№3 Java Spring Core, Configuration, Log, Mapping, Validation, Bean.

Задание

1. **Используя Spring Framework разработайте web- приложение**
   a. В системе предусмотрите два типа пользователей *администратор* и *обычный пользователь*. В дальнейшем будет поддерживаться авторизация Spring Security и хранение данных со Spring Data. Реализуйте в этом задании пока функции администратора. Соблюдайте архитектуру MVC.
   b. *Администратор* может зарегистрировать в системе новый «**объект»** с характеристиками (характеристика определит самостоятельно, например, id (личный номер), опты работы, технологии, дата рождения, должность) или удалить его (возможно перенести в список удаленных, архив). Храните объекты в файле json(xml).

   Варианты «объектов»:

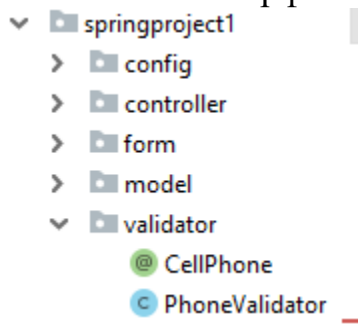
1	*Студент*
2	*Работник (компания)*
3	*Товар*
4	*Фильм*
5	*Альбом (музыкальный)*
6	*Транспорт*
7	*Игра*
8	*Приложение*
9	*Книга*
10	*Задача*
11	*Недвижимость*
12	*Документ*
13	*Компьютер*
14	*Проект*

   Таким образом, «объектами» управляет *администратор*. Пользователь сможет просмотреть список «объектов».
   c. Все действия *администратора* (удаление, добавление, редактирование и т.п.) должны записываться в лог на сервер.
   d. Разместите все отображаемые в этом сценарии страницы в отдельной папке, например, WEB-INF. Для отображения страниц измените настройку конфигурации ViewResolver (т.к. Spring Boot автоматически его конфигурирует).
   e. Используя Java Bean Validation API (JSR-303) объявите правила проверки при создании «объекта» пользователем. Для этого задайте правила проверки для класса модели (@Null,

@DecimalMin, @Digits, Pattern, Email, @Past @DateTimeFormat @NotBlank). Укажите, что валидация должна выполняться в методах контроллера (аннотация @Valid к аргументу метода). Измените вид(ы) формы для отображения ошибок проверки.

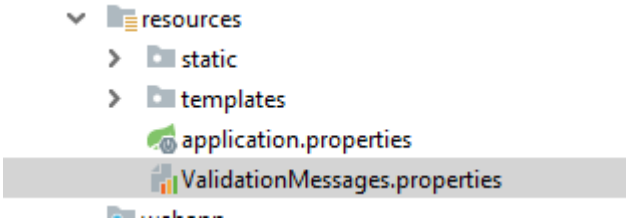
По ссылке можно получить информацию по валидацию https://docs.oracle.com/javaee/7/tutorial/partbeanvalidation.htm#sthref1322

f. Напишите пользовательский валидатор (аннотацию) используя интерфейс Validator. Разместите его отдельном пакете:



Примените его для поля при добавлении нового «объекта» в список.

g. Сообщения об ошибках вынесите в отдельный файл ресурсов V**alidationMessages**.



Например, содержимое файла V**alidationMessages** может быть следующее:

```
#message for user validator
Phone = "Phone is not valid"
valid.phone.cellphone = "Phone format is not valid"

valid.name.notNull = "Name is required"
valid.firstname.size.min3 = "Name must be at least 3 characters long"
valid.lastname.notBlank="LastName is required"
valid.street.notBlank = "Street is required"
valid.city.notBlank = "City is required"
valid.zip.digits = "Invalid zip cpde"
valid.email.notBlank = "Email is required"
valid.email.email = "Enter the email in correct format"
valid.birthday.past = "Date is not valid"
```

Добавим опеределение LocalValidatorFactoryBean в конфигурации приложения. Ниже показан пример.

```
@SpringBootApplication

public class SpringProject1Application {

   @Bean
   public MessageSource messageSource() {
      ReloadableResourceBundleMessageSource messageSource = new
```

```
ReloadableResourceBundleMessageSource();
      messageSource.setBasename("classpath:ValidationMessages");
      messageSource.setDefaultEncoding("UTF-8");
      return messageSource;
   }

   @Bean
   public LocalValidatorFactoryBean validator() {
      LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
      bean.setValidationMessageSource(messageSource());
      return bean;
   }

   public static void main(String[] args) {

       SpringApplication.run(SpringProject1Application.class, args);
   }

}
```

Измените класс модели. Для примера с person это может выглядеть так:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PersonForm {

    @NotNull(message="{valid.name.notNull}")
    @Size(min=3, message="{valid.firstname.size.min3}")
    private String firstName;
    @NotBlank(message="{valid.lastname.notBlank}")
    private String lastName;
    @NotBlank(message="{valid.street.notBlank}")
    private String street;
    @NotBlank(message="{valid.city.notBlank}")
    private String city;
    @Digits(integer=6, fraction=0, message="{valid.zip.digits}")
    private String zip;
    @NotBlank(message="{valid.email.notBlank}")
    @Email (message = "{valid.email.email}")
    private String email;

  //  @Pattern(regexp="^(0?[1-9]|[12][0-9]|3[01])[\\/\\-](0?[1-
9]|1[012])[\\/\\-]\\d{4}$",message="Must be formatted DD/MM/YYYY")
  //  private String birthday;

    //ISO 8601 date format (yyyy-MM-dd)
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    @Past(message = "{valid.birthday.past}")
    private Date birthday;


    @CellPhone (message = "{valid.phone.cellphone}")
    private String phone;


}
```

h. Изучите технику создания и использования Bean. Продемонстрируйте это на примере:

   1) Создайте в соответствии с вариантом bean. В итоге вы создадите некоторое количество сконфигурированных классов.

2) Используйте ApplicationContext. Сконфигурируйте bean через factory-method. Сконфигурируйте init, destroy методы с помощью конфигурации контекстов или с помощью InitializingBean, DisposableBean. Сконфигурируйте default-init и default-destroy методы. Поменяйте область применения bean (prototype, singleton,..)
3) Сконфигурируйте beans с использованием автосвязывания по имени, по типу, по конструктору.
4) Сконфигурируйте beans с использованием @Autowired (учитывая ограничения).
5) Сконфигурируйте beans с использованием Qualifying Ambiguous Dependencies.
6) Сконфигурируйте beans с использованием атрибута component-scan и include-filter.

Вопросы:

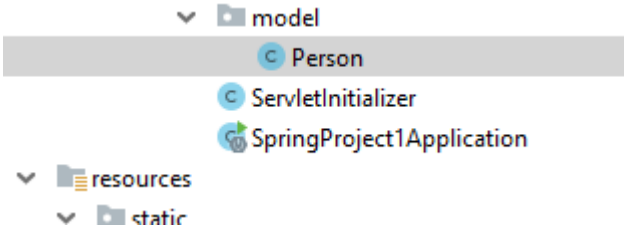
1. Перечислите Spring модули и их назначение.
2. Расскажите о составе Spring Framework.
3. Объясните принцип IoC (Inversion of Control)?
4. Объясните принцип DI (Dependency injection)? В чем между ними разница?
5. Какие существуют формы внедрения DI?
6. Что такое Spring Boot и для чего он используется?
7. Как создать и зарегистрировать контроллер?
8. Поясните значения аннотаций: @Configuration, @Bean, @Component, @Service, @Repository, @Controller.
9. Что такое ModelAndView?
10. Поясните как работает DispatcherServlet, HandlerMapping, ViewResolver?
11. Расскажите как использовать SLF4J (Simple Logging Facade for Java).
12. Поясните как можно использовать @RequestMapping, @GetMapping. @GetMapping и др.
13. Перечислите и охарактеризуйте все аннотации валидации.
14. Как создать пользовательский валидатор.
15. Что такое Spring Bean? Как получить экземпляр бина?
16. Какие существуют способы настройки класса в качестве spring bean?
17. Как прописать конфигурацию, чтобы выполнялся автоматический поиск бинов и управление контейнером?
18. В каком месте и для чего может использоваться аннотация @Autowired и @Qualifier?
19. В каком месте и для чего может использоваться аннотация @Configuration, @ComponentScan?
20. Какие scope может иметь Spring Bean?
21. Для чего используют BeanFactory и ApplicationContext?

Пояснения к выполнению

1) Для начала воспользуйтесь spring initializr (с maven поддержкой) с https://start.spring.io/ сгенерируйте и изучите проект. Напишите страницу приветствия для вашей IT компании (должно фигурировать компании и логотип). Добавьте страницу или область логин-пароль.

Можно создавать проект и без инициализатора. Добавьте поддержку Spring через maven зависимости. Настройте базовый скрипт сборки mave, добавьте модули spring framework org.springframework ->spring-context, spring-core, (модуль для работы с контекстом и Dependency Injection(DI)) и др.

2) Класс объекта разместите в пакете model, например

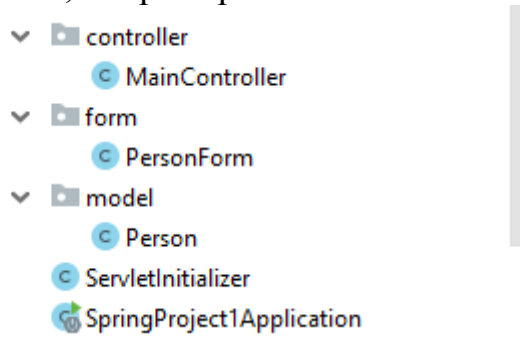


3) Для определения конструкторов, геттеров и сеттеров используйте Lombok, например:

```
@Data
@AllArgsConstructor
public class Person {
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String zip;
    private String email;
    private String birthday;
}
```

Посмотрит pom файл зависимости, если они не установлены, добавьте.

4) Для обработки запросов создайте класс контроллера (@Controller) в новом пакете *controller* , например:

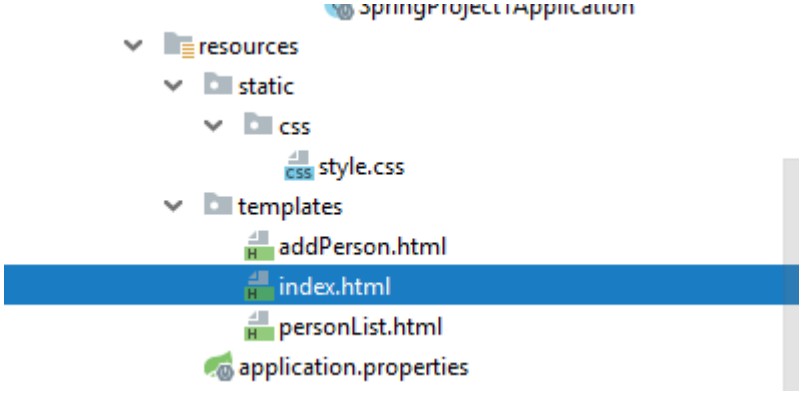



```
@Controller
public class MainController {

    //
    // Вводится (inject) из application.properties.
    @Value("${welcome.message}")
```

```
    private String message;

    @Value("${error.message}")
    private String errorMessage;

    @RequestMapping(value = {"/", "/index"}, method = RequestMethod.GET)
    public ModelAndView index(Model model) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("index");
        model.addAttribute("message", message);

        return modelAndView;
    }
}
```

Здесь используется аннотация @Value - удобный способ для “впрыскивания” значений из конфигурации Spring Boot в код. Можно задать значение по-умолчанию.

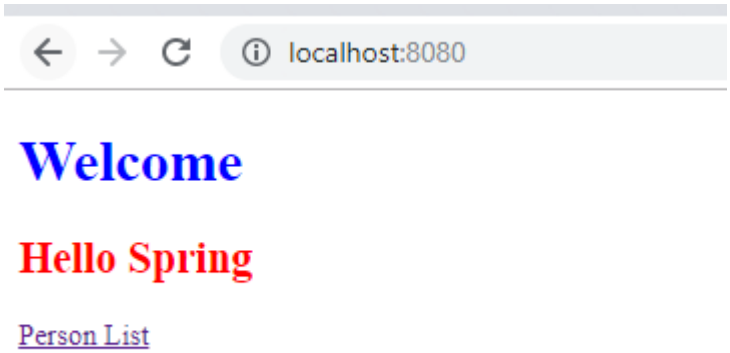
5) Расположите в папке **src/main/resources/templates** страницы



Например, страница приветсвия, которая возвращает ссылку на список объектов

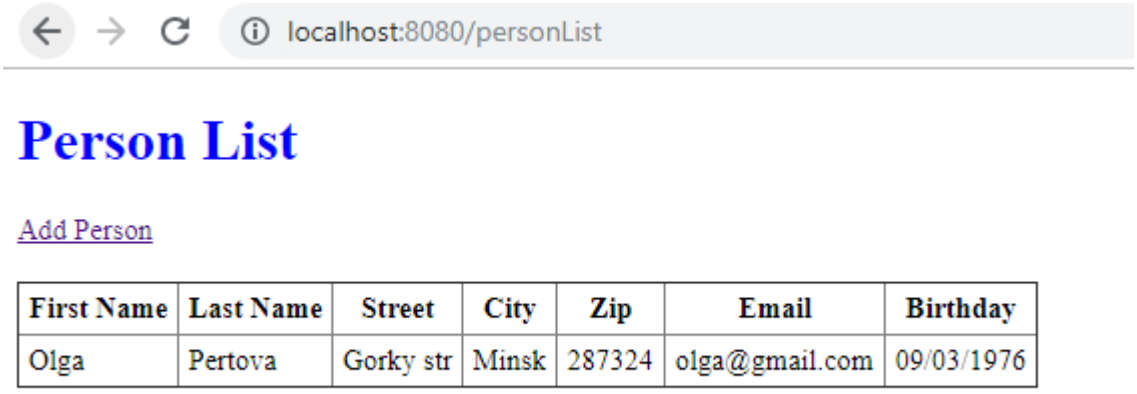
index.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Welcome</title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Welcome</h1>
<h2 th:utext="${message}">..!..</h2>

<a th:href="@{/personList}">Person List</a>

</body>
```



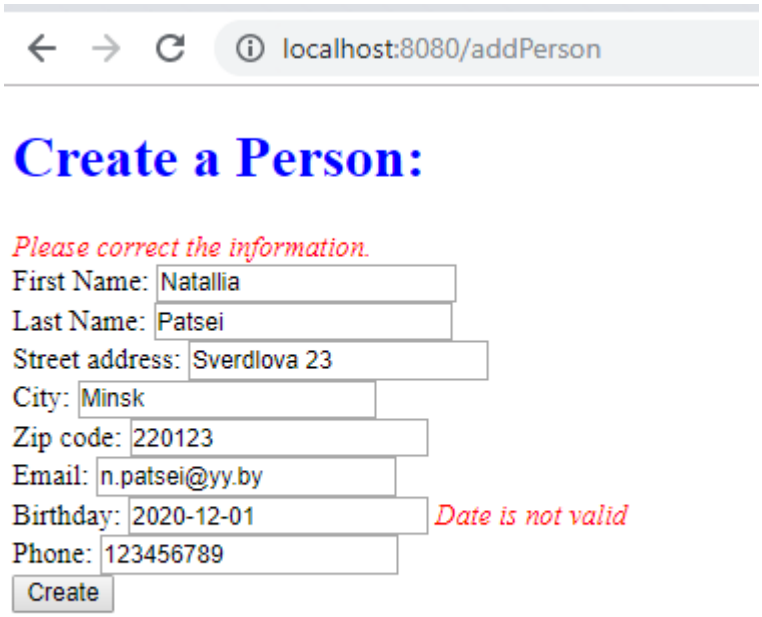
И страница, которая возвращает таблицу объектов

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Person List</title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Person List</h1>
<a href="addPerson">Add Person</a>
<br/><br/>
<div>
    <table border="1">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Street</th>
            <th>City</th>
            <th>Zip</th>
            <th>Email</th>
            <th>Birthday</th>
        </tr>

        <tr th:each ="person : ${persons}">
            <td th:utext="${person.firstName}">...</td>
            <td th:utext="${person.lastName}">...</td>
            <td th:utext="${person.street}">...</td>
            <td th:utext="${person.city}">...</td>
            <td th:utext="${person.zip}">...</td>
            <td th:utext="${person.email}">...</td>
            <td th:utext="${person.birthday}">...</td>
        </tr>
    </table>
</div>
</body>
</html>
```



Для создания объектов:

```
 <!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Add Person</title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Create a Person:</h1>


<form th:action="@{/addPerson}"
      th:object="${personForm}" method="POST">

    First Name:
    <input type="text" th:field="*{firstName}" />
    <br/>
    Last Name:
    <input type="text" th:field="*{lastName}" />
    <br/>
    Street address:
    <input type="text" th:field="*{street}"/>
    <br/>
    City:
    <input type="text" th:field="*{city}"/>
    <br/>
    Zip code:
    <input type="text" th:field="*{zip}"/>
    <br/>
    Email:
    <input type="text" th:field="*{email}"/>
    <br/>
    Birthday:
    <input type="text" th:field="*{birthday}"/>
    <br/>
    <input type="submit" value="Create" />

</form>
<br/>
<div th:if="${errorMessage}" th:utext="${errorMessage}"
     style="color:red;font-style:italic;">
</div>
</body>
```



Для представления можете использовать другие известные вам шаблоны.

4) Допишите методы добавления, удаления…в контроллер, например :

```
@Slf4j
@Controller
@RequestMapping
public class MainController {

    @GetMapping(value = {"/", "/index"})
    public ModelAndView index(Model model) {
...
    @GetMapping(value = {"/personList"})
    public ModelAndView personList(Model model) {
...
    @GetMapping(value = {"/addPerson"})
    public  ModelAndView showAddPersonPage(Model model) {
...
    @PostMapping(value = {"/addPerson"})
    public ModelAndView savePerson(Model model, //
...
```

5) Для проверки объекта используйте аннотации на полях класса модели:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PersonForm {

    @NotBlank(message="Name is required")
    @Size(min=3, message="Name must be at least 3 characters long")
    private String firstName;

      @Pattern(regexp="^(0?[1-9]|[12][0-9]|3[01])[\\/\\-](0?[1-
9]|1[012])[\\/\\-]\\d{4}$",message="Must be formatted DD/MM/YYYY")
     private String birthday;

     @NotNull
     @Min(18)
     private Integer age;
```

Есть еще много полезных аннотаций валидации. Например для даты:

```
@DateTimeFormat(style="S-")
```

Для валюты:

```
@NumberFormat(style= NumberFormat.Style.CURRENCY)
```

Для задания даты позднее текущей

```
@NotNull @Future
```

Диапазон

```
@Size(min = 2, max = 30)
```

Все аннотации валидации включают атрибут message, который определяет сообщение для пользователя, если введенная информация не соответствует требованиям правил проверки.

В контроллере укажите на необходимость проверки, когда форма отправляется в метод-обработчики. Для этого добавтье аннотацию @Valid к аргументу метода:

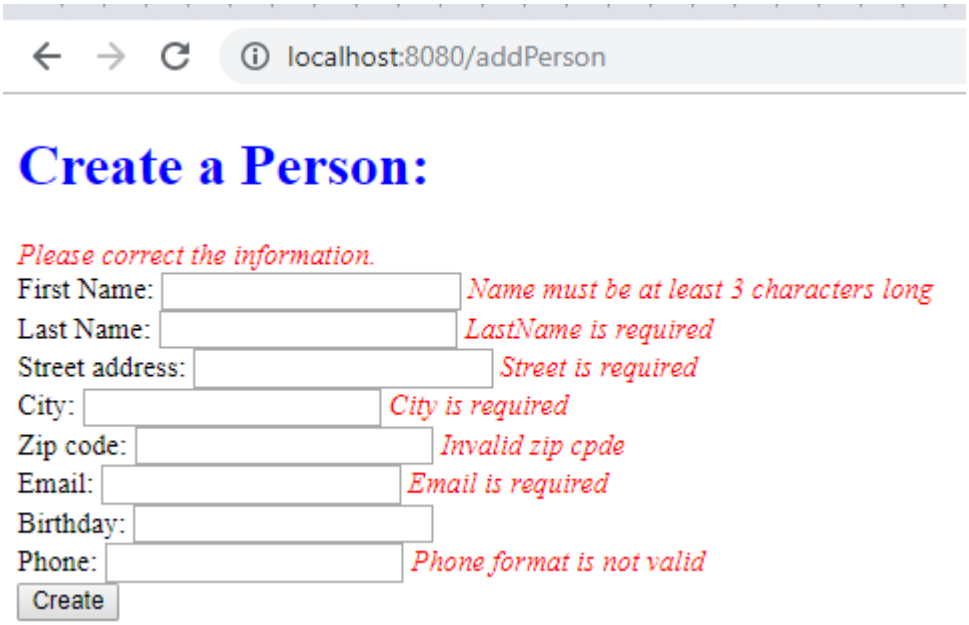
```
@PostMapping(value = {"/addPerson"})
public ModelAndView savePerson(  Model model,
                  @Valid @ModelAttribute("personForm") PersonForm personForm,
                  Errors errors) {
    ModelAndView modelAndView = new ModelAndView();
  if (errors.hasErrors()) {
      modelAndView.setViewName("addPerson");
  }
  else {
            modelAndView.setViewName("personList");
            String firstName = personForm.getFirstName();
            String lastName = personForm.getLastName();
            String street = personForm.getStreet();
            String city = personForm.getCity();
            String zip = personForm.getZip();
            String email = personForm.getEmail();
            String birthday = personForm.getBirthday();
            Person newPerson = new Person(firstName, lastName, street, city,
zip, email, birthday);
            persons.add(newPerson);
            model.addAttribute("persons", persons);
            log.info("/addPerson - POST  was called");
            return modelAndView;
        }
        return modelAndView;
    }
```

6) TBhymeleaf предлагает удобный доступ к объекту Errors через свойство fields и его атрибут th: errors:

```
<form th:action="@{/addPerson}"
      th:object="${personForm}" method="POST">

    <div th:if="${#fields.hasErrors()}">
 <span class="validationError">
 Please correct the information.
 </span>
    </div>
```

```
First Name:
<input type="text" th:field="*{firstName}" />
<span class="validationError"
      th:if="${#fields.hasErrors('firstName')}"
      th:errors="*{firstName}">
</span>
<br/>
```

В стиль добавьте, чтобы эти сообщения выводились курсивом или красным цветом.



7) В Spring есть свой интерфейс Validator. (org.springframework.validation.Validator) как и в Java (javax.validation.Validator). Его имплементация выполняет проверку данных. Это уже не декларативный подход, но в нем есть своя гибкость и расширяемость. Таким образом, есть две возможности - использовать спецификацию JSR-303 и его класс Validator или сделать реализацию интерфейса org.springframework.validation.Validator.

Определите пользовательскую аннотацию, например CellPhone. Она будет проверять формат телефона

```
@Documented
@Constraint(validatedBy = PhoneValidator.class)
@Target ({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface CellPhone {

    String message() default "{Phone}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Большая часть определения аннотации стандартна и соответствует спецификации JSR-303.

String message() — здесь мы указываем сообщение об ошибке по умолчанию. groups() и payload() в данном случае не пригодятся, поэтому они пустые.

@Target – указывает, что именно мы можем пометить этой аннотацией.
ElementType.PACKAGE – только для пакетов;
ElementType.TYPE – только для классов;
ElementType.CONSTRUCTOR – только для конструкторов;
ElementType.METHOD – только для методов;
ElementType.FIELD – только для атрибутов(*переменных*) класса;
ElementType.PARAMATER – только для параметров метода;
ElementType.LOCAL_VARIABLE – только для локальных переменных.

`@Target ({ElementType.FIELD}) -` сообщает о там, что аннотация будет применяться к полю.

`@Documented` – указывает, что помеченная таким образом аннотация должна быть добавлена в javadoc поля/метода.
@Retention – позволяет указать жизненный цикл аннотации: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения.
RetentionPolicy.CLASS – будет присутствовать в скомпилированном файле;
RetentionPolicy.RUNTIME – будет присутствовать только в момент выполнения;
RetentionPolicy.SOURCE – будет присутствовать только в исходном коде.

Наиболее важной частью является аннотация @Constraint, где мы предоставляем класс, который будет использоваться для проверки, т.е. PhoneValidator.

Реализация класса проверки выполняется в классе PhoneValidator.class

```
public class PhoneValidator implements ConstraintValidator<CellPhone, String> {

    @Override
    public void initialize(CellPhone paramA) {
    }
    @Override
    public boolean isValid(String phoneNo, ConstraintValidatorContext ctx) {
        if(phoneNo == null){
            return false;
        }
        //задание номера телефона в формате  "123456789"
        if (phoneNo.matches("\\d{9}"))
                        return true;
            //номер телефона может разделяться -, . или пробелом
        else if(phoneNo.matches("\\d{2}[-\\.\\s]\\d{3}[-\\.\\s]\\d{2}[-
\\.\\s]\\d{2}"))
                         return true;
            //может быть код оператора в скобках ()
        else if(phoneNo.matches("\\(\\d{2}\\)\\d{3}\\d{2}\\d{2}"))
                        return true;
            //может быть код страны в скобках ()
        else if(phoneNo.matches("\\(\\d{3}\\)\\d{2}\\d{3}\\d{4}"))
```

```
                    return true;
                //return false если ничего не подходит
            else return false;
        }

}
```

Необходимо чтобы класс реализовал интерфейс javax.validation.ConstraintValidator.
Если используются ресурс, например DataSource, то можно инициализировать его в методе initialize ().
Основную логику проверки выполняет метод **isValid(String phoneField, ConstraintValidatorContext cxt)**. Значение поля передается в качестве первого аргумента Метод проверки isValid возвращает true, если данные верны, иначе - false.
Прверка выполняется на основе регулярного выражения.

Аннотация готова, добавляем ее к полю и уже можно проверить, все поля на которых есть аннотации будут проверены соответствующими правилами.

```
@CellPhone (message = "Phone format is not valid")
private String phone;
```

Проверка выполняется при добавлении нового объекта. Поэтому изменим метод

```
@PostMapping(value = {"/addPerson"})
public ModelAndView savePerson(  Model model, //
        @Valid @ModelAttribute("personForm") PersonForm personForm,
Errors errors) {
   ModelAndView modelAndView = new ModelAndView();
   if (errors.hasErrors()) {
       modelAndView.setViewName("addPerson");
   }
```