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

Publication date 2017-04-07
Copyright © 2014-2017 Gerd Wagner

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

Foreword ................................................................................................................................... vi

1. Integrity Constraints and Data Validation ................................................................................. 1
   1. Introduction .................................................................................................................... 1

2. Integrity Constraints ........................................................................................................ 1
   2.1. String Length Constraints ..................................................................................... 3
   2.2. Mandatory Value Constraints ................................................................................ 3
   2.3. Range Constraints ................................................................................................ 4
   2.4. Interval Constraints .............................................................................................. 5
   2.5. Pattern Constraints ............................................................................................... 5
   2.6. Cardinality Constraints ......................................................................................... 6
   2.7. Uniqueness Constraints ......................................................................................... 7
   2.8. Standard Identifiers (Primary Keys) ...................................................................... 7
   2.9. Referential Integrity Constraints ............................................................................ 8
   2.10. Frozen and Read-Only Value Constraints ............................................................ 8
   2.11. Beyond property constraints ................................................................................ 9

3. Responsive Validation ..................................................................................................... 9

4. Constraint Validation in MVC Applications ................................................................... 10

5. Adding Constraints to a Design Model ........................................................................... 11

2. Implementing Constraint Validation in a Plain JS Web App .................................................... 13
   1. Introduction .................................................................................................................. 13

2. New Issues ................................................................................................................... 13

3. Make a JavaScript Class Model ..................................................................................... 14

4. Set up the Folder Structure Adding Some Library Files ................................................... 15
   4.1. Provide general utility functions and JavaScript fixes in library files ...................... 15
   4.2. Create a start page .............................................................................................. 16

5. Write the Model Code ................................................................................................... 16
   5.1. Summary ........................................................................................................... 16
   5.2. Code the model class as a constructor function .................................................... 17
   5.3. Code the property checks ................................................................................... 17
   5.4. Code the property setters .................................................................................... 18
   5.5. Add a serialization function ................................................................................ 19
   5.6. Data management operations .............................................................................. 19

6. Write the View Code .................................................................................................... 21
   6.1. The data management UI pages .......................................................................... 21
   6.2. Initialize the app ................................................................................................ 21
   6.3. Set up the user interface ..................................................................................... 21

7. Run the App and Get the Code ...................................................................................... 24

8. Possible Variations and Extensions ................................................................................ 24
   8.1. Adding an object-level custom validation function ............................................... 24
   8.2. Implicit versus explicit form field labels ................................................................ 24

9. Points of Attention ........................................................................................................ 24
   9.1. Boilerplate code ................................................................................................. 24
   9.2. Configuring the UI for integrity maintenance ....................................................... 25
   9.3. Using implicit JS setters ..................................................................................... 25

10. Practice Project ........................................................................................................... 25
List of Figures

1.1. The object type Person with an interval constraint ................................................................. 5
1.2. The object type Book with a pattern constraint ..................................................................... 6
1.3. Two object types with cardinality constraints ........................................................................ 6
1.4. The object type Book with a uniqueness constraint ............................................................. 7
1.5. The object type Book with a standard identifier declaration ................................................... 8
1.6. The object type Book with a frozen value constraint ............................................................. 8
1.7. A platform-independent design model defining the object type Book with two invariants ...... 11
2.1. Deriving a JavaScript class model from an information design model ........................................ 15
2.2. Loading JavaScript files in the validation app's start page index.html ................................... 16
2.3. The object type Movie defined with several constraints. ....................................................... 26
List of Tables

1.1. Sample data for Book ........................................................................................................ 11
2.1. Data type mapping ............................................................................................................. 17
2.2. Sample data ....................................................................................................................... 26
Foreword

This tutorial is Part 2 of our series of six tutorials 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 (Create/Retrieve/Update/Delete). It extends the minimal app discussed in the Minimal App Tutorial 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 4 [http://web-engineering.info/tech/JsFrontendApp/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 [http://web-engineering.info/tech/JsFrontendApp/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 [http://web-engineering.info/tech/JsFrontendApp/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, 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. For those UI widgets, for which invalid user input cannot be prevented by limiting the input options, to detect and reject invalid user input as early as possible, by performing constraint validation in the UI.

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 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 pre-defined 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 \( 1..u \) where the lower multiplicity \( 1 \) is a non-negative integer and the upper multiplicity \( u \) is either a positive integer not smaller than \( 1 \) 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.

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,
Integrity Constraints
and Data Validation

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:

```
Book
| isbn : String |
| title : String[5,80] |
```

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

```
Book
| isbn : String |
| title : String{min:5, max:80} |
```

### 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*:

```
Person
| name[1] : String |
| age[0..1] : Integer |
```

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
};

In JPA, the mandatory property name is annotated withNotNull 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:

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

In JavaScript, we can 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 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 DataAnnotations, 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:

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

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

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

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

```java
@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

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

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

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

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

```javascript
Person.checkIsbn = function (id) {
  if (!/\b\d{9}([\d]|X)\b/.test( id)) {
    return "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:

```java
@Pattern(regexp="^[\d]{9}([\d]|X)$")
private String isbn;
```

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

```csharp
[Required]
[RegularExpression("^[\d]{9}([\d]|X)$")]
public string isbn { 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**

<table>
<thead>
<tr>
<th>Person</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>age[0..1] : Integer</td>
<td></td>
</tr>
<tr>
<td>nickNames[0..3] : String</td>
<td>members[3..5] : Person</td>
</tr>
</tbody>
</table>
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 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:

```javascript
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

```java
@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](image)

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:

```sql
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)

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

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn: String [id]</td>
</tr>
<tr>
<td>title: String</td>
</tr>
</tbody>
</table>

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:

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

In JavaScript, we cannot easily code 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 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 example shows how to define a frozen value constraint for the isbn attribute:

Figure 1.6. The object type Book with a frozen value constraint

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn: String [id, frozen]</td>
</tr>
<tr>
<td>title: String</td>
</tr>
</tbody>
</table>

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:

```javascript
Object.defineProperty( obj, "teamSize", {value: 5, writable: false, enumerable: true})
```
where the property slot `obj.teamSize` is made un-writable. 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:

```javascript
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
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

A general approach for supporting 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 `add` and in the `update` method.

### 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`, \ldots)
Integrity Constraints and Data Validation

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 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 in the back-end and front-end components.

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 1.7 below.

Figure 1.7. A platform-independent design model defining the object type Book with 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 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

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Year</th>
<th>Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>006251587X</td>
<td>Weaving the Web</td>
<td>2000</td>
<td>3</td>
</tr>
<tr>
<td>0465026567</td>
<td>Gödel, Escher, Bach</td>
<td>1999</td>
<td>2</td>
</tr>
<tr>
<td>ISBN</td>
<td>Title</td>
<td>Year</td>
<td>Edition</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------</td>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>0465030793</td>
<td>I Am A Strange Loop</td>
<td>2008</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2. Implementing Constraint Validation in a Plain JS Web App

1. Introduction

The minimal JavaScript front-end web application that we have discussed in the Minimal App Tutorial [http://web-engineering.info/tech/JsFrontendApp/minimal-tutorial.html] 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 (UI) built with HTML5.

We show how to perform responsive validation with the HTML5 form validation API [https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation]. Since using the new input field types and validation attributes (like required, min, max and pattern) implies defining constraints in the UI, they are not really useful in a best-practice approach where constraints are only checked, but not defined, in the UI.

Consequently, we will not use the new HTML5 validation features, but only use two methods of the HTML5 form validation API:

1. `setCustomValidity`, which 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;

2. `checkValidity`, which 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.

However, in the case of two special kinds of attributes, having calendar dates or colors as values, it is desirable to use the new UI widgets defined by HTML5 for picking a date or picking a color (with corresponding input field types). Unfortunately, in 2017, the HTML5 date picker widget is still not supported by all major browsers.

2. New Issues

Compared to the Minimal App [http://web-engineering.info/tech/JsFrontendApp/MinimalApp/index.html] discussed in the Minimal App Tutorial [http://web-engineering.info/tech/JsFrontendApp/minimal-tutorial.html] we have to deal with a number of new issues:

1. In the model code we have to add for every property of a class
   
   a. a check function that can be invoked for validating the constraints defined for the property, and

   b. a setter method that invokes the check function and is to be used for setting the value of the property.

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

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

   b. 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 data store. 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 proposed 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.

3. Make a JavaScript Class Model

Using the information design model shown in Figure 1.7 above as the starting point, we make a JavaScript class 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 attribute, such as Book::isbn, two check operations are needed:

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

   b. An extended check operation, such as checkIsbnAsId, for checking, in addition to the basic constraints, the mandatory value and uniqueness constraints that are required for a standard 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 class model shown on the right-hand side of the mapping arrow in the following figure.
Essentially, the JavaScript class 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.

4. Set up the Folder Structure Adding Some Library Files

The MVC folder structure of our validation app extends the structure of the minimal app by adding a lib folder containing the generic code libraries browserShims.js, errorTypes.js and util.js. Thus, we get the following folder structure:

```
<table>
<thead>
<tr>
<th>publicLibrary</th>
</tr>
</thead>
<tbody>
<tr>
<td>css</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>main.css</td>
</tr>
<tr>
<td>normalize.min.css</td>
</tr>
<tr>
<td>lib</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>browserShims.js</td>
</tr>
<tr>
<td>errorTypes.js</td>
</tr>
<tr>
<td>util.js</td>
</tr>
<tr>
<td>src</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>m</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>index.html</td>
</tr>
</tbody>
</table>
```

We discuss the contents of the added files in the following sub-sections.

4.1. 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 classes for error (or exception) types corresponding to the basic types of property constraints discussed above: `StringLengthConstraintViolation`, `MandatoryValueConstraintViolation`, `RangeConstraintViolation`, `IntervalConstraintViolation`, `PatternConstraintViolation`, `UniquenessConstraintViolation`. In addition, a class `NoConstraintViolation` is defined for being able to return a validation result object in the case of no constraint violation.

4.2. Create a start page

The start page `index.html` takes care of loading CSS page styling files with the help of the following two `link` elements:

```html
<link rel="stylesheet" href="css/normalize.min.css">
<link rel="stylesheet" href="css/main.css">
```

Then it loads the following JavaScript files:

1. `browserShims.js` and `util.js` from the `lib` folder, discussed in Section 4.1,
2. `errorTypes.js` from the `lib` folder, defining exception classes.
3. `initialize.js` from the `src/c` folder, defining the app's MVC namespaces, as discussed in the *Minimal App Tutorial*.
4. `Book.js` from the `src/m` folder, a model class file that provides data management and other functions discussed in Section 5.

![Figure 2.2. Loading JavaScript files in the validation app's start page `index.html`.](image)

5. Write the Model Code

How to Code a JavaScript Class Model

The JavaScript class model shown on the right hand side in Figure 2.1 can be coded 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.

5.1. Summary

1. Code the model class as a JavaScript constructor function.
2. *Code 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 class model, are properly coded in the check functions.
3. *Code 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. Code the add and remove operations, if there are any, as instance-level methods.

5. Code any other operation.

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

5.2. Code the model class as a constructor function

The class `Book` is coded as a corresponding 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.

```javascript
function Book( slots) {
    // assign default values
    this.isbn = "";  // string
    this.title = "";  // string
    this.year = 0;   // number (int)
    // assign properties only if the constructor is invoked with an argument
    if (arguments.length > 0) {
        this.setIsbn( slots.isbn);
        this.setTitle( slots.title);
        this.setYear( slots.year);
        // optional property
        if (slots.edition) 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 2.1. Data type mapping**

<table>
<thead>
<tr>
<th>Platform-independent datatype</th>
<th>JavaScript datatype</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>string</td>
<td>CHAR(n) or VARCHAR(n)</td>
</tr>
<tr>
<td>Integer</td>
<td>number (int)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Decimal</td>
<td>number (float)</td>
<td>REAL, DOUBLE PRECISION or DECIMAL(p,s)</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>Date</td>
<td>Date</td>
<td>DATE</td>
</tr>
</tbody>
</table>

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:

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

5.3. Code the property checks

Code the property check functions in the form of class-level ('static') methods. In JavaScript, this means to define them as method slots of the constructor, as in `Book.checkIsbn` (recall that a constructor, is a JS function and a JS object, since in JavaScript, functions are objects, and as an object, it can have slots).
Take care that all constraints of a property as specified in the class model are properly coded in its
check function. This concerns, in particular, the mandatory value and uniqueness constraints implied by
the standard identifier declaration (with \{id\}), 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 4.1 and defined in the file
errorTypes.js is returned.

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

```javascript
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:

```javascript
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;
};
```

We assume that all check functions and setters can deal both with proper data values (that are of the
attribute's range type) and also with string values that are supposed to represent proper data values,
but have not yet been converted to the attribute's range type. This approach allows avoiding de-serialization
in the user interface ("view") code and performing it in the model layer, instead.

For instance, in our example app, we have the integer-valued attribute year. When the user has entered
a value for this attribute in a corresponding form field, in the Create or Update user interface, the form
field holds a string value. This value is passed to the Book.add or Book.update method, which
invokes the setYear and checkYear methods. Only after being validated, this string value is de-
serialized (converted to an integer) and assigned to the year attribute.

5.4. Code the property setters

Code 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 coded in the following way:

```javascript
Book.prototype.setIsbn = function (id) {
  var validationResult = Book.checkIsbnAsId(id);
  if ((validationResult instanceof NoConstraintViolation)) {
    ... // assign the validated ISSN to the isbn property
  } else {
    throw validationResult;
  }
};
```
Implementing Constraint Validation in a Plain JS Web App

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

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

### 5.5. 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:

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

### 5.6. 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.convertRec2Obj` and `Book.retrieveAll` 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.
5. `Book.destroy` for deleting a `Book` instance.
6. `Book.createTestData` for creating a few example book records to be used as test data.

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.convertRec2Obj`, `Book.add`, `Book.update` and `Book.createTestData`;
2. we create more informative status and error log messages for better observing what’s going on; and
3. 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. Otherwise, the new book object is added to `Book.instances` and a status message is created, as shown in the following program listing:
Implementing Constraint Validation in a Plain JS Web App

When an object of a model class is to be updated, we first create a clone of it for being able to restore it if the update attempt fails. In the object update attempt, we only assign those properties of the object the value of which has changed, and we report this in a status log.

Normally, all properties defined by a model class, except the standard identifier attribute, can be updated. It is, however, possible to also allow updating the standard identifier attribute. This requires special care for making sure that all references to the given object via its old standard identifier are updated as well.

When a constraint violation is detected in one of the setters invoked in `Book.update`, the object update attempt fails, and instead the error message of the constraint violation object thrown by the setter and caught in the `update` method is shown, and the previous state of the object is restored. Otherwise, a status message is created, as shown in the following program listing:

```javascript
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");
        }
            // slots.edition has a non-empty value that is new
            book.setEdition( slots.edition);
            updatedProperties.push("edition");
        } else if (!slots.edition && book.edition !== undefined) {
            // slots.edition has an empty value that is new
            delete book.edition;
            updatedProperties.push("edition");
        }
    } catch (e) {...}
    ...
};
```

Notice that optional properties, like `edition`, need to be treated in a special way. If the user doesn't enter any value for them in a `Create` or `Update` user interface, the form field's value is the empty string "". In the case of an optional property, this means that the property is not assigned a value in the `add` use case, or that it is `un-set` if it has had a value in the `update` use case. This is different from the case of a mandatory property, where the empty string value obtained from an empty form field may or may not be an admissible value.

If there is a constraint violation exception, an error message is written to the log and the object concerned is reset to its previous state:

```javascript
Book.update = function (slots) {
```
try {
  ...
} 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 + ":")
  }
}

6. Write the View Code

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 retrieveAndListAllBooks.html or createBook.html in the app folder. The start page index.html has been discussed in Section 4.2.

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.

6.1. 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 a use-case-specific view code file src/v/useCase.js and then sets the setupUserInterface procedure of the use-case as an event handler for the page load event, which takes care of initializing the use case when the UI page has been loaded.

6.2. Initialize the app

For initializing the app, its namespace and MVC sub-namespaces have to be defined. For our example app, the main namespace is defined to be pl, standing for "Public Library", with the three sub-namespaces m, v and c being initially empty objects:

```javascript
var pl = { m:{}, v:{}, c:{} };
```

We put this code in the file initialize.js in the c folder.

6.3. Set up the user interface

For setting up the user interfaces of the data management use cases, we have to distinguish the case of "Retrieve/List All" 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 "Retrieve/List All" we only have to render a table displaying all books, as in the case of the Minimal App [http://web-engineering.info/tech/JsFrontendApp/MinimalApp-with-CSS/index.html] discussed in Part 1 of this tutorial.

For the Create, Update and Delete use cases, we need to add event listeners for:
Implementing Constraint Validation in a Plain JS Web App

1. responsive validation on form field input events,

2. handling the event when the user clicks the save (or delete) button,

3. making sure the main memory data is saved when a beforeunload event occurs, that is, when the browser window/tab is closed.

For the use case Create, we obtain the following code (in `v/createBook.js`):

```javascript
pl.v.createBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        saveButton = formEl.commit;
    // load all book records
    Book.retrieveAll();
    // add event listeners for responsive validation
    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);
    });
    ...
  }
};
```

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).

In addition to the event handlers for responsive constraint validation, we need two more event handlers:

```javascript
pl.v.createBook = {
  setupUserInterface: function () {
    ...
    // Set an event handler for the submit/save button
    saveButton.addEventListener("click",
      pl.v.createBook.handleSaveButtonClickEvent);
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
      e.preventDefault();
      formEl.reset();
    });
    // Set a handler for the event when the browser window/tab is closed
    window.addEventListener("beforeunload", Book.saveAll);
  },
  handleSaveButtonClickEvent: function () {...}
};
```

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,
Implementing Constraint Validation in a Plain JS Web App

the event handler handleSaveButtonClickEvent() performs the property checks again with the help of setCustomValidity, as shown in the following program listing:

```javascript
handleSaveButtonClickEvent: function () {
  var formEl = document.forms['Book'];
  var slots = {isbn: formEl.isbn.value,
               title: formEl.title.value,
               year: formEl.year.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);
  if (formEl.edition.value) {
    slots.edition = formEl.edition.value;
    formEl.edition.setCustomValidity( Book.checkEdition( slots.edition).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 handleSaveButtonClickEvent 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.

In the user interface of the use case Update, which is handled in v/updateBook.js, we do not have an input, but rather an output field for the standard identifier attribute isbn, since it is not supposed to be modifiable. Consequently, we don't need to validate any user input for it. However, we need to set up a selection list (in the form of an HTML select element) allowing the user to select a learning unit in the first step, before its data can be modified. This requires to add a change event listener on the select element such that the fields of the UI can be filled with the data of the selected object.

```javascript
pl.v.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].setCustomValidity("");  // no error
        });
      } else formEl.reset();
    });
    // add event listeners for responsive validation
    ...
    // Set an event handler for the submit/save button
    ...
    // neutralize the submit event
    ...
    // Set a handler for the event when the browser window/tab is closed
    ...
  },
  handleSaveButtonClickEvent: function () {...}
};
```

There is no need to set up responsive validation for the standard identifier attribute isbn, but for all other form fields, as shown above for the Create use case.
The logic of the setupUserInterface method for the Delete use case is similar. We only need to take care that the object to be deleted can be selected by providing a selection list, like in the Update case. No validation is needed for the Delete case.

7. Run the App and Get the Code


8. Possible Variations and Extensions

8.1. 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.

8.2. Implicit versus explicit form field labels

The explicit labeling of form fields 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, where these id references are not needed. In this approach, the input element is 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.

9. Points of Attention

9.1. Boilerplate code

An 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 our mODELcLASSjs tutorial [http://web-
Implementing Constraint Validation in a Plain JS Web App

[243x797]engineering.info/tech/mODELcLASSjs/validation-tutorial.html], we present an approach how to put these methods in a generic form in a meta-class, such that they can be reused in all model classes of an app.

9.2. Configuring the UI for integrity maintenance

Some of the new HTML5 input field types (like number, date or color) are intended to allow web browsers rendering corresponding input elements in the form of UI widgets (like date or color pickers) that limit the user's input options such that only valid input is possible. In terms of usability, it's preferable to prevent users from entering invalid data instead of allowing to enter it and only then checking its validity and reporting errors. Unfortunately, and this is quite disappointing, many browsers do, in 2017, still not have reasonable UI widget implementations for these HTML5 input field types yet.

9.3. Using implicit JS setters

Since ES5, JavaScript has its own form of setters, which are implicit and allow having the same semantics as explicit setter methods, but with the simple syntax of direct access. In addition to having the advantage of a simpler syntax, implicit JS setters are also safer than explicit setters because they decrease the likelihood of a programmer circumventing a setter by using a direct property assignment when instead a setter should be used. In other OOP languages, like Java, this is prevented by declaring properties to be 'private'. But JavaScript does not have this option.

The following code defines implicit setter and getter methods for the property title:

```javascript
Object.defineProperty( Book.prototype, "title", { 
set: function(t) {
    var validationResult = Book.checkTitle( t);
    if (validationResult instanceof NoConstraintViolation) {
        this._title = t;
    } else {
        throw validationResult;
    }
},
get: function() {
    return this._title;
}
});
```

Notice that, also in the constructor definition, the internal property _title, used for storing the property value, is not used for setting/getting it, but rather the virtual property title:

```javascript
Book = function (slots) {
    this.learnUnitNo = 0;
    this.title = "";
    if (arguments.length > 0) {
        this.learnUnitNo = slots.learnUnitNo;
        this.title = slots.title;
        // optional property
        if (slots.subjectArea) this.subjectArea = slots.subjectArea;
    }
};
```

We will start using implicit setter and getter methods, along with ES6 class definitions, in Part 3.

10. 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 2.3 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 2.3. The object type Movie defined with several constraints.**

<table>
<thead>
<tr>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>movieId[1] : PositiveInteger {id}</td>
</tr>
<tr>
<td>title[1] : NonEmptyString(120)</td>
</tr>
<tr>
<td>releaseDate[0..1] : Date</td>
</tr>
</tbody>
</table>

«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, 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 coded 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 2.2. Sample data**

<table>
<thead>
<tr>
<th>Movie ID</th>
<th>Title</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pulp Fiction</td>
<td>1994-05-12</td>
</tr>
</tbody>
</table>
### Implementing Constraint Validation in a Plain JS Web App

<table>
<thead>
<tr>
<th>Movie ID</th>
<th>Title</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Star Wars</td>
<td>1977-05-25</td>
</tr>
<tr>
<td>3</td>
<td>Casablanca</td>
<td>1943-01-23</td>
</tr>
<tr>
<td>4</td>
<td>The Godfather</td>
<td>1972-03-15</td>
</tr>
</tbody>
</table>

In this project, and in all further projects, 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].