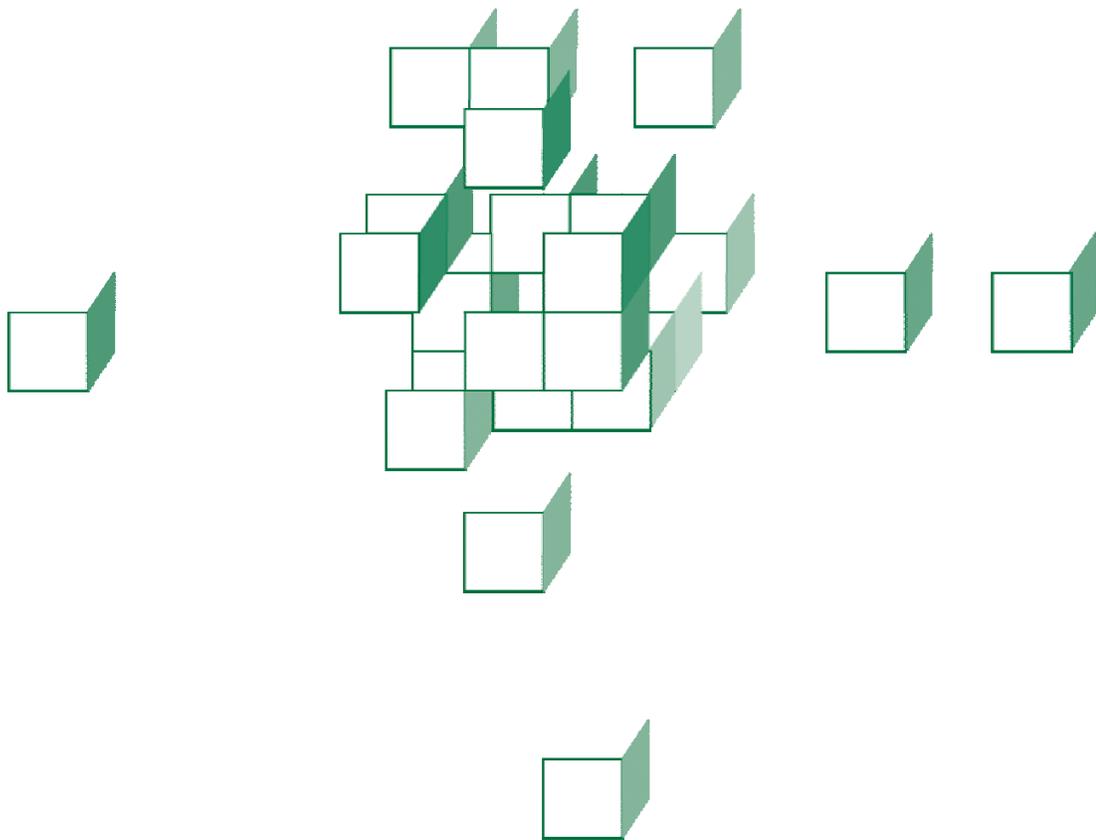


**CHAPTER 6**  
**ERROR HANDLING AND**  
**VALIDATION**



## OBJECTIVES

*After completing “Error Handling and Validation,” you will be able to:*

- **Handle error conditions during request processing using Spring-MVC exception-handling techniques:**
  - Dedicated exception classes, annotated to translate instances automatically into HTTP responses
  - Declared exception-handling methods
- **Validate request content using Java-EE Bean Validation.**
- **Report validation errors clearly using custom exception handlers.**

# Error Handling for REST Services

---

- Thus far we've taken a fairly relaxed approach to error conditions in our exercises in REST services.
  - Some services are defensive in detecting bad input or a failure to achieve the requested result, and do their own error messaging, for example in plain text.
  - Some are letting internal errors translate automatically to HTTP 500 responses – which is to say, they're not doing much at all!
- We'll start to practice more robust error handling, and return more informative responses. Some good codes to keep in mind:
  - HTTP 404 when an object is not found – just as we would get a 404 from the container if a URL were not found
  - HTTP 400 for bad inputs – a “bad request” response with some clarifying error message in the body
  - HTTP 409 for a conflict, such as when a caller tries to create a new object with an ID that already exists

# Strategies

---

- We could build error-handling into a service in a few ways.
- We might implement each method to do its own checking and handling.
  - But this would be **clumsy**, hard to **maintain**, and even gets in the way of using **@ResponseStatus** to define a static response code.
  - Will every method now need to declare **ResponseEntity<>** for some type **T** as its return type, so that we can have dynamic control over the response code?
- We could define helper methods to centralize production of error responses.
  - This might address the re-use problem, but still leaves us with awkward choices in the method signature.
- We could use Java exception handling: write methods as we currently do, and throw exceptions to express failure cases.
  - But, who catches those exceptions?
  - How are they then translated to HTTP? We may be facing an unacceptable **loss of control** over the response.

# HandlerExceptionResolver

---

- Happily, **DispatcherServlet** offers more hooks than we've seen so far, and one of those is for exception handling.
- You can declare one or more exception handlers in the context configuration, and any exceptions that fall through to the servlet will be handed over to them.
  - The interface to implement is **org.springframework.web.servlet.HandlerExceptionResolver**:

```
public interface HandlerExceptionResolver
{
    public ModelAndView resolveException
        (HttpServletRequest request,
         HttpServletResponse response,
         Object handler, Exception ex);
}
```

- Any and all beans of this type will be discovered and wired to the servlet.
  - The handler becomes essentially a fallback controller: note the similarity of the handling method to controller **handleRequest**.
- **Exception handlers are called for errors occurring during controller execution**; note that this excludes data binding, validation, and view resolution.

## @ExceptionHandler

---

- You can also identify one or more methods on your controller as **@ExceptionHandler** for that controller.

```
@ExceptionHandler  
public String handleError (Exception ex) { ... }
```

- You can be specific about the type of exception you want to handle, just by stating a specific **Exception** subclass as the type of one of your parameters.
  - Or you can use the **value** element of the annotation itself to identify the exception class your method is meant to handle.
- The method signature is similarly flexible to that of a **@RequestMapping** method.
  - **Parameter types** include **exceptions**, **HTTP request**, **response**, and **session** objects, Spring web request object, or stream objects.
  - **Return types** can be the usual means of identifying model and/or view, including **String**, **ModelAndView**, etc.
  - You can annotate for RESTful responses, too, with **@ResponseStatus**, **@ResponseBody**, etc.
- The top-level configuration option described earlier is more global, while this approach is controller-scoped.

## @ControllerAdvice for Global Error Handling

---

- Some sorts of error handling will naturally apply to more than one Spring-MVC controller.
- You can define your own **HandlerExceptionResolver**, as shown earlier in the chapter.
- A more modern approach is to annotate a class as providing **@ControllerAdvice**.
  - Such a class can define its own **@InitBinder**, **@ModelAttribute**, and **@ExceptionHandler** methods – only the last of which is really interesting to a REST web service.
  - Any such methods will be applied to request handling by many controllers, as if defined within those controller classes.
- You can apply controller advice to ...

- All controllers in the context – which is the default

**@ControllerAdvice**

- Controllers of certain classes (or subtypes of them):

**@ControllerAdvice**

**(assignableTypes={A.class,B.class})**

- Controllers that bear a specific annotation:

**@ControllerAdvice(annotations=MyQualifier.class)**

- Controllers found under a given base package:

**@ControllerAdvice(basePackage="com.me.controllers")**

## Dedicated Exception Classes

---

- You can also annotate an exception type, in order to fix an HTTP response code to it for purposes of Spring MVC.

```
@ResponseStatus  
(  
    value=HttpStatus.BAD_REQUEST,  
    reason="Missing information in request."  
)  
public static class BadRequestException { ... }
```

- If an exception of such a type is caught as part of request processing, Spring MVC's **ResponseStatusExceptionHandler** will translate this to a stock error response with the given code.
- This is a nice tool for exception classes that you define yourself, but it's of limited use, because you will usually have to contend with exceptions that you can't annotate.
  - You could define your own library of REST-specific exceptions.
  - Then you'd spend a lot of code in your request-handling methods translating a caught exception – null-pointer, illegal-argument, etc. – into one of your own.
- The **@ExceptionHandler** method is a more powerful and general-purpose tool.

- Let's look at a new version of the **DB.java** service.
  - Recall that the very first version of this service had a nice, natural mapping of returned values to response bodies as plain text.
  - Then, in order to handle error cases, we converted most methods over to using **ResponseEntity<String>** instead. This allowed us to control the response code, headers, and body dynamically; but it complicated the method signatures and implementations.
  - In **Basic\_Step3** is a version that goes back to the simpler style, and lets dedicated exception classes do most of the work when errors are encountered.
  - We'll review this version, and try it out; and then find that we can refine it a little further, leading to answer code in **Basic\_Step4**.
- See **src/cc/rest/DB.java**, and note first the new exception types:

```
public class DB
{
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public static class NotFound
        extends RuntimeException {}

    @ResponseStatus(HttpStatus.CONFLICT)
    public static class Conflict
        extends RuntimeException {}
    ...
}
```

- Then, in each request-handling method, we're back to returning the success-case value or message; and when there's trouble we just throw the appropriate exception type:

```
@RequestMapping
(value="/{key}",method=RequestMethod.GET)
public String get
(@PathVariable("key") String key)
{
    System.out.println ("GET key=" + key);
    if (!values.containsKey (key))
        throw new NotFound ();

    return values.get (key).toString ();
}
...
@RequestMapping
(value="/{key}",method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public String post
(
    @PathVariable("key") String key,
    @RequestBody String value
)
{
    System.out.println
        ("POST key=" + key + "&value=" + value);
    if (values.containsKey (key))
        throw new Conflict ();

    values.put (key, Integer.parseInt (value));
    return value;
}
...
```

1. Deploy this version of the service, and test as follows.

- You'll see the HTML error pages, as below, in the **HTML** view of the HTTPPad tool.

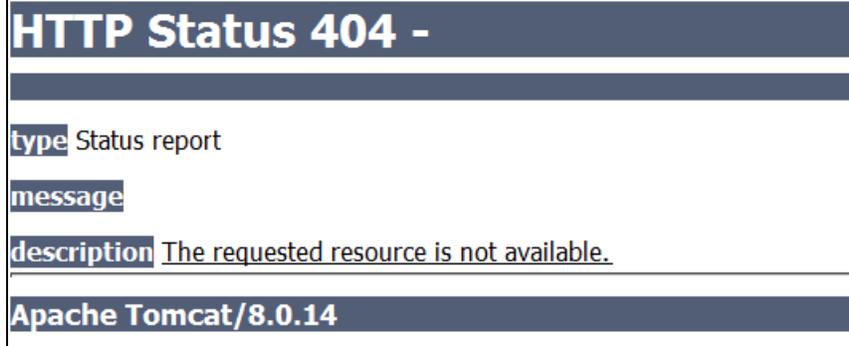
```
GET /Basic/Number/A HTTP/1.1
```

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain;charset=ISO-8859-1
```

```
1
```

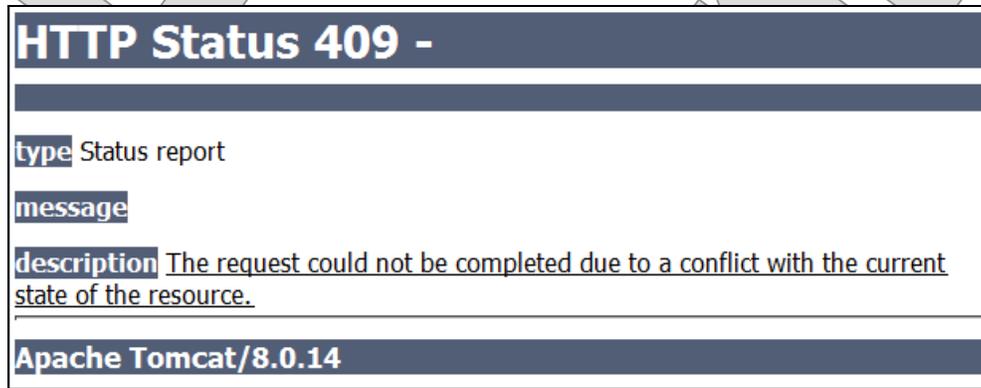
```
GET /Basic/Number/Z HTTP/1.1
```



```
POST /Basic/Number/A HTTP/1.1
```

```
Content-Type: text/plain
```

```
10
```



- So, it's perhaps a step in the right direction, but this version leaves some things to be desired.
  - We might want to stick to plain text as our response type, given that this is what the rest of the service uses.
  - We can't express an error message – just the code.
- 2. Create a new nested class **CodedException** that will encapsulate a desired HTTP response code and a string message:

```
public static class CodedException
    extends RuntimeException
    {
        public HttpStatus responseCode;
        public String message;

        public CodedException
            (HttpStatus responseCode, String message)
        {
            this.responseCode = responseCode;
            this.message = message;
        }
    }
}
```

- Now, we can keep the dedicated types, and just re-wire them as extensions of the new class. Adjust **NotFound** to take a key, and to pass the 404 error code and a synthesized error message along to the superclass constructor. You can also drop the **@ResponseStatus** annotation now:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public static class NotFound
    extends CodedException
{
    public NotFound (String key)
    {
        super (HttpStatus.NOT_FOUND,
            "No such key: " + key);
    }
}
```

- Do the same for the **Conflict** class:

```
@ResponseStatus(HttpStatus.CONFLICT)
public static class Conflict
    extends CodedException
{
    public Conflict (String key)
    {
        super (HttpStatus.CONFLICT,
            "A value is already defined for key: " +
            key);
    }
}
```

5. Now we'll need to adjust the calls to all of these constructors, because each now requires an argument where it didn't, before ...

– In **get**, **put**, and **delete**:

```
if (!values.containsKey (key))  
    throw new NotFound (key);
```

– In **post**:

```
if (values.containsKey (key))  
    throw new Conflict (key);
```

6. So far, we'd just get the same responses if we were to test again, because Spring MVC won't know what to do with the extra information in our **CodedException** type. This is a job for an exception handler!

```
@ExceptionHandler  
public ResponseEntity<String> handle  
    (CodedException ex)  
{  
    HttpHeaders headers = new HttpHeaders ();  
    headers.add  
        ("Content-Type", MediaType.TEXT_PLAIN_VALUE);  
  
    return new ResponseEntity<>  
        (ex.message, headers, ex.responseCode);  
}
```

- Spring MVC will call this method whenever it catches a **CodedException**, and it will treat the behaviors of this method just as it would do with a normal request-handling method.
- So now we can manage code, content type, and body.

7. Build and test the updated service, and see the difference in the error responses:

```
GET /Basic/Number/Z HTTP/1.1
```

```
HTTP/1.1 404 Not Found  
Content-Type: text/plain
```

```
No such key: Z
```

```
POST /Basic/Number/A HTTP/1.1  
Content-Type: text/plain
```

```
10
```

```
HTTP/1.1 409 Conflict  
Content-Type: text/plain
```

```
A value is already defined for key: A
```

8. Try one more request, and see how your service handles it:

```
PUT /Basic/Number/A HTTP/1.1  
Content-Type: text/plain
```

```
ten
```

```
HTTP Status 500 - Request processing  
failed; nested exception is  
java.lang.NumberFormatException:  
For input string: "ten"
```

9. We can do better than that ... right? Add a second exception handler to the service:

```
@ExceptionHandler
public ResponseEntity<String> handle
    (NumberFormatException ex)
{
    HttpHeaders headers = new HttpHeaders ();
    headers.add
        ("Content-Type", MediaType.TEXT_PLAIN_VALUE);

    return new ResponseEntity<>
        ("Expecting a numeric value.",
         headers, HttpStatus.BAD_REQUEST);
}
```

10. Test to see that you can handle this built-in exception about as easily as you do your own, defined exception classes:

```
PUT /Basic/Number/A HTTP/1.1
Content-Type: text/plain
```

```
ten
```

```
HTTP/1.1 400 Bad Request
Content-Type: text/plain
```

```
Expecting a numeric value.
```

**Suggested time: 30-45 minutes**

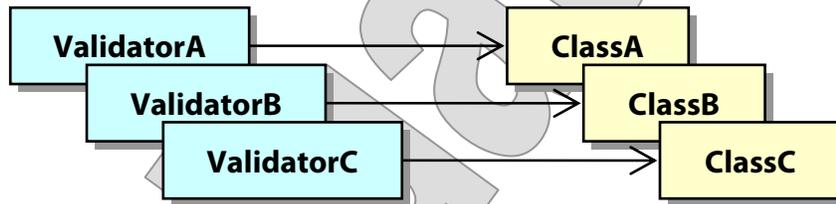
In this lab you will add error handling to the Billing service, mostly by way of generic handling in **WebService<T>**. Then you'll define and handle a specific exception type for the **InvoiceWebService**.

Detailed instructions are found at the end of the chapter.

Evaluation  
Only

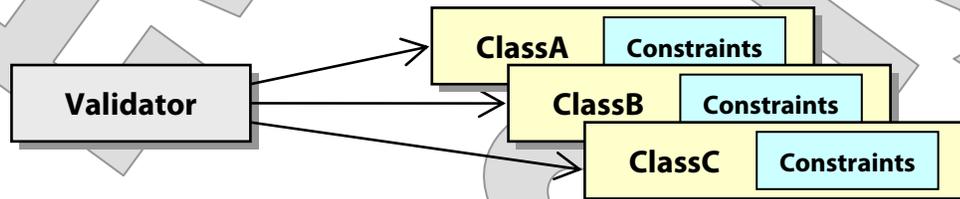
# Validation in Spring MVC

- Spring offers strong support for two means of validating inputs to an application, each based on a different theory of how and where constraints should be defined.
- Early in the evolution of object-oriented design, the idea that a class should be able to validate itself was discredited.
- For a long time Spring has espoused the idea that a separate validator class is defined and dedicated to a given target class.



– This gave rise to the **Validator** interface and to **ValidationUtils**.

- An interesting middle-ground approach has gained popularity more recently, in which, yes, a separate validator is responsible for the process, but is itself completely generic, and is informed by metadata expressed on the target class:



– This was proven in concept by the **Hibernate Validator** and is now a Java-EE standard known as **Bean Validation**.

- Spring supports both approaches, but for RESTful services, where inputs will mostly be in the form of HTTP entities, the **Bean Validation** approach makes the most sense.

# Java-EE Bean Validation

---

- A new validation standard enters the Java EE platform as of edition 6: this is known as **Bean Validation**, or sometimes by its original JSR number, 303.
- By this standard, any JavaBean can carry source-code annotations that declare validation constraints on its properties.
- These annotations can then be observed and enforced by a validation tool – at any time, in any tier of the application.
- So the advantage is that we can define validation constraints once, instead of having to write them out in different languages for different parts of a large application.
- In these exercises we're using the Hibernate Validator, version 5.1, which implements the Bean Validation 1.1 standard.

- The **BeanValidation** project holds a simple Java SE application that validates values on two different JavaBeans.
- One of these is familiar – see **src/cc/math/Ellipsoid.java**:

```
@DecimalMin
(
    value="0",
    inclusive=false,
    message="Semi-axis A must be a positive number"
)
private double a = 1;
```

- The other, in **src/cc/web/PersonalInfo.java**, sets various constraints on its properties: rejecting **null** values and enforcing a regular expression, setting a value range, etc:

```
@NotNull
@Pattern
(
    regexp="([A-Za-z\\'\\-]+)( [A-Za-z\\'\\-]+)+",
    message="Must include at least ..."
)
private String name;

@Min
(
    value=18,
    message="Age must be at least 18"
)
@Max
(
    value=120,
    message="Age must be no greater than 120"
)
private int age;
```

- An application class creates a few instances of each type and applies the Bean Validator to them; we won't dig into this code as it's not directly relevant to JSF practice, since the JSF implementation will carry out this process for us.
- **Build and test with `run` from the command line.**
  - Or, in Eclipse, just find `cc.validation.ValidateEverything`, and Run as Java Application.

```
Ellipsoid "sphere":  
  Validation succeeded.
```

```
Ellipsoid "twoD":  
  Semi-axis B must be a positive number.
```

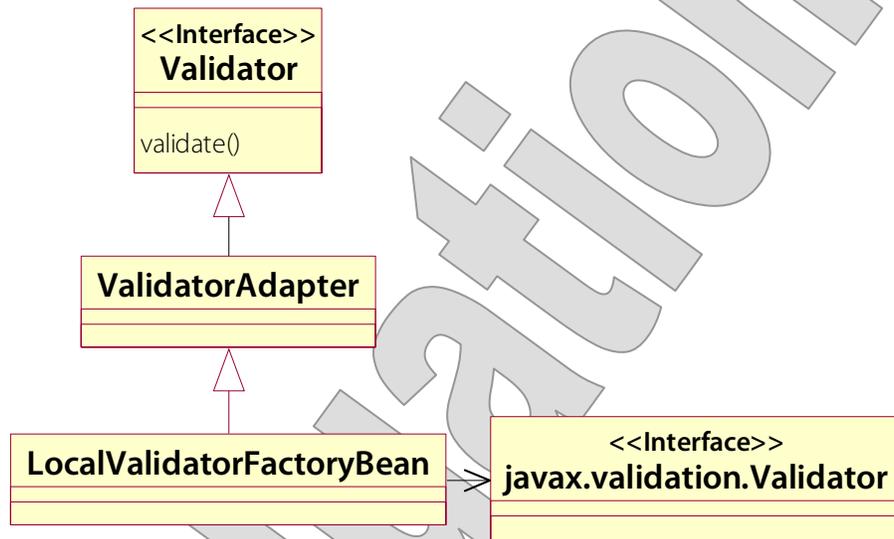
```
Ellipsoid "senseless":  
  Semi-axis C must be a positive number.
```

```
Good PersonalInfo:  
  Validation succeeded.
```

```
Bad PersonalInfo:  
  Must include at least first and last name.  
  Please keep reference to 40 characters or less.  
  Invalid e-mail address.  
  Invalid SSN.  
  Age must be at least 18.
```

# Configuration in Spring

- Spring adopts Bean Validation as an option by providing an adapter: the **LocalValidatorFactoryBean**.



- An object of this type will satisfy calling code that expects an **org.springframework.validation.Validator**, and it turns around and delegates actual validation to a Java-EE Bean Validator.
- This class is both **ApplicationContextAware** and an **InitializingBean** – so it's much easier to configure than to create programmatically.
- Then inject it into your controller, and you are ready to use it.
  - Call the validator manually.
  - Or configure it via an **@InitBinder**, calling **setValidator** on the data binder you are given.
  - A Bean Validation implementation – a/k/a a Bean Validator – must be found on the class path.

## Automatic Support with @Valid

---

- An even easier way to use Bean Validation is to get implicit support via `<mvc:annotation-driven>` configuration.
  - You can set a **global validator** explicitly on this element – and this can be any Spring validator.
  - But if you don't, you will get the **Bean Validation adapter**.
- Then, for an entity parameter annotated with **@Valid**, the Bean Validator will be invoked.
  - It will then apply whatever constraints are defined on the class to the object at hand.
  - It will report violations to an **Errors** object, which is ultimately a map with “field names” or identifiers for content parts as keys and lists of error messages as values.
  - This will be found via a **MethodArgumentNotValidException** (package **org.springframework.web.bind**) that is thrown by the framework after the binding-and-validation process completes – meaning, before the method that declared the **@Valid** constraint is invoked.
  - Note that **@Valid** is not a Spring annotation, but a Bean Validation one; find it in the package **javax.validation**.

- In **Ellipsoid\_Step4**, we've adopted the Bean-Validation constraints shown earlier for the **Ellipsoid** class.
- We are `<mvc:annotation-driven>`, and required JARs are in the compile-time and server class paths.
- So all that remains is to require that the **ellipsoid** parameter be **@Valid**.
  - This will work both for command objects (as it was originally meant to in Spring MVC) and for request entities – although the means of error reporting are very different.
  - We have just annotated the **getEllipsoid** method, at the bottom of the source file:

```
@RequestMapping(value="/Analyze",
    consumes=MediaType.APPLICATION_JSON_VALUE,
    produces=MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public Ellipsoid getEllipsoid
    (@RequestBody @Valid Ellipsoid ellipsoid)
{
    return ellipsoid;
}
```

- With just this, we would get an HTTP 400, Bad Request, if any of the input values were zero or negative.

- But we can do a bit better by declaring an exception-handling method:

```
@ExceptionHandler
public ResponseEntity<String> handleError
(MethodArgumentNotValidException ex)
{
    return new ResponseEntity<String>
        ("All semi-axis lengths must be " +
         "positive numbers.",
         HttpStatus.BAD_REQUEST);
}
```

- This will at least give a meaningful message as the body of the HTTP 406 response.
- Deploy this application and test using HTTPPad – you can see this final test request at the bottom of **HTTP/TestScript.txt**:

```
GET /Ellipsoid/REST/Analyze HTTP/1.1
Content-Type: application/json
{ "a": -1.0, "b": 3.0, "c": 3.0 }
```

- The parsed **Ellipsoid** will fail the validation constraint on **a**, throwing a **MethodArgumentNotValidException**.
- Our **handleError** method will then send the response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
```

All semi-axis lengths must be positive numbers.

- We'll add Bean Validation as a standard feature to our **WebService<T>**, and apply a few constraints to Billing domain classes to see validation in play.
- Then we'll re-factor a prepared exception handler to a new **@ControllerAdvice** class, which will be re-usable in other projects.
  - Do your work in **Billing\_Step3**.
  - The completed demo is found in **Billing\_Step4**.
- In **src/cc/rest/WebService.java**, we already have an exception handler for the **MethodArgumentNotValidException**, and so we're ready to report Bean Validation exceptions as plain-text HTTP 400 responses:

```
@ExceptionHandler
public ResponseEntity<String> onInvalidEntity
(MethodArgumentNotValidException ex)
{
    String message = String.format
        ("Validation of the request entity failed, "
         + "with the following messages:%n");
    for (FieldError error :
         ex.getBindingResult ().getFieldErrors ())
        message += String.format
            ("  %s: %s%n", error.getField (),
             error.getDefaultMessage ());

    return new ResponseEntity<String>
        (message, HttpStatus.BAD_REQUEST);
}
```

1. Deploy the service, and test with the following request, as found at the bottom of `HTTP/CustomerCRUD.txt`:

```
POST /Billing/REST/Customer/Create HTTP/1.1
Content-Type: application/json
```

```
{"firstName":"Millicent","lastName":"Mobry","address1":"993 Moodicus Parkway","address2":"","city":"Mursky","state":"MT","zip":"78311","email":"not an e-mail address"}
```

```
HTTP/1.1 201 Created
Content-Type: application/json;charset=UTF-8
```

```
{"firstName":"Millicent","lastName":"Mobry","address1":"993 Moodicus Parkway","address2":"","city":"Mursky","state":"MT","name":"Millicent Mobry","id":6,"email":"not an e-mail address","zip":"78311"}
```

- So the customer's e-mail address would be in our database in this obviously useless form.
2. Open `src/cc/billing/Customer.java` and un-comment a prepared constraint on the `eMail` field:

```
— /*
  @Pattern
  (
    regexp="\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*
    \\.(\\w{2}|(com|net|org|edu|int|mil|gov|arpa|
    biz|aero|name|coop|info|pro|museum))",
    message="Invalid e-mail address"
  )
— */
private String eMail;
```

3. In `src/cc/rest/WebService.java`, add the `@Valid` annotation to the parameter to each of the **add** methods, and to the **update** method:

```
public T add (@RequestBody @Valid T newObject)
...
public void add (@RequestParam int ID,
    @RequestBody @Valid T newObject)
    throws CRUDService.ConflictException
...
public void update
    (@RequestBody @Valid T modifiedObject)
    throws CRUDService.NotFoundException
```

4. Deploy and test, with the same request:

```
HTTP/1.1 400 Bad Request
Content-Type: text/plain;charset=ISO-8859-1
```

Validation of the request entity failed,  
with the following messages:

```
eMail: Invalid e-mail address
```

- So the Bean Validator was engaged by Spring MVC, because it saw the **@Valid** annotation on the parameter.
- When it found the e-mail value for this instance violated the **@Pattern** constraint, it threw its own exception.
- This was caught, an error message packed into an **Errors** object, and this was wrapped in a **MethodArgumentNotValidException** – which the prepared handler method caught and translated to a plain-text HTTP 400.

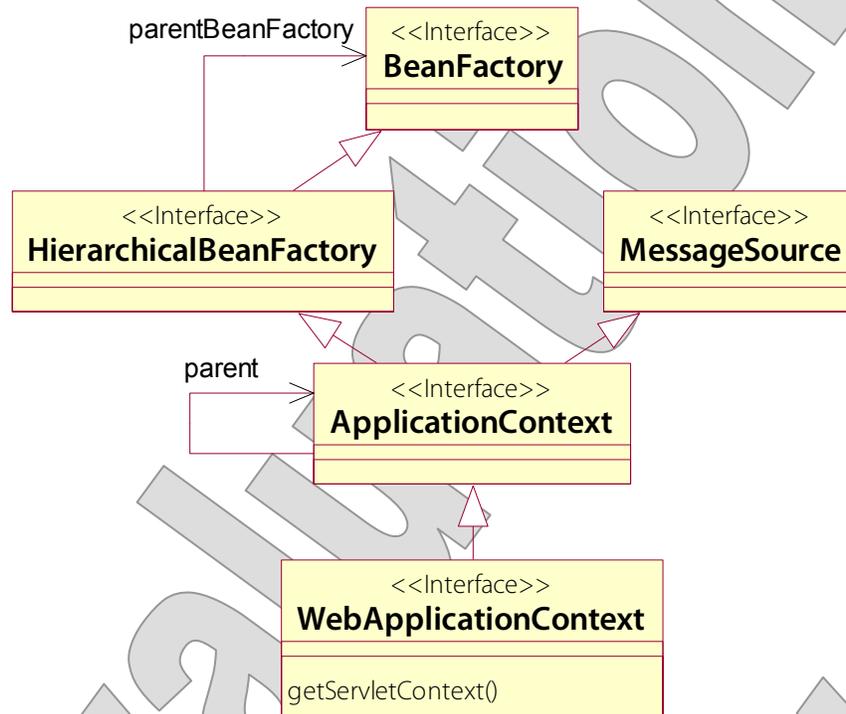
5. Create a new class `cc.rest.BeanValidationExceptionHandler`.
6. Cut the `onInvalidEntity` method from `WebService`, and paste it as the only content of the new class.
7. Annotate the class as `@ControllerAdvice`, so that the method will still be available to all controllers. So the full new source file is:

```
@ControllerAdvice
public class BeanValidationExceptionHandler
{
    @ExceptionHandler
    public ResponseEntity<String> onInvalidEntity
        (MethodArgumentNotValidException ex)
    {
        String message = String.format
            ("Validation of the request entity failed, "
            + "with the following messages:\n");
        for (FieldError error :
            ex.getBindingResult ().getFieldErrors ())
            message += String.format
                ("  %s: %s\n", error.getField (),
                error.getDefaultMessage ());
        return new ResponseEntity<String>
            (message, HttpStatus.BAD_REQUEST);
    }
}
```

8. Test again and see that the error handling is consistent.
  - And now the new class could be dropped into any other project that uses Bean Validation, and would work “out of the box.”
  - The answer code also includes a constraint annotation in `Invoice.java`, asserting that invoice amounts can’t be negative.

# Configuring Message Sources for Localization

- Recall that any **ApplicationContext** is also a **MessageSource**.



- The web application context created by the **DispatcherServlet** can be configured with a default message source – just define a bean “messageSource”:

```
<bean
  id="messageSource"
  class="org.springframework.context
    .support.ResourceBundleMessageSource"
  >
  <property name="basename" value="messages" />
</bean>
```

- You can manually load and use as many other bundles as you like, but this one becomes the default for the application context, and will serve for localizable error-code lookups.

## Resolving Error Codes

---

- The **Errors** interface offers overloads of various error-reporting methods.
- Here are the ways to report a field-level error on **Errors**:

```
public void rejectValue (String field, String code);  
public void rejectValue (String field, String code,  
    String defaultMessage);  
public void rejectValue (String field, String code,  
    Object[] args, String defaultMessage);
```

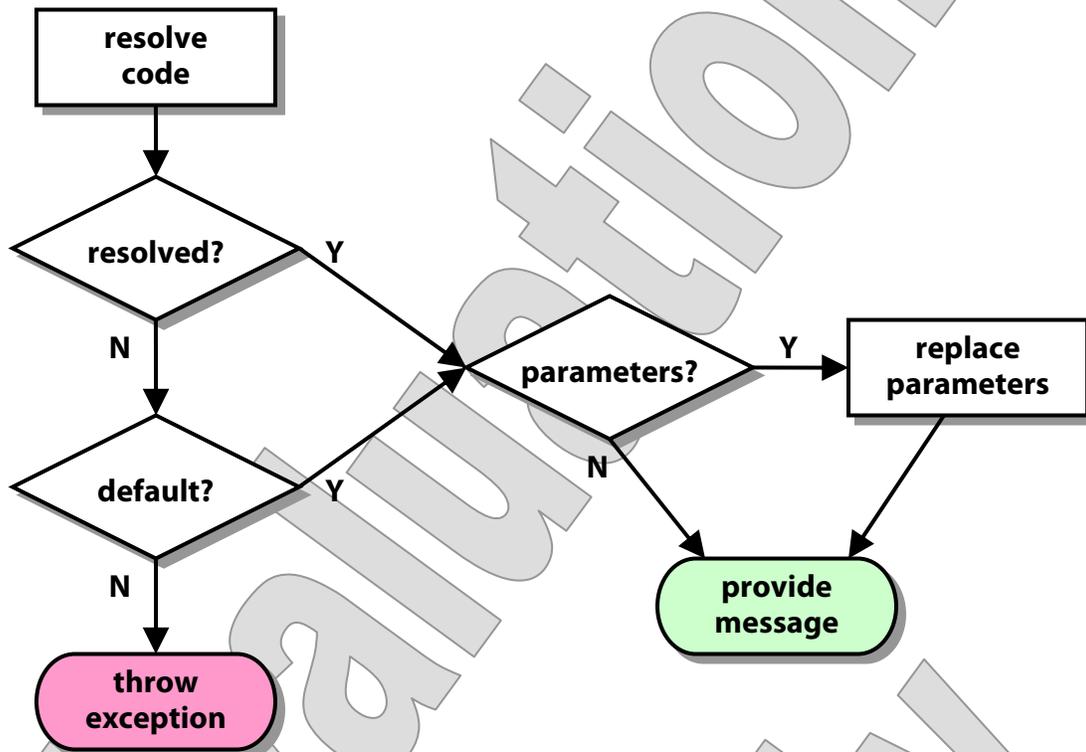
- The first option is “flying without a net,” in the sense that no default message is given by the application code. When the JSP tries to show the error message, if the code is not resolved by the applicable **MessageSource**, an exception will be thrown.
- The second option supplies a default to be used in this case.
- The third option also allows the code to provide arguments for replaceable parameters in the message, encoded as follows:

```
myMessage=Field {0} is wrong because {1}.
```

- When an object fails to pass a Bean Validation constraint, the **message** element of the constraint annotation will become the default message in the **Error** that gets reported.

## Heuristic for Resolving Message Codes

- In order to resolve error objects to localized error messages, follow a heuristic like this:



- The default message is used if the code can't be resolved.
  - If there is no default, an exception will be thrown.
  - Parameters will be replaced by provided arguments in either the resolved message or the default message.
- This is how `<form:errors>` tries to show error messages when building web pages with input forms in them, and similar code appears in producing the **message** property of various validation exception classes.
  - We'll stick to the default message in most of our lab work.

**Suggested time: 30 minutes**

In this lab you will configure a bit of validation and error reporting for the Tracking service. You'll disallow non-positive quantities for line items in the order. Then you'll enforce this constraint for both of the methods that supply new data: **order** and **addToOrder**.

Detailed instructions are found at the end of the chapter.

Evaluation  
Only

## SUMMARY

- Spring's readiness to catch exceptions as thrown from your request-handling methods, and route them to your declared exception handlers, helps to keep those primary methods clean, and focused on service logic.
- You can configure exception handlers
  - Per exception by annotating exception classes, where possible
  - Per controller with `@ExceptionHandler` methods
  - For multiple controllers via `@ControllerAdvice`
- The Spring framework has traveled a long road in its quest for the perfect validation architecture.
- Ultimately, it found the answer in a Java EE standard, Bean Validation (which in turn came out of the Hibernate community).
- Spring MVC puts Bean Validation to work in a very simple way.
- The default error handling for REST services leaves something to be desired, especially compared to the relatively rich system enjoyed by web-application developers.
- But developing a custom exception handler is not hard, and so with a bit of extra effort you can get whatever depth and format of error reporting makes sense for your service.