Java Back-End Web App Tutorial
Part 2: Adding Constraint Validation

Learn how to build a back-end web application with constraint validation, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database.

Gerd Wagner <G.Wagner@b-tu.de>
Mircea Diaconescu <M.Diaconescu@b-tu.de>
Java Back-End Web App Tutorial Part 2: Adding Constraint Validation: Learn how to build a back-end web application with constraint validation, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database
by Gerd Wagner and Mircea Diaconescu

Warning: This tutorial may still contain errors and may still be incomplete in certain respects. Please report any issue to Gerd Wagner at G.Wagner@b-tu.de or Mircea Diaconescu at M.Diaconescu@b-tu.de.

This tutorial is also available in the following formats: PDF [validation-tutorial.pdf]. See also the project page [http://web-engineering.info], or run the example app [http://yew.informatik.tu-cottbus.de/tutorials/java/validationapp/] from our server, or download it as a ZIP archive file [ValidationApp.zip].

Publication date 2015-07-24
Copyright © 2014-2015 Gerd Wagner, Mircea Diaconescu

This tutorial article, along with any associated source code, is licensed under The Code Project Open License (CPOL) [http://www.codeproject.com/info/cpol10.aspx], implying that the associated code is provided "as-is", can be modified to create derivative works, can be redistributed, and can be used in commercial applications, but the article must not be distributed or republished without the authors' consent.
Table of Contents

Foreword ................................................................................................................................... vi
1. Integrity Constraints and Data Validation ...................................................................................... 1
   1. Introduction .................................................................................................................... 1
2. Integrity Constraints .................................................................................................................... 1
   2.1. String Length Constraints ..................................................................................... 2
   2.2. Mandatory Value Constraints ................................................................................ 2
   2.3. Range Constraints ................................................................................................ 3
   2.4. Interval Constraints .............................................................................................. 4
   2.5. Pattern Constraints ............................................................................................... 4
   2.6. Cardinality Constraints ......................................................................................... 5
   2.7. Uniqueness Constraints ......................................................................................... 6
   2.8. Standard Identifiers (Primary Keys) ...................................................................... 6
   2.9. Referential Integrity Constraints ............................................................................ 7
   2.10. Frozen Value Constraints .................................................................................... 7
   2.11. Beyond property constraints ................................................................................ 7
3. Responsive Validation .................................................................................................................. 7
4. Constraint Validation in MVC Applications ................................................................................... 8
5. Criteria for Evaluating the Validation Support of Frameworks .................................................... 9
2. Implementing Constraint Validation in a Java Web App ................................................................ 10
   1. Using Java Annotations for Persistent Data Management and Constraint Validation ........... 10
      1.1. JPA database constraint annotations .................................................................... 11
      1.2. Bean validation annotations ................................................................................ 12
   2. New Issues ................................................................................................................... 13
   3. Make a JPA Entity Class Model .................................................................................... 13
   4. Style the User Interface with CSS .................................................................................. 14
   5. Write the Model Code ................................................................................................... 14
      5.1. Type mapping .................................................................................................... 14
      5.2. Encode the constraints as annotations .............................................................. 15
      5.3. Checking uniqueness constraints ......................................................................... 16
      5.4. Dealing with model related exceptions .............................................................. 16
      5.5. Requiring non-empty strings ............................................................................... 17
6. Validation in the View Layer .................................................................................................... 18
   6.1. Form validation for the Create Object use case ................................................... 18
   6.2. Form validation for the Update Object use case ................................................... 19
7. Defining a Custom Validation Annotation ..................................................................................... 20
8. Run the App and Get the Code ............................................................................................. 21
9. Possible Variations and Extensions ............................................................................................. 21
   9.1. Object-level constraint validation ........................................................................... 21
   9.2. Alternative class level custom constraint validation ............................................... 23
List of Figures

1.1. The object type Person with an interval constraint .............................................................. 4
1.2. The object type Book with a pattern constraint ..................................................................... 4
1.3. Two object types with cardinality constraints ....................................................................... 5
1.4. The object type Book with a uniqueness constraint ............................................................... 6
1.5. The object type Book with a standard identifier declaration ................................................... 6
2.1. A platform-independent design model with the class Book and two invariants ...................... 10
2.2. Deriving a JavaBean data model from an information design model ...................................... 13
List of Tables

2.1. JavaBean validation annotations for properties .............................................................. 12
2.2. Datatype mapping to Java ............................................................................................ 15
2.3. Datatype mapping to MySQL ..................................................................................... 15
Foreword

This tutorial is Part 2 of our series of six tutorials [http://web-engineering.info/JavaJpaJsfApp] about model-based development of distributed web applications with Java in combination with the Java Persistence API (JPA) and Java Server Faces (JSF) as a back-end platform. It shows how to build a simple web app with constraint validation.

A distributed web app is composed of at least two parts: a front-end part, which, at least, renders the user interface (UI) pages, and a back-end part, which, at least, takes care of persistent data storage. A back-end web app is a distributed web app where essentially all work is performed by the back-end component, including data validation and UI page creation, while the front-end only consists of a web browser's rendering of HTML-forms-based UI pages. Normally, a distributed web app can be accessed by multiple users, possibly at the same time, over HTTP connections.

In the case of a Java/JPA/JSF back-end app, the back-end part of the app can be executed by a server machine that runs a web server supporting the Java EE specifications Java Servlets, Java Expression Language (EL), JPA and JSF, such as the open source server Tomcat/TomEE [http://tomee.apache.org/apache-tomee.html].

This tutorial provides theoretically underpinned and example-based learning materials and supports learning by doing it yourself.

The minimal Java app that we have discussed in the first part of this tutorial has been limited to support the minimum functionality of a data management app only. However, it did not take care of preventing the users from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a Java model class with the help of annotations, and how to perform constraint validation both in the model part of the app and in the user interface built with JSF facelets.

The simple form of a data management application presented in this tutorial takes care of only one object type ("books") for which it supports the four standard data management operations (Create/Read/Update/Delete). It extends the minimal app discussed in the Minimal App Tutorial [minimal-tutorial.html] by adding constraint validation, but it needs to be enhanced by adding further important parts of the app's overall functionality. The other parts of the tutorial are:

- Part 1 [minimal-tutorial.html]: Building a minimal app.
- Part 3 [enumeration-tutorial.html]: Dealing with enumerations.
- Part 4 [unidirectional-association-tutorial.html]: Managing unidirectional associations between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.
- Part 5 [bidirectional-association-tutorial.html]: Managing bidirectional associations, such as the associations between books and publishers and between books and authors, not only assigning authors and a publisher to a book, but also the other way around, assigning books to authors and to publishers.
- Part 6 [subtyping-tutorial.html]: Handling subtype (inheritance) relationships between object types.

You may also want to take a look at our open access book Building Java Web Apps with JPA and JSF [http://web-engineering.info/JavaJpaJsfApp-Book], which includes all parts of the tutorial in one document, and complements them with additional material.
Chapter 1. Integrity Constraints and Data Validation

1. Introduction

For detecting non-admissible and inconsistent data and for preventing such data to be added to an application's database, we need to define suitable integrity constraints that can be used by the application's data validation mechanisms for catching these cases of flawed data. Integrity constraints are logical conditions that must be satisfied by the data in the model objects stored in the application's database. For instance, if an application is managing data about persons including their birth dates and their death dates, if they have already died, then we must make sure that for any person object with a death date, this date is not before that person object's birth date.

Since integrity maintenance is fundamental in database management, the data definition language part of the relational database language SQL supports the definition of integrity constraints in various forms. On the other hand, however, there is hardly any support for integrity constraints and data validation in common programming languages such as PHP, Java, C# or JavaScript. It is therefore important to take a systematic approach to constraint validation in web application engineering and to choose an application development framework that provides sufficient support for it.

Unfortunately, many web application development frameworks do not provide sufficient support for defining integrity constraints and performing data validation. Integrity constraints should be defined in one (central) place in an app, and then be used for validating data in other parts of the app, such as in the database and in the user interface.

HTML5 provides support for validating user input in an HTML-based user interface (UI). Here, the goal is to provide immediate feedback to the user whenever invalid data has been entered into a form field. We call this UI mechanism responsive validation.

2. Integrity Constraints

The visual language of UML class diagrams supports defining integrity constraints either with the help of special modeling elements, such as multiplicity expressions, or with the help of invariants shown in a special type of rectangle attached to the model element concerned. UML invariants can be expressed in plain English or in the Object Constraint Language (OCL). We use UML class diagrams for making design models with integrity constraints that are independent of a specific programming language or technology platform.

UML class diagrams provide special support for expressing multiplicity (or cardinality) constraints. This type of constraint allows to specify a lower multiplicity (minimum cardinality) or an upper multiplicity (maximum cardinality), or both, for a property or an association end. In UML, this takes the form of a multiplicity expression \( l . . u \) where the lower multiplicity \( l \) is a non-negative integer and the upper multiplicity \( u \) is either a positive integer or the special value *, standing for unbounded. For showing property multiplicity constrains in a class diagram, multiplicity expressions are enclosed in brackets and appended to the property name in class rectangles, as shown in the Person class rectangle in the class diagram below.

Integrity constraints may take many different forms. For instance, property constraints define conditions on the admissible property values of an object. They are defined for an object type (or class)
such that they apply to all objects of that type. We concentrate on the most important kinds of property constraints:

- **String Length Constraints** require that the length of a string value for an attribute is less than a certain maximum number, or greater than a minimum number.

- **Mandatory Value Constraints** require that a property must have a value. For instance, a person must have a name, so the name attribute must not be empty.

- **Range Constraints** require that an attribute must have a value from the value space of the type that has been defined as its range. For instance, an integer attribute must not have the value "aaa".

- **Interval Constraints** require that the value of a numeric attribute must be in a specific interval.

- **Pattern Constraints** require that a string attribute's value must satisfy a certain pattern defined by a regular expression.

- **Cardinality Constraints** apply to multi-valued properties, only, and require that the cardinality of a multi-valued property's value set is not less than a given minimum cardinality or not greater than a given maximum cardinality.

- **Uniqueness Constraints** require that a property's value is unique among all instances of the given object type.

- **Referential Integrity Constraints** require that the values of a reference property refer to an existing object in the range of the reference property.

- **Frozen Value Constraints** require that the value of a property must not be changed after it has been assigned initially.

In the following sections we discuss the different types of property constraints listed above in more detail. We also show how to express them in UML class diagrams, in SQL table creation statements and, as an example of how to do it yourself in a programming language, we also show how to express them in JavaScript model class definitions, where we encode constraint validations in class-level ("static") check functions. Any systematic approach also requires to define a set of error (or 'exception') classes, including one for each of the standard property constraints listed above.

### 2.1. String Length Constraints

The length of a string value for a property such as the title of a book may have to be constrained, typically rather by a maximum length, but possibly also by a minimum length. In an SQL table definition, a maximum string length can be specified in parenthesis appended to the SQL datatype `CHAR` or `VARCHAR`, as in `VARCHAR(50)`.

### 2.2. Mandatory Value Constraints

A mandatory value constraint requires that a property must have a value. This can be expressed in a UML class diagram with the help of a multiplicity constraint expression where the lower multiplicity is 1. For a single-valued property, this would result in the multiplicity expression `1..1`, or the simplified expression `1`, appended to the property name in brackets. For example, the following class diagram defines a mandatory value constraint for the property `name`:

```plaintext
<table>
<thead>
<tr>
<th>Class: Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
Integrity Constraints
and Data Validation

Whenever a class rectangle does not show a multiplicity expression for a property, the property is mandatory (and single-valued), that is, the multiplicity expression 1 is the default for properties.

In an SQL table creation statement, a mandatory value constraint is expressed in a table column definition by appending the key phrase NOT NULL to the column definition as in the following example:

```sql
CREATE TABLE persons(
    name  VARCHAR(30) NOT NULL,
    age   INTEGER
);
```

According to this table definition, any row of the `persons` table must have a value in the column `name`, but not necessarily in the column `age`.

In JavaScript, we can encode a mandatory value constraint by a class-level check function that tests if the provided argument evaluates to a value, as illustrated in the following example:

```javascript
Person.checkName = function (n) {
    if (n === undefined) {
        return ...; // error message "A name must be provided!"
    } else {
        ...
    }
};
```

2.3. Range Constraints

A range constraint requires that a property must have a value from the value space of the type that has been defined as its range. This is implicitly expressed by defining a type for a property as its range. For instance, the attribute `age` defined for the object type `Person` in the class diagram above has the range `Integer`, so it must not have a value like "aaa", which does not denote an integer. However, it may have values like -13 or 321, which also do not make sense as the age of a person. In a similar way, since its range is `String`, the attribute `name` may have the value "", which is a valid string that does not make sense as a name.

We can avoid allowing negative integers like -13 as age values, and the empty string as a name, by assigning more specific datatypes as range to these attributes, such as `NonNegativeInteger` to `age`, and `NonEmptyString` to `name`. Notice that such more specific datatypes are neither predefined in SQL nor in common programming languages, so we have to implement them either in the form of user-defined types, as supported in SQL-99 database management systems such as PostgreSQL, or by using suitable additional constraints such as interval constraints, which are discussed in the next section. In a UML class diagram, we can simply define `NonNegativeInteger` and `NonEmptyString` as custom datatypes and then use them in the definition of a property, as illustrated in the following diagram:

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name[1]: NonEmptyString</td>
</tr>
<tr>
<td>age[0..1]: NonNegativeInteger</td>
</tr>
</tbody>
</table>

In JavaScript, we can encode a range constraint by a check function, as illustrated in the following example:

```javascript
Person.checkName = function (n) {
    ...
};
```
Integrity Constraints
and Data Validation

```javascript
if (typeof(n) !== "string" || n.trim() === ") {
    return ...; // error message "Name must be a non-empty string!"
} else {
...
}
```

This check function detects and reports a constraint violation if the given value for the `name` property is not of type "string" or is an empty string.

### 2.4. Interval Constraints

An interval constraint requires that an attribute's value must be in a specific interval, which is specified by a minimum value or a maximum value, or both. Such a constraint can be defined for any attribute having an ordered type, but normally we define them only for numeric datatypes or calendar datatypes. For instance, we may want to define an interval constraint requiring that the `age` attribute value must be in the interval [0,120]. In a class diagram, we can define such a constraint as an "invariant" and attach it to the `Person` class rectangle, as shown in Figure 1.1 below.

**Figure 1.1. The object type Person with an interval constraint**

![Person class diagram with an invariant: `age` is between 0 and 120](image)

In an SQL table creation statement, an interval constraint is expressed in a table column definition by appending a suitable `CHECK` clause to the column definition as in the following example:

```sql
CREATE TABLE persons(
    name  VARCHAR(30) NOT NULL,
    age   INTEGER  CHECK (age >= 0 AND age <= 120)
)
```

In JavaScript, we can encode an interval constraint in the following way:

```javascript
Person.checkAge = function (a) {
    if (a < 0 || a > 120) {
        return ...; // error message "Age must be between 0 and 120!";
    } else {
        ...
    }
};
```

### 2.5. Pattern Constraints

A pattern constraint requires that a string attribute's value must match a certain pattern, typically defined by a regular expression. For instance, for the object type `Book` we define an `isbn` attribute with the datatype `String` as its range and add a pattern constraint requiring that the `isbn` attribute value must be a 10-digit string or a 9-digit string followed by "X" to the `Book` class rectangle shown in Figure 1.2 below.

**Figure 1.2. The object type Book with a pattern constraint**

![Book class diagram with an invariant: `isbn` is a 10-digit string or a 9-digit string followed by "X"](image)
Integrity Constraints and Data Validation

In an SQL table creation statement, a pattern constraint is expressed in a table column definition by appending a suitable CHECK clause to the column definition as in the following example:

```sql
CREATE TABLE books (  
    isbn   VARCHAR(10) NOT NULL CHECK (isbn ~ '^[\d]{9}(\d|X)$'),  
    title  VARCHAR(50) NOT NULL  
) ;
```

The ~ (tilde) symbol denotes the regular expression matching predicate and the regular expression `^[\d]{9}(\d|X)$` follows the syntax of the POSIX standard (see, e.g. the PostgreSQL documentation [http://www.postgresql.org/docs/9.0/static/functions-matching.html]).

In JavaScript, we can encode a pattern constraint by using the built-in regular expression function `test`, as illustrated in the following example:

```javascript
Person.checkIsbn = function (id) {
    if (!/\b\d{9}(\d|X)\b/.test(id)) {
        return ...; // error message like "The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!"
    } else {
        ...
    }
};
```

### 2.6. Cardinality Constraints

A cardinality constraint requires that the cardinality of a multi-valued property’s value set is not less than a given **minimum cardinality** or not greater than a given **maximum cardinality**. In UML, cardinality constraints are called **multiplicity constraints**, and minimum and maximum cardinalities are expressed with the lower bound and the upper bound of the multiplicity expression, as shown in Figure 1.3 below, which contains two examples of properties with cardinality constraints.

![Figure 1.3. Two object types with cardinality constraints](image)

The attribute definition `nickNames[0..3]` in the class `Person` specifies a minimum cardinality of 0 and a maximum cardinality of 3, with the meaning that a person may have no nickname or at most 3 nicknames. The reference property definition `members[3..5]` in the class `Team` specifies a minimum cardinality of 3 and a maximum cardinality of 5, with the meaning that a team must have at least 3 and at most 5 members.

It's not obvious how cardinality constraints could be checked in an SQL database, as there is no explicit concept of cardinality constraints in SQL, and the generic form of constraint expressions in SQL, assertions, are not supported by available DBMSs. However, it seems that the best way to implement a minimum (resp. maximum) cardinality constraint is an on-delete (resp. on-insert) trigger that tests the number of rows with the same reference as the deleted (resp. inserted) row.

In JavaScript, we can encode a cardinality constraint validation for a multi-valued property by testing the size of the property's value set, as illustrated in the following example:

```javascript
Person.checkNickNames = function (nickNames) {
    if (nickNames.length > 3) {
        return ...; // error message like "There must be no more than 3 nicknames!"
    } else {
        ...
    }
};
```
With Bean Validation annotations, we can specify `@Size(max=3) List<String> nickNames` or `@Size(min=3, max=5) List<Person> members.`

### 2.7. Uniqueness Constraints

A uniqueness constraint requires that a property's value is unique among all instances of the given object type. For instance, for the object type `Book` we can define the `isbn` attribute to be unique in a UML class diagram by appending the keyword `unique` in curly braces to the attribute's definition in the `Book` class rectangle shown in Figure 1.4 below.

**Figure 1.4. The object type Book with a uniqueness constraint**

```
Book
isbn : String {unique}
title : String
```

In an SQL table creation statement, a uniqueness constraint is expressed by appending the keyword `UNIQUE` to the column definition as in the following example:

```
CREATE TABLE books(
    isbn   VARCHAR(10) NOT NULL UNIQUE,
    title  VARCHAR(50) NOT NULL
)
```

In JavaScript, we can encode this uniqueness constraint by a check function that tests if there is already a book with the given `isbn` value in the `books` table of the app's database.

### 2.8. Standard Identifiers (Primary Keys)

An attribute (or, more generally, a combination of attributes) can be declared to be the standard identifier for objects of a given type, if it is mandatory and unique. We can indicate this in a UML class diagram with the help of a (user-defined) stereotype «stdid» assigned to the attribute `isbn` as shown in Figure 1.5 below.

**Figure 1.5. The object type Book with a standard identifier declaration**

```
Book
<stdid> isbn : String
	title : String
```

Notice that a standard identifier declaration implies both a mandatory value and a uniqueness constraint on the attribute concerned.

Standard identifiers are called `primary keys` in relational databases. We can declare an attribute to be the primary key in an SQL table creation statement by appending the phrase `PRIMARY KEY` to the column definition as in the following example:

```
CREATE TABLE books(
    isbn   VARCHAR(10) PRIMARY KEY,
    title  VARCHAR(50) NOT NULL
)
```

In JavaScript, we cannot easily encode a standard identifier declaration, because this would have to be part of the metadata of the class definition, and there is no standard support for such metadata in
Integrity Constraints
and Data Validation

JavaScript. However, we should at least check if the given argument violates the implied mandatory value or uniqueness constraints by invoking the corresponding check functions discussed above.

2.9. Referential Integrity Constraints

A referential integrity constraint requires that the values of a reference property refer to an existing object in the range of the reference property. Since we do not deal with reference properties in this part of the tutorial, we postpone the discussion of referential integrity constraints to the next part of our tutorial.

2.10. Frozen Value Constraints

A frozen value constraint defined for a property requires that the value of this property must not be changed after it has been assigned initially. This includes the special case of a read-only value constraint applying to mandatory properties that are initialized at object creation time. Typical examples of properties with a frozen value constraint are event properties, since the semantic principle that the past cannot be changed prohibits that the property values of past events can be changed.

In Java, a frozen value constraint can be enforced by declaring the property to be final. However, while in Java a final property must be mandatory, a frozen value constraint may also apply to an optional property.

In JavaScript, a read-only property can be defined with

```javascript
Object.defineProperty( obj, "teamSize", {value: 5, writable: false, enumerable: true})
```

by making it unwritable, while an entire object `o` can be frozen by stating `Object.freeze( o)`.

We postpone the further discussion of frozen value constraints to Part 5 of our tutorial.

2.11. Beyond property constraints

So far, we have only discussed how to define and check property constraints. However, in certain cases there may be also integrity constraints that do not just depend on the value of a particular property, but rather on

1. the values of several properties of a particular object
2. both the values of a property before and after a change attempt
3. the set of all instances of a particular object type
4. the set of all instances of several object types

We plan to discuss these more advanced types of integrity constraints in a forthcoming book on web application engineering.

3. Responsive Validation

This problem is well-known from classical web applications where the front-end component submits the user input data via HTML form submission to a back-end component running on a remote web server. Only this back-end component validates the data and returns the validation results in the form of a set of error messages to the front end. Only then, often several seconds later, and in the hard-to-digest form of a bulk message, does the user get the validation feedback. This approach is no longer considered
Integrity Constraints and Data Validation

acceptable today. Rather, in a responsive validation approach, the user should get immediate validation feedback on each single data input, typically in response to input or change events.

So, we need a data validation mechanism in the user interface (UI). Fortunately, the new HTML5 form validation API [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/] supports constraint validation in the UI. Alternatively, the jQuery Validation Plugin [http://jqueryvalidation.org/] can be used as a (non-HTML5-based) form validation API.

The HTML5 form validation API essentially provides new input type values and a set of new attributes for form control elements for the purpose of supporting responsive validation directly performed by the browser. Since using the new validation attributes (like required, min, max and pattern) implies defining constraints in the UI, they are not really useful in a general approach where constraints are only checked, but not defined, in the UI.

Consequently, we only use two methods of the HTML5 form validation API for validating constraints in the HTML-forms-based user interface of our app. The first of them, setCustomValidity, allows to mark a form input field as either valid or invalid by assigning either an empty string or a non-empty message string to it. The second method, checkValidity, is invoked on a form and tests, if all input fields have a valid value.

See this Mozilla tutorial [https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation] or this HTML5Rocks tutorial [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/] for more about the HTML5 form validation API.

4. Constraint Validation in MVC Applications

Integrity constraints should be defined in the model classes of an MVC app since they are part of the business semantics of a model class (representing a business object type). However, a more difficult question is where to perform data validation? In the database? In the model classes? In the controller? Or in the user interface ("view")? Or in all of them?

A relational database management system (DBMS) performs data validation whenever there is an attempt to change data in the database, provided that all relevant integrity constraints have been defined in the database. This is essential since we want to avoid, under all circumstances, that invalid data enters the database. However, it requires that we somehow duplicate the code of each integrity constraint, because we want to have it also in the model class to which the constraint belongs.

Also, if the DBMS would be the only application component that validates the data, this would create a latency, and hence usability, problem in distributed applications because the user would not get immediate feedback on invalid input data. Consequently, data validation needs to start in the user interface (UI).

However, it is not sufficient to perform data validation in the UI. We also need to do it in the model classes, and in the database, for making sure that no flawed data enters the application's persistent data store. This creates the problem of how to maintain the constraint definitions in one place (the model), but use them in two or three other places (at least in the model classes and in the UI code, and possibly also in the database). We call this the multiple validation problem. This problem can be solved in different ways. For instance:

1. Define the constraints in a declarative language (such as Java Persistency and Bean Validation Annotations or ASP.NET Data Annotations) and generate the back-end/model and front-end/UI
validation code both in a back-end application programming language such as Java or C#, and in JavaScript.

2. Keep your validation functions in the (PHP, Java, C# etc.) model classes on the back-end, and invoke them from the JavaScript UI code via XHR. This approach can only be used for specific validations, since it implies the penalty of an additional HTTP communication latency for each validation invoked in this way.

3. Use JavaScript as your back-end application programming language (such as with NodeJS), then you can encode your validation functions in your JavaScript model classes on the back-end and execute them both before committing changes on the back-end and on user input and form submission in the UI on the front-end side.

The simplest, and most responsive, solution is the third one, using only JavaScript both in the back-end and front-end components.

5. Criteria for Evaluating the Validation Support of Frameworks

The support of MVC frameworks for constraint validation can be evaluated according to the following criteria. Does the framework support

1. the declaration of all important kinds of property constraints as defined above (String Length Constraints, Mandatory Value Constraints, Range Constraints, etc.) in model classes?

2. the provision of an object validation function to be invoked in the UI on form submission, and in the model layer before save?

3. validation in the model class on assign property and before save object?

4. informative generic validation error messages referring to the object and property concerned?

5. custom validation error messages with parameters?

6. responsive validation in the user interface on input and on submit based on the constraints defined in the model classes, preferably using the HTML5 constraint validation API, or at least a general front-end API like the jQuery Validation Plugin?

7. two-fold validation with defining constraints in the model and checking them in the model and in the view?

8. three-fold validation with defining constraints in the model and checking them in the model, the view and in the database system?

9. reporting database validation errors that are passed from the database system to the app?
Chapter 2. Implementing Constraint Validation in a Java Web App

The minimal web app that we have discussed in Part 1 has been limited to support the minimum functionality of a data management app only. For instance, it did not take care of preventing the user from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a Java model class, and how to have constraints validated transparently on critical lifecycle events of bean objects (such as persist) with JPA, and on form submission with JSF.

We again consider the single-class data management problem that was considered in Part 1 of this tutorial. So, again, the purpose of our app is to manage information about books. But now we also consider the data integrity rules (also called 'business rules') that govern the management of book data. These integrity rules, or constraints, can be expressed in a UML class diagram as shown in Figure 2.1 below.

Figure 2.1. A platform-independent design model with the class Book and two invariants

In this simple model, the following constraints have been expressed:

1. Due to the fact that the isbn attribute is declared to be the standard identifier of Book, it is mandatory and unique.

2. The isbn attribute has a pattern constraint requiring its values to match the ISBN-10 format that admits only 10-digit strings or 9-digit strings followed by "X".

3. The title attribute is mandatory.

4. The year attribute is mandatory and has an interval constraint, however, of a special form since the maximum is not fixed, but provided by the calendaric function nextYear() .

In addition to these constraints, there are the implicit range constraints defined by assigning the datatype NonEmptyString as range to isbn and title, and Integer to year.

1. Using Java Annotations for Persistent Data Management and Constraint Validation

The integrity constraints of a distributed app have to be checked both in model classes and in the underlying database, and possibly also in the UI. However this requirement for three-fold validation should not imply having to define the same constraints three times in three different languages: in Java,
Implementing Constraint Validation in a Java Web App

in SQL and in HTML5/JavaScript. Rather, the preferred approach is to define the constraints only once, in the model classes, and then reuse these constraint definitions also in the underlying database and in the UI. Java/JPA/JSF apps support this goal to some degree. There are two types of constraint annotations:

1. **Database constraint annotations**, which are part of JPA, specify constraints to be used for generating the database schema (with CREATE TABLE statements) such that they are then checked by the DBMS, and not by the Java runtime environment;

2. **Bean validation annotations** specify constraints to be checked by the Java runtime environment.

In this section we discuss how to use some of the predefined constraint annotations and how to define a custom constraint annotation for the \texttt{year} property of the \texttt{Book} class, since its value has an upper bound defined by an expression (‘next year’).

### 1.1. JPA database constraint annotations

The JPA database constraint annotations specify constraints to be used by the underlying database system after generating the database schema, but not for Java validation. Consider the \@Id annotation in the following definition of an entity class \texttt{Item}:

```java
@Entity @Table( name="items")
public class Item {
    @Id @Column( length=10)
    private String itemCode;
    private int quantity;
    // define constructors, setters and getters
}
```

The \@Id annotation of the \texttt{itemCode} attribute is mapped to a SQL primary key declaration for this attribute in the corresponding database table schema. As a consequence, the values of the \texttt{itemCode} column allowed in the generated \texttt{items} table are mandatory and have to be unique. However, these conditions are not checked in the Java runtime environment. JPA generates the following CREATE TABLE statement:

```sql
CREATE TABLE IF NOT EXISTS items (
    itemCode varchar(255) NOT NULL PRIMARY KEY,
    quantity int(11) DEFAULT NULL
)
```

Since nothing has been specified about the length of \texttt{itemCode} strings, the length is set to 255 by default. However, in our case we know that \texttt{itemCode} has a fixed length of 10, which can be enforced by using the \@Column annotation, which has the following parameters:

- **name** allows to specify a name for the column to be used when the table is created (by default, the attribute name of the entity class is used);

- **nullable** is a boolean parameter that defines if the column allows NULL values (by default, it is true);

- **length** is a positive integer, specifying the maximum number of characters allowed for string values of that column (by default, this is 255);

- **unique** is a boolean parameter that defines if the values of the column must be unique (by default, it is false).

Using the \@Column annotation, the improved Java/JPA code of the model class is:

```java
@Entity @Table( name="items")
public class Item {
    @Id @Column( length=10)
    private String itemCode;
    private int quantity;
    // define constructors, setters and getters
}
```
private String itemCode;
@Column( nullable=false)
private int quantity;
...
// define constructors, setters and getters
}

As a result, the generated CREATE TABLE statement now contains the additional constraints expressed for the columns itemCode and quantity:

```sql
CREATE TABLE IF NOT EXISTS items (
    itemCode varchar(10) NOT NULL PRIMARY KEY,
    quantity int(11) NOT NULL
)
```

### 1.2. Bean validation annotations

In Java's Bean Validation [http://docs.oracle.com/javaee/7/tutorial/doc/bean-validation001.htm#GIRCZ] approach, Java runtime validation can be defined in the form of bean validation annotations placed on a property, method, or class.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Annotations</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>String Length Constraints</td>
<td>@Size</td>
<td>@Size( min=8, max=80) String message;</td>
</tr>
<tr>
<td>Cardinality Constraints (for @Size arrays, collections and maps)</td>
<td>@Size( min=2, max=3) List&lt;Member&gt; coChairs;</td>
<td></td>
</tr>
<tr>
<td>Mandatory Value Constraints</td>
<td>@NotNull</td>
<td>@NotNull String name</td>
</tr>
<tr>
<td>Range Constraints for numeric @Digits attributes</td>
<td>@Digits( integer=6, fraction=2) BigDecimal price;</td>
<td></td>
</tr>
<tr>
<td>Interval Constraints for integer- @Min and @Max valued attributes</td>
<td>@Min and @Max</td>
<td>@Min(5) int teamSize</td>
</tr>
<tr>
<td>Interval Constraints for decimal- @DecimalMin and @DecimalMax valued attributes</td>
<td>@DecimalMin and @DecimalMax(&quot;30.00&quot;) double voltage</td>
<td></td>
</tr>
<tr>
<td>Pattern Constraints</td>
<td>@Pattern</td>
<td>@Pattern( regexp=&quot;\b\d{10}\b&quot;) String isbn;</td>
</tr>
</tbody>
</table>

In addition, there are annotations that require a date value to be in the future (@Future) or in the past (@Past).

All bean validation annotations have an optional message attribute for defining a specific error message. In the following example, we add two @NotNull annotations with messages, a @Size and a @Min annotation to the database constraint annotations. The @NotNull annotations constrain the itemCode and the quantity attributes to be mandatory, while the @Min annotation constrains the quantity attribute to have a minimum value of 0:

```java
@Entity
@Table( name="items")
public class Item {
    @Id @Column( length=8)
    @NotNull( message="An item code is required!")
    @Size( min=8, max=8)
    private String itemCode;
    @Column( nullable=false)
```
2. New Issues

Compared to the minimal app discussed in Part 1 (Minimal App Tutorial [minimal-tutorial.html]) we have to deal with a number of new issues:

1. In the **model** layer we have to take care of adding for every property the constraints that must be checked before allowing a record to be saved to the database

2. In the **user interface (view)** we have to take care of form validation providing feedback to the user whenever data entered in the form is not valid.

Checking the constraints in the user interface on user input is important for providing immediate feedback to the user. Using JSF and Bean Validation requires to submit the form before the validation checks are performed. It would be preferable to define the validation checks in the model classes only and use them in the user interface before form submission, without having to duplicate the validation logic in the JSF facellets. However, at this point in time, JSF does not support this, and the validation is performed only after the form is submitted.

Using HTML5 validation attributes in the JSF facellets to enforce HTML5 validation before submitting the form requires an undesirable duplication of validation logic. The effect of such a duplication would be duplicate maintenance of the validation code, once in the model classes and once more in the user interface. In a simple application like our example app, this is doable, but in a larger application this quickly becomes a synchronization nightmare.

3. Make a JPA Entity Class Model

Using the information design model shown in Figure 2.1 above as the starting point, we make a **JavaBean data model** with getters/setters and corresponding Java datatypes.

Figure 2.2. Deriving a JavaBean data model from an information design model

The data model defines the following constraints:

1. The *isbn* attribute is declared to be a standard identifier, implying that it is **mandatory** and **unique**.
2. The isbn attribute has a **pattern constraint** requiring its values to match the ISBN-10 format that admits only 10-digit strings or 9-digit strings followed by "X".

3. The title attribute is **mandatory**.

4. The year attribute is **mandatory** and has an **interval constraint**, however, of a special form since the maximum is not fixed, but provided by the calendaric function `nextYear()`.

### 4. Style the User Interface with CSS

We style the UI with the help of the CSS library Pure [http://purecss.io/](http://purecss.io/) provided by Yahoo. We only use Pure's basic styles, which include the browser style normalization of normalize.css [http://necolas.github.io/normalize.css/], and its styles for forms. In addition, we define our own style rules, as for in the case of table element and for error messages in style.css.

Adding the Yahoo CSS library to our HTML5 requires to edit the WebContent/WEB-INF/templates/page.xhtml file and add the style element referencing the online CSS file:

```xml
<h:head>
    <title><ui:insert name="title">Public Library</ui:insert></title>
    <link rel="stylesheet" href="http://yui.yahooapis.com/pure/0.5.0/pure-min.css" />
</h:head>
```

Following the same procedure, our custom CSS file is added:

```xml
<h:head>
    <title><ui:insert name="title">Public Library</ui:insert></title>
    <link rel="stylesheet" href="http://yui.yahooapis.com/pure/0.5.0/pure-min.css" />
    <link href="#{facesContext.externalContext.requestContextPath}/resources/css/style.css" rel="stylesheet" type="text/css" />
</h:head>
```

The `facesContext.externalContext.requestContextPath` expression allows, via JSF, to get the system and file system independent path to WebContent folder. Our CSS file is named `style.css` and is located under WebContent/resources/css folder.

Notice how HTML5 code (e.g., the link element) are used on the same HTML source file, near the JSF specific elements (e.g., `h:head`).

Specific CSS classes can be applied to JSF view elements (e.g., `h:form`), by using the `@styleClass` attribute. For example, using `styleClass="pure-form pure-form-aligned"` the YUI pure-form and pure-form-aligned CSS classes are applied to the resulting HTML form element:

```xml
<h:form id="createBookForm" styleClass="pure-form pure-form-aligned">
    ...
</h:form>
```

### 5. Write the Model Code

The JavaBeans data model shown on the right hand side in Figure 2.2 can be encoded step by step for getting the code of the model classes of our Java web app.

#### 5.1. Type mapping

When defining the properties, we first need to map the platform-independent data types of the information design model to the corresponding implicit Java supported data types according to the following table.
Implementing Constraint Validation in a Java Web App

### Table 2.2. Datatype mapping to Java

<table>
<thead>
<tr>
<th>Platform-independent datatype</th>
<th>Java datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Integer</td>
<td>int, long, Integer, Long</td>
</tr>
<tr>
<td>Decimal</td>
<td>double, Double, Java.math.BigDecimal</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean, Boolean</td>
</tr>
<tr>
<td>Date</td>
<td>java.util.Date</td>
</tr>
</tbody>
</table>

Notice that for precise computations with decimal numbers, the special datatype `java.math.BigDecimal` [http://www.opentaps.org/docs/index.php/How_to_Use_Java_BigDecimal:_A_Tutorial] is needed.

A second datatype mapping is needed for obtaining the corresponding MySQL data types:

### Table 2.3. Datatype mapping to MySQL

<table>
<thead>
<tr>
<th>Platform-independent datatype</th>
<th>MySQL datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Integer</td>
<td>INT</td>
</tr>
<tr>
<td>Decimal</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Boolean</td>
<td>BOOL</td>
</tr>
<tr>
<td>Date</td>
<td>DATETIME or TIMESTAMP</td>
</tr>
</tbody>
</table>

### 5.2. Encode the constraints as annotations

In this section we add database constraint annotations and bean validation annotations for implementing the property constraints defined for the `Book` class in the JavaBean data model. For the standard identifier attribute `isbn`, we add the database constraint annotations `@Id` and `@Column( length=10)`, as well as the bean validation annotations `@NotNull` and `@Pattern( regexp="\b\d{10}\b")`. Notice that, for readability, we have simplified the ISBN pattern constraint.

For the attribute `title`, we add the database constraint annotation `@Column( nullable=false)`, as well as the bean validation annotations `@NotNull` and `@Size( max=255)`. For the attribute `year`, we add the database constraint annotation `@Column( nullable=false)`, as well as the bean validation annotations `@NotNull` and `@Min( value=1459)`. Notice that we cannot express the constraint that `year` must not be greater than next year with a standard validation annotation. Therefore, we'll define a custom annotation for this constraint in Section 7 below.

Encoding the integrity constraints with database constraint annotations and bean validation annotations results in the following annotated bean class:

```java
@Entity
@Table( name="books")
@ManagedBean( name="book")
@ViewScoped
public class Book {
    @Id @Column( length=10)
    @NotNull( message="An ISBN value is required!"")
    @Pattern( regexp="\b\d{10}\b", message="The ISBN must be a 10-digit string!")
    private String isbn;
```
Implementing Constraint Validation in a Java Web App

private String title;
@Column( nullable=false)
@NotNull( message="A title is required!")
@Size( max=255)
private Integer year;
... // define constructors, setters and getters

Notice that for the year property, the Java Integer wrapper class is used instead of the primitive int data type. This is required for the combined use of JSF and JPA, because if the value of an empty year input field is submitted in the create or update forms, the value which is passed to the year property by JSF via the setYear method is null (more details on Section 5.5, “Requiring non-empty strings”), which is not admitted for primitive datatypes by Java.

We only provide an overview of the methods. For more details, see our minimal app tutorial [minimal-tutorial.html].

5.3. Checking uniqueness constraints

For avoiding duplicate Book records we have to check that the isbn values are unique. At the level of the database, this is already checked since the isbn column is the primary key, and the DBMS makes sure that its values are unique. However, we would like to perform this check in our Java app before the data is passed to the DBMS and create suitable error messages. Unfortunately, there is no predefined bean validation annotation for this purpose, and it is not clear how to do this with a custom validation annotation. Therefore we need to write a static method, Book.checkIsbnAsId, for checking if a value for the isbn attribute is unique. This check method can then be called by the controller for validating any isbn attribute value before trying to create a new Book record. The Book.checkIsbnAsId method code is shown below:

```java
public static void checkIsbnAsId( EntityManager em, String isbn) throws UniquenessConstraintViolation {
    Book book = Book.getObjectByStdId( em, isbn);
    if ( book != null) {  // book was found, so isbn is not unique
        throw new UniquenessConstraintViolation("There is already a book record with this ISBN!");
    }
}
```

The method throws a UniquenessConstraintViolation exception in case that a Book record was found for the given ISBN value. The exception can then be caught and a corresponding error message displayed in the UI. In the sequel of this tutorial we show how to define the controller validation method and inform JSF facelets that it must be used to validate the isbn form input field.

5.4. Dealing with model related exceptions

The Book.checkIsbnAsId method from the above section is designed to be used in combination with a controller so the user gets an error message when trying to duplicate a Book entry (i.e. the provided isbn value already exists). However, if the Book.add method is used directly (i.e. by another piece of code, where the uniqueness constraint is not performed by calling
Implementing Constraint Validation in a Java Web App

Book.checkIsbnAsId), then uniqueness constraint validation may fail. Let's have a look on the Book.add code:

```java
public static void add(EntityManager em, UserTransaction ut,
    String isbn, String title, int year) throws NotSupportedException,
    SystemException, IllegalStateException, SecurityException,
    HeuristicMixedException, HeuristicRollbackException, RollbackException,
    EntityExistsException {
    ut.begin();
    Book book = new Book(isbn, title, year);
    em.persist(book);
    ut.commit();
}
```

The method throws a set of exceptions to reflect problems occurred when trying to run the persist or the commit method. One of the exceptions (i.e. EntityExistsException) is thrown by the ut.commit call. The method which calls Book.add may catch this exception and perform specific actions, such as rolling back the transaction. In our case, the Book.add is called by the add action method of the BookController class, and the action performed is to show the exception traceback in the console, as well as calling the ut.rollback which takes care of cancelling any database change performed by the current transaction. The rest of the exceptions are caught by using their super class (i.e. Exception) and the exception traceback is displayed in the console.

```java
public String add(String isbn, String title, int year) {
    try {
        Book.add(em, ut, isbn, title, year);
    } catch(EntityExistsException e) {
        try {
            ut.rollback();
        } catch(Exception e1) {
            e1.printStackTrace();
        }
        e.printStackTrace();
    } catch(Exception e) {
        e.printStackTrace();
    }
    return "create";
}
```

Note: the EntityExistsException is part of the javax.persistence package (i.e. javax.persistence.EntityExistsException). TomEE uses the Apache OpenJPA [http://openjpa.apache.org/] implementation of the JPA API, which means that the EntityExistsException class (and other exceptions classes too) are part of the org.apache.openjpa.persistence package. Therefore, using this exception with our code, requires to use import org.apache.openjpa.persistence.EntityExistsException; instead of import javax.persistence.EntityExistsException; as well as adding the openjpa-xxx.jar (located in the lib subfolder of the TomEE installation folder) to the Java application classpath for being able to have the code compiled with Eclipse or other IDE tools.

5.5. Requiring non-empty strings

Normally a mandatory string-valued attribute, such as title, requires a non-empty string, which is expressed in our model above by the range NonEmptyString. For treating empty strings as null, the context parameter javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL must be set to true in web.xml:

```xml
<context-param>
    <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>
</context-param>
```
6. Validation in the View Layer

After we have defined the validations in the Java model layer and the database layer we need to take care of validation in the user interface. In particular, we need to make sure that human readable error messages are displayed.

6.1. Form validation for the Create Object use case

The WebContent/views/books/create.xhtml file contains the JSF code (see below) used to define the view which allows to create a new Book record. We like to improve the code so we can see the validation errors as well as enforce the uniqueness validation check.

```xml
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="content">
    <h:form id="createBookForm" styleClass="pure-form pure-form-aligned">
      <h:panelGrid columns="3">
        <h:outputLabel for="isbn" value="ISBN: " />
        <h:inputText id="isbn" value="#{book.isbn}" validator="#{bookController.checkIsbnAsId}" />
        <h:message id="isbnMessages" for="isbn" errorClass="error" />
        <h:outputLabel for="title" value="Title: " />
        <h:inputText id="title" value="#{book.title}" />
        <h:message id="titleMessages" for="title" errorClass="error" />
        <h:outputLabel for="year" value="Year: " />
        <h:inputText id="year" p:type="number" value="#{book.year}" />
        <h:message id="yearMessages" for="year" errorClass="error" />
      </h:panelGrid>
    </h:form>
  </ui:define>
</ui:composition>
```

There are only a few changes comparing with the same view used for the minimal application, when no validation was performed. The first change is the new h:message element which is bound to a specific form element by using the h:message/@for attribute. We create a such an element for each of our form input elements. Please notice that we don't have to do nothing else for seeing the validation errors for all integrity constraint checks which are performed by using the Validation API annotations (builtin and custom defined ones). As soon as a constraint validation fails, the message set by using the message property of the integrity constraint annotation (e.g. @Pattern, @NotNull, etc) is displayed in the JSF generated HTML5 span element, created as a result of using the h:message element.

For all the integrity constraints we have used Validation API annotations (builtin and the UpToNextYear custom one we've created), but for the uniqueness constraint we have used custom code, therefore no error message will be shown for it. In the view code we can see that a new attribute, h:inputText/@validator was used for the isbn input field. It specifies which custom method is used to perform validation of the provided value in this form field. In our case, we use the checkIsbnAsId method defined in the BookController controller as shown below:

```java
public void checkIsbnAsId( FacesContext context, UIComponent component,
  Object value) throws ValidatorException {
  String isbn = (String) value;
  try {
    Book.checkIsbnAsId( em, isbn);
  }
```
The controller check method throws a ValidatorException which is also used to deliver the error message (the third parameter of the ValidatorException constructor) to the corresponding JSF facetlet for being displayed in the UI. Methods used as validators must have a specific syntax. The first two parameters of type FacesContext, respectively UIComponent are used by the container to invoke the method with references to the right view component and context, and they can be used in more complex validation methods. The last one, of type Object is mostly used by the method validation code and it contains the value which was provided in the form field. This value has to be casted to the expected value type (i.e. String in our case). It is important to know that, if the a cast to a non compatible type is requested, the validation method fails and an exception is thrown.

6.2. Form validation for the **Update Object** use case

In the update object test case, the WebContent/views/books/update.xhtml file was updated so it uses the h:message elements for being able to display validation errors:

```xml
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="content">
    <h:form id="updateBookForm" styleClass="pure-form pure-form-aligned">
      <h:panelGrid columns="3">
        <h:outputLabel for="selectBook" value="Select book:" />
        <h:selectOneMenu id="selectBook" value="#{book.isbn}">
          <f:selectItem itemValue="" itemLabel="---" />
          <f:selectItems value="#{bookController.books}" var="b" itemValue="#{b.isbn}" itemLabel="#{b.title}" />
        </h:selectOneMenu>
        <h:outputLabel for="isbn" value="ISBN:" />
        <h:outputText id="isbn" value="#{book.isbn}" />
        <h:outputLabel for="title" value="Title:" />
        <h:inputText id="title" value="#{book.title}" />
        <h:message id="titleMessages" for="title" errorClass="error" />
        <h:outputLabel for="year" value="Year:" />
        <h:inputText id="year" value="#{book.year}" p:type="number" value="#{book.year}" />
        <h:message id="yearMessages" for="year" errorClass="error" />
      </h:panelGrid>
    </h:form>
  </ui:define>
</ui:composition>
```

Since we do not allow to change the isbn value, it is created by using h:outputText, it does not require to be validated in the view, so a validator is not defined as in the create object use case.

The result of an h:outputText JSF element is a HTML5 span element. This means that the form submission which occurs on pressing the "Update" h:commandButton contains no information about that specific element. If the validation fail, we expect to see the form content together with the error messages. To get the expected result, we need to use @ViewScoped annotation for the bean class (i.e. pl.model.Book) instead of @RequestScoped, otherwise our bean instance referenced by the book variable is initialized with a new value on every request (this is what @RequestScoped annotation is for), and then #{book.isbn}" is equal with null. As a result, our ISBN value is not shown. The @ViewScoped annotation specifies that the bean instance is alive as long as the associated
view is alive, so the ISBN value stored by the book is available for all this time and it can be displayed in the view.

By contrast, the h:inputText JSF elements results in input HTML5 elements which are part of the form submission content, so the response contains the already existing values because these values are known in this case. From this we learn that choosing a bean scope is important, and it can create logical errors if we do not pay attention to it.

7. Defining a Custom Validation Annotation

One other integrity constraint we have to consider is about the values of the year property, which should be in [1459, nextYear] range. At first we may have the intention to use @Min and @Max to specify the range, but we discover soon that it is not possible to specify an expression as value of the @Max annotation (as well as for the other annotations too), so we cannot specify the nextYear value expression. Sure, we are still able to specify the lower range value (i.e. 1459) by using @Min(value = 1459).

Fortunately, the Bean Validation API allows to define custom validation annotations with custom code performing the constraint checks. This means that we are free to express any kind of validation logic in this way. Creating and using a custom validation annotation requires to:

1. create the annotation interface - we call this UpToNextYear and the corresponding code is shown below:

```java
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = UpToNextYearImpl.class)
public @interface UpToNextYear {
    String message() default "The value of year must be between 1459 and next year!";
    Class<?>[] groups() default {};
    Class<?> extends Payload[] payload() default {};
}
```

It requires to define the three builtin methods, message (returns the default key or text message in case the constraint is violated), groups (allows the specification of validation groups, to which this constraint belongs) and payload (used by clients of the Validation API to assign custom payload objects to a constraint - this attribute is not used by the API itself). Notice the @Target annotation, which specifies on which kind of element type we can apply the annotation (e.g. fields/properties and methods in our case). The @Constraint annotation allows to specify (using the validatedBy property) the implementation class which will perform the validation, i.e. UpToNextYearImpl in our case.

2. create an implementation class where the validation code is defined:

```java
public class UpToNextYearImpl implements ConstraintValidator<UpToNextYear, Integer> {
    private Calendar calendar;

    @Override
    public void initialize(UpToNextYear arg0) {
        this.calendar = Calendar.getInstance();
        calendar.setTime(new Date());
    }

    @Override
    public boolean isValid(Integer year, ConstraintValidatorContext context) {
        if (year == null ||
```
Implementing Constraint Validation in a Java Web App

```java
year > this.calendar.get(Calendar.YEAR) + 1) {
    return false;
}
return true;
}
```

The implementation class implements the `ConstraintValidator` interface which requires two Java generics parameters: first is the annotation interface defined before (i.e. `UpToNextYear`), the second one represents the type of elements, which the validator can handle (i.e. `Integer`, so implicitly also the compatible primitive type `int`). The `initialize` method (as also the name specifies) allows to create all kind of initialization are required for performing the validation check. The `isValid` method is responsible to perform the validation check: it must returns `true` if validation passed and false otherwise. The first parameter of the `isValid` method represents the value (current property or returned method value) to be validated and its type must be compatible with the type defined by the second generics parameter of the `ConstraintValidator` (i.e. `Integer` in our case).

3. annotate the property or method - in our case we like to validate the `year` property so the corresponding code is:

```java
@Entity
@Table(name = "books")
public class Book {
    // ...
    @Min(value = 1459)
    @UpToNextYear
    private Integer year;
    // ...
}
```

8. Run the App and Get the Code

Running your application is very simple. First stop (if already started, otherwise skip this part) your Tomcat/TomEE server by using `bin/shutdown.bat` for Windows OS or `bin/shutdown.sh` for Linux. Next, download and unzip the ZIP archive file [ValidationApp.zip] containing all the source code of the application and also the ANT script file which you have to edit and modify the `server.folder` property value. Now, execute the following command in your console or terminal: `ant deploy -Dappname=validationapp`. Last, start your Tomcat web server (by using `bin/startup.bat` for Windows OS or `bin/startup.sh` for Linux). Please be patient, this can take some time depending on the speed of your computer. It will be ready when the console display the following info:

```
INFO: Initializing Mojarra [some library versions and paths are show here] for context '/validationapp'.
```

Finally, open a web browser and type: `http://localhost:8080/validationapp/faces/views/books/index.xhtml`

You may want to download the ZIP archive [lib.jar] containing all the dependency libraries, or run the validation app [http://yew.informatik.tu-cottbus.de/tutorials/java/validationapp/] directly from our server.

9. Possible Variations and Extensions

9.1. Object-level constraint validation

As an example of a constraint that is not bound to a specific property, but must be checked by inspecting several properties of an object, we consider the validation of the attribute `Author::dateOfDeath`. 
First, any value for this attribute must be in the past, which can be specified with the @Past Bean Validation annotation, and second, any value of dateOfBirth must be after the dateOfBirth value of the object concerned. This object-level constraint cannot be expressed with a pre-defined Bean Validation annotation. We can express it with the help of a custom class-level annotation, like the following AuthorValidator annotation interface:

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=AuthorValidatorImpl.class)
public @interface AuthorValidator {
    String message() default "Author data is invalid!";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Comparing with a property level custom constraint validation definition, there is only one difference, the parameter of @Target annotation. While in the case of a property and method level custom constraint validation the values are ElementType.FIELD and ElementType.METHOD, for the case of a class it must be ElementType.TYPE. For a better understanding of the rest of the parameter, please read Part 2 (Validation Tutorial [validation-tutorial.html]).

The corresponding implementation class, i.e., AuthorValidatorImpl, has the same structure as for the property validation case, but now, we can get access to the class instance and access all the properties and their corresponding values, so we can compare two or many properties when required. In our case, we have to compare the values of dateOfBirth and dateOfDeath values, as shown below, by using the required isValid method:

```java
public class AuthorValidatorImpl implements ConstraintValidator<AuthorValidator, Author> {
    @Override
    public void initialize(AuthorValidator arg0) {} 
    
    @Override
    public boolean isValid(Author author, ConstraintValidatorContext context) {
        if (author.getDateOfDeath() != null && author.getDateOfBirth().after( author.getDateOfDeath())) {
            return false;
        }
        return true;
    }
}
```

Using this class level validator with JSF requires a bit of tweaking, this feature not being directly supported by JSF (as in the case of property and method validators). For this, in the JSF view, for the specific form element to be validated, we have to specify who is doing the validation, by using the @validator attribute and a JSF expression which points out to the controller method invoked to perform the validation:

```html
<ui:composition template="/WEB-INF/templates/page.xhtml">
    <ui:define name="content">
        <h:form id="createAuthorForm">
            <h:panelGrid columns="3">
                <h:outputLabel for="dateOfDeath" value="Date of death: " />
                <h:inputText id="dateOfDeath" p:type="date" value="#{author.dateOfDeath}" validator="#{authorController.checkDateOfDeath}" f:convertDateTime pattern="yyyy-MM-dd" />
                <h:message for="dateOfDeath" errorClass="error" />
            </h:panelGrid>

            <h:commandButton value="Create" action="#{authorController.add( author.personId, author.name, author.dateOfBirth, author.dateOfDeath)}" />

        </h:form>
    </ui:define>
</ui:composition>
```
Implementing Constraint Validation in a Java Web App

The corresponding checkDateOfDeath method is responsible for manually invoking the Java Validation API validator, capture the corresponding validation exceptions and translate them to javax.faces.validator.ValidatorException exceptions which are then managed by JSF and displayed in the view. The method code is shown below:

```java
public void checkDateOfDeath( FacesContext context, UIComponent component,
    Object value) {
    boolean isCreateForm = (UIForm) context.getViewRoot().
        findComponent( "createAuthorForm") != null;
    String formName = isCreateForm ? "createAuthorForm:" : "updateAuthorForm:";
    UIInput personIdInput = isCreateForm ? (UIInput) context.getViewRoot().
        findComponent( formName + "personId") : null;
    UIOutput personIdOutput = isCreateForm ? null :
        (UIOutput) context.getViewRoot().findComponent( formName + "personId");
    UIInput nameInput = (UIInput) context.getViewRoot().findComponent( formName + "name");
    UIInput dateOfBirthInput = (UIInput) context.getViewRoot().
        findComponent( formName + "dateOfBirth");
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();
    Author author = new Author();
    if ( isCreateForm) {
        author.setPersonId( (Integer) personIdInput.getValue());
    } else {
        author.setPersonId( (Integer) personIdOutput.getValue());
    }    
    author.setName( (String) nameInput.getValue());
    author.setDateOfBirth( (Date) dateOfBirthInput.getValue());
    author.setDateOfDeath( (Date) value);
    Set<ConstraintViolation<Author>> constraintViolations = validator.validate(author);
    for ( ConstraintViolation<Author> cv : constraintViolations) {
        if ( cv.getMessage().contains("date of death")) {
            throw new ValidatorException( new FacesMessage( FacesMessage.SEVERITY_ERROR,
                cv.getMessage(), cv.getMessage()));
        }
    }
}
```

While the method looks complicated, it is responsible for the following simple tasks:

- get access to form data and extract the user input values, by using context.getViewRoot().findComponent( componentName) method. Notice that the component name has the pattern: formName:formElementName.
- create the Author instance and set the corresponding data as extracted from the form, by using the FacesContext instance provided by the JSF specific validator method
- manually invoke the Java Validation API validator by using the javax.validation.Validator class.
- loop trough the validator exception, select the ones which corresponds to the custom validated field and map them to javax.faces.validator.ValidatorException exceptions. The selection can be made by looking for specific data in the exception message.

As a result, the custom Java Validation API class validator is not used, and the JSF view is able to render the corresponding error messages when the validation fails, in the same way as is possible for single property validation situations.

### 9.2. Alternative class level custom constraint validation

An alternative method for class level custom validation is to use JSF custom class validators. The advantage of this is that they are directly supported in the JSF views, and the disadvantage is that if
Implementing Constraint Validation in a Java Web App

another UI engine has to be used outside JSF one, then the validation will be of no use. Also, using the Java Validation API, the validation is mainly provided at the model level, which is the desired method in most of the case, this way avoiding the duplicate logic for validations (i.e., the code duplication for the view related code and the one related to the model).

For our example, the validator for the Author class which is responsible for the validation of dateOfDeath property by comparing it with the dateOfBirth is shown below:

```java
@FacesValidator( "AuthorValidator")
public class AuthorValidator implements Validator  {
    @Override
    public void validate( FacesContext context, UIComponent component,
            Object value) throws ValidatorException  {
        Date dateOfDeath  = (Date)value;
        boolean isCreateForm = (UIForm) context.getViewRoot().
                        findComponent( "createAuthorForm") != null;
        String formName = isCreateForm ? "createAuthorForm:" : "updateAuthorForm:";
        UIInput dateOfBirthInput = (UIInput) context.getViewRoot().
                        findComponent( formName + "dateOfBirth");
        Date dateOfBirth = (Date)dateOfBirth.getValue();
        if (dateOfBirth.after( dateOfDeath)) {
            throw new ValidatorException ( new FacesMessage(
                    "The date of death should be after the date of birth!")");
        }
    }
}
```

Then, in the JSF view, for the corresponding field, the validator has to be specified:

```html
<h:outputLabel for="dateOfDeath" value="Date of death: "/>
<h:inputText id="dateOfDeath" p:type="date" value="#{author.dateOfDeath}"
            <f:validator validatorId = "AuthorValidator" />
            <f:convertDateTime pattern="yyyy-MM-dd" />
</h:inputText>
<h:message for="dateOfDeath" errorClass="error" />
```

As discussed before, this method only works with the JSF framework, so it must be used only if the application is not supposed to be updated in the future to other UI frameworks. Even in such case, the Java Validation API custom validation can still be used in addition, and he helps to prevent model level validation failures.