Java EE Web App Tutorial Part 2: Adding Constraint Validation

Learn how to build a back-end web application with constraint validation, using Java EE 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 EE Web App Tutorial Part 2: Adding Constraint Validation: Learn how to build a back-end web application with constraint validation, using Java EE 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 [ValidationApp/index.html] from our server, or download it as a ZIP archive file [ValidationApp.zip].

Publication date 2018-06-18

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 2 of our series of six tutorials [http://web-engineering.info/JavaJpaJsfApp] about model-based development of back-end web applications with Java EE using the *Java Persistence API* (JPA) and *Java Server Faces* (JSF). 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 web 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 (**C**reate/**R**ead/**U**pdate/**D**elete). 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 entered by a user and stored in the application's database.

For instance, if an application is managing data about persons including their birth dates and their death dates, then we must make sure that for any person record with a death date, this date is not before that person'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, like choosing 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 configuring the user interface and for validating data in different parts of the app, such as in the user interface and in the database. In terms of usability, the goals should be:

1. To prevent the user from entering invalid data in the user interface (UI) by limiting the input options, if possible.

2. To detect and reject invalid user input as early as possible by performing constraint validation in the UI for those UI widgets where invalid user input cannot be prevented by limiting the input options.

3. To prevent that invalid data pollutes the app's main memory state and persistent database state by performing constraint validation also in the model layer and in the database.

HTML5 provides support for validating user input in an HTML-forms-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. This UI mechanism of ***responsive validation*** is an important feature of modern web applications. In traditional web applications, the 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.

2. Integrity Constraints

Integrity constraints (or simply *constraints*) are logical conditions on the data of an app. They may take many different forms. The most important type of constraints, **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 cases 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 match 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 (also called 'Key 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.

The visual language of UML class diagrams supports defining integrity constraints either in a special way for special cases (like with predefined keywords), or, in the general case, with the help of *invariants*, which are conditions expressed either in plain English or in the *Object Constraint Language (OCL)* and shown in a special type of rectangle attached to the model element concerned. We use UML class diagrams for modeling constraints in *design models* 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 not smaller than `l` or the special value * standing for *unbounded*. For showing property multiplicity (or cardinality) constrains in a class diagram, multiplicity expressions are enclosed in brackets and appended to the property name, as shown in the `Person` class rectangle below.

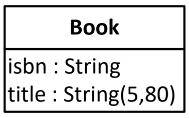
In the following sections, we discuss the different types of property constraints listed above in more detail. We also show how to express some of them in computational languages such as *UML* class diagrams, *SQL* table creation statements, *JavaScript* model class definitions, or the annotation-based languages *Java Bean Validation* annotations and *ASP.NET Data Annotations*.

Any systematic approach to constraint validation also requires to define a set of error (or 'exception') classes, including one for each of the standard property constraints listed above.

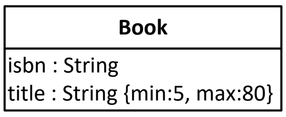
2.1. String Length Constraints

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)`.

UML does not define any special way of expressing string length constraints in class diagrams. Of course, we always have the option to use an *invariant* for expressing any kind of constraint, but it seems preferable to use a simpler form of expressing these property constraints. One option is to append a maximum length, or both a minimum and a maximum length, in parenthesis to the datatype name, like so

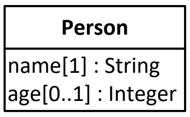


Another option is to use min/max constraint keywords in the property modifier list:



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`:



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:

```
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 code 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:

```
Person.checkName = function (n) {
  if (n === undefined) {
    return "A name must be provided!"; // constraint violation error message
  } else return "";  // no constraint violation
};
```

With Java Bean Validation, a mandatory property like `name` is annotated with `NotNull` in the following way:

```
@Entity
public class Person {
    @NotNull
    private String name;
    private int age;
}
```

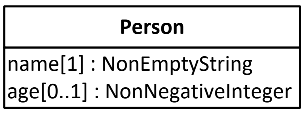
The equivalent ASP.NET Data Annotation is `Required` as shown in

```
public class Person{
    [Required]
    public string name { get; set; }
    public int age { get; set; }
}
```

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 "" (the empty string), 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:



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

```
Person.checkName = function (n) {
  if (typeof(n) !== "string" || n.trim() === "") {
    return "Name must be a non-empty string!";
  } else return "";
};
```

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.

In a Java EE web app, for declaring empty strings as non-admissible user input we must set the context parameter

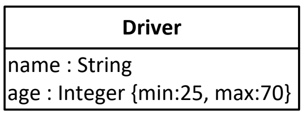
```
javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
```

to `true` in the web deployment descriptor file `web.xml`.

In ASP.NET, empty strings are non-admissible by default.

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 [25,70]. In a class diagram, we can define such a constraint by using the property modifiers `min` and `max`, as shown for the `age` attribute of the `Driver` class in the following diagram.



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:

```
CREATE TABLE drivers(
  name  VARCHAR NOT NULL,
  age   INTEGER CHECK (age >= 25 AND age <= 70)
)
```

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

```
Driver.checkAge = function (a) {
  if (a < 25 || a > 70) {
    return "Age must be between 25 and 70!";
  } else return "";
};
```

In Java Bean Validation, we express this interval constraint by adding the annotations `Min(0)` and `Max(120)` to the property `age` in the following way:

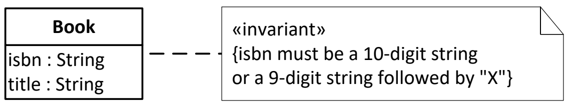
```
@Entity
public class Driver {
    @NotNull
    private String name;
    @Min(25) @Max(70)
    private int age;
}
```

The equivalent ASP.NET Data Annotation is `Range(25,70)` as shown in

```
public class Driver{
    [Required]
    public string name { get; set; }
    [Range(25,70)]
    public int age { get; set; }
}
```

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 the following diagram.


Book
isbn : String
title : String
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}

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:

```
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 code a pattern constraint by using the built-in regular expression function `test`, as illustrated in the following example:

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

In Java EE Bean Validation, this pattern constraint for `isbn` is expressed with the annotation `Pattern` in the following way:

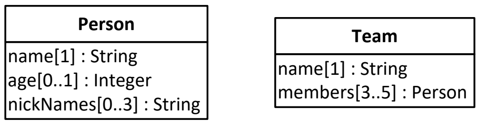
```
@Entity
public class Book {
    @NotNull
    @Pattern(regexp="^\\(\d{9}(\d|X))$")
    private String isbn;
    @NotNull
    private String title;
}
```

The equivalent ASP.NET Data Annotation is `RegularExpression` as shown in

```
public class Book{
    [Required]
    [RegularExpression(@"^(\d{9}(\d|X))$")]
    public string isbn { get; set; }
    public string title { get; set; }
}
```

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 the following diagram, which contains two examples of properties with cardinality constraints.



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 (or maximum) cardinality constraint is an on-delete (or on-insert) trigger that tests the number of rows with the same reference as the deleted (or inserted) row.

In JavaScript, we can code 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:

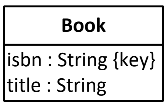
```
Person.checkNickNames = function (nickNames) {
  if (nickNames.length > 3) {
    return "There must be no more than 3 nicknames!";
  } else return "";
};
```

With Java Bean Validation annotations, we can specify

```
@Size( max=3)
List<String> nickNames
@Size( min=3, max=5)
List<Person> members
```

2.7. Uniqueness Constraints

A *uniqueness constraint* (or *key constraint*) requires that a property's value (or the value list of a list of properties in the case of a composite key constraint) is unique among all instances of the given object type. For instance, in a UML class diagram with the object type `Book` we can define the `isbn` attribute to be *unique*, or, in other words, a *key*, by appending the (user-defined) property modifier keyword `key` in curly braces to the attribute's definition in the `Book` class rectangle shown in the following diagram.



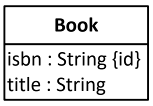
In an SQL table creation statement, a uniqueness constraint is expressed by appending the keyword `UNIQUE` to the column definition as in the following example:

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

In JavaScript, we can code 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)

A unique attribute (or a composite key) can be declared to be the standard identifier for objects of a given type, if it is mandatory (or if all attributes of the composite key are mandatory). We can indicate this in a UML class diagram with the help of the property modifier `id` appended to the declaration of the attribute `isbn` as shown in the following diagram.



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

Often, practitioners do not recommended using a composite key as a standard ID, since composite identifiers are more difficult to handle and not always supported by tools. Whenever an object type does not have a key attribute, but only a composite key, it may therefore be preferable to add an artificial standard ID attribute (also called *surrogate ID*) to the object type. However, each additional surrogate ID has a price: it creates some cognitive and computational overhead. Consequently, in the case of a simple composite key, it may be preferable not to add a surrogate ID, but use the composite key as the standard ID.

There is also an argument against using any real attribute, such as the `isbn` attribute, for a standard ID. The argument points to the risk that the values even of natural ID attributes like `isbn` may have to be changed during the life time of a business object, and any such change would require an unmanageable effort to change also all corresponding ID references. However, the business semantics of natural ID attributes implies that they are frozen. Thus, the need of a value change can only occur in the case of a data input error. But such a case is normally detected early in the life time of the object concerned, and at this stage the change of all corresponding ID references is still manageable.

Standard IDs 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 object-oriented programming languages, like JavaScript and Java, we cannot code a standard ID declaration, because this would have to be part of the metadata of a class definition, and there is no support for such metadata. However, we should still check the implied mandatory value and uniqueness constraints.

2.9. Referential Integrity Constraints

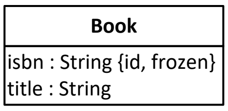
A referential integrity constraint requires that the values of a reference property refer to an object that exists in the population of the property's range class. Since we do not deal with reference properties in this chapter, we postpone the discussion of referential integrity constraints to Part 4 of our tutorial.

2.10. Frozen and Read-Only 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. This includes the special case of **read-only value constraints** on mandatory properties that are initialized at object creation time.

Typical examples of properties with a frozen value constraint are standard identifier attributes and event properties. In the case of events, the semantic principle that the past cannot be changed prohibits that the property values of events can be changed. In the case of a standard identifier attribute we may want to prevent users from changing the ID of an object since this requires that all references to this object using the old ID value are changed as well, which may be difficult to achieve (even though SQL provides special support for such ID changes by means of its `ON UPDATE CASCADE` clause for the change management of foreign keys).

The following diagram shows how to define a frozen value constraint for the `isbn` attribute:



In Java, a *read-only* value constraint can be enforced by declaring the property to be `final`. In JavaScript, a *read-only* property slot can be implemented as in the following example:

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

where the property slot `obj.teamSize` is made unwritable. An entire object `obj` can be frozen with `Object.freeze( obj)`.

We can implement a frozen value constraint for a property in the property's setter method like so:

```
Book.prototype.setIsbn = function (i) {
  if (this.isbn === undefined) this.isbn = i;
  else console.log("Attempt to re-assign a frozen property!");
}
```

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 (object-level constraints),

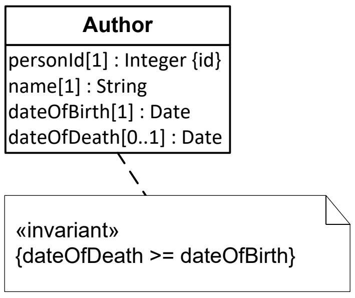
2. the value of a property before and its value after a change attempt (dynamic constraints),
3. the set of all instances of a particular object type (type-level constraints),
4. the set of all instances of several object types.

OCL

The *Object Constraint Language* (OCL) was defined in 1997 as a formal logic language for expressing integrity constraints in UML version 1.1. Later, it was extended for allowing to define also (1) derivation expressions for defining derived properties, and (2) preconditions and postconditions for operations, in a class model.

In a class model, property constraints can be expressed within the property declaration line in a class rectangle (typically with keywords, such as `id`, `max`, etc.). For expressing more complex constraints, such as object-level or type-level constraints, we can attach an *invariant* declaration box to the class rectangle(s) concerned and express the constraint in unambiguous plain English or in pseudo-code . A simple example of an object-level constraint expressed as an invariant is shown in Figure 1.1.

Figure 1.1. An example of an object-level constraint



A general approach for implementing *object-level constraint validation* consists of taking the following steps:

1. Choose a fixed name for an object-level constraint validation function, such as `validate`.
2. For any class that needs object-level constraint validation, define a `validate` function returning either a `ConstraintViolation` or a `NoConstraintViolation` object.
3. Call this function, if it exists, for the given model class,
   a. in the UI/view, on form submission;
   b. in the model class, before save, both in the `create` and in the `update` method.

Constraints affecting two or more model classes could be defined in the form of static methods (in a model layer method library) that are invoked from the `validate` methods of the affected model classes.

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 acceptable today. Rather, in a *responsive validation* approach, the user should get immediate validation feedback on each single data input. Technically, this can be achieved with the help of event handlers for the user interface events `input` or `change`.

Responsive validation requires a data validation mechanism in the user interface (UI), such as the HTML5 form validation API [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/]. 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 types of `input` fields (such as `number` or `date`) and a set of new attributes for form control elements for the purpose of supporting responsive validation 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 field as either valid or invalid by assigning either an empty string or a non-empty (constraint violation) message string.

The second method, `checkValidity`, is invoked on a form before user input data is committed or saved (for instance with a form submission). It tests, if all fields have a valid value. For having the browser automatically displaying any constraint violation messages, we need to have a `submit` event, even if we don't really submit the form, but just use a `save` button.

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 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.

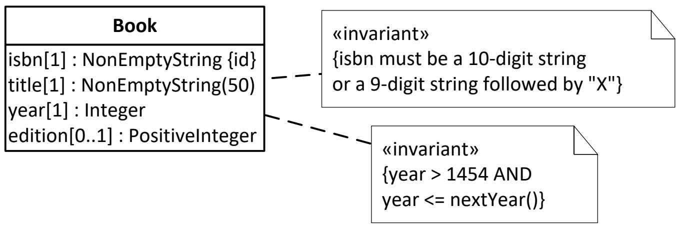
3. Use JavaScript as your back-end application programming language (such as with NodeJS), then you can code your validation functions in your JavaScript model classes on the back-end and execute them both before committing changes on the back-end and on user input and form submission in the UI on the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

The simplest, and most responsive, solution is the third one, using only JavaScript both for the back-end and front-end components of a web app.

5. Adding Constraints to a Design Model

We again consider the book data management problem that was considered in Part 1 of this tutorial. But now we also consider the **data integrity rules** (or '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 1.2 below.

Figure 1.2. A design model defining the object type `Book` with two invariants


Book
isbn[1] : NonEmptyString {id}
title[1] : NonEmptyString(50)
year[1] : Integer
edition[0..1] : PositiveInteger
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}
«invariant»
{year > 1454 AND
year <= nextYear()}

In this 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**, as indicated by its multiplicity expression [1], and has a **string length constraint** requiring its values to have at most 50 characters.

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 calendar function `nextYear()`, which we implement as a utility function.

Notice that the `edition` attribute is not mandatory, but *optional*, as indicated by its multiplicity expression [0..1]. In addition to the constraints described in this list, there are the implicit range constraints defined by assigning the datatype `NonEmptyString` as range to `isbn` and `title`, `Integer` to `year`, and `PositiveInteger` to `edition`. In our plain JavaScript approach, all these property constraints are coded in the model class within property-specific *check* functions.

The meaning of the design model can be illustrated by a sample data population respecting all constraints:

Table 1.1. Sample data for `Book`

ISBN	**Title**	**Year**	**Edition**
006251587X	Weaving the Web	2000	3
0465026567	Gödel, Escher, Bach	1999	2
0465030793	I Am A Strange Loop	2008	

6. Summary

1. Constraints are logical conditions on the data of an app. The simplest, and most important, types of constraints are property constraints and object-level constraints.
2. Constraints should be defined in the model classes of an MVC app, since they are part of their business semantics.
3. Constraints should be checked in various places of an MVC app: in the UI/view code, in model classes, and possibly in the database.
4. Software applications that include CRUD data management need to perform two kinds of bi-directional object-to-string type conversions:
    a. Between the model and the UI: converting model object property values to UI widget values, and, the other way around, converting input widget values to property values. Typically, widgets are form fields that have string values.
    b. Between the model and the datastore: converting model objects to storage data sets (called serialization), and, the other way around, converting storage data sets to model objects (called de-serialization). This involves converting property values to storage data values, and, the other way around, converting storage data values to property values. Typically, datastores are either JavaScript's local storage or IndexedDB, or SQL databases, and objects have to be mapped to some form of table rows. In the case of an SQL database, this is called "Object-Relational Mapping" (ORM).
5. Do not perform any string-to-property-value conversion in the UI code. Rather, this is the business of the model code.
6. For being able to observe how an app works, or, if it does not work, where it fails, it is essential to log all critical application events, such as data retrieval, save and delete events, at least in the JavaScript console.
7. Responsive validation means that the user, while typing, gets immediate validation feedback on each input (keystroke), and when requesting to save the new data.

Chapter 2. Implementing Constraint Validation in a Java EE 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 EE model class (called *entity class*), and how to have constraints automatically validated on critical life cycle events of entity objects with JPA, and on form submission with JSF.

1. 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, 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 EE apps support this goal to some degree. There are two types of constraint annotations:

1. **JPA constraint annotations** 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 `year` property of the `Book` class, since its value has an upper bound defined by an expression ('next year').

1.1. JPA constraint annotations

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

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

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

```
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 `itemCode` strings, the length is set to 255 by default. However, in our case we know that `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:

```
@Entity @Table( name="items")
public class Item {
  @Id @Column( length=10)
  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`:

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

1.2. Bean Validation annotations

In the Java EE 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 2.1. Bean Validation annotations for properties

Constraint Type	Annotations	Examples
String Length Constraints	`@Size`	`@Size( min=8, max=80) String message;`
Cardinality Constraints (for arrays, collections and maps)	`@Size`	`@Size( min=2, max=3) List<Member> coChairs;`
Mandatory Value Constraints	`@NotNull`	`@NotNull String name`

Constraint Type	Annotations	Examples
Range Constraints for numeric attributes	`@Digits`	`@Digits( integer=6, fraction=2) BigDecimal price;`
Interval Constraints for integer-valued attributes	`@Min` and `@Max`	`@Min(5) int teamSize`
Interval Constraints for decimal-valued attributes	`@DecimalMin` and `@DecimalMax`	`@DecimalMax("30.00") double voltage`
Pattern Constraints	`@Pattern`	`@Pattern( regexp="\\b\\d{10}\\b") String isbn;`

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 custom error message. In the following example, we add two `@NotNull` annotations with messages, a `@Size` and a `@Min` annotation to the JPA 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:

```
@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)
  @NotNull( message="A quantity is required!")
  @Min(0)
   private int quantity;
}
```

Notice that that we need some duplicate logic in this example because the same constraints may have to be defined twice: as a JPA constraint annotation and as a Bean Validation annotation. For instance, for a mandatory attribute like `quantity` we have both a `@Column( nullable=false)` JPA constraint annotation and a `@NotNull` Bean Validation annotation.

2. New Issues

Compared to the Minimal App [http://web-engineering.info/tech/JavaJpaJsfApp/MinimalApp/index.html] discussed in the Minimal App Tutorial [http://web-engineering.info/tech/JavaJpaJsfApp/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.

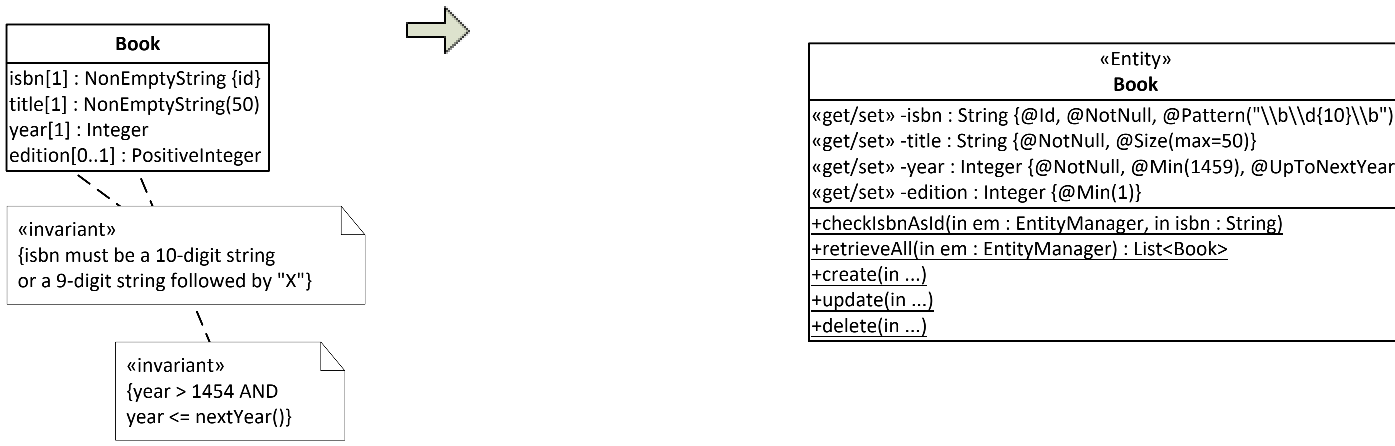
Checking the constraints in the user interface on user input is important for providing immediate feedback to the user. Using JSF and Bean Validation requires to submit the form before the validation checks are performed. It would be preferable to define the validation checks in the model classes only and use them in the user interface before form submission, without having to duplicate the validation logic in the JSF facelets. 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 facelets 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 maintenance nightmare.

3. Make an Entity Class Model

Using the information design model shown in Figure 1.2 above as the starting point, we make an *Entity class model* with getters/setters and corresponding Java datatypes.

Figure 2.1. Deriving an Entity class model from an information design model


Book
isbn[1] : NonEmptyString {id}
title[1] : NonEmptyString(50)
year[1] : Integer
edition[0..1] : PositiveInteger
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}
«invariant»
{year > 1454 AND
year <= nextYear()}
«Entity»
Book
«get/set» -isbn : String {@Id, @NotNull, @Pattern("\\b\\d{10}\\b")
«get/set» -title : String {@NotNull, @Size(max=50)}
«get/set» -year : Integer {@NotNull, @Min(1459), @UpToNextYear
«get/set» -edition : Integer {@Min(1)}
+checkIsbnAsId(in em : EntityManager, in isbn : String)
+retrieveAll(in em : EntityManager) : List<Book>
+create(in ...)
+update(in ...)
+delete(in ...)

Notice that for the `year` and `edition` attributes, the datatype wrapper class `Integer` is used instead of the primitive datatype `int`. This is required when using JSF, since if a form with an empty input field is submitted in the Create or Update use case, the value `null` is assigned to the corresponding attribute (by JSF invoking the attribute's setter), which is not admitted for primitive datatypes by Java.

The entity class model defines getters and setters for all properties and the following property constraint annotations:

1. `@Id` and `@NotNull` declare the `isbn` attribute to be a standard identifier, implying that it is *mandatory* and *unique*.
2. `@Pattern("\\b\\d{10}\\b")` declares a *pattern constraint* on the `isbn` attribute requiring its values to match the ISBN-10 format (simplified to the case of 10-digit strings).
3. `@NotNull` and `@Size(max=50)` declare that the `title` attribute is *mandatory* and has a *string length maximum constraint* of at most 50 characters.
4. `@NotNull`, `@Min(1459)` and the custom annotation `@UpToNextYear` declare that the `year` attribute is *mandatory* and has an *interval constraint* of a special form where the minimum is 1459 and the maximum is not fixed, but provided by a custom annotation implementing the calendar arithmetic function *nextYear()*.
5. `@Min(1)` declares that the `edition` attribute has an *interval constraint* with minimum value 1.

Since there is no predefined Bean Validation annotation for checking the uniqueness of an ID value provided when creating a new entity object, we define a static method `checkIsbnAsId` that can be invoked in a corresponding controller method when creating a new entity object.

In addition, the entity class model defines the static CRUD data management methods `retrieveAll`, `create`, `update` and `delete`.

4. Write the Model Code

The Entity class model shown on the right hand side in Figure 2.1 can be coded step by step for getting the code of the entity classes of our Java EE web app.

4.1. Type mapping

When defining the properties, we first need to map the platform-independent datatypes of the information design model to the corresponding Java datatypes according to the following table.

Table 2.2. Java datatype mapping

Platform-independent datatype	Java datatype
String	String
Integer	int, long, Integer, Long
Decimal	double, Double, java.math.BigDecimal
Boolean	boolean, Boolean
Date	java.util.Date

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 datatypes:

Table 2.3. MySQL datatype mapping

Platform-independent datatype	MySQL datatype
String	VARCHAR
Integer	INT
Decimal	DECIMAL
Boolean	BOOL
Date	DATETIME or TIMESTAMP

4.2. Code the constraints as annotations

In this section we add JPA constraint annotations and Bean Validation annotations for implementing the property constraints defined for the `Book` class in the Java entity class model. For the standard identifier attribute `isbn`, we add the JPA 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 JPA constraint annotation `@Column( nullable=false)`, as well as the Bean Validation annotations `@NotNull` and `@Size( max=50)`.

For the attribute `year`, we add the JPA constraint annotation `@Column( nullable=false)`, as well as the Bean Validation annotations `@NotNull` and `@Min( 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 6 below.

Coding the integrity constraints with JPA constraint annotations and Bean Validation annotations results in the following annotated bean class:

```
@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;
  @Column( nullable=false)
  @NotNull( message="A title is required!")
  @Size( max=255)
   private String title;
  @Column( nullable=false)
  @NotNull( message="A year is required!")
  @Min( value=1459, message="The year must not be before 1459!")
   private Integer year;

  ...  // define constructors, setters and getters

  public static Book getObjectByStdId(...) {...}
  public static List<Book> getAllObjects(...) {...}
  public static void create(...) throws Exception {...}
  public static void update(...) throws Exception {...}
  public static void delete(...) throws Exception {...}
}
```

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

4.3. 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 check this in our Java app before the data is passed to the DBMS. 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:

```
public static void checkIsbnAsId( EntityManager em, String isbn)
    throws UniquenessConstraintViolation,
    MandatoryValueConstraintViolation {
  if (isbn == null) {
    throw new MandatoryValueConstraintViolation(
        "An ISBN value is required!");
  } else {
    Book book = Book.retrieve( em, isbn);
    // book was found, uniqueness constraint validation failed
    if ( book != null) {
    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.

Notice that in this case we also need to check the `isbn` value and reject `null` values, because the `@NotNull` validation triggers only later, when the `isbn` property of the `Book` is set, thus at this point we could get `NullPointerException`, from the `Book.retrieve` method.

4.4. Dealing with model-related exceptions

The `Book.checkIsbnAsId` method discussed in the previous sub-section is designed to be used in combination with a controller so the user gets an error message when trying to duplicate a `Book` record (i.e., if the provided `isbn` value is already used in an existing record). However, if the `Book.create` method is used directly (i.e. by another piece of code, where the uniqueness constraint is not performed by calling `Book.checkIsbnAsId`), then uniqueness constraint validation may fail. Lets have a look on the `Book.create` code:

```
public static void create( 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 may throw a number of exceptions when trying to execute 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.create` may catch this exception and perform specific actions, such as rolling back the transaction. In our case, the `Book.create` is called by the `create` action method of the `BookController` class, and the action performed is to show the exception stack trace 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 stack trace is displayed in the console.

```
public String create( String isbn, String title, int year) {
  try {
    Book.create( 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 `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 class path for being able to have the code compiled with Eclipse or other IDE tools.

4.5. 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 no value, the context parameter `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` must be set to `true` in `web.xml`:

```
<context-param>
  <param-name>
    javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
  </param-name>
  <param-value>true</param-value>
</context-param>
```

5. Write the View Code

After we have defined the constraints in the Java EE 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 the user gets informed about issues by rendering visual indicators and informative validation error messages.

5.1. Validation in the *Create* use case

The `WebContent/views/books/create.xhtml` file contains the JSF facelet code for creating a new `Book` record. We now use the JSF `validator` attribute for performing the uniqueness validation and JSF `message` elements for displaying validation error messages.

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="headerTitle">
    <h1>Create a new book record</h1>
  </ui:define>
  <ui:define name="main">
    <h:form id="createBookForm">
      <div>
        <h:outputLabel for="isbn" value="ISBN: ">
        <h:inputText id="isbn" value="#{book.isbn}"
            validator="#{bookCtrl.checkIsbnAsId}" />
        </h:outputLabel>
        <h:message for="isbn" errorClass="error" />
      </div>
      <div>
        <h:outputLabel for="title" value="Title: ">
        <h:inputText id="title" value="#{book.title}" />
        </h:outputLabel>
        <h:message for="title" errorClass="error" />
      </div>
      ...
    </h:form>
  </ui:define>
</ui:composition>
```

There are only a few changes compared to the same view used for the minimal app, where no validation was performed. The first change is the new `h:message` element which is bound to a specific form element by the `for` attribute. We create such an element for each of our form input elements. Notice that we don't have to do anything else for seeing the validation errors for all integrity constraint checks which are performed by using the (built-in and custom) Bean Validation annotations. 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 an HTML `span` element generated by JSF as a result of using the `h:message` element.

For all the integrity constraints we have used Bean Validation annotations, 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, `validator` in `h:inputText`, 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` as shown below:

```
public void checkIsbnAsId( FacesContext context,
    UIComponent component, Object value)
    throws ValidatorException {
  String isbn = (String) value;
  try {
    Book.checkIsbnAsId( em, isbn);
  } catch ( UniquenessConstraintViolation e) {
    throw new ValidatorException( new FacesMessage(
        FacesMessage.SEVERITY_ERROR, e.getMessage(),
        e.getMessage()));
  } catch ( MandatoryValueConstraintViolation e) {
    throw new ValidatorException( new FacesMessage(
        FacesMessage.SEVERITY_ERROR, e.getMessage(),
        e.getMessage()));
  }
}
```

The controller's 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 facelet for being displayed in the UI. Methods used as *JSF 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`, represents the value to be validated by the method. This value has to be casted to the expected type (to `String`, in our example). It is important to know that, if a cast to a non-compatible type is performed, the validation method fails and an exception is thrown.

5.2. Validation in the *Update* use case

In the *Update* use case, the facelet file `update.xhtml` in `WebContent/views/books` was updated so it uses the `h:message` elements for being able to display validation errors:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
 <ui:define name="headerTitle">
  <h1>Update a book record</h1>
 </ui:define>
 <ui:define name="main">
  <h:form id="updateBookForm">
   <div>
    <h:outputLabel for="selectBook" value="Select book: ">
     <h:selectOneMenu id="selectBook" value="#{book.isbn}">
      ...
     </h:selectOneMenu>
    </h:outputLabel>
    <h:message for="selectBook" errorClass="error" />
   </div>
   <div>
    <h:outputLabel for="isbn" value="ISBN: ">
     <h:outputText id="isbn" value="#{book.isbn}" />
    </h:outputLabel>
   </div>
   ...
  </h:form>
 </ui:define>
</ui:composition>
```

Since we do not allow to change the ISBN of a book, we create an output field for the `isbn` attribute with the JSF element `h:outputText`. This implies that no validation is performed.

Using an `h:outputText` element for showing the value of an entity attribute results in an HTML `span` element. This implies that the HTTP form submission message contains no information about that attribute. If the validation fails, we expect to see the form content together with the error messages. To get the expected result, we need to use the annotation `@ViewScoped` for the entity class `pl.m.Book` instead of `@RequestScoped`, otherwise our bean instance referenced by the `book` variable is initialized with a new value on every request, implying that the expression `#{book.isbn}` evaluates to `null` and the ISBN value is not displayed. The `@ViewScoped` annotation specifies that the entity bean is alive as long as the associated view is alive, so the ISBN value stored by the `book` is available during this time and it can be displayed in the view.

By contrast, `h:inputText` elements result in HTML `input` 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. This consideration shows that it is important to choose the right bean scope.

6. Defining a Custom Validation Annotation

One other integrity constraint we have to consider is about the allowed values of the `year` property, which must be in the interval [1459, nextYear()] where nextYear() is a function invocation expression. We may have the idea to use `@Min` and `@Max` to specify the interval constraint, but this is not possible because the `@Max` annotation (as well as any other annotation) does not allow expressions, but only data literals. So, while we can express the interval's lower bound with `@Min( value=1459)`, we need another solution for expressing the upper bound.

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 the following steps:

1. Create the annotation interface `UpToNextYear` with the following code:

   ```
   @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 {};
   }
   ```

   The interface needs to define three methods, `message` (returns the default key or error message if the constraint is violated), `groups` (allows the specification of validation groups, to which this constraint belongs) and `payload` (used by clients of the Bean Validation API to assign custom payload objects to a constraint - this attribute is not used by the API itself). Notice the `@Target` annotation, which defines the element types that can be annotated (fields/properties and methods in our case). The `@Constraint` annotation allows to specify the implementation class that will perform the validation, i.e. `UpToNextYearImpl` in our case.

2. Create an implementation class with the validation code:

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

```
      return true;
    }
  }
```

The implementation class implements the `ConstraintValidator` interface, which requires two type parameters: the annotation interface defined before (i.e. `UpToNextYear`), and the type of elements the validator can handle (i.e. `Integer`, so implicitly also the compatible primitive type `int`). The `initialize` method allows initializing variables required for performing the validation check. The `isValid` method is responsible for performing the validation: it must return `true` if the validation succeeds, and `false` otherwise. The first parameter of the `isValid` method represents the value to be validated and its type must be compatible with the type defined by the second type parameter of the `ConstraintValidator` (`Integer` in our case).

3. Annotate the property or method concerned:

```
  @Entity
  @Table( name = "books")
  public class Book {
    // ...

    @Min( value = 1459)
    @UpToNextYear
    private Integer year;

    //...
  }
```

7. Run the App and Get the Code

You can run the validation app [http://web-engineering.info/tech/JavaJpaJsf/ValidationApp/] on our server or download the code [http://web-engineering.info/tech/JavaJpaJsf/ValidationApp.zip] as a ZIP archive file.

Follow our instructions [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications] for getting your environment prepared for running Java EE web applications.

8. Possible Variations and Extensions

8.1. 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 `dateOfDeath` must be after the `dateOfBirth` value of the object concerned. This object-level constraint cannot be expressed with a predefined Bean Validation annotation. We can express it with the help of a custom class-level annotation, like the following `AuthorValidator` annotation interface:

```
@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 {};
}
```

Compared to a property constraint annotation definition, there is only one difference, the parameter of the `@Target` annotation. While in the case of a property and method level custom constraint annotation the values are `ElementType.FIELD` and `ElementType.METHOD`, for the case of a class it must be `ElementType.TYPE`.

The corresponding implementation class, i.e., `AuthorValidatorImpl`, has the same structure as in the case of a property constraint annotation , but now, we can access all properties of an entity bean, so we can compare two or more properties when required. In our case, we have to compare the values of `dateOfBirth` and `dateOfDeath` in the `isValid` method:

```
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 class-level JPA validators in facelets requires a bit of tweaking because they are not directly supported by JSF. For the specific form field to be validated, we have to specify a controller method in charge of the validation, as the value of the `@validator` attribute:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
 <ui:define name="main">
  <h:form id="createAuthorForm">
   <div>
    <h:outputLabel for="dateOfDeath" value="Date of death: ">
     <h:inputText id="dateOfDeath" p:type="date"
        value="#{author.dateOfDeath}"
        validator="#{authorCtrl.checkDateOfDeath}">
      <f:convertDateTime pattern="yyyy-MM-dd" />
     </h:inputText>
    </h:outputLabel>
    <h:message for="dateOfDeath" errorClass="error" />
   </div>
   <div>
    <h:commandButton value="Create"
       action="#{authorCtrl.create( author.personId, author.name,
       author.dateOfBirth, author.dateOfDeath)}"/>
   </div>
  </h:form>
 </ui:define>
```

```
</ui:composition>
```

The controller method `checkDateOfDeath` has to invoke the Bean Validation API validator, catch the validation exceptions and translate them to exceptions of type `javax.faces.validator.ValidatorException`, which are then managed by JSF and displayed in the view. Its code is as follows:

```
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 with the help of the `context.getViewRoot().findComponent` 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 Bean 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 Bean Validation class validator is not used, and the facelet is able to render the corresponding error messages when the validation fails, in the same way as is possible for single property validation situations.

8.2. JSF custom validators

An alternative approach to object-level validation is using JSF custom validators. They have the advantage that they are directly supported in facelets, but the downside of this approach is that it violates the onion architecture principle by defining business rules in the UI instead of defining them in the model..

For our example, the validator for the Author class that is responsible for validating `dateOfDeath` by comparing it with `dateOfBirth` is shown below:

```
@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 facelet, for the corresponding field, the validator has to be specified:

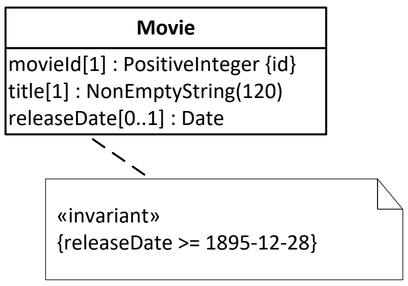
```
<h:outputLabel for="dateOfDeath" value="Date of death: " />
<h:inputText id="dateOfDeath" p:type="date"
  value="#{author.dateOfDeath}">
  <f:validator validatorId = "AuthorValidator" />
  <f:convertDateTime pattern="yyyy-MM-dd" />
</h:inputText>
<h:message for="dateOfDeath" errorClass="error" />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

9. Practice Project

If you have any questions about how to carry out the following projects, you can ask them on our discussion forum [http://web-engineering.info/forum/JavaJpaJsfApp].

9.1. Validate movie data

The purpose of the app to be built is managing information about movies. The app deals with just one object type: `Movie`, as depicted in the following class diagram.


Movie
movieId[1] : PositiveInteger {id}
title[1] : NonEmptyString(120)
releaseDate[0..1] : Date
«invariant»
{releaseDate >= 1895-12-28}

In this model, the following constraints have been expressed:

1. Due to the fact that the `movieId` attribute is declared to be the *standard identifier* of `Movie`, it is **mandatory** and **unique**.
2. The `title` attribute is **mandatory**, as indicated by its multiplicity expression [1], and has a **string length constraint** requiring its values to have at most 120 characters.
3. The `releaseDate` attribute has an **interval constraint**: it must be greater than or equal to 1895-12-28.

Notice that the `releaseDate` attribute is not mandatory, but *optional*, as indicated by its multiplicity expression [0..1]. In addition to the constraints described in this list, there are the implicit range constraints defined by assigning the datatype `PositiveInteger` to `movieId`, `NonEmptyString` to `title`, and `Date` to `releaseDate`.