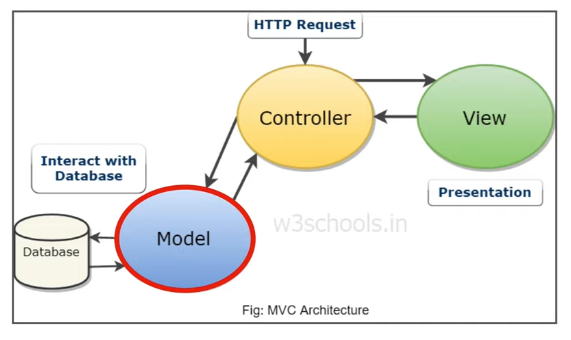
# **MVC(model-view-controller)** – паттерн, который предполагает выделение блоков, отвечающих за решение разных задач. Основное преимущество архитектуры MVC – возможность **менять один из компонентов приложения, существенно не влияя на остальные**.

**Model –** компонент отвечающий за данные.

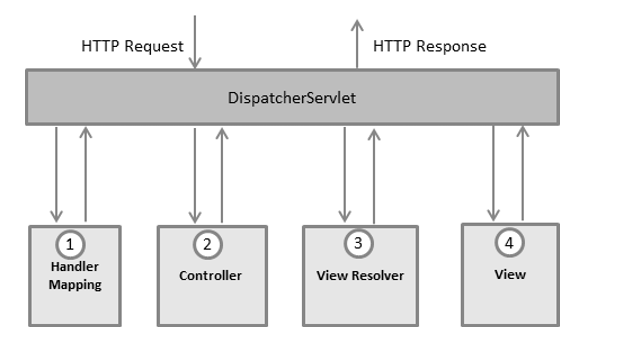
**View –** компонент отвечающий за взаимодействие с пользователем. Определяет внешний вид приложения и пользовательский интерфейс.

**Controller –** обрабатывает запросы пользователя. Связывает model и view – получает данные из model и передает их во view.



# **Основное отличие MVP и MVC** в том, что в MVC обновлённая модель сама говорит виду, что нужно показать другие данные. Если же этого не происходит и приложению нужен посредник в виде представителя, то паттерн стоит называть MVP.

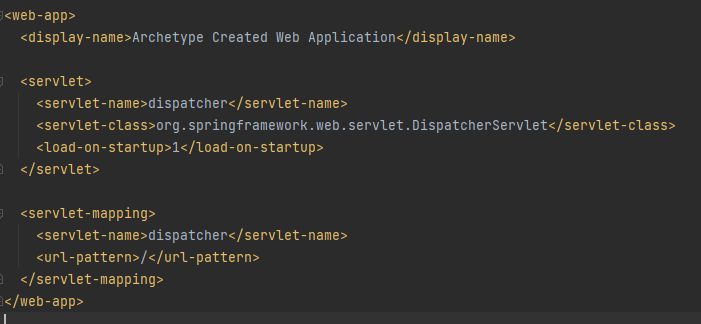
# **DispatcherServlet –** принимает и обрабатывает все HTTP-запросы и ответы на них.



# **Последовательность обработки запроса:**

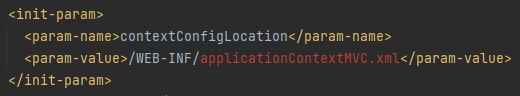
* После получения HTTP-запроса DispatcherServlet обращается к интерфейсу **HandlerMapping,** который определяет, какой контроллер должен быть вызван**.** HandlerMapping отправляет сведения о контроллере назад в DispatcherServlet.
* DispatcherServlet передает запрос в нужный контроллер. Выбранный контроллер обрабатывает запрос, вызвав соответствующий метод для подготовки данных и создав некую бизнес-логику (или напрямую извлекает информацию из БД).
* Контроллер возвращает в DispatcherServlet необходимые данные модели и имя представления.
* При помощи интерфейса **ViewResolver** DispatcherServlet определяет, какое представление нужно использовать на основе полученного имени.
* После того, как view создан, DispatcherServlet отправляет данные модели в виде атрибутов в представление, которое в конечном итоге отображается в браузере.

Конфигурация, как и в приложении на сервлетах, может быть задана в файле web.xml. Tomcat так же считывает информацию из него, и выполняет нужные настройки.

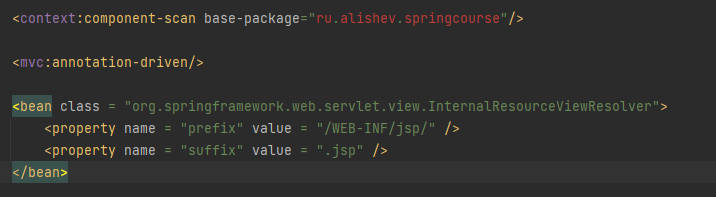


Как правило там мы указываем только DispatcherServlet и обрабатываемый им урл.

Можно также указать различные init-параметры, например нахождение файла с конфигурацией spring-context.



# В конфигурации контекста мы указываем какие пакеты сканировать на предмет наличия аннотаций, указываем что необходимо обрабатывать mvc аннотации, конфигурируем бин для **ViewResolver** (**InternalResourceViewResolver** по умолчанию).



**AbstractCachingViewResolver –** абстрактный класс, обеспечивающий функцию кэширования для представления. Представление парсится один раз и кэшируется. Нужное представление автоматически извлекается из кэша.

**UrlBasedViewResolver -** наследуется непосредственно от класса AbstractCachingViewResolver. Обеспечивает подход к поиску представлений на основе сшивки URL. может задать префикс и суффикс соответственно через свойства prefix и suffix, а URL конечного представления - это префикс + логическое представление + суффикс.

**InternalResourceViewResolver** наследуется от класса UrlBasedViewResolver. преобразует все возвращаемые имена представлений в объекты InternalResourceView.

**ViewResolverComposite –** класс, реализующий интерфейс ViewResolver. Он содержит в себе список ViewResolvers, и делегирует им работу. Он по очереди вызывает резолверы, пока не получит View.

# **Конфигурация с помощью Java-кода**

Для этого необходимо создать класс, который реализует интерфейс **WebApplicationInitializer**.

В его методе **onStartup** мы прописываем создание Dispatcher-сервлета и его настройку.

Этот интерфейс был уже реализован в абстрактном классе **AbstractAnnotationConfigDispatcherServletInitializer.**



Конфигурацию спринг контекста также можно вынести в java-файл:



# **Определение контроллера**

Пометив класс аннотацией **@Controller** мы указываем что данный класс является контроллером, и HandlerMapping будет проверять, можно ли обработать запрос этим контроллером.

# **@RequestMapping(“/url”)** – указывает какой url будет обрабатывать контроллер. Может также использоваться над методом контроллера. Урл указанный над классом будет считаться базовым, а над методами будут дополнять его.

Аннотация @RequestMapping имеет параметр **method**, который определяет, какой http-метод будет обрабатывать данный метод.

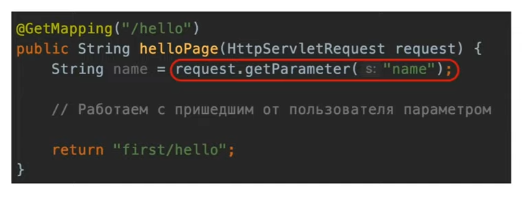
**Атрибуты** **@RequestMapping:**

* **сonsumes** – определяет тип содержимого тела запроса, который будет обрабатываться этим контроллером.  
  
* **produces -** определяет формат возвращаемого методом значения. Для работы этого параметра нужно чтобы в headers запроса был указан заголовок Accept (В каком виде принять ответ: html, json и т.д.). Значение produces должно совпадать с заголовком accept, чтобы запрос был обработан контроллером.   
  
* **params –** позволяет отфильтровать запросы по наличию/отсутствию определенного параметра в запросе или его значению. **params="myParam=myValue", params="!myParam=myValue", params="myParam", params="!myParam"**
* **headers** позволяет отфильтровать запросы по наличию/отсутствию определённого заголовка в запросе или по его значению. **headers="myHeader=myValue", headers="!myHeader=myValue", headers="myHeader", headers="!myHeader"**

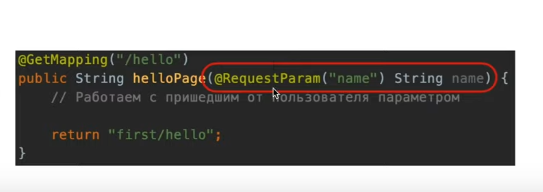
Есть также аннотации для конкретного http-метода: @PostMapping, @GetMapping, @PutMapping, @DeleteMapping.

# **ПАРАМЕТРЫ ЗАПРОСА**

Для обработки параметров запроса в URL можно использовать старый метод как в сервлетах: получать параметры из объекта **HttpServletRequest.**



# Либо можно воспользоваться аннотацией **@RequestParam** которая устанавливается над параметром метода контроллера.



@RequestParam можно использовать для извлечения параметров запроса, параметров формы и даже файлов из запроса.

@RequestParam(**“name”**) – позволяет указать имя извлекаемого параметра, если переменная в коде имеет другое имя.

@RequestParam(**required = false**) – помечает параметр как необязательный. Если он не был передан, в переменную будет записан null. Если параметр примитивного типа, то бросается исключение (null не может быть преобразован в примитив).

@RequestParam(**defaultValue = "test"**) – указать дефолтное значение для параметра, если он не был передан в запросе. При этом не обязательно использовать *required = false*.

@RequestParam **Map<String,String> allParams** - Можно положить все параметры в мапу, чтобы не перечислять их все.

@RequestParam **List<String> id** – если один параметр имеет несколько значений, то их можно положить в список.

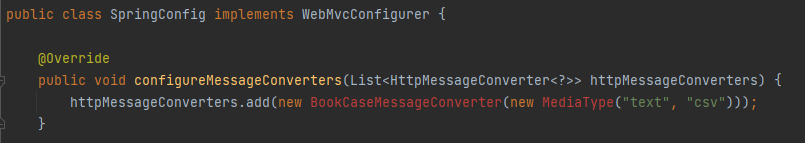
# **@RequestBody -**если аннотировать параметр метода с помощью этой аннотации, Spring попытается преобразовать содержимое тела входящего запроса в объект.

# **@ResponseBody –** Spring попытается преобразовать возвращаемое методом значение и автоматически записать его в тело ответа http.



Преобразование работает следующим образом: Есть список **HttpMessageConverters,** эти объекты отвечают за преобразование тела запроса в определенный класс и обратно в тело запроса, в зависимости от MIME типа. Spring перебирает эти объекты и находит первый, соответствующий MIME типу и вызывает метод для преобразования.

Добавить кастомный конвертер можно переопределив конфигурацию:



# **@PathVariable –** аннотация позволяет получить определенную часть URI.

URI: http://localhost:8080/getById/23

Следующий код поместит в переменную id значение 23.

@GetMapping("getById/{id}") **public** User **getUserById**(@PathVariable("id") String id) { //some logic

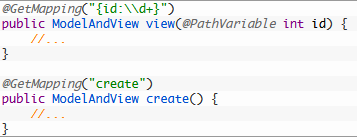
}

Атрибуты @PathVariable:

String **name** - Имя переменной пути для привязки.

Boolean **required** -Требуется ли переменная пути.

Чтобы отличить такие мапинги: **/person/create**и **/person/{id}** можно использовать регулярные выражения:



Регулярные выражения могут также пригодится если последний PathVariable содержит точку. Точка интерпретируется спрингом как расширение файла и часть после точки отбрасывается.

Если у нас есть несколько PathVariable, то мы можем положить их в мапу



# **@RequestHeader –** аннотация, позволяющая получить значение из заголовка запроса.

# **ResponseEntity –** специальный класс, который представляет http-ответ.. Мы можем использовать его для более тонкой настройки http-ответа: тело, статус ответа и заголовки.

Мы можем использовать любой объект в качестве тела.



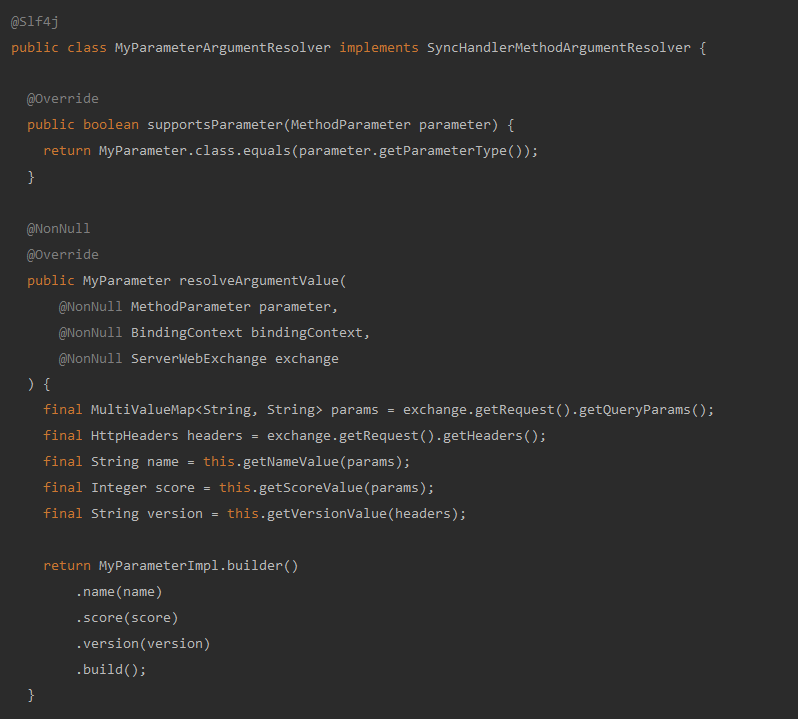
# **@ResponseStatus.(HttpStatus.OK) -** аннотация, позволяющая задать код http ответа.

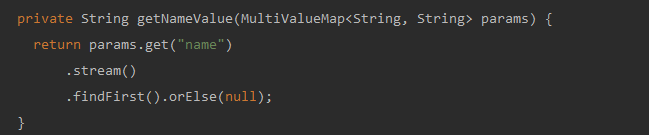
# **HandlerMethodArgumentResolver –** интерфейс, позволяющий объединить значения из параметров запроса, тела, заголовков и cookies в один объект.

Pageable в Spring как раз таки реализует этот интерфейс.

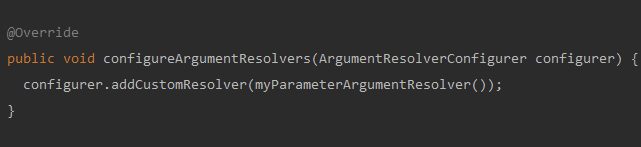
Этот интерфейс имеет 2 метода:

* **supportsParameter –** используется для проверки, должен ли параметр метода обрабатываться с помощью данного резолвера. Обычно там просто проверяем что он нужного типа.
* **resolveArgumentValue –** выполняет преобразование параметров в объект.





Резолвер регистрируется в Configuration.



# **МОДЕЛЬ**

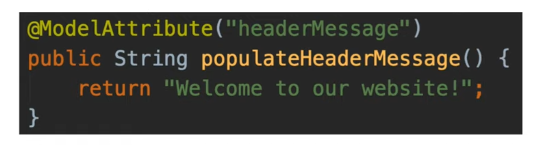
Интерфейс **Model** предоставляет доступ к модели. Мы можем указать объект Model в параметрах контроллера и Spring автоматически его внедрит.

Мы можем **добавлять** в модель **атрибуты** в виде пар ключ – значение.



Затем мы сможем извлечь эти атрибуты в представлении.

Аннотация **@ModelAttribute** может аннотировать:

* **Метод** – в данном случае значение, возвращаемое этим методом будет добавлено в модель каждого метода текущего контроллера. Например ниже в каждую модель будет добавлен атрибут с ключом «headerMessage».  
  
* **Аргумент метода** – в данном случае будет создан объект person, удовлетворяющий модели с данными например из html формы. Этот объект затем будет положен в модель. Если значение для поля не пришло, оно будет установлено по умолчанию.  
  

**ModelMap –** интерфейс аналогичен Model. Но так же позволяет передавать коллекцию значений, и обращаться с этими значениями так, если бы они находились внутри мапы.

**ModelAndView -** Этот интерфейс позволяет нам передавать всю информацию, требуемую Spring MVC, в одном возвращаемом объекте. Представление может принимать строку, либо объект View может быть указан напрямую.



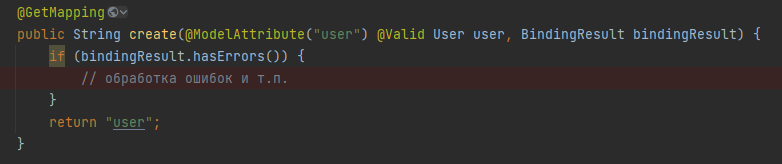
# **Валидация**

Для валидации удобно использовать **hibernate-validator.**

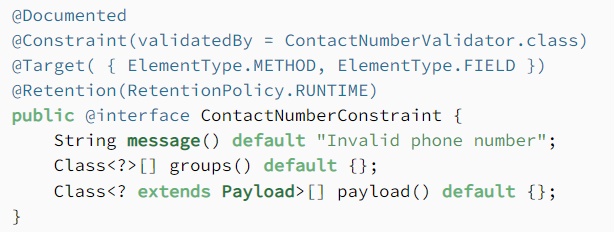
Для валидации входных данных используются следующие аннотации:

* **@NotEmpty(message = “message”)** – поле не должно быть пустым.
* **@Size(min = 2, max = 30, message = “message”)** – размер строки.
* **@Min(value = 12.23, message=””) –** минимальное допустимое значение для числа.
* **@Email(“message”)** – аннотация для валидации email адреса.
* **@Negative -** Значение должно быть отрицательным, либо быть null.
* **@Positive** - Значение должно быть положительным, либо быть null.
* **@Future -** Значение переменной должно быть будущим временем.
* **@FutureOrPresent -** Значение переменной должно быть будущим либо настоящим временем.
* **@Past -** Значение переменной должно быть прошедшим временем.
* **@NotBlank -** Значение типа CharSequence не должно быть null, пустым или состоять из одних лишь пробельных символов.
* **@Pattern -** Значение типа CharSequence должно соответствовать указанному регулярному выражению.

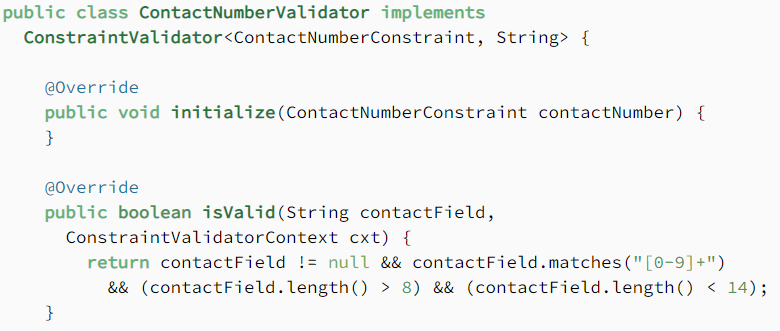
# Затем с помощью аннотации **@Valid** выполняется валидация объекта. Эту аннотацию обрабатывает по умолчанию Hibernate Validator. Если аргумент не проходит валидацию падает ошибка **BindException.**

Результат помещается в объект типа **BindingResult**, с помощью которого мы можем проверить наличие ошибок, например методом hasErrors. 

В *hibernate-validator* мы можем определить и свою аннотацию. Свою аннотацию нужно пометить **@Constraint,** чтобы указать какой класс будет валидировать поле.



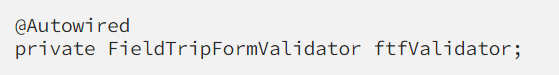
Сам класс валидатора наследуется от класса **ConstraintValidator**, и в нем нужно переопределить метод **isValid**.



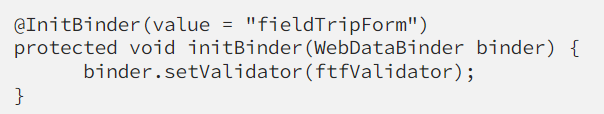
В голом Spring для написания своего валидатора нужно реализовать интерфейс **Validator** и переопределить 2 его метода:

* **supports(Class)** – может ли данный валидатор валидировать объект указанного класса.
* **validate(Object, org.springframework.validation.Errors) –** валидирует указанный объект, и в случае ошибок регистрирует их в объекте Errors.
* **public** **class** PersonValidator **implements** Validator {
* */\*\**
* *\* This Validator validates just Person instances*
* *\*/*
* **public** **boolean** supports(Class clazz) {
* **return** Person.**class**.equals(clazz);
* }
* **public** **void** validate(Object obj, Errors e) {
* ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
* Person p = (Person) obj;
* **if** (p.getAge() < 0) {
* e.rejectValue("age", "negativevalue");
* } **else** **if** (p.getAge() > 110) {
* e.rejectValue("age", "too.darn.old");
* }
* }
* }

Затем нужно добавить валидатор в класс контроллера:



И привязать валидатор к контроллеру:



Теперь параметр аннотированый **@Valid** будет валидироваться.

С помощью @**InitBinder** можно не только привязывать валидаторы, но и преобразовывать параметры запроса и форматировать их.

Можно также запретить привязку некоторых полей, когда например в классе есть поля, которые не предусмотрены для записи со стороны клиента. Допустим в классе User есть поле Role, если клиент пришлет Update запрос с полем role, то роль будет изменена, что как бы не предусматривалось приложением. Указать допустимые поля можно следующим образом:

binder.setAllowedFields("oldEmailAddress", "newEmailAddress");

# **ПРЕДСТАВЛЕНИЕ**

Spring MVC поддерживает множество типов представлений для различных технологий отображения страницы. В том числе — JSP, HTML, PDF, Excel, XML, Velocity templates, XSLT, JSON, каналы Atom и RSS, JasperReports и проч. Но чаще всего используются шаблоны JSP, написанные при помощи JSTL.

**View –** интерфейс, отвечающий за рендеринг контента и раскрытие модели.

Представления должны быть бинами. Скорее всего, они будут инстанцироваться как бины с помощью ViewResolver.

# **ОБРАБОТКА ОШИБОК**

До Spring 3.2 основным способом обработки ошибок в приложении были **HandlerExceptionResolver** и аннотация **@ExceptionHandler.**

C помощью *@ExceptionHandler* можно обрабатывать исключения **на уровне** отдельного **контроллера**. Для этого мы объявляем метод с логикой обработки ошибок, и ставим над ним аннотацию.

@ExceptionHandler(BusinessException.class)

public Response **handleException**(BusinessException e) {

return new Response(e.getMessage());

}

Основной недостаток этого подхода в том, что обработчик ошибок определяется для каждого контроллера и нельзя объявить глобально для всего приложения. Это ограничение можно обойти определив *@ExceptionHandler* в базовом классе, однако такой подход не всегда возможен.

**HandlerExceptionResolver –** общий интерфейс для обработчиков исключений. Все исключения выброшенные в приложении будут обработаны одним из подклассов *HandlerExceptionResolver.* Можно определить свою реализацию, либо воспользоваться готовыми.

**ExceptionHandlerExceptionResolver**— этот резолвер является частью механизма обработки исключений с помощью аннотации *@ExceptionHandler*.

**DefaultHandlerExceptionResolver**— используется для обработки стандартных исключений Spring и устанавливает соответствующий код ответа, в зависимости от типа исключения:

| **Exception** | **HTTP Status Code** |
| --- | --- |
| BindException | 400 (Bad Request) |
| ConversionNotSupportedException | 500 (Internal Server Error) |
| HttpMediaTypeNotAcceptableException | 406 (Not Acceptable) |
| HttpMediaTypeNotSupportedException | 415 (Unsupported Media Type) |
| HttpMessageNotReadableException | 400 (Bad Request) |
| HttpMessageNotWritableException | 500 (Internal Server Error) |
| HttpRequestMethodNotSupportedException | 405 (Method Not Allowed) |
| MethodArgumentNotValidException | 400 (Bad Request) |
| MissingServletRequestParameterException | 400 (Bad Request) |
| MissingServletRequestPartException | 400 (Bad Request) |
| NoSuchRequestHandlingMethodException | 404 (Not Found) |
| TypeMismatchException | 400 (Bad Request) |
|  |  |

**ResponseStatusExceptionResolver**— позволяет настроить код ответа для любого исключения с помощью аннотации @ResponseStatus.

@ResponseStatus(value = HttpStatus.INTERNAL\_SERVER\_ERROR)

public class **ServiceException** extends **Exception** {

public **ServiceException**(String message) {

super(message);

}

}

**Кастомный HandlerExceptionResolver – в нем, в отличии от предыдущих, можно передать тело ответа, а не только статус код.**

@Component

public class **CustomExceptionResolver** extends **AbstractHandlerExceptionResolver** {

@Override

protected ModelAndView **doResolveException**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {

ModelAndView modelAndView = new ModelAndView(new MappingJackson2JsonView());

if (ex instanceof CustomException) {

modelAndView.setStatus(HttpStatus.BAD\_REQUEST);

modelAndView.addObject("message", "CustomException was handled");

return modelAndView;

}

modelAndView.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR);

modelAndView.addObject("message", "Another exception was handled");

return modelAndView;

}

}

В версии 3.2 была введена аннотация **@ControllerAdvice,** позволяющая обрабатывать исключения глобально и централизованно.

Эта аннотация ставится над классом, и все методы @ExceptionHandler, @InitBinder или @ModelAttribute объявленные в advice будут использоваться в контролллерах.

Эта аннотация ставится над классом, и в этом классе определяются методы обработчики.

@ControllerAdvice

public class **DefaultAdvice** {

@ExceptionHandler(BusinessException.class)

public ResponseEntity<Response> **handleException**(BusinessException e) {

Response response = new Response(e.getMessage());

return new ResponseEntity<>(response, HttpStatus.OK);

}

}

**Атрибуты аннотации @ControllerAdvice:**

* **annotations –** указывает, что advice будет обрабатывать ошибки только тех контроллеров, которые аннотированы указанными аннотациями.

@ControllerAdvice(annotations = CustomExceptionHandler.class)

* **basePackages (**алиас для value**) –** список пакетов, контроллеры которых будет обрабатывать advice.
* **assignableTypes –** список контроллеров, обрабатываемых адвайсом

В Spring 5 добавился новый класс **ResponseStatusException**, который удобен для обработки базовых ошибок REST AP.

Выбрасывая *ResponseStatusException* можно также возвращать пользователю определённый код статуса, в зависимости от того, что произошло в логике приложения. При этом не нужно создавать кастомное исключение и прописывать аннотацию *@ResponseStatus* — просто выбрасываем исключение и передаём нужный статус-код.

throw new ResponseStatusException(HttpStatus.INTERNAL\_SERVER\_ERROR, "ResponseStatusException in testResponseStatusException");

# **Работа с cookies**

Мы все еще можем получить Cookie с помощью объекта **HttpServletRequest.getCookies()**, и записать используя **HttpServletResponse.addCookie().**

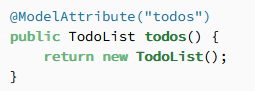
Для получения значения из куки можно воспользоваться аннотацией **@CookieValue**:

public String readCookieExample(*@CookieValue*(value = "cookieName", required = false) Cookie cookieName){}

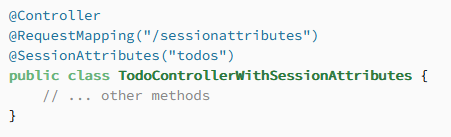
# **Работа с сессией**

Для работы с сессией используются аннотация **@SessionAttribute** и **@SessionAttributes.**

Объект, который нужно сохранить в сессию создается в методе @ModelAttribute.



Теперь этот объект будет доступен в любом методе контроллера. Чтобы сохранить его состояние на время сессии, используем аннотацию @SessionAttributes.



Затем можем прокинуть этот объект в метод контроллера

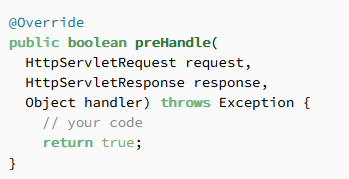
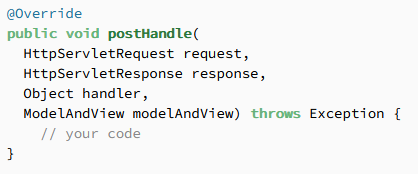
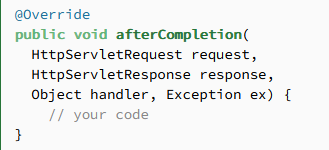


# **Interceptors**

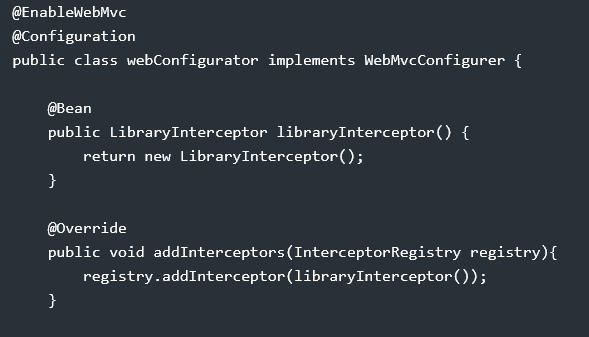
**Перехватчики** перехватывают запросы и обрабатывают их. Они помогают избежать повторяющегося кода обработчика, такого как логирование и проверка авторизации, валидации.

Перехватчик может работать до отправки запроса обработчику, после получения ответа от обработчика и после завершения (отображения представления).

Для написания перехватчика нужно реализовать интерфейс **HandlerInterceptor**. Он содержит 3 основных метода:

1. **prehandle**() - вызывается перед выполнением фактического обработчика. он возвращает true, чтобы позволить фреймворку отправить запрос дальше в метод обработчика (или в следующий перехватчик)  
   
2. **postHandle**() - вызывается после выполнения обработчика. Мы можем использовать этот метод для добавления дополнительных атрибутов к ModelAndView или для определения времени, затраченного методом обработчика на обработку запроса клиента.  
   
3. **afterCompletion**() - вызывается после завершения выполнения запроса и генерации представления. Например, мы можем использовать этот метод для добавления аватара вошедшего в систему пользователя в модель.  
   

Чтобы добавить перехватчик используется метод конфигурации **addInterceptors.**



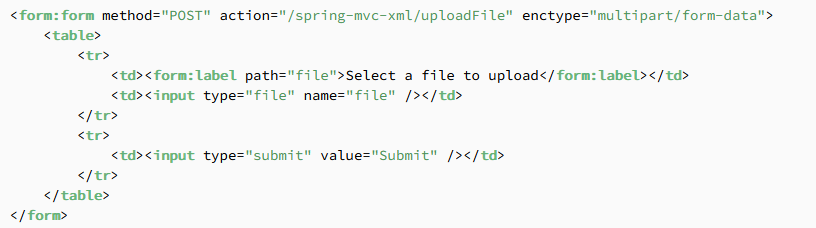
Мы можем добавить **шаблоны путей**, которые мы хотим перехватить с помощью этого перехватчика, используя **.addPathPatterrns().**

Мы можем определить более одного перехватчика для определенного метода обработчика и указать **порядок выполнения** с помощью **.order().**

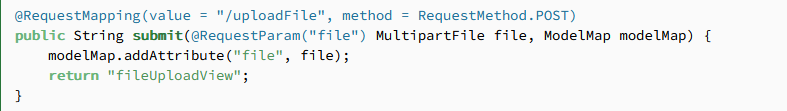
# **Загрузка файлов**

В Spring есть встроенный клаcc **MultipartFile**, позволяющий работать с файлами.

Создаем форму с input типа file:



В контроллере получаем файл с помощью @RequestParam



**Настроить максимальный размер** загружаемого файла можно в application.properties



Мы так же можем контролировать разрешена ли загрузка файлов и путь к каталогу, куда будут загружаться файлы.

