템플릿 관련 정보는 나중에 통합해서 템플릿 용 요약 및 정리 본에 따로 마련.

**1 개발을 위한 준비**

**알아야 할 키워드 : @Controller, @Log4j2, ServletConfig, dispatch servlet, front controller**

**컨트롤러 src>main>java>org>scoula>controller>???.controller**Front controller(dispatch servlet)에 대한 설정을 하는 ServletConfig.java에서 사용하게 된다.  
ServletConfig에서 컨트롤러들을 사용하니까 @ComponentScan(basePackages={컨트롤러들이 모여있는 경로들}를 사용하여 @Controller가 적용된 컨트롤러 클래스들을 스캔하자.  
컨트롤러 담당 클래스에 @Controller 적용. @Controller는 @Component의 자식이다. 그러므로 해당 타입과 클래스의 캐멀케이스 이름으로 컨텍스트에 빈으로서 등록함.

컨트롤러에서는 무슨 일을 하냐. 요청과 메서드(포워딩할 페이지 이름 혹은 리다이렉트 할 요청을 반환하는)을 매핑하는 역할을 한다.

ServletConfig에서 컨트롤러 메서드로부터 받은 결과(포워딩할 페이지 이름 혹은 리다이렉트 할 요청)를 가지고 뷰리솔버를 통해 포워딩하거나 리다이렉트를 진행한다.

결과적으로 src>main>webapp>web-inf>views 밑에 있는 .jsp파일을 응답으로 보낸다.

**2 스프링의 특징과 의존성 주입**

**알아야 할 키워드 : RootConfig, @ComponentScan,**

그냥 범용적인 사용 클래스를 빈에 넣고 싶다면 @Component를 적용해라. 생성자와 setter가 있어야 한다. 롬복을 활용해라.

컨트롤러가 아닌 범용 클래스들은 RootConfig에서 스캔하고 사용하니 @ComponentScan을 통해 인지하게 경로들을 추가해줘야 한다

**3 spring mvc 기본구조**

**알아야 할 키워드 : RootConfig, ServletConfig, frontcontroller, dispatch servlet, WebConfig, filter, handler mapping, handler intercepter(adapter), controller, service(B.L.), DAO(mapper)(repository), view resolver, view.**

RootConfig.java 일반 설정, 범용 데이터 관리, 디비 연결 설정  
ServletConfig.java frontcontroller설정(dispatch servlet). 컨트롤러 관리.  
WebConfig.java 앱 관련 설정(필터링, 캐릭터 셋 등등)

RootConfig에 등록된 빈은 ServletConfig에 등록된 빈을 사용할 수 없다. 하지만 반대는 가능하다.

**모델2**  
요청 🡪 컨트롤러(요청 처리를 위한 로직 연동) 🡪 모델(순수한 데이터 처리 로직) 🡪 컨트롤러가 다시 받고 🡪 처리한 데이터를 view로 전달 🡪 응답

**Spring mvc 라이프 사이클**

1. **요청이 필터 통과**
2. **프론트컨트롤러가 수신**
3. **언어, 지역, 시간, 파일 처리 해석 등등 설정**
4. **핸들러 매핑이 요구를 분석하고 어떤 요구인지, 어떤 컨트롤러로 보내야하는지 파악. 요청마다 컨트롤러의 메서드가 매핑되어 있으므로 컨트롤러의 메서드의 요청이 넘어간다.**
5. **핸들러 인터셉터(핸들러 어댑터)가 알맞은 컨트롤러에서 DI, 데이터 준비 등 전처리**
6. **컨트롤러가 모델, 포매팅, BL(service)을 진행해 연산 결과를 얻음. 모델 얻음.**
7. **BL과정 중 디비에서 DAO(repository)를 통한 데이터 조회 및 수정이 있을 수 있다.**
8. **BL이 끝난 결과를 원하는 형식으로 프론트컨트롤러에게 넘김. 다양한 방법으로. 단일 데이터, 객체, 뷰 이름 등등**
9. **넘긴 결과를 뷰로 보여주기 위해 뷰 리솔버를 통해서 알맞은 뷰를 찾아냄.**
10. **이때 뷰는 프론트컨트롤러에서 얻은 데이터(모델)를 활용할 수 있음.**
11. **뷰를 응답으로 내보내기 전 필터링.**
12. **최종적인 뷰 응답으로 내보냄.**

**Spring mvc life cycle 정리 : filter 🡪 dispatchservlet 🡪 handlermapping 🡪 handlerIntercepter(adapter) 🡪 controller 🡪service(BL) [🡨🡪 DAO(repository)(Mapper)(DB 연동)] 🡪 viewresolver 🡪 view 🡪 filter 🡪 response**

**4 스프링 mvc의 controller. 컨트롤러 안에서 무슨 일이 일어나는지에 대하여.**

**알아야 할 키워드 : controller, handler mapping, handler intercepter(adapter), Service(BL), DAO(mapper), ServletConfig, view resolver, forward, redirect, view name, url request, request method, DTO  
@Controller, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping,  
@RequestParam, @DateTimeFormat**

위에 내용들을 종합하여 controller가 하는 일들은  
1) 자신이 담당한 요청이 들어올 경우 매핑되어 – handler mapping  
2) 사용자가 보낸 데이터(쿼리스트링 혹은 다른 형태의 데이터)를 가져오고 – 핸들러 인터셉터(어댑터)  
3) 서비스(BL)을 진행하여 결과를 도출하고 – model  
4) 과정 중에서 DB와 연동은 DAO를 통해서 데이터 조회 및 수정  
5) servletconfig에 있는 뷰 리솔버가 필요로 하는 뷰 이름이나 리다이렉트 요청 및 필요 데이터를 저장 및 반환한다.

**먼저1)항목, 요청 매핑 – handler mapping**  
@Controller  
@RequestMapping(“/sample”) //공통 URL 지정. /sample 로 시작하는 요청은 모두 해당 컨트롤러가 맡는다.  
public class SampleController{  
 @RequestMapping(“/뒤에붙을url나머지”)// /sample/뒤에붙은url나머지 요청은 해당 메서드가 맡는다.  
 public void basic(){//반환형이 void일 경우 맡은 url가 문자열로 반환된다. 결과적으로 url.jsp 뷰로.  
   
 }  
}

메서드 레벨 RequestMapping에서 좀 더 세부적인 조건을 추가할 수 있다. 메서드별로.  
@RequestMapping(“/뒤에붙을url나머지”, method={RequestMethod.GET}) 메서드  
// /sapmle/뒤에붙을url나머지가 get메서드로 요청 들어왔을 때만 해당 메서드와 매핑  
혹은 @GetMapping으로 줄여서 사용할 수 있다. @PostMapping, @PutMapping, @DeleteMapping도 있다.

**2)항목, DI 및 데이터 준비 등등 – handler adapter(인터셉터) // request.getParameter와 대응함.**  
핸들러 어댑터가 컨트롤러의 메서드의 매개변수에 DI를 해줄 상황 4가지.  
첫)매개변수타입이 생성자와 세터가 있는 DTO객체 타입인 경우 추가 코드 없이 그냥 주입해줌. 들어온 (요청) 데이터 필드명과 일치한 DTO객체의 필드에 주입. 내부에선 세터를 호출하여 주입.  
  
둘)매개변수타입이 객체가 아닌 단일 변수에 주입하고 싶다면  
@RequestParam(“들어온 (요청) 데이터 필드명”) Integer age 이런식으로 매개변수 앞에 어노테이션.  
primitive타입이어도 래퍼 객체로 받아야 0(디폴트값)과 null을 구분할 수 있음. Null은 정보가 없다라는 의미.  
셋)그렇다면 동일한 이름(들어온 (요청) 데이터 필드명)의 값(파라미터)이 여러 개 전달되는 경우  
매개변수타입을 ArrayList<>나 배열을 이용하면 돼. 체크박스 같은거. // /ex02?name=harry&name=kwon  
@RequestParam(“들어온 (요청) 데이터 필드명”)ArrayList<String> names  
객체들을 담는 리스트들도 이 경우에 해당한다.   
//요청 /sample/ex02?list[0].name=d&list[0].value=10&list[1].name=k 로 들어와도 ArrayList<??DTO> objs 로.  
  
넷)값에 대한 포맷이나 수정이 필요한 경우. 커스텀 바인딩을 사용하면 된다. 대표적인 예) Date타입  
적용할 타입 변수는 DTO의 필드멤버로 선언하고 @DateTimeFormat을 사용하면 된다.  
@Data public class DateDTO{ @DateTimeFormat(pattern=”입력될date값패턴”) Date reg\_date; }  
요청 /sample?reg\_date=22/03/12 일때, @DateTimeFormat의 pattern은 yy/MM/dd.

**5 스프링 mvc의 controller.**

**1-1컨트롤러가 뷰에 연산 결과 데이터(모델)을 어떻게 넘기는지.  
1-2컨틀롤러가 리다이렉트시 재요청에 필요한 데이터를 어떻게 넘기는지.  
1-3컨트롤러의 메서드의 반환값이 어떤 의미를 갖는지.**

**2파일 업로드를 위한 설정과 과정**

**3Spring mvc에서의 예외처리.**

**알아야 할 키워드 : Model, @ModelAttribute, RedirectAttributes, ResponseEntity, HttpHeaders, servletconfig, multipart, webconfig, customizeRegistration, enctype, 404, 500, @ControllerAdvice, @ExceptionHandler**

1-1)컨트롤러가 뷰(JSP)에 연산 결과 데이터(모델)을 어떻게 넘기는지 대하여  
리퀘스트 스코프나 세션 스코프에 데이터를 저장하여 jsp에서 사용한다.  
Model이라는 데이터 전달자를 이용한다. Controller의 메서드의 매개변수로 Model타입 변수를 사용하여 저장한다.  
@GetMapping(“/”) public void func(Model model){ model.addAttribute(키, 밸류); }  
//request스코프객체.setAttribute(키,밸류); 랑 대응한다.  
model타입 변수를 통해 requset스코프에 저장된 정보는 jsp에서 표현할 수 있다. 보통 EL표기를 사용한다.  
src>main>webapp>web-inf>views>???.jsp에서 ${키이름}==변수 혹은 객체 으로 표현. 뷰의 {{ 이름 }} 와 대응함.  
앞서 본 명시적으로 request 스코프에 attribute 추가 메서드를 호출하여 데이터를 저장하는 방법 말고 좀더 elegant한 방법 없을까  
매개변수에 생성자 & 세터가 있는 DTO객체 타입 변수가 handler intercepter에 의해서 DI되고 요청의 데이터와 매핑될 때 자동으로 request 스코프에 담긴다. 따로 model 데이터 전달자 변수를 통해 해당 데이터를 추가할 필요없다.  
DTO타입으로 데이터를 받지 않고 단일 매개변수에 데이터를 매핑하여 DI하고 싶다면 @RequsetParam 변수 레벨 어노테이션을 단일 매개변수에 적용하라 했지? 이렇게 얻은 데이터는 따로 model 데이터 전달자 변수를 통해 request 스코프에 추가해줘야 한다. 하지만 @ModelAttribute 변수 레벨 어노테이션을 단일 매개변수에 적용하면 @RequestParam 적용효과 + 해당 데이터가 request 스코프에 저장된다.   
// /ex04?page=9  
@GetMapping(“/ex”)public void ex4(@ModelAttribute(요청쿼리키이름&리퀘스트스코프에등록할키이름)int page){…}

1-2)컨트롤러가 리다이렉트시 재요청에 필요한 데이터를 어떻게 넘기는지  
response.sendRedirect(“/url?쿼리스트링”); 과 대응하는 spring mvc 방식을 알려줄게요.   
RedirectAttributes타입의 매개변수를 사용하면 해당 변수를 통해서 재요청 때 필요한 쿼리스트링을 만들어주거나 세션스코프 플래쉬로 데이터를 저장할 수 있다.  
@GetMapping(“/ex”) public String ex(RedirectAttributes ra){   
 ra.addAttribute(저장할키값,밸류);//쿼리스트링으로 데이터 저장.  
 ra.addFlashAttribute(저장할키값,밸류);//세션스코프에 잠시만 존재하게끔 데이터 저장.  
 return “redirect:공통매핑url/새로운요청경로”; //쿼리스트링 붙이지 않고 리다이렉트 요청 경로만 반환.   
 //후에 재요청 url구성은 공통매핑url/새로운요청경로?아까쿼리스트링으로저장한내용 형태임.  
}  
1-3)컨트롤러의 메서드의 반환 값의 의미에 대하여  
String, void, VO, DTO, ResponseEntity, Model, ModelAndView, HttpHeaders이 반환 타입으로 가능하다.  
String 🡺 jsp뷰의 경로/이름으로 해석.  
 forward시 뷰이름문자열 반환, redirect시 “redirect:요청url”문자열(쿼리스트링은 따로 저장했죠?) 반환  
 forward시 요청 “/sample/ex” 🡪 반환 “index” 🡪 뷰페이지로 이동 “/sample/ex/index.jsp”  
void 🡺 호출한 URL과 동인 이름의 jsp로 해석  
 요청 /sample/ex 🡪 반환 /web-inf/views/sample/ex05.jsp  
VO, DTO 객체 타입 🡺 JSON 타입의 데이터로 변환하여 응답 ‘바디’에 넣어서 브라우저로 응답 //rest통신 할때 사용됨.  
 implementation 'com.fasterxml.jackson.core:jackson-databind:2.9.4’//json문자열과 자바객체 파싱해줌.  
 build.gradle에 dependency 추가. Jackson-databind라이브러리 추가함.  
 객체타입 리턴할 때 JSON응답으로 보내고 싶다면 반환형 앞에 @ResponseBody 어노테이션 사용.  
 public @ResponseBody SampleDTO ex( ){ DTO dto = new DTO; … return dto; } bl결과를 담은 dto 내용이 json 변환 반환  
ResponseEntity타입 🡺 Http헤더 정보와 내용을 가공하여 브러우저로 응답 //rest통신에 많이 쓰여  
 브라우저로 직접 응답하는 경우에 사용하고, 응답 헤더와 바디를 설정할 수 있다.  
 ResponseEntity<바디부분에들어가는객체타입> 메서드이름( ) {   
 String msg=json문자열;  
 HttpHeaders header=new HttpHeaders( ); header.add(“Content-Type”, ”application/json;charset=UTF-8”);  
 return new ResponseEntity<>(msg, header, HttpStatus.OK); // 헤더와 바디 편집후 응답 생성  
 }  
Model 🡺 모델로 데이터를 변환한 걸 반환 //특수한 경우  
ModelAndView 🡺 모델로 데이터를 변환하고 뷰이름을 같이 지정한 것을 묶어서 반환 //특수한 경우  
HttpHeaders 🡺 응답에 내용 없이 http헤더 메시지만 전달 //특수한 경우

2)파일 업로드를 위한 설정과 과정 – 바이너리 데이터는 멀티 파트란 인코딩을 사용한다  
먼저 servletconfig에 MultipartResolver를 생성하며 Bean객체로 컨텍스트에 등록하는 메서드가 필요  
@Bean public MultipartResolver multipartResolver( ){  
 StandardServletMulitpartResolver resolver=new StandardServiceMultipartResolver( );  
 return resolver;  
}

그리고 webconfig에 파일업로드위치, 한파일최대사이즈, 한요청마다최대사이즈, 메모리쓸지 임시파일에 기록할지에 대한 기준 용량.에 대한 지정을하고 이를 등록하는 코드가 필요하다.  
@Override protected void customizeRegistration(ServletRegistration.Dynamic registration){  
 MultipartConfigElement multipartConfig=new MultipartConfigElement(  
 LOCATION, MAX\_FILE\_SIZE, MAX\_REQUEST\_SIZE, FILE\_SIZE\_THRESHHOLD  
 );  
 registration.setMultipartConfig(multipartConfig);  
}

파일 업로드 페이지 src > main > webapp > web-inf > views > sample > uploadform.jsp  
<form action=”/sample/uploadformpost” method=”post” enctype=”multipart/form-data”>  
 <input type=”file” name=”files”/>  
 <input type=”file” name=”files”/>  
 <input type=”submit”/>  
</form>  
업로드할 파일들을 받는 컨트롤러 메서드로 받을 수 있다.  
@PostMapping(“/uploadformpost”) public void uploadpost(ArrayList<MultipartFile> files){ … }

3)Spring mvc의 예외처리  
가장 흔하게 발생하는 404에러와 500에러.   
매번 에러 코드를 직접 처리하는 것보다 한 곳에 모아서 중앙집중식으로 처리, 관리하는 것이 편하다.

참고로 500에러는 internal server error이고 처리 과정 중 예상치 못한 오류로. 서버 내부 문제.

@ControllerAdvice는 전역적으로 컨트롤러에 영향을 미치는 역할을 한다. 경로를 따로 지정하지 않으면, 프로젝트의 모든 패키지에 있으면서 @Controller 어노테이션이 있는 클래스에서 발생하는 예외상황을 잡을 수 있다.  
그리고 @ControllerAdvice가 적용된 클래스에는 @ExceptionHandler를 적용한 메서드를 정의하고 처리할 예외를 명시해주면, 발생된 예외와 일치하는 메서드가 실행된다.

정의한 ExceptionManager 클래스에서 @ExceptionHandler(AppException.class) 와 같이 사용했다. 위에서 설명했듯, AppException이 컨트롤러에서 감지되면, @ExceptionHandler(AppException.class)가 적용된 메서드 로직이 실행되는 것이다. @ExceptionHandler( 처리맡을예외클래스.class )

에러를 중앙에서 처리하는 컨트롤러어드바이스를 구성해보자.   
@ControllerAdvice public class CommonExceptionAdvice{  
 @ExceptionHandler(Exception.class)//모든 예외가 이 메소드로 오겠네.  
 public String except(Exception ex, Model model){//Exception, Model 타입 DI 요구됨.  
 model.addAttribute(“exc1”,ex); //사용 예}  
 return “error\_page”;//사용자한테 보여줄 에러 안내 페이지  
 }  
}// 물론 해당 컨트롤러어드바이스는 다른 패키지에 소속시키는게 좋다.

Servletconfig에 @ComponentScan 경로에 컨트롤러어드바이스 있는 경로 추가하고.

src> main >webapp>web-inf>views>error\_page.jsp  
… <h4> ${ exc1.getMessage( ) } </h4> …

참고로 404에러는 웹에서 페이지나 파일이 존재하지 않거나 찾을 수 없을 때 발생하는 오류 코드.  
404에러는 서버에서 예외를 발생시키 않는다. 즉 NoHandlerFoundException가 발생할 상황에 예외가 발생하게끔 webconfig에 지정하자. Webconfig의 아까 만든 customRegistration메서드에   
registration.setInitParameter(“throwExceptionIfNoHandlerFound”, “true”); 코드를 추가하여 404상황에 예외가 발생하게끔 설정하자.  
에러를 중앙에서 처리하는 컨트롤러어드바이스에 404에러를 처리하는 메서드를 추가하자.  
@ControllerAdvice public class CommonExceptionAdvice{  
 @ExceptionHandler(NoHandlerFoundException.class)//특정 404 예외가 이 메소드로 오겠네.  
 public String except(NoHandlerFoundException ex, Model model, HttpRequest request){   
 model.addAttribute(“uri”, request.getRequestURI( )); //사용 예  
 model.addAttribute(“exception”,ex);  
 return “custom\_404”;//사용자한테 보여줄 에러 안내 페이지  
 }

src> main >webapp>web-inf>views>custom\_404.jsp  
… <h1> NOT FOUND ${ uri } ${ exception.getMessage( ) } </h1> …