



Validation

CS544: Enterprise Architecture

Overview

- In this lecture we will look at validating data.
- We can do this by specifying constraints for what we expect the data to be like.
- Our data is generally stored in Entities, Hibernate's Entity validation became a Java standard.

Validation

ABOUT VALIDATION

Validation

- Validation is the act of ensuring that the your data is what you expect it to be.
- Ensuring that the program operates on clean, correct and useful data*
- Important for:
 - Consistency, Reliability, Security

*http://en.wikipedia.org/wiki/Data_validation

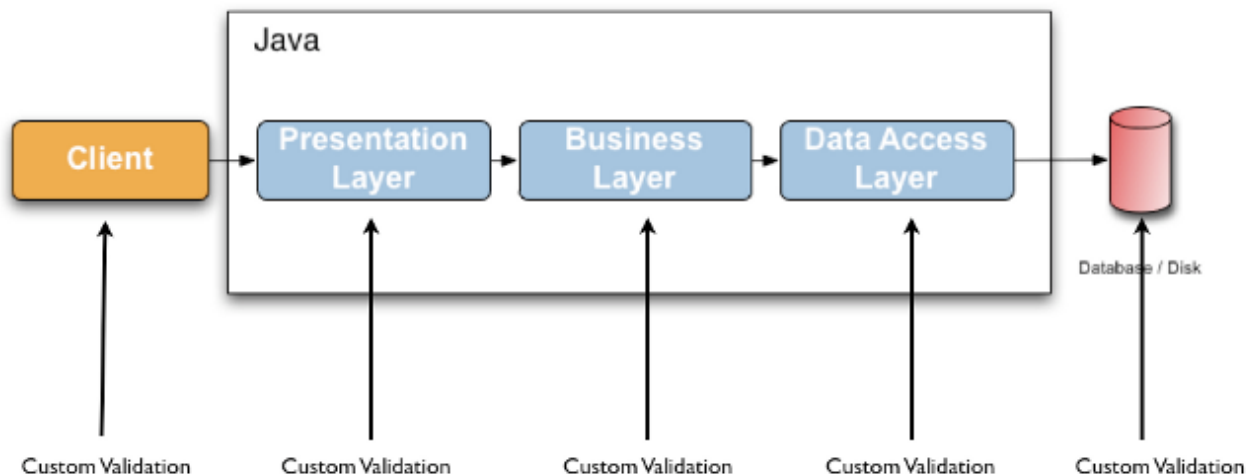
Hibernate Bean Validation*

- Validating data is a common task that occurs throughout the application layers
 - Presentation layer (user input)
 - Business layer (input from clients)
 - Data Access Layer (check before persisting)

Many of the slides are based on the hibernate validation reference documentation:
http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/

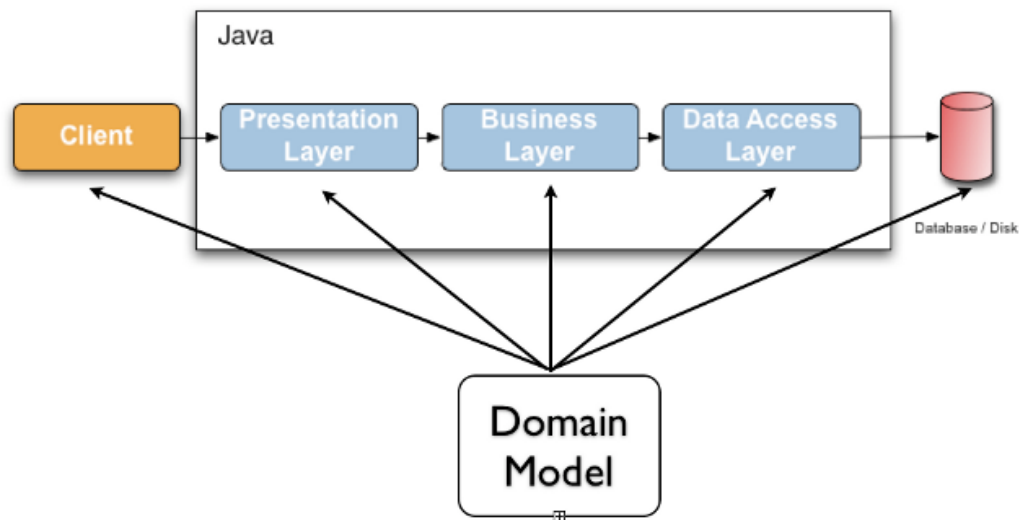
Validation Logic

- The same validation logic could be implemented in each layer
 - time consuming
 - error-prone (violates DRY)



Validation in Domain

- Validation logic can be added to the domain
 - But code would clutter domain classes
 - Such code is really just metadata about each class
 - Meta data is best expressed by annotations



Code Validation VS. Annotations

Code

```
@Entity
public class Car {

    ...

    private int year;

    public void setYear(int year) {
        if (year > 1940 && year < 2015){
            this.year = year;
        } else {
            throw new IllegalArgumentException
                ("Invalid year for a Car");
        }
    }
}
```

Annotations

```
@Entity
public class Car {

    ...

    @Range(min = 1940, max = 2015)
    private int year;
}
```


Validation

VALIDATION ANNOTATIONS

Declaring Bean Constraints

- Constraints can be declared on:
 - Fields (validator framework will use reflection)
 - Properties (Class needs to adhere to JavaBean)
 - Constraint Inheritance (super class / interface)
 - References / creating a valid Object Graph
 - Class Level Constraints (always custom)
 - Useful if for checking related properties
 - Eg. `Car.passengers <= Car.seats`

Provided Constraints 1/3

Annotation	Data Types	Description
@Null	Any	Check if it's null (affects column)
@NotNull	Any	Check that it's not null
@NotBlank	String	Not null, trimmed length > 0
@Valid	Any non-primitive	Go into the object and validate it
@AssertFalse	Boolean	Check that it's false
@AssertTrue	Boolean	Check that it's true
@Future	Date or Calendar	Check that it's in the future
@Future OrPresent	Date or Calendar	Future or Present
@Past	Date or Calendar	Check that it's in the past
@PastOrPresent	Date or Calendar	Past or Present
@Size(min=,max=)	String / Collection	Check size is >= min and <= max, column length set to max
@Pattern(regex=,flag=)	String	Check that it matches the regex

Numeric Constraints

Annotation	Data Types	Description
@Positive	Numeric types	
@PositiveOrZero	Numeric types	
@Negative	Numeric types	
@NegativeOrZero	Numeric types	
@Min(value=)	Numeric types	Check that it's not lower
@Max(value=)	Numeric types	Check that it's not higher
@DecimalMin(value=,inclusive=)	Numeric types	Check that it's not lower
@DecimalMax(value=,inclusive=)	Numeric types	Check that it's not higher
@Digits(integer=,fraction=)	Numeric types	Checks if it has less digits / fractional points then given

@Min, @Max and @Digits also affect DDL, adding constraints on the table column

@DecimalMin and @DecimalMax do not, but their min/max values can be specified As strings which allows you to check beyond Long.MAX_VALUE / Long.MIN_VALUE

Additional Constraints

Annotation	Data Types	Description
@CreditCardNumber(..)	String	Credit Cards
@EAN	String	Barcode
@Email	String	Email address
@URL(...)	String	URL
@Length(min=,max=)	String	Column length set to max
@LuhnCheck(...)	String	Checksum (mod 10) CC
@Mod10Check(...)	String	Checksum (mod 10)
@Mod11Check(...)	String	Checksum (mod 11) (also used in ISBN)
@ISBN	String	Checks if valid ISBN number
@NotEmpty	String / Collection	Not null or empty
@Range(min=,max=)	Numeric	Checks \geq min and \leq max
@SafeHtml(...)	String	Requires jsoup, checks for <script> etc
@ScriptAssert(...)	Any Type	Executes JSR 233 script against target

Fields and Properties

Fields

```
public class Car {  
  
    @NotNull  
    private String manufacturer;  
  
    @AssertTrue  
    private boolean isRegistered;  
  
    public Car(String manufacturer,  
                boolean isRegistered) {  
        this.manufacturer = manufacturer;  
        this.isRegistered = isRegistered;  
    }  
  
    //getters and setters...  
}
```

Properties

```
public class Car {  
    private String manufacturer;  
    private boolean isRegistered;  
    public Car(String manufacturer, boolean  
                isRegistered) {  
        this.manufacturer = manufacturer;  
        this.isRegistered = isRegistered;  
    }  
  
    @NotNull  
    public String getManufacturer() {  
        return manufacturer;  
    }  
    public void setManufacturer(String manufacturer) {  
        this.manufacturer = manufacturer;  
    }  
  
    @AssertTrue  
    public boolean isRegistered() {  
        return isRegistered;  
    }  
    public void setRegistered(boolean isRegistered) {  
        this.isRegistered = isRegistered;  
    }  
}
```

Containers / Collections

- Bean Validation 2.0 also adds support for:
 - container constraints
 - container cascades
- Example:

```
private Map<@Valid @NotNull OrderCategory, List<@Valid @NotNull Order>> OrderByCategory
```

Inheritance

```
public class Car {  
    private String manufacturer;  
  
    @NotNull  
    public String getManufacturer() {  
        return manufacturer;  
    }  
  
    // ...  
}
```

```
public class RentalCar extends Car {  
    private String rentalStation;  
  
    @NotNull  
    public String getRentalStation() {  
        return rentalStation;  
    }  
  
    //...  
}
```

When validating RentalCar, both
manufacturer and rentalStation
will be validated

Also works with interfaces

Object Graph

```
public class Car {  
  
    public class Car {  
  
        @NotNull  
        @Valid  
        private Person driver;  
  
        //...  
    }  
}
```

```
public class Person {  
  
    @NotNull  
    private String name;  
  
    //...  
}
```

When validating Car, @Valid makes the validator 'cascade' into Person, and check name is @NotNull

Class Level

```
@ValidPassengerCount  
public class Car {  
  
    private int seatCount;  
  
    private List<Person> passengers;  
  
    //...  
}
```

You can make custom classlevel annotations to check the relationship between properties

@ValidPassengerCount for example could check that:
passengers.size() <= seatcount

Custom Constraint Annotation

```
@Target({ TYPE, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = { ValidPassengerCountValidator.class })
@Documented
public @interface ValidPassengerCount {

    String message()
    default "{org.hibernate.validator.referenceguide.chapter06.classlevel." +
        "ValidPassengerCount.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };
}
```

Custom Validator

```
public class ValidPassengerCountValidator
    Implements ConstraintValidator<ValidPassengerCount, Car> {

    @Override
    public void initialize(ValidPassengerCount constraintAnnotation) {
    }

    @Override
    public boolean isValid(Car car, ConstraintValidatorContext context) {
        if (car == null) {
            return true;
        }
        return car.getPassengers().size() <= car.getSeatCount();
    }
}
```

Validation

PROGRAMMATIC VALIDATION

Programmatic Validation

```
public class App {  
    public static void main(String[] args) {  
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  
        Validator validator = factory.getValidator();  
  
        Car car = new Car( null, true );  
  
        Set<ConstraintViolation<Car>> constraintViolations =  
            validator.validate( car );  
  
        assertEquals( 1, constraintViolations.size() );  
        assertEquals( "may not be null",  
            constraintViolations.iterator().next().getMessage() );  
    }  
}
```

Create a validator

Car from slide 15, property constraints

Validate something

Check if it worked

Checking a Single Prop / Value

- If you don't want to validate an entire object:
 - You can validate individual values
 - You can validate individual properties
 - JavaBean property name (take off get, lowercase first)
 - These will not follow @Valid annotations

```
Set<ConstraintViolation<Car>> constraintViolations = validator.validateValue(  
    Car.class,  
    "manufacturer",  
    null );
```

Checks if manufacturer is null

```
Set<ConstraintViolation<Car>> constraintViolations = validator.validateProperty(  
    Car.class,  
    "manufacturer",  
    null );
```

Checks if manufacturer is null

Constraint Violation Methods

Method	Description	Example
getMessage()	The error message	"may not be null"
getMessageTemplate()	The name in the bundle	{...NotNull.message}
getRootBean()	Root of object graph	Car
getRootBeanClass()	Class or root bean	Car.class
getLeafBean()	'leaf' the constraint is on	Person
getPropertyPath()	From root to property	Car.Person.name
getInvalidValue()	Value failing the constraint	null
getConstraintDescriptor()	Access to annotation etc.	@NotNull

Validation

SPRING MVC INTEGRATION

web.xml

```
<web-app ... >
  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/rootconfig.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <filter>
    <filter-name>OpenSessionInView</filter-name>
    <filter-class>org.springframework.orm.hibernate4.support.OpenSessionInViewFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>OpenSessionInView</filter-name>
    <url-pattern>*</url-pattern>
  </filter-mapping>
</web-app>
```

SpringMVC Servlet

Root Config for Service & DAO

Open Session in View Filter

SpringMVC-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- find controller beans -->
    <mvc:annotation-driven />
    <context:component-scan base-package="cs544" />

    <!-- enable custom validation messages -->
    <bean id="messageSource1"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="messages" />
    </bean>

    <!--Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- Forwards requests to the "/addCar" resource to the "addCar" view -->
    <mvc:view-controller path="/index" view-name="index"/>
    <mvc:view-controller path="/login" view-name="login"/>
    <!-- Lets us find resources (.html etc) through the default servlet -->
    <mvc:default-servlet-handler/>
    <!-- Handles HTTP GET requests for /resources/** -->
    <mvc:resources mapping="/resources/**" location="/resources/" />
</beans>
```

View


```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Add a Car</title>
    <link href="resources/style.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <form:form modelAttribute="car" action="addCar" method="post">
      <form:errors path="*" cssClass="errorblock" element="div" />
      <table>
        <tr>
          <td>Make:</td>
          <td><form:input path="make" /> </td>
          <td><form:errors path="make" cssClass="error" /> </td>
        </tr>
        <!-- Model year and color removed to keep the slide shorter -->
      </table>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```

modelAttribute or
commandName
refers to the bean
from which data
will be used

Spring Form error tags
display validation errors
for their fields

path attribute specifies
what 'their' field is

Form / Validation Errors



The diagram illustrates a form validation process. It shows two browser windows side-by-side, with a large blue arrow pointing from the left window to the right window, indicating a transition from a valid state to an error state.

Left Window (Valid State):

- Browser tab: Add a Car
- Address bar: localhost:8080/validation/addCar
- Form fields:
 - Make:
 - Model:
 - Year:
 - Color:
- Submit button

Right Window (Error State):

- Browser tab: Add a Car
- Address bar: localhost:8080/validation/addCar
- Validation Error Message (highlighted in a red box):
may not be empty
may not be empty
must be between 1940 and 2015
- Form fields with error messages:
 - Make: may not be empty
 - Model: may not be empty
 - Year: must be between 1940 and 2015
 - Color:
- Submit button

Reminder: show with POST/Redirect/GET

Controller

```
@Controller
public class CarController {
    @Autowired
    private CarService carService;

    @GetMapping("/addCar")
    public String addCar(@ModelAttribute("car") Car car) {
        return "addCar";
    }

    @PostMapping("/addCar")
    public String add(@Valid Car car, BindingResult result, RedirectAttributes attr){
        if (result.hasErrors()) {
            attr.addFlashAttribute("org.springframework.validation.BindingResult.car", result);
            attr.addFlashAttribute("car", car);
            return "addCar";
        } else {
            carService.add(car);
            return "redirect:/cars";
        }
    }
}
```

The Car 'command' object always has to be available, even to just display the form

Results of validating and putting it into the Car object (binding). Always has to be specified directly after the thing to validate

Important: add package name for the bindingResult to show on the redirect page

Car Class

```
@Entity
public class Car {
    @Id
    @GeneratedValue
    private int id;

    @NotEmpty
    private String make;

    @NotEmpty
    private String model;

    @Range(min = 1940, max = 2015)
    private int year;

    private String color;
```

Spring Form Tags

Tag	Description
<form:form>	Creates an HTML form, that can also hold Spring Form Tags
<form:errors>	path attribute can specify which field, shows all without
<form:input>	Wrapper for HTML element
<form:password>	Wrapper for HTML element
<form:hidden>	Wrapper for HTML element
<form:textarea>	Wrapper for HTML element
<form:select>	Wrapper for HTML element
<form:option>	Wrapper for HTML element
<form:options>	Creates multiple option elements from a list
<form:checkbox>	Wrapper for HTML element
<form:checkboxes>	Creates multiple checkboxes from a list
<form:radiobutton>	Wrapper for HTML element
<form:radiobuttons>	Creates multiple radio buttons from a list
<form:label>	Wrapper for HTML element

Summary

- Validation allows us to ensure that our program operates on clean, correct, and useful data. *Seek the highest first*
- Using annotations we can easily declare what constraints should be applied to our data fields/references/classes.
- While there are many built in annotations, the hibernate validation framework is setup to easily allow for custom annotations.
- A ConstraintViolation object is created for every item that fails to validate.
- Based on this information we can then take the appropriate action to fix the incorrect data.
- Spring MVC will integrate with a validator implementation if it detects it on the classpath
- Spring Form error tags display any error messages related to their field

Main Point

- Validation allows us to ensure that our program operates on clean, correct, and useful data – we can do this easily using constraint annotations and integration with Spring MVC
- Science of Consciousness: Harmony Exists in Diversity – we can receive any and all data, as long as we check that it's the correct right data and send back a warning for wrong data