**Spring MVC аннотации**

**1. @Controller** – эта аннотация помечает класс контроллер.

**@Controller** тот же **@Component**, но с дополнительными возможностями.

Наследуется от аннотации **@Component**, поэтому **@ComponentScan** работает с **@Controller** так же как и с **@Component.**

Внутри контроллеров находятся методы:

- Обычно (но не всегда), каждый метод соответствует одному URL`у.

- Обычно (но не всегда, могут так же возвращать JSON или другие структуры данных которые будут приниматься другим приложением), методы возвращают строку (String) — название представления, которое надо показать пользователю.

- У метода может быть любое название.  
  
2. **Маппинги**

Маппинги связывают метод контроллера с адресом, по которому можно к этому методу обратиться(из браузера, например)

Всего 5 новых видов маппинга и 1 старый — в зависимости от того, какой HTTP запрос (с каким HTTP методом) должен прийти в этот метод контроллера:

**@RequestMapping** – Эта аннотация может использоваться на уровне классе для выражения общих отображений или на уровне метода (устаревший способ — лучше использовать новые аннотации) для сужения до конкретного отображения конечной точки. Проще говоря, аннотация используется для сопоставления веб-запросов с методами Spring Controller.

**@GetMapping** – Используется для реалзиации использования GET запросов.

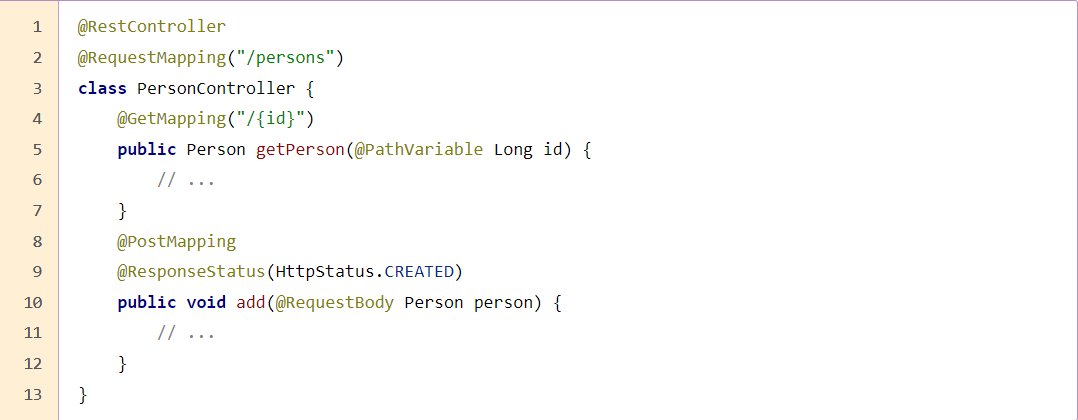
**@PostMapping** – Используется для реалзиации использования POST запросов.

**@PutMapping** – Используется для реалзиации использования PUT запросов.

**@DeleteMapping** – Используется для реалзиации использования DELETEзапросов.

**@PatchMapping** – Используется для реалзиации использования PATCH запросов.

Новые аннотации являются кастомными аннотациями, которые указываются потому, что, пожалуй, большинство методов контроллера нужно сопоставить с конкретным HTTP-методом, а не использовать аннотацию **@RequestMapping**, которая по умолчанию производит сопоставление со всеми HTTP-методами. Для выражения общих отображений на уровне класса по-прежнему необходима **@RequestMapping**.

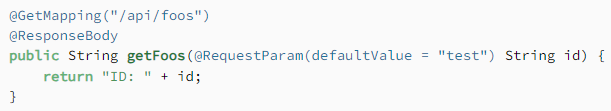
Пример:

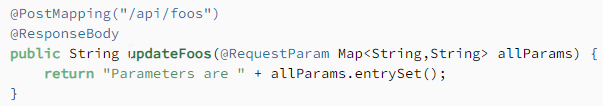
**3.** **@RequestParam** - используется для чтения значений из параметров запроса. Это обычно используется, когда параметры передаются в URL после символа ?.

Например, http://localhost:8080/spring-mvc-basics/api/foos?id=abc.  

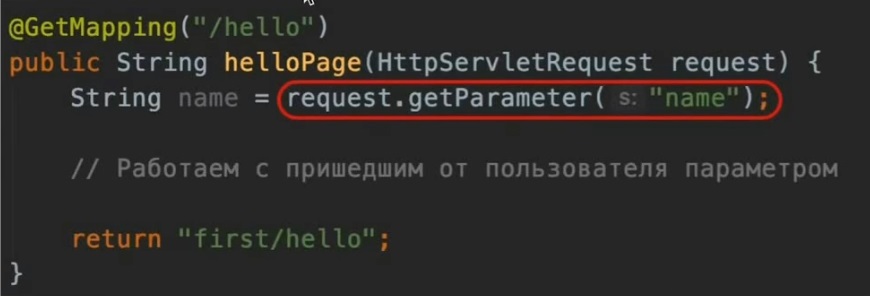

В этом случае, id=abc извлекается из URL и связывается с параметром метода id.

Параметры метода, аннотированные @RequestParam, являются обязательными по умолчанию.  
Это означает, что если параметр отсутствует в запросе, мы получим ошибку.  
Однако мы можем настроить @RequestParam как необязательный с атрибутом required:   
  
  
  
  
  
  
  
Мы также можем установить значение по умолчанию для @RequestParam, используя атрибут defaultValue:

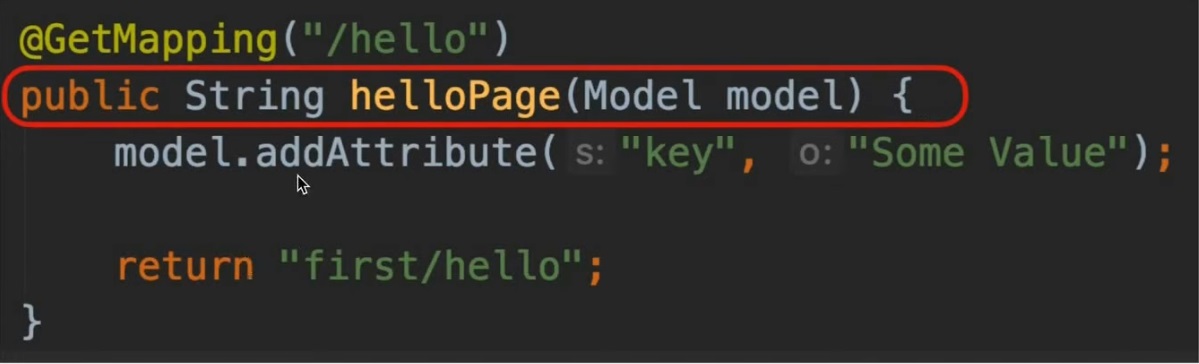
  
  
  
  
  
Мы также можем иметь несколько параметров без определения их имен или количества, просто используя Map:



Так же получить параметры из запроса можно с помощью объекта **HttpServletRequest,** мы этот объект пишем в параметрах нашего метода в контроллере и из этого параметра с помощью метода *getParametr()* можно получить параметры запроса. Так же в этом объекте есть вся информацию о запросе.



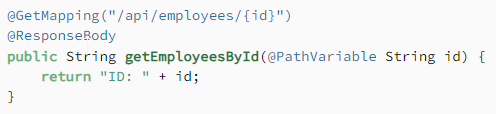
**4. Получение доступа к модели в контроллере.**

В параметрах метода контроллера пишем Model и Spring автоматически внедряет этот объект необходимый в метод контроллера и уже внутри этого метода контроллера мы можем обратиться к этому объекту и использовать его. С помощью метода *addAtribute()* мы можем в модель положить пару ключ значение и эта одель будет отправлена на представление, где уже с помощью шаблонизатора Thymeleaf мы сможем получить значение по ключу.  


**5. @PathVariable** - используется для связывания параметров из URL с параметрами метода. В приведенном примере, *id* может быть извлечено с помощью @PathVariable.

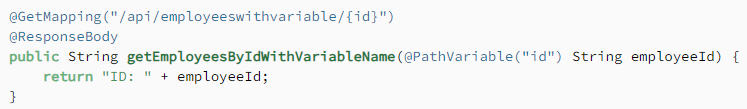
Например, http://localhost:8080/api/employees/111 .

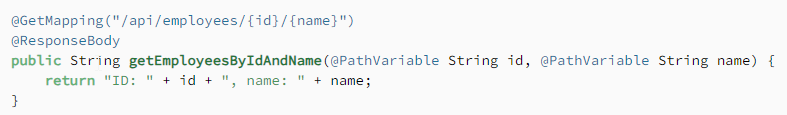
Число 111 в этом случае является идентификатором продукта (id).



В этом примере, {id} в URL сопоставляется с параметром id метода getEmployeesById.

Если имя переменной пути отличается, мы можем указать его в аргументе аннотации@PathVariable:



В зависимости от варианта использования мы можем иметь более одной переменной пути в нашем URI запроса для метода контроллера, который также имеет несколько параметров метода:

**6. @ModelAttribute** ­- эта аннотация может аннотировать метод или аргумент метода.

**Аннотация на уровне метода:**

В модель в каждом методе текущего контроллера добавляет ключ-значение.

Используется для добавления тех пар ключ-значение, которые нужны во всех моделях этого контроллера.



Любая модель из этого контроллера по умолчанию будет иметь значение с ключом headerMessage.

Можем в качестве значения добавлять любой объект, а не только строки.



Любая модель из этого контроллера по умолчанию будет иметь значение с ключом messageObject.

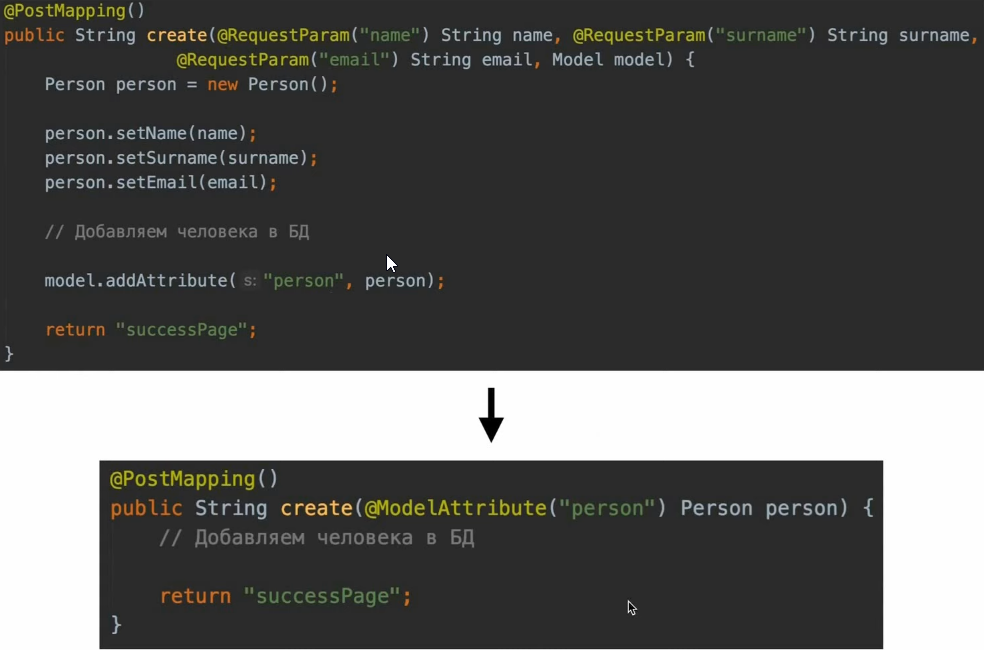
**Аннотация на уровне аргумента метода:**

Если метод контроллера на вход принимает какие то поля из которых нам нужно потом собрать какой нибудь объект(как на первом скриншоте), то вместо того чтобы, принимать все параметры в отдельности можно использовать аннотацию **@ModelAttribute,** которая сама создаст объект, и положит его в Model. Тоесть аннотация **@ModelAttribute** берет на себя три вещи:

1. Создание нового объекта.

2. Заполнение объекта из полей HTML формы.

3. Добавление созданого объекта в Model.



Если отправить запрос без полей, то в модель будут положен новый объект со значениями по умолчанию(0, null и т.д).

**7. @Valid** – валидация форм. Нужна для проверки правильности заполнения полей на формах.

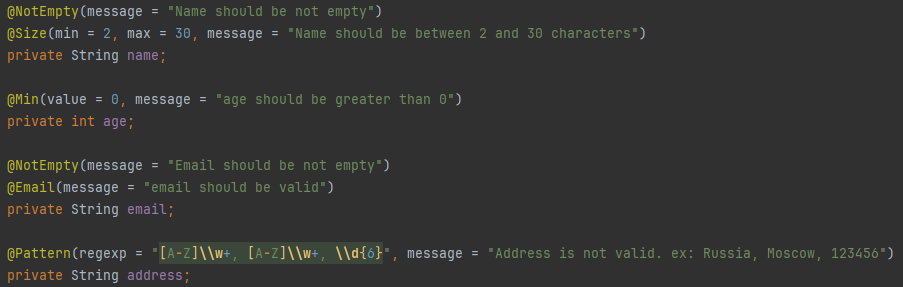
Для того чтобы была возможность использовать валидацию форма нужно подключить зависимость **hibernate-validator** в pom.xml.

Чтобы установить правила валидации существует множество аннотаций, с помощью этих аннотаций можно устанавливать правила валидации прямо на полях, вот часть из некоторых аннотаций:

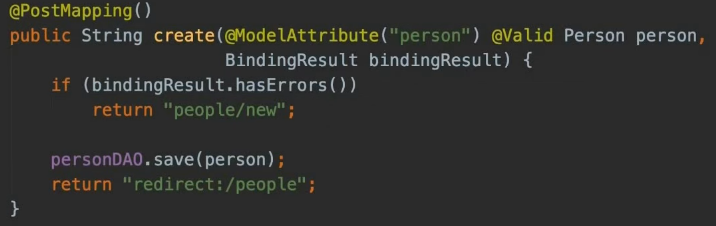
**@NotEmpty(<message>)** - указывает, что поле не может быть пустым. В message указывается сообщеине в случае если пользователь введет не верное значение.  
**@Size(<min>, <max>, <message>)** – указывает диапозон длины поля.

**@Min(<value>, <message>)** - устанавливает минимальное значение для числового поля.

**@Email(<message>) –** проверяет, что в этом поле лежит именно email.

**@Pattern(<regExp>, <message>)** - задает формат поля с помощью регулярного выражения.

Теперь, чтобы значения на форме валидировались нужно добавить аннотацию **@Valid** в контроллер, в аргументы того метода где нужно проверять поля которые мы указали в классе. Если у нас какие-то условия валидации нарушаются, то у нас появляется ошибка и эта ошибка помещается в отдельный объект **BindingResult. ВАЖНО:** объект **BindingResult** всегда должен идти после той модели которая валидируется, тоесть всегда если у нас есть какой то класс с аннотацией **@Valid**, то **BindingResult** должен всегда идти следующим аргументом после этого аругмента и именно в этот аругмент **BindingResult** будет внедрён тот объект с ошибками.



У объекта **BindingResult** есть специальный метод (***hasErrors()***) который говорит есть ли ошибки или нет.

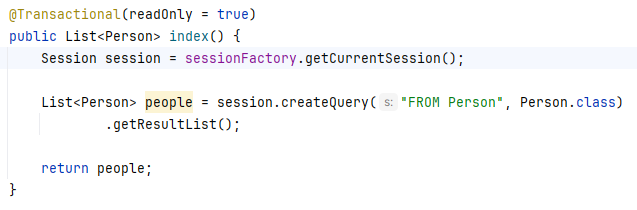
Так как аннотация **@ModelAttribute** автоматически добавляет объект в модель, то когда мы вернём форму “people/new” – на этой форме будет объект класса **Person** и у этого объекта уже будут ошибки, которые внедрились автоматически с помощью аннотации **@Valid** и эти ошибки мы с можем показать с помощью **Thymeleaf**.

**8.** В **Spring** приложении транзакции мы не открываем самостоятельно, тоесть не нужно вызывать метод ***beginTransaction()*** как это делается в чистом **hibernate**.

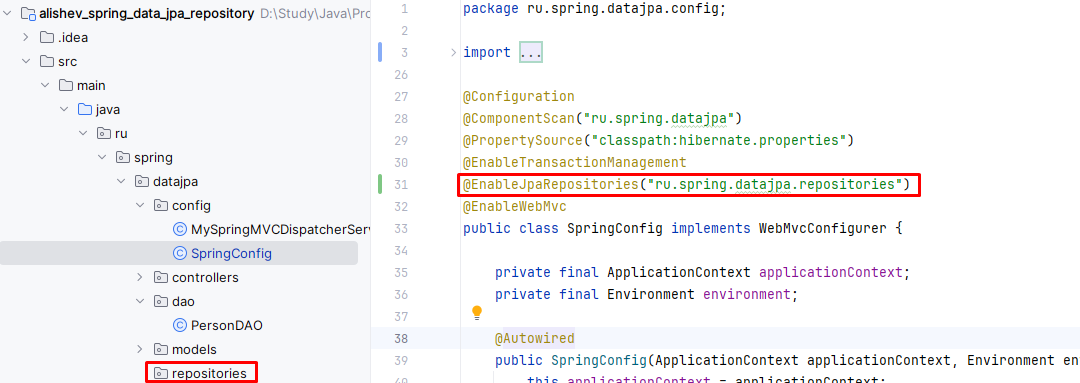
Чтобы начать транзакцию внутри какого-то метода нужно использовать специальную аннотацию которая называется **@Transactional**. Тоесть всё что будет происходит в этом методе будет присходить внутри транзакции и как только в этом методе все строчки будут пройдены транзакция автоматически будет закомченна(на сессии будет вызван метод **commit()**).

У аннотации **@Transactional** есть аргумент **readOnly.** Если в методе мы только читаем данные из базы данных, то этот аругмент следует поставить в положение **true**. Это может ускорить работы с базой данных.

Если целый класс помечается аннотацией **@Transactional**, то все публичные методы внутри этого класса автоматически будут иметь аннотацию **@Transactional**.



**9.** **@EnableJpaRepositories.** С помощью этой аннотации мы включаем поддержку JPA репозиториев. Внтури аннотации мы должны указать путь к папке где будут лежать Jpa репозитории. **Репозиторий** — это тот интерфейс который помечен аннотацией репозиторий.



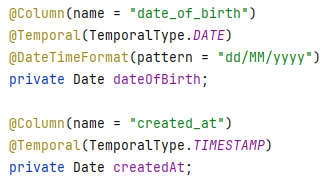
**10. @Temporal -** специальная аннотация которая относиться к датам в **Spring**`е. С этой аннотацией Spring понимает, что это поле является датой.

В аругментах этой аннотации нужно выбрать тот тип который у нас используется в базе данных. Например, **TemporalType.DATE** – только время, **TemporalType.TIMESTAMP** - дата и время.

**11. @DateTimeFormat** – с помощью этой аннотации мы сможем автоматически парсить строку даты из формы, тоесть в форме мы будем вводитть строку такого вида дд.мм.гггг, а Spring будет автоматически из этой строки делать объъект класса **Date**.

В аргументах мы должнгы уазать тот паттерн в котором у нас будет находится дата.

Если в форму мы будем передавать дату в другом формате или дата будет не валидная, то будет передавать сообщение об ошибке, но оно будет чисто техническое, нужно чтобы сообщение об ошибке было красивое и было понятно пользователю, то для этого можно реализовать **Spring validato**r.



**12. @SpringBootApplication –** это главная аннотация в Spring Boot, которая в себе содержит все остальные аннотации, тость одна аннотация для настройки всего грубо говоря. Эта аннотация проводит автоконфигурацию приложения, настраивает сканирование компонентов и т. д. Эта аннотация содержит в себе такие аннотаци такие как **@ComponentScan**, **@Configuration** и т. д.

Благодаря этой аннотации если мы создадим класс и пометим его аннотацией **@Component**, то автоматически он будет отсканирован и из этого класса будет создан бин.

В этой аннотации **@SpringBootApplication,** аннотация **@ComponentScan** настроена таким образом, что он сканирует все компоненты в текущей папке, тоесть в папке в которой находится наш класс и во всех подпапках причем рекурсивно.

Класс **SpringBootAppApplication** тоесть тот класс с аннотацией **@SpringBootApplication** должен находится в корне нашего проекта, тоесть все остальные классы должны находится на одном уровне с этим классом или на уровень ниже.

