JavaScript Front-End Web App Tutorial Part 2: Adding Constraint Validation

Learn how to build a front-end web application with responsive constraint validation using plain JavaScript

Gerd Wagner <G.Wagner@b-tu.de>

JavaScript Front-End Web App Tutorial Part 2: Adding Constraint Validation: Learn how to build a front-end web application with responsive constraint validation using plain JavaScript

by Gerd Wagner

Warning: This tutorial manuscript may contain errors and may still be incomplete. Please report any issue to Gerd Wagner at G.Wagner@b-tu.de.

This tutorial is also available in the following formats: PDF [validation-tutorial.pdf]. You may run the example app [ValidationApp/index.html] from our server, or download the code [ValidationApp.zip] as a ZIP archive file. See also our Web Engineering project page [http://web-engineering.info/].

Publication date 2015-11-12

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/JsFrontendApp] about model-based development of front-end web applications with plain JavaScript. It shows how to build a single-class front-end web application with constraint validation using plain JavaScript, and no third-party framework or library. While libraries and frameworks may help to increase productivity, they also create black-box dependencies and overhead, and they are not good for learning how to do it yourself.

A front-end web application can be provided by any web server, but it is executed on the user's computer device (smartphone, tablet or notebook), and not on the remote web server. Typically, but not necessarily, a front-end web application is a single-user application, which is not shared with other users.

The *minimal* JavaScript app that we have discussed in the first part of this 6-part tutorial has been limited to support the minimum functionality of a data management app only. However, it did not take care of preventing 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 JavaScript *model class*, and how to perform constraint validation both in the model/storage code of the app and in the user interface built with HTML5.

The simple form of a JavaScript 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**etrieve/**U**pdate/**D**elete). It extends the minimal app discussed in the Minimal App Tutorial [minimal-tutorial.html] by adding ***constraint validation*** (and some CSS styling), 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**, such as the 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, also assigning books to authors and to publishers.
- Part 6 [subtyping-tutorial.html]: Handling **subtype** (inheritance) relationships between object types.

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, if they have already died, 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 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 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	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 with the help of invariants 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 either a non-negative integer or the special value `*`, standing for unbounded, and the upper multiplicity `u` is either a positive integer not smaller than `l` or the special value `*`. 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 below.

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, integrity constraints and data validation are not supported at all in common programming languages such as PHP, Java, C# or JavaScript. It is therefore important to choose an application development framework that provides sufficient support for constraint validation. Notice that in HTML5, there is some support for validation of user input data in form fields.

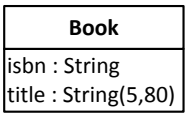
In the following sections, we discuss the different types of property constraints listed above in more detail. We also show how to express them in ***UML*** class diagrams, in ***SQL*** table creation statements, in ***JavaScript*** model class definitions, as well as in the annotation-based frameworks *Java Persistency API (***JPA***)* and ***ASP.NET****'s DataAnnotations*.

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.

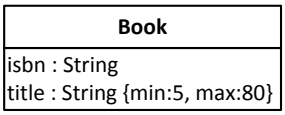
2.1. String Length Constraints

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

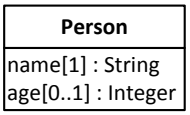


Another option is to use min/max constraint keywords in the property modifier list:



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 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:

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

```
  } else return "";  // no constraint violation
};
```

In JPA, the mandatory property `name` is annotated with `NotNull` in the following way:

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

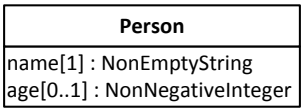
The equivalent ASP.NET's Data Annotation is `Required` as shown in

```
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 encode 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 JPA, for declaring empty strings as non-admissible we must set the context parameter

```
javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
```

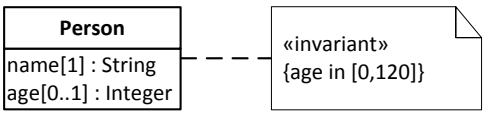
to `true` in the web deployment descriptor file `web.xml`.

In ASP.NET's Data Annotations, 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 [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



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 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:

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

InJPA, 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 Person {
    @NotNull
    private String name;
    @Min(0)@Max(120)
    private int age;
}
```

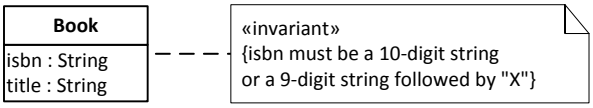
The equivalent ASP.NET's Data Annotation is `Range(0,120)` as shown in

```
public class Person{
    [Required]
    public string name { get; set; }
    [Range(0,120)]
    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 Figure 1.2 below.

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



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 encode 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 'X'!";
  } else return "";
};
```

In JPA, 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;
}
```

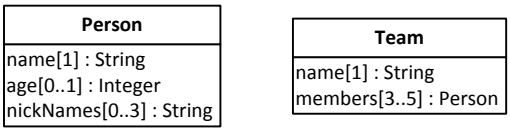
The equivalent ASP.NET's Data Annotation is `RegularExpression` as shown in

```
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 Figure 1.3 below, which contains two examples of properties with cardinality constraints.

Figure 1.3. Two object types 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 (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:

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

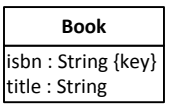
With Java Bean Validation annotations, we can specify

```
@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 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 Figure 1.4 below.

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



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:

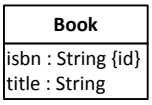
```
CREATE TABLE books(
  isbn   VARCHAR(10) NOT NULL UNIQUE,
  title  VARCHAR(50) NOT NULL
)
```

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 the property modifier `id` appended to the declaration of the attribute `isbn` as shown in Figure 1.5 below.

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



Notice that such 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 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

```
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 6 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 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 in response to the form 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 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?

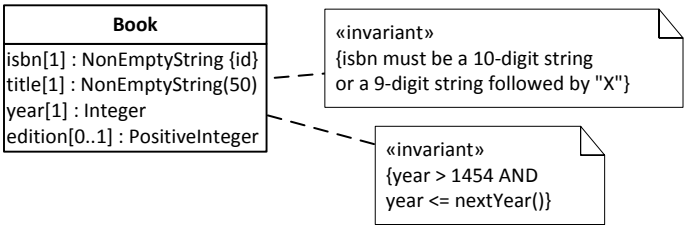
Chapter 2. Constraint Validation in a Plain JS Front-End App

1. Introduction

The *minimal* JavaScript front-end web application 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 JavaScript *model class*, and how to perform constraint validation both in the *model* part of the app and in the user interface built with HTML5.

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 object type `Book` and two invariants



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 calendaric 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 encoded in the model class within property-specific *check* functions.

The meaning of the design model can be illustrated by a sample data population for the model class `Book`:

Table 2.1. 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	

2. Using the HTML5 Form Validation API

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 are marked as valid.

Notice that in our general approach there is no need to use the new HTML5 attributes for validation, such as `required`, since we do all validations with the help of `setCustomValidity` and our property check functions, as we explain below.

3. New Issues

Compared to the minimal app [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/MinimalApp/index.html] discussed in Part 1 (Minimal App Tutorial [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/minimal-tutorial.html]) we have to deal with a number of new issues:

1. In the *model* code we have to take care of

   a. adding for every property of a class a ***check*** function that can be invoked for validating the constraints defined for the property, and a ***setter*** method that invokes the check function and is to be used for setting the value of the property,

   b. performing validation before any data is saved.

2. In the *user interface "(vie"w)* code we have to take care of

   a. styling the user interface with CSS rules,

   b. ***responsive validation*** on user input for providing immediate feedback to the user,

   c. validation on form submission for preventing the submission of flawed data to the model layer.

   For improving the break-down of the view code, we introduce a utility method (in `lib/util.js`) that fills a `select` form control with `option` elements the contents of which is retrieved from an entity table such as `Book.instances`. This method is used in the `setupUserInterface` method of both the `updateBook` and the `deleteBook` use cases.

Checking the constraints in the user interface on user input is important for providing immediate feedback to the user. But it is not safe enough to perform constraint validation only in the user interface, because this could be circumvented in a distributed web application where the user interface runs in the web browser of a front-end device while the application's data is managed by a back-end component on a remote web server. Consequently, we need multiple constraint validation, first in the user interface *on input* (or *on change*) and *on form submission*, and subsequently in the model layer before saving/

sending data to the persistent datastore. And in an application based on a DBMS we may also use a third round of validation before persistent storage by using the validation mechanisms of the DBMS. This is a must, when the application's database is shared with other apps.

Our solution to this **multiple validation problem** is to keep the constraint validation code in special *check* functions in the model classes and invoke these functions both in the user interface on user input and on form submission, as well as in the *create* and *update* data management methods of the model class via invoking the setters. Notice that *referential integrity* constraints (and other relationship constraints) may also be violated through a *delete* operation, but in our single-class example we don't have to consider this.

4. Make a JavaScript Data Model

Using the information design model shown in Figure 2.1 above as the starting point, we make a *JavaScript* data model by performing the following steps:

1. Create a ***check*** operation for each non-derived property in order to have a central place for implementing all the constraints that have been defined for a property in the design model. For a standard identifier (or *primary key*) attribute, such as `Book::isbn`, two check operations are needed:

   a. A check operation, such as `checkIsbn`, for checking all basic constraints of an identifier attribute, except the *mandatory value* and the *uniqueness* constraints.

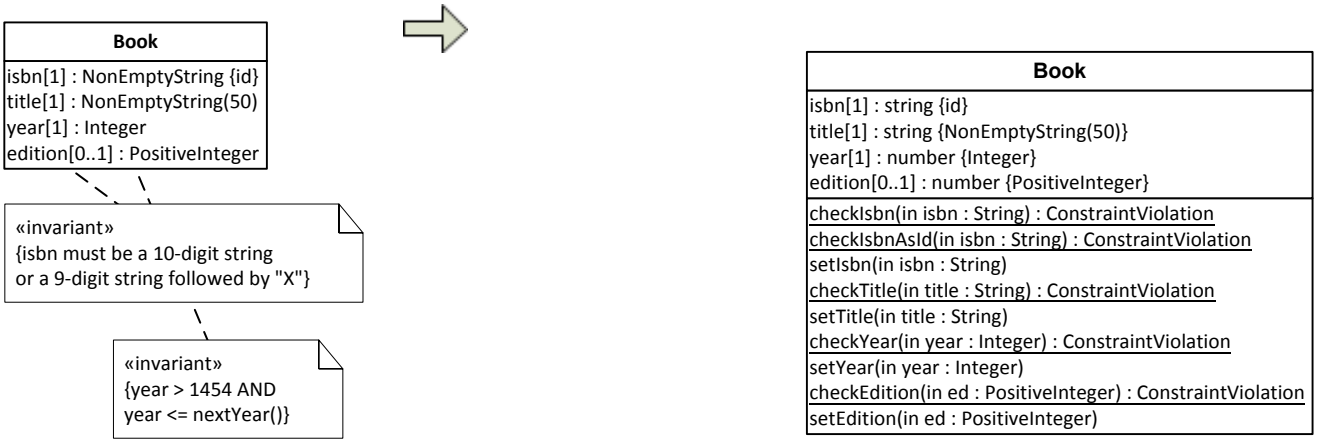
   b. A check operation, such as `checkIsbnAsId`, for checking in addition to the basic constraints the *mandatory value* and *uniqueness* constraints that are required for an identifier attribute.

   The `checkIsbnAsId` operation is invoked on user input for the `isbn` form field in the *create book* form, and also in the `setIsbn` method, while the `checkIsbn` operation can be used for testing if a value satisfies the syntactic constraints defined for an ISBN.

2. Create a **setter** operation for each non-derived **single-valued** property. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.

This leads to the *JavaScript data model* shown on the right hand side of the mapping arrow in the following figure.

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



Essentially, the JavaScript data model extends the design model by adding checks and setters for each property. The attached invariants have been dropped since they are taken care of in the checks. Property ranges have been turned into JavaScript datatypes (with a reminder to their real range in curly braces). Notice that the names of check functions are underlined, since this is the convention in UML for class-level (as opposed to instance-level) operations.

5. Set up the Folder Structure with Library Files

The MVC folder structure of our validation app extends the structure of the minimal app by adding two folders, `css` for adding the CSS file `main.css` and `lib` for adding the generic code libraries `browserShims.js`, `errorTypes.js` and `util.js`. Thus, we end up with the following folder structure containing five initial files:

```
publicLibrary
  css
    main.css
  lib
    browserShims.js
    errorTypes.js
    util.js
  src
    ctrl
    model
    view
  index.html
```

We discuss the contents of the four initial files in the following sections.

5.1. Style the user interface with CSS

We style the UI with the help of the CSS library Pure [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 for `table` and `menu` elements in `main.css`.

5.2. Provide general utility functions and JavaScript fixes in library files

We add three library files to the `lib` folder:

1. `browserShims.js` contains a definition of the string trim function for older browsers that don't support this function (which was only added to JavaScript in ECMAScript Edition 5, defined in 2009). More browser shims for other recently defined functions, such as `querySelector` and `classList`, could also be added to `browserShims.js`.

2. `util.js` contains the definitions of a few utility functions such as `isNonEmptyString(x)` for testing if `x` is a non-empty string.

3. `errorTypes.js` defines general classes for error (or exception) types: NoConstraintViolation, MandatoryValueConstraintViolation, RangeConstraintViolation, IntervalConstraintViolation, PatternConstraintViolation, UniquenessConstraintViolation, OtherConstraintViolation.

5.3. Create a start page

The start page of the app first takes care of the page styling by loading the *Pure CSS* base file (from the Yahoo site) and our `main.css` file with the help of the two `link` elements (in lines 6 and 7), then it loads the following JavaScript files (in lines 8-12):

1. `browserShims.js` and `util.js` from the `lib` folder, discussed in Section 5.2,
2. `initialize.js` from the `src/ctrl` folder, defining the app's MVC namespaces, as discussed in Part 1 (the *minimal app tutorial*).
3. `errorTypes.js` from the `lib` folder, defining exception classes.
4. `Book.js` from the `src/model` folder, a model class file that provides data management and other functions discussed in Section 6.

Figure 2.3. The validation app's start page `index.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Plain JS Validation App</title>
    <link rel="stylesheet" type="text/css"
        href="http://yui.yahooapis.com/combo?pure/0.6.0/base-min.css" />
    <link rel="stylesheet" type="text/css" href="css/main.css" />
    <script src="lib/browserShims.js"></script>
    <script src="lib/util.js"></script>
    <script src="lib/errorTypes.js"></script>
    <script src="src/ctrl/initialize.js"></script>
    <script src="src/model/Book.js"></script>
  </head>
  <body>
    <h1>Public Library</h1> <h2>Validation Example App</h2>
    <p>This app supports the following operations:</p>
    <menu>
      <li><a href="listBooks.html"><button type="button">
          List all books</button></a></li>
      <li><a href="createBook.html"><button type="button">
          Add a new book</button></a></li>
      <li><a href="updateBook.html"><button type="button">
          Update a book</button></a></li>
      <li><a href="deleteBook.html"><button type="button">
          Delete a book</button></a></li>
      <li><button type="button" onclick="Book.clearData()">
          Clear database</button></li>
      <li><button type="button" onclick="Book.createTestData()">
          Create test data</button></li>
    </menu>
</body>
</html>
```

6. Write the Model Code

How to Encode a JavaScript Data Model

The JavaScript 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 layer of our JavaScript front-end app. These steps are summarized in the following section.

6.1. Summary

1. Encode the model class as a JavaScript constructor function.

2. ***Encode the check functions***, such as `checkIsbn` or `checkTitle`, in the form of class-level ('static') methods. Take care that all constraints, as specified in the JavaScript data model, are properly encoded in the check functions.

3. ***Encode the setter operations***, such as `setIsbn` or `setTitle`, as (instance-level) methods. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.

4. Encode the add and remove operations, if there are any, as instance-level methods.

5. Encode any other operation.

These steps are discussed in more detail in the following sections.

6.2. Encode the model class as a constructor function

The class `Book` is encoded by means of a corresponding JavaScript *constructor function* with the same name `Book` such that all its (non-derived) properties are supplied with values from corresponding key-value slots of a `slots` parameter.

```
function Book( slots) {
  // assign default values
  this.isbn = "";   // string
  this.title = "";  // string
  this.year = 0;    // number (int)
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    if (slots.edition) {  // optional
      this.setEdition( slots.edition);
    }
  }
};
```

In the constructor body, we first assign default values to the class properties. These values will be used when the constructor is invoked as a default constructor (without arguments), or when it is invoked with only some arguments. It is helpful to indicate the range of a property in a comment. This requires to map the platform-independent data types of the information design model to the corresponding implicit JavaScript data types according to the following table.

Table 2.2. Datatype mapping

Platform-independent datatype	JavaScript datatype
String	string
Integer	number (int)
Decimal	number (float)
Boolean	boolean
Date	Date

Since the setters may throw constraint violation errors, the constructor function, and any setter, should be called in a try-catch block where the catch clause takes care of processing errors (at least logging suitable error messages).

As in the minimal app, we add a class-level property `Book.instances` representing the collection of all `Book` instances managed by the app in the form of an entity table:

```
Book.instances = {};
```

6.3. Encode the property checks

Encode the property check functions in the form of class-level ('static') methods. In JavaScript, this means to define them as function slots of the constructor, as in `Book.checkIsbn`. Take care that all constraints of a property as specified in the data model are properly encoded in its check function. This concerns, in particular, the *mandatory value* and *uniqueness* constraints implied by the *standard identifier* declaration (with «`stdid`»), and the *mandatory value* constraints for all properties with multiplicity 1, which is the default when no multiplicity is shown. If any constraint is violated, an error object instantiating one of the error classes listed above in Section 5.3 and defined in the file `model/errorTypes.js` is returned.

For instance, for the `checkIsbn` operation we obtain the following code:

```
Book.checkIsbn = function (id) {
  if (!id) {
    return new NoConstraintViolation();
  } else if (typeof(id) !== "string" || id.trim() === "") {
    return new RangeConstraintViolation(
        "The ISBN must be a non-empty string!");
  } else if (!/\b\d{9}(\d|X)\b/.test( id)) {
    return new PatternConstraintViolation(
        "The ISBN must be a 10-digit string or "+
        " a 9-digit string followed by 'X'!");
  } else {
    return new NoConstraintViolation();
  }
};
```

Notice that, since `isbn` is the standard identifier attribute of `Book`, we only check the syntactic constraints in `checkIsbn`, but we check the *mandatory value* and *uniqueness* constraints in `checkIsbnAsId`, which itself first invokes `checkIsbn`:

```
Book.checkIsbnAsId = function (id) {
  var constraintViolation = Book.checkIsbn( id);
  if ((constraintViolation instanceof NoConstraintViolation)) {
    if (!id) {
      constraintViolation = new MandatoryValueConstraintViolation(
          "A value for the ISBN must be provided!");
    } else if (Book.instances[id]) {
      constraintViolation = new UniquenessConstraintViolation(
          "There is already a book record with this ISBN!");
    } else {
      constraintViolation = new NoConstraintViolation();
    }
  }
  return constraintViolation;
};
```

6.4. Encode the property setters

Encode the setter operations as (instance-level) methods. In the setter, the corresponding check function is invoked and the property is only set, if the check does not detect any constraint violation. Otherwise, the *constraint violation* error object returned by the check function is thrown. For instance, the `setIsbn` operation is encoded in the following way:

```
Book.prototype.setIsbn = function (id) {
  var validationResult = Book.checkIsbnAsId( id);
```

```
  if (validationResult instanceof NoConstraintViolation) {
    this.isbn = id;
  } else {
    throw validationResult;
  }
};
```

There are similar setters for the other properties (`title`, `year` and `edition`).

6.5. Add a serialization function

It is helpful to have an object serialization function tailored to the structure of an object (as defined by its class) such that the result of serializing an object is a human-readable string representation of the object showing all relevant information items of it. By convention, these functions are called `toString()`. In the case of the `Book` class, we use the following code:

```
Book.prototype.toString = function () {
  return "Book{ ISBN:" + this.isbn + ", title:" +
      this.title + ", year:" + this.year +"}";
};
```

6.6. Data management operations

In addition to defining the model class in the form of a constructor function with property definitions, checks and setters, as well as a `toString()` serialization function, we also need to define the following data management operations as class-level methods of the model class:

1. `Book.convertRow2Obj` and `Book.loadAll` for loading all managed Book instances from the persistent data store.

2. `Book.saveAll` for saving all managed Book instances to the persistent data store.

3. `Book.add` for creating a new Book instance and adding it to the collection of all Book instances.

4. `Book.update` for updating an existing Book instance.

5. `Book.destroy` for deleting a Book instance.

6. `Book.createTestData` for creating a few example book records to be used as test data.

7. `Book.clearData` for clearing the book data store.

All of these methods essentially have the same code as in our *minimal app* discussed in Part 1, except that now

1. we may have to catch constraint violations in suitable try-catch blocks in the procedures `Book.convertRow2Obj`, `Book.add`, `Book.update` and `Book.createTestData`; and

2. we can use the `toString()` function for serializing an object in status and error messages.

Notice that for the change operations `add` (create) and `update`, we need to implement an all-or-nothing policy: whenever there is a constraint violation for a property, no new object must be created and no (partial) update of the affected object must be performed.

When a constraint violation is detected in one of the setters called when `new Book(...)` is invoked in `Book.add`, the object creation attempt fails, and instead a constraint violation error message is created

in line 6. Otherwise, the new book object is added to `Book.instances` and a status message is created in lines 10 and 11, as shown in the following program listing:

```
Book.add = function (slots) {
  var book = null;
  try {
    book = new Book( slots);
  } catch (e) {
    console.log( e.constructor.name +": "+ e.message);
    book = null;
  }
  if (book) {
    Book.instances[book.isbn] = book;
    console.log( book.toString() + " created!");
  }
};
```

Likewise, when a constraint violation is detected in one of the setters invoked in `Book.update`, a constraint violation error message is created (in line 16) and the previous state of the object is restored (in line 19). Otherwise, a status message is created (in lines 23 or 25), as shown in the following program listing:

```
Book.update = function (slots) {
  var book = Book.instances[slots.isbn],
      noConstraintViolated = true,
      updatedProperties = [],
      objectBeforeUpdate = util.cloneObject( book);
  try {
    if (book.title !== slots.title) {
      book.setTitle( slots.title);
      updatedProperties.push("title");
    }
    if (book.year !== parseInt( slots.year)) {
      book.setYear( slots.year);
      updatedProperties.push("year");
    }
    if (slots.edition && book.edition !== parseInt(slots.edition)) {
      book.setEdition( slots.edition);
      updatedProperties.push("edition");
    }
  } catch (e) {
    console.log( e.constructor.name +": "+ e.message);
    noConstraintViolated = false;
    // restore object to its state before updating
    Book.instances[slots.isbn] = objectBeforeUpdate;
  }
  if (noConstraintViolated) {
    if (updatedProperties.length > 0) {
      console.log("Properties " + updatedProperties.toString() +
          " modified for book " + slots.isbn);
    } else {
      console.log("No property value changed for book " +
          slots.isbn + " !");
    }
  }
};
```

7. The View and Controller Layers

The user interface (UI) consists of a start page `index.html` that allows the user choosing one of the data management operations by navigating to the corresponding UI page such as `listBooks.html` or `createBook.html` in the app folder. The start page `index.html` has been discussed in Section 5.3.

After loading the Pure [http://purecss.io/] base stylesheet and our own CSS settings in `main.css`, we first load some browser shims and utility functions. Then we initialize the app in `src/ctrl/`

`initialize.js` and continue loading the error classes defined in `lib/errorTypes.js` and the model class `Book`.

We render the data management menu items in the form of buttons. For simplicity, we invoke the `Book.clearData()` and `Book.createTestData()` methods directly from the buttons' `onclick` event handler attribute. Notice, however, that it is generally preferable to register such event handling functions with `addEventListener(...)`, as we do in all other cases.

7.1. The data management UI pages

Each data management UI page loads the same basic CSS and JavaScript files like the start page `index.html` discussed above. In addition, it loads two use-case-specific view and controller files `src/view/`*`useCase`*`.js` and `src/ctrl/`*`useCase`*`.js` and then adds a use case initialize function (such as `pl.ctrl.listBooks.initialize`) as an event listener for the page load event, which takes care of initializing the use case when the UI page has been loaded (see Section 7.3).

For the "list books" use case, we get the following code in `listBooks.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>JS Front-End Validation App Example</title>
  <link rel="stylesheet"
        href="http://yui.yahooapis.com/pure/0.3.0/pure-min.css" />
  <link rel="stylesheet" href="css/main.css" />
  <script src="lib/browserShims.js"></script>
  <script src="lib/util.js"></script>
  <script src="lib/errorTypes.js"></script>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/listBooks.js"></script>
  <script src="src/ctrl/listBooks.js"></script>
  <script>
    window.addEventListener("load", pl.ctrl.listBooks.initialize);
  </script>
</head>
<body>
  <h1>Public Library: List all books</h1>
  <table id="books">
    <thead>
      <tr><th>ISBN</th><th>Title</th><th>Year</th><th>Edition</th></tr>
    </thead>
    <tbody></tbody>
  </table>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

For the "create book" use case, we get the following code in `createBook.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>JS Front-End Validation App Example</title>
  <link rel="stylesheet"
        href="http://yui.yahooapis.com/combo?pure/0.3.0/base-
              min.css&pure/0.3.0/forms-min.css" />
  <link rel="stylesheet" href="css/main.css" />
  <script src="lib/browserShims.js"></script>
  <script src="lib/util.js"></script>
  <script src="lib/errorTypes.js"></script>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/createBook.js"></script>
```

```
  <script src="src/ctrl/createBook.js"></script>
  <script>
    window.addEventListener("load", pl.ctrl.createBook.initialize);
  </script>
  </head>
  <body>
  <h1>Public Library: Create a new book record</h1>
  <form id="Book" class="pure-form pure-form-aligned">
    <div class="pure-control-group">
      <label for="isbn">ISBN</label>
      <input id="isbn" name="isbn" />
    </div>
    <div class="pure-control-group">
      <label for="title">Title</label>
      <input id="title" name="title" />
    </div>
    <div class="pure-control-group">
      <label for="year">Year</label>
      <input id="year" name="year" />
    </div>
    <div class="pure-control-group">
      <label for="edition">Edition</label>
      <input id="edition" name="edition" />
    </div>
    <div class="pure-controls">
      <p><button type="submit" name="commit">Save</button></p>
      <nav><a href="index.html">Back to main menu</a></nav>
    </div>
  </form>
</body>
</html>
```

Notice that for styling the form elements in `createBook.html`, and also for `updateBook.html` and `deleteBook.html`, we use the Pure [http://purecss.io/] CSS form styles. This requires to assign specific values, such as "pure-control-group", to the `class` attributes of the form's `div` elements containing the form controls. We have to use explicit labeling (with the `label` element's `for` attribute referencing the `input` element's `id`), since *Pure* does not support implicit labels where the `label` element contains the `input` element.

7.2. Initialize the app

For initializing the app, its namespace and MVC subnamespaces have to be defined. For our example app, the main namespace is defined to be `pl`, standing for "Public Library", with the three subnamespaces `model`, `view` and `ctrl` being initially empty objects:

```
var pl = { model:{}, view:{}, ctrl:{} };
```

We put this code in the file `initialize.js` in the `ctrl` folder.

7.3. Initialize the data management use cases

For initializing a data management use case, the required data has to be loaded from persistent storage and the UI has to be set up. This is performed with the help of the controller procedures `pl.ctrl.createBook.initialize` and `pl.ctrl.createBook.loadData` defined in the controller file `ctrl/createBook.js` with the following code:

```
pl.ctrl.createBook = {
  initialize: function () {
    pl.ctrl.createBook.loadData();
    pl.view.createBook.setupUserInterface();
  },
  loadData: function () {
    Book.loadAll();
```

```
  }
};
```

All other data management use cases (read/list, update, delete) are handled in the same way.

7.4. Set up the user interface

For setting up the user interfaces of the data management use cases, we have to distinguish the case of "list books" from the other ones (create, update, delete). While the latter ones require using an HTML form and attaching event handlers to form controls, in the case of "list books" we only have to render a table displaying all the books, as shown in the following program listing of `view/listBooks.js`:

```
pl.view.listBooks = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("table#books>tbody");
    var i=0, book=null, row={}, key="",
        keys = Object.keys( Book.instances);
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      book = Book.instances[key];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = book.isbn;
      row.insertCell(-1).textContent = book.title;
      row.insertCell(-1).textContent = book.year;
      if (book.edition) {
        row.insertCell(-1).textContent = book.edition;
      }
    }
  }
};
```

For the *create*, *update* and *delete* use cases, we need to attach the following event handlers to form controls:

1. a function, such as `handleSubmitButtonClickEvent`, for handling the event when the user clicks the save/submit button,

2. functions for validating the data entered by the user in form fields (if there are any).

In addition, in line 28 of the following `view/createBook.js` code, we add an event handler for saving the application data in the case of a `beforeunload` event, which occurs, for instance, when the browser (or browser tab) is closed:

```
pl.view.createBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit;
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    formEl.isbn.addEventListener("input", function () {
        formEl.isbn.setCustomValidity(
            Book.checkIsbnAsId( formEl.isbn.value).message);
    });
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.checkTitle( formEl.title.value).message);
    });
    formEl.year.addEventListener("input", function () {
        formEl.year.setCustomValidity(
            Book.checkYear( formEl.year.value).message);
    });
    formEl.edition.addEventListener("input", function () {
        formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    });
```

```
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
        e.preventDefault();;
        formEl.reset();
    });
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  handleSubmitButtonClickEvent: function () {
    ...
  }
};
```

Notice that for each form input field we add a listener for `input` events, such that on any user input a validation check is performed because `input` events are created by user input actions such as typing. We use the predefined function `setCustomValidity` from the HTML5 form validation API for having our property check functions invoked on the current value of the form field and returning an error message in the case of a constraint violation. So, whenever the string represented by the expression `Book.checkIsbn( formEl.isbn.value).message` is empty, everything is fine. Otherwise, if it represents an error message, the browser indicates the constraint violation to the user by rendering a red outline for the form field concerned (due to our CSS rule for the `:invalid` pseudo class).

While the validation on user input enhances the usability of the UI by providing immediate feedback to the user, validation on form data submission is even more important for catching invalid data. Therefore, the event handler `handleSubmitButtonClickEvent()` performs the property checks again with the help of `setCustomValidity`, as shown in the following program listing:

```
handleSubmitButtonClickEvent: function () {
  var formEl = document.forms['Book'];
  var slots = { isbn: formEl.isbn.value,
          title: formEl.title.value,
          year: formEl.year.value,
          edition: formEl.edition.value
  };
  // set error messages in case of constraint violations
  formEl.isbn.setCustomValidity( Book.checkIsbnAsId( slots.isbn).message);
  formEl.title.setCustomValidity( Book.checkTitle( slots.title).message);
  formEl.year.setCustomValidity( Book.checkYear( slots.year).message);
  formEl.edition.setCustomValidity(
      Book.checkEdition( formEl.edition.value).message);
  // save the input data only if all of the form fields are valid
  if (formEl.checkValidity()) {
    Book.add( slots);
  }
}
```

By invoking `checkValidity()` on the form element, we make sure that the form data is only saved (by `Book.add`), if there is no constraint violation. After this `handleSubmitButtonClickEvent` handler has been executed on an invalid form, the browser takes control and tests if the pre-defined property `validity` has an error flag for any form field. In our approach, since we use `setCustomValidity`, the `validity.customError` would be true. If this is the case, the custom constraint violation message will be displayed (in a bubble) and the `submit` event will be suppressed.

For the use case *update book*, which is handled in `view/updateBook.js`, we provide a book selection list, so the user need not enter an identifier for books (an ISBN), but has to select the book to be updated. This implies that there is no need to validate the ISBN form field, but only the title and year fields. We get the following code:

```
pl.view.updateBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit,
        selectBookEl = formEl.selectBook;
```

```
    // set up the book selection list
    util.fillSelectWithOptions( Book.instances,
     selectBookEl, "isbn", "title");
    // when a book is selected, populate the form with its data
    selectBookEl.addEventListener("change", function () {
      var book=null, bookKey = selectBookEl.value;
      if (bookKey) {  // set form fields and reset CustomValidity
        book = Book.instances[bookKey];
        ["isbn","title","year","edition"].forEach( function (p) {
          formEl[p].value = book[p] !== undefined ? book[p] : "";
          formEl[p].setCustomValidity("");  // no error
        });
      } else {
        formEl.reset();
      }
    });
    ...  // add event listeners
  },
```

We also need to set up a number of event listeners, including input event listeners for responsive validation:

```
pl.view.updateBook = {
  setupUserInterface: function () {
    ...
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.checkTitle( formEl.title.value).message);
    });
    formEl.year.addEventListener("input", function () {
        formEl.year.setCustomValidity(
            Book.checkYear( formEl.year.value).message);
    });
    formEl.edition.addEventListener("input", function () {
        formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    });
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
        e.preventDefault();;
        formEl.reset();
    });
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
```

When the save button on the *update book* form is clicked, the `title` and `year` form field values are validated by invoking `setCustomValidity`, and then the book record is updated if the form data validity can be established with `checkValidity()`:

```
  handleSubmitButtonClickEvent: function () {
    var formEl = document.forms['Book'];
    var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        edition: formEl.edition.value
    };
    // set error messages in case of constraint violations
    formEl.title.setCustomValidity(
        Book.checkTitle( slots.title).message);
    formEl.year.setCustomValidity(
        Book.checkYear( slots.year).message);
    formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    if (formEl.checkValidity()) {
      Book.update( slots);
    }
  }
```

The logic of the `setupUserInterface` method for the *delete* use case is similar.

8. Run the App and Get the Code

You can run the validation app [ValidationApp/index.html] from our server or download the code [ValidationApp.zip] as a ZIP archive file.

9. Evaluation

We evaluate the approach presented in this chapter of the tutorial according to the criteria defined in the previous chapter.

Table 2.3. Evaluation

Evaluation criteria	Mark
custom validation function per class	0
custom validation function per property	1
direct support of mandatory value and range constraints	1
direct support of pattern, string length and interval constraints	1
direct support of cardinality, uniqueness and referential integrity constraints	1
model validation on assign and before save	1
generic validation error messages	0
responsive validation on input and on submit	1
two-fold validation	1
three-fold validation	n.a.
reporting database validation errors	n.a.
in total	80 %

10. Possible Variations and Extensions

10.1. Adding an object-level custom validation function

We can add a custom validation function `validate` to each model class, such that object-level validation (across two or more properties) can be performed before save

10.2. Simplifying forms with implicit labels

The explicit labeling of form fields used in this tutorial requires to add an `id` value to the `input` element and a `for`-reference to its `label` element as in the following example:

```
<div class="pure-control-group">
  <label for="isbn">ISBN:</label>
```

```
  <input id="isbn" name="isbn" />
</div>
```

This technique for associating a label with a form field is getting quite inconvenient when we have many form fields on a page because we have to make up a great many of unique `id` values and have to make sure that they don't conflict with any of the `id` values of other elements on the same page. It's therefore preferable to use an approach, called *implicit labeling*, that does not need all these `id` references. In this approach we make the `input` element a child element of its `label` element, as in

```
<div>
  <label>ISBN: <input name="isbn" /></label>
</div>
```

Having `input` as a child of its `label` doesn't seem very logical (rather, one would expect the `label` to be a child of an `input` element). But that's the way, it is defined in HTML5.

A small disadvantage of using implicit labels is the lack of support by popular CSS libraries, such as Pure CSS. In the following parts of this tutorial, we will use our own CSS styling for implicitly labeled form fields.

11. Points of Attention

11.1. Database size and memory management

Notice that in this tutorial, we have made the assumption that all application data can be loaded into main memory (like all book data is loaded into the map `Book.instances`). This approach only works in the case of local data storage of smaller databases, say, with not more than 2 MB of data, roughly corresponding to 10 tables with an average population of 1000 rows, each having an average size of 200 Bytes. When larger databases are to be managed, or when data is stored remotely, it's no longer possible to load the entire population of all tables into main memory, but we have to use a technique where only parts of the table contents are loaded.

11.2. Boilerplate code

Another issue with the do-it-yourself code of this example app is the *boilerplate code* needed

1. per model class for the storage management methods `add`, `update`, `destroy`, etc.;

2. per model class and property for getters, setters and validation checks.

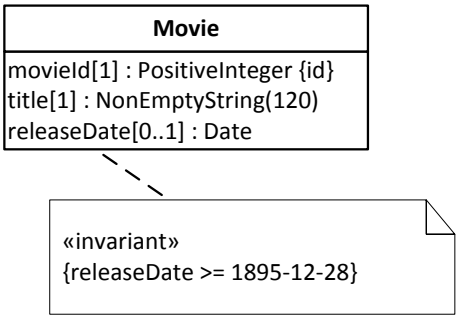
While it is good to write this code a few times for learning app development, you don't want to write it again and again later when you work on real projects. In another tutorial [mODELcLASS-validation-tutorial.html], we present an approach how to put these methods in a generic form in a meta-class called `mODELcLASS`, such that they can be reused in all model classes of an app.

12. Practice Project

The purpose of the app to be built is managing information about movies. Like in the book data management app discussed in the tutorial, you can make the simplifying assumption that all the data can be kept in main memory. Persistent data storage is implemented with JavaScript's Local Storage API.

The app deals with just one object type: `Movie`, as depicted in Figure 2.4 below. In the subsequent parts of the tutorial, you will extend this simple app by adding enumeration-valued attributes, as well as actors and directors as further model classes, and the associations between them.

Figure 2.4. The object type `Movie` defined with several constraints.



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`, as expressed by the property annotation `{id}` shown after the property range, 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`. In our plain JavaScript approach, all these property constraints are encoded in the model class within property-specific *check* functions.

Following the tutorial, you have to take care of

1. adding for every property a ***check*** function that validates the constraints defined for the property, and a ***setter*** method that invokes the check function and is to be used for setting the value of the property,

2. ***performing validation*** before any data is saved in the `Movie.add` and `Movie.update` methods.

in the *model* code of your app, while In the *user interface* ("view") code you have to take care of

1. styling the user interface with CSS rules (by integrating a CSS library such as Yahoo's Pure CSS),

2. ***validation on user input*** for providing immediate feedback to the user,

3. ***validation on form submission*** for preventing the submission of invalid data.

You can use the following sample data for testing your app:

Table 2.4. Sample data

Movie ID	Title	Release date
1	Pulp Fiction	1994-05-12
2	Star Wars	1977-05-25
3	Casablanca	1943-01-23

Movie ID	Title	Release date
4	The Godfather	1972-03-15

In this assignment, and in all further assignment, you have to make sure that your pages comply with the XML syntax of HTML5 (by means of XHTML5 validation), and that your JavaScript code complies with our Coding Guidelines [http://oxygen.informatik.tu-cottbus.de/webeng/Coding-Guidelines.html] and is checked with JSLint (http://www.jslint.com [http://www.jslint.com/]).

If you have any questions about how to carry out this project, you can ask them on our discussion forum [http://web-engineering.info/forum/12].