실행되기 때문에 서블릿의 쓰레드에서 공통적으로 필요로 하는 작업들이 진행된다. 그 작업들 중에는 디스패처 서블릿이 컨트롤러로 요청을 위임하고 받은 결과를 처리하기 위한 도구들을 준비하는 과정이 있다. 디스패처 서블릿은 요청을 처리하기 위해 다음과 같은 도구들을 필요로 한다.

Multipart 파일 업로드를 위한 MultipartResolver Locale을 결정하기 위한 LocaleResolver 요청을 처리할 컨트롤러를 찾기 위한 HandlerMapping 요청을 컨트롤러로 위임하기 위한 HandlerAdapter 뷰를 반환하기 위한 ViewResolver 기타 등등

스프링은 Lazy-Init 전략을 사용해 애플리케이션을 빠르게 구동하도록 하고 있어서, 첫 요청이 와서 서블릿 초기화가 진행될 때 애플리케이션 컨텍스트로부터 해당 빈을 찾아서 설정(Set)해준다. 그리고 이는 스프링의 첫 요청을 느리게 만드는 원인이다. 디스패처 서블릿의 초기화 화 로직은 다음과 같이 구현되어 있다. (부모 클래스 부분은 생략한 것이다.)

@Override protected void onRefresh(ApplicationContext context) { initStrategies(context); }

protected void initStrategies(ApplicationContext context) { initMultipartResolver(context); initLocaleResolver(context); initThemeResolver(context); initHandlerMappings(context); initHandlerAdapters(context); initHandlerExceptionResolvers(context); initRequestToViewNameTranslator(context); initViewResolvers(context); initFlashMapManager(context); }

위의 코드에서 도구들을 초기화하는 메소드 이름이 initStrategies인 이유는 전략 패턴이 적용되었기 때문이다. 대표적으로 뷰를 반환하기 위한 도구인 ViewResolver를 중심으로 살펴보도록 하자. ViewResolver에는 전략 패턴이 적용되었으므로 인터페이스이다. ViewResolver 외에 다른 모든 도구들도 전략 패턴을 적용하므로 인터페이스를 갖고 있고, 인터페이스 타입으로 선언되어 있다.

public interface ViewResolver {

```
@Nullable
View resolveViewName(String viewName, Locale locale) throws Exception;
```

}

스프링은 기본적으로 ContentNegotiatingViewResolver, BeanNameViewResolver, InternalResourceViewResolver를 빈으로 등록해둔다. 그리고 개발자가 추가한 Thymeleaf나 Mustache와 같은 템플릿 엔진을 위한 ViewResolver도 추가될 수 있다. (스프링 부트에서는 해당 의존성을 추가하면 자동으로 등록된다.) 결국 실제로 어떤 구현체가 사용될지는 애플리케이션 실행 후에야 알 수 있다. 그래서 스프링은 유연하게 도구들을 사용할 수 있도록 전략 패턴을 적용하였으며 여러 ViewResolver가 동작 가능하도록 List로 ViewResolver를 가지고 있다.

그런데 스프링은 ViewResolver에 추가적으로 컴포지트 클래스인 ViewResolverComposite를 만들어 컴포지트 패턴까지 적용하고 있는데, 그 이유는 WebMvcConfigurer와 관련이 있다. 스프링에서는 인터셉터나 CORS 처리 등 웹 기능을 확장하기 위해서 WebMvcConfigurer 인터페이스를 구현한 설정 클래스를 만들어준다. 그리고 여기서도 ViewResolver를 직접 등록해줄 수 있는데, 여기서 등록된 빈들은 CompositeViewResolver에 등록이 된다. 예를 들어 다음과 같은 설정 클래스를 추가했다고 하자.

@Configuration public class WebMvcConfig implements WebMvcConfigurer {

```
@Override
public void configureViewResolvers(ViewResolverRegistry registry) {
    registry.viewResolver(new MyViewResolver());
}

static class MyViewResolver implements ViewResolver {
    @Override
    public View resolveViewName(String viewName, Locale locale) throws
Exception {
        return null;
    }
}

@Bean
public MangKyuViewResolver mangKyuViewResolver() {
```

```
        return new MangKyuViewResolver();
    }

    static class MangKyuViewResolver implements ViewResolver {
        @Override
        public View resolveViewName(String viewName, Locale locale) throws
    Exception {
            return null;
        }
    }
```

}

MyViewResolver는 WebMvcConfigurer를 통해 등록되었으므로 ViewResolverComposite에 추가가 되고, MangKyuViewResolver는 직접 빈을 등록해준 것이므로 List 안에 등록이 된다. 최종적으로 디스패처 서블릿의 List는 다음과 같이 구성 된다.

ContentNegotiatingViewResolver BeanNameViewResolver MangKyuViewResolver ViewResolverComposite MyViewResolver InternalResourceViewResolver

첫 요청이 느린 문제를 해결하는 방법은 스프링 애플리케이션이 실행된 후에 아무런 API를 호출해 서블릿 초기화를 시키면 된다. 존재하지 않는 URI라 할지라도 디스패처 서블릿은 첫 요청을 받아들이기 위해 초기화 과정을 진행한다.

개발자라면 애플리케이션이 실행된 후에 1회 초기화 과정을 자동화하기를 원할 수 있는데 관련 내용은 이 포스팅에서 참고하도록 하자.

위의 내용들은 개인적으로 공부를 하면서 작성을 한 내용이라 충분히 틀리거나 잘못된 내용들이 있을 수 있습니다. 혹시 수정 또는 추가할 내용들을 발견하셨다면 댓글 남겨주세요! 반영해서 수정하도록 하겠습니다:)