JavaScript Frontend Web App Tutorial Part 2: Adding Constraint Validation

A medium-level tutorial about developing a single-class frontend web application with constraint validation using plain JavaScript

Gerd Wagner <G.Wagner@b-tu.de>
JavaScript Frontend Web App Tutorial Part 2: Adding Constraint Validation: A medium-level tutorial about developing a single-class frontend web application with 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]. See also the project page [http://web-engineering.info], or run the example app [ValidationApp/index.html] from our server, or download the app as a ZIP archive file [ValidationApp.zip].

Publication date 2014-04-11
Copyright © 2014 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>vi</td>
</tr>
<tr>
<td>1. Integrity Constraints and Data Validation</td>
<td>1</td>
</tr>
<tr>
<td>1. String Length Constraints</td>
<td>2</td>
</tr>
<tr>
<td>2. Mandatory Value Constraints</td>
<td>2</td>
</tr>
<tr>
<td>3. Range Constraints</td>
<td>2</td>
</tr>
<tr>
<td>4. Interval Constraints</td>
<td>3</td>
</tr>
<tr>
<td>5. Pattern Constraints</td>
<td>4</td>
</tr>
<tr>
<td>6. Uniqueness Constraints</td>
<td>4</td>
</tr>
<tr>
<td>7. Standard Identifiers (Primary Keys)</td>
<td>4</td>
</tr>
<tr>
<td>8. Referential Integrity Constraints</td>
<td>5</td>
</tr>
<tr>
<td>9. Constraint Validation in MVC Applications</td>
<td>5</td>
</tr>
<tr>
<td>2. Implementing Constraint Validation in a JavaScript Frontend Web App</td>
<td>7</td>
</tr>
<tr>
<td>1. Using the HTML5 Form Validation API</td>
<td>7</td>
</tr>
<tr>
<td>2. New Issues</td>
<td>8</td>
</tr>
<tr>
<td>3. Make a JavaScript Data Model</td>
<td>8</td>
</tr>
<tr>
<td>4. Set up the folder structure and create four initial files</td>
<td>9</td>
</tr>
<tr>
<td>4.1. Style the user interface with CSS</td>
<td>10</td>
</tr>
<tr>
<td>4.2. Provide general utility functions and JavaScript fixes in library files</td>
<td>10</td>
</tr>
<tr>
<td>4.3. Create a start page</td>
<td>10</td>
</tr>
<tr>
<td>5. Write the Model Code</td>
<td>11</td>
</tr>
<tr>
<td>5.1. Summary</td>
<td>11</td>
</tr>
<tr>
<td>5.2. Encode the model class as a constructor function</td>
<td>12</td>
</tr>
<tr>
<td>5.3. Encode the property checks</td>
<td>12</td>
</tr>
<tr>
<td>5.4. Encode the property setters</td>
<td>13</td>
</tr>
<tr>
<td>5.5. Add a serialization function</td>
<td>13</td>
</tr>
<tr>
<td>5.6. Data management operations</td>
<td>14</td>
</tr>
<tr>
<td>6. The View and Controller Layers</td>
<td>15</td>
</tr>
<tr>
<td>6.1. The data management UI pages</td>
<td>15</td>
</tr>
<tr>
<td>6.2. Initialize the app</td>
<td>17</td>
</tr>
<tr>
<td>6.3. Initialize the data management use cases</td>
<td>17</td>
</tr>
<tr>
<td>6.4. Set up the user interface</td>
<td>17</td>
</tr>
<tr>
<td>7. Run the App and Get the Code</td>
<td>20</td>
</tr>
</tbody>
</table>
List of Figures

1.1. The object type Person with an interval constraint .................................................. 3
1.2. The object type Book with a pattern constraint ......................................................... 4
1.3. The object type Book with a uniqueness constraint .................................................... 4
1.4. The object type Book with a standard identifier declaration ..................................... 5
2.1. A platform-independent design model with the class Book and two invariants .......... 7
2.2. Deriving a JavaScript data model from an information design model ....................... 9
2.3. The validation app's start page index.html ................................................................. 11
List of Tables

2.1. Datatype mapping .................................................................................................................. 12
Foreword

This tutorial is Part 2 of our series of five tutorials about engineering a frontend web application with plain JavaScript. It shows how to build a single-class frontend web application with constraint validation using plain JavaScript (and no third-party framework or library). A frontend 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 frontend 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 5-part tutorial has been limited to support the minimum functionality of a data management app only. However, it did not take care of preventing the users from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a 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.

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/Read/Update/Delete). 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:

• Part 3 [unidirectional-association-tutorial.html]: Managing a unidirectional association between the object types Book and Publisher.

• Part 4 [bidirectional-association-tutorial.html]: Managing a bidirectional association between the object types Book and Author.

• Part 5 [subtypes-tutorial.html]: Handling subtype (inheritance) relationships between object types.

You may also want to take a look at Part 1 (our Minimal App Tutorial [minimal-tutorial.html] without CSS styling and without constraint validation), and at our Complete App Tutorial [tutorial.html], which includes all five parts of the tutorial in one document, dealing with multiple object types ("books", "publishers" and "authors") and taking care of constraint validation, associations and subtypes/inheritance.
Chapter 1. Integrity Constraints and Data Validation

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

Integrity constraints may take many different forms. For instance, property constraints define conditions on the admissible property values of an object of a certain 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 has a maximum, or a minimum.
- **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 an attribute's value must be in a specific interval.
- **Pattern Constraints** require that a string attribute's value must satisfy a certain pattern, typically defined by a regular expression.
- **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.

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.

Since integrity maintenance is fundamental in database management, the data definition language part of the relational database language SQL supports the definition of integrity constraints in various forms. On the other hand, however, there is hardly any support for integrity constraints and data validation in common programming languages such as PHP, Java, C# or JavaScript. It is therefore important to take a systematic approach to constraint validation in web application engineering and
to choose an application development framework that provides sufficient support for it. Notice that in HTML5, there is some support of data validation for user input 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 and, as an example of how to do it yourself in a programming language, we also show how to express them in JavaScript model class definitions, where we encode constraint validations in class-level ("static") check functions. Any systematic approach also requires to define a set of error (or 'exception') class definitions, including one for each of the standard property constraints listed above.

1. String Length Constraints

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

2. 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 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 ...; // error message "A name must be provided!"
    } else {
        ...
    }
};
```

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 to these attributes, such as NonNegativeInteger to age, and NonEmptyString to name. Notice that such more specific type definitions are neither predefined in SQL nor in common programming languages, so we have to implement them either by 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 our own datatypes and then use them in the definition of a property, as illustrated in the following diagram:

```plaintext
Person
name[1] : NonEmptyString
age[0..1] : NonNegativeInteger
```

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

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

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

### 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**

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

```
«invariant»
{age in [0,120]}
```

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

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

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

```javascript
Person.checkAge = function (a) {
```
```
if (a < 0 || a > 120) {
    return ...; // error message "Age must be between 1 and 120!";
} else {
    ...
}
```
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 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.

### 7. Standard Identifiers (Primary Keys)

An attribute (or, more generally, a combination of attributes) can be declared to be the standard identifier for objects of a given type, if it is mandatory and unique. We can indicate this in a UML class diagram with the help of a (user-defined) stereotype «stdid» assigned to the attribute `isbn` as shown in Figure 1.4 below.

**Figure 1.4. The object type Book with a standard identifier declaration**

```
<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn : String</td>
</tr>
<tr>
<td>title : String</td>
</tr>
</tbody>
</table>
```

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

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

```sql
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.

### 8. 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.

### 9. Constraint Validation in MVC Applications

Unfortunately, many MVC application development frameworks do not provide sufficient support for integrity constraints and data validation.

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? 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. This problem is well-known from classical web applications where the frontend component submits the user input data via HTML form submission to a backend component running on a remote web server. Only this backend component validates the data and returns the validation results in the form of a set of error messages to the front end. Only then, typically 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, the user should get immediate validation feedback on each single input data item.

So, we need a validation mechanism in the user interface (UI). Fortunately, the new HTML5 form validation API [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/] provides help (also the jQuery Validation Plugin [http://jqueryvalidation.org/] supports form validation). But it is not sufficient to perform data validation in the user interface. We also need to do it in the model classes, and in the DBMS, 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 places (at least in the model classes and in the UI, and possibly also in the database). I have called this the multiple validation problem in this stackoverflow post [http://stackoverflow.com/questions/22054347/how-to-avoid-duplicate-code-for-constraint-validation-in-the-model-and-view-of-a] where I also sketch a solution for JavaScript frontend apps. The multiple validation problem can be solved in different ways:

1. Define the constraints in a declarative language (such as Java and ASP.NET annotations) and generate the backend/model and frontend/UI validation code (typically, in a backend application programming language such as Java or C#, and in JavaScript).

2. Keep your validation functions in the model on the backend, and invoke them from the UI via XHR.

3. Use JavaScript as your backend application programming language (such as with NodeJS), then you can encode your validations in JavaScript and execute them on your backend or in the user interface on the frontend.

The simplest, and most responsisve, approach is the third one, using only JavaScript both in the backend and frontend components.
Chapter 2. Implementing Constraint Validation in a JavaScript Frontend Web App

The minimal JavaScript frontend web app that we have discussed in Part 1 has been limited to support the minimum functionality of a data management app only. For instance, it did not take care of preventing the user from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a 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 as shown in Figure 2.1 below.

![Figure 2.1. A platform-independent design model with the class Book and two invariants](image)

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

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

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

3. The title attribute is mandatory.

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

In addition to these constraints, there are the implicit range constraints defined by assigning the datatype NonEmptyString as range to isbn and title, and Integer to year. In our plain JavaScript approach, all these property constraints are encoded in the model class within property-specific check functions.

1. 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 to it. The second method, checkValidity, is invoked on a form and tests, if all input fields have a valid value.
Implementing Constraint Validation
in a JavaScript Frontend Web App

Notice that in our 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.

Constraint_validation] or this HTML5Rocks tutorial [http://www.html5rocks.com/en/tutorials/forms/
constraintvalidation/] for more about the HTML5 form validation API.

2. New Issues

Compared to the minimal app [http://oxygen.informatik.tu-cottbus.de/IT/JsFrontendApp/
MinimalApp/index.html] discussed in Part 1 (Minimal App Tutorial [https://oxygen.informatik.tu-
cottbus.de/IT/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 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,
   b. performing validation before any data is saved.

2. In the user interface ("view") code we have to take care of
   a. styling the user interface with CSS rules,
   b. 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 associative array 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 frontend device while the application's data is managed by a backend component
on a remote web server. Consequently, we need a two-fold validation of constraints, first in the user
interface, and subsequently in the model code responsible for data storage.

   Our solution to this 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 certain relationship (such as referential integrity) 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 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:
Implementing Constraint Validation in a JavaScript Frontend Web App

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` function is invoked on user input for the isbn form field in the create book form, and also in the `setIsbn` method, while the `checkIsbn` function 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.

3. Create add and remove operations for each non-derived multi-valued property (if there are any).

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**

The JavaScript data model extends the design model by adding checks and setters for each property. Notice that the names of check functions are underlined, since this is the convention in UML for class-level ("static") methods.

4. Set up the folder structure and create four initial 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` and `util.js`. Thus, we end up with the following folder structure containing four initial files:

```
publicLibrary
  css
```
Implementing Constraint Validation in a JavaScript Frontend Web App

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

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

4.1. 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.

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

We add two library files to the lib folder:

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

2. 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.

4.3. Create a start page

The start page of the app first takes care of the 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 4.2,

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 src/model folder, defining the following classes for error (or exception) types: NoConstraintViolation, MandatoryValueConstraintViolation, RangeConstraintViolation, IntervalConstraintViolation, PatternConstraintViolation, UniquenessConstraintViolation, OtherConstraintViolation.

4. Book.js from the src/model folder, a model class file that provides data management and other functions discussed in Section 5.
5. 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 frontend app. These steps are summarized in the following section.

5.1. 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 of the property, 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.

5. Encode any other operation.

These steps are discussed in more detail in the following sections.
5.2. 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.

```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);
    }
}
```

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>
<thead>
<tr>
<th>Platform-independent datatype</th>
<th>JavaScript datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>Integer</td>
<td>number (int)</td>
</tr>
<tr>
<td>Decimal</td>
<td>number (float)</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean</td>
</tr>
<tr>
<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 application in the form of an associative array:

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

5.3. 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 JavaScript 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 4.3 and defined in the file model/errorTypes.js is returned.

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

```javascript
Book.checkIsbn = function (id) {
```
Implementing Constraint Validation in a JavaScript Frontend Web App

```javascript
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 (!/\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;
};

5.4. 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:

```javascript
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 properties title and year.

5.5. Add a serialization function

It is helpful to have a serialization function tailored to the structure of a 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 +"}";
}

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() function, we also need to define the following data management operations as class-level methods of the model class:

1. Book.convertRow2Obj and Book.loadAllInstances for loading all managed Book instances from the persistent data store.
2. Book.saveAllInstances for saving all managed Book instances to the persistent data store.
5. Book.deleteRow 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.convertRow2Obj, Book.createRow, Book.updateRow 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 createRow and updateRow, we need to implement an all-or-nothing policy: as soon as 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.createRow, 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:

```javascript
Book.createRow = function (slots) {
    var book = null;
    try {
        book = new Book( slots);
    } catch (e) {
        console.log( e.name +": " + e.message);
        book = null;
    }
    if (book) {
        console.log( book.toString() + " created!");
    }
};
```

Likewise, when a constraint violation is detected in one of the setters invoked in Book.updateRow, a constraint violation error message is created (in line 16) and the previous state of the object is restored.
Implementing Constraint Validation in a JavaScript Frontend Web App

(in line 19). Otherwise, a status message is created (in lines 23 or 25), as shown in the following program listing:

```javascript
Book.updateRow = 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");
        }
    } catch (e) {
        console.log(e.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. 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 4.3.

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 src/model/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.

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 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 6.3).
Implementing Constraint Validation in a JavaScript Frontend Web App

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

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta charset="UTF-8" />
    <title>JS Frontend 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="src/ctrl/initialize.js"></script>
    <script src="src/model/errorTypes.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></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`:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta charset="UTF-8" />
    <title>JS Frontend 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="src/ctrl/initialize.js"></script>
    <script src="src/model/errorTypes.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>
    <button type="submit">Save</button>
</form>
</body>
</html>
```
Implementing Constraint Validation in a JavaScript Frontend Web App

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.

6.2. 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:

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

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

6.3. 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:

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

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

6.4. 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
Implementing Constraint Validation

in a JavaScript Frontend Web App

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:

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

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 20 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:

```javascript
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);
    });
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
      e.preventDefault();
      formEl.reset();
    });
    window.addEventListener("beforeunload", function () {
      Book.saveAllInstances();
    });
  },
  handleSubmitButtonClickEvent: function () {
    ...
  }
};
```
Implementing Constraint Validation
in a JavaScript Frontend Web App

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:

```javascript
handleSubmitButtonClickEvent: function () {
  var formEl = document.forms['Book'];
  var slots = { isbn: formEl.isbn.value, title: formEl.title.value, year: formEl.year.value};
  // check all input fields and create 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);
  // save the input data only if all of the form fields are valid
  if (formEl.checkValidity()) {
    Book.createRow( slots);
  }
}
```

By invoking checkValidity() on the form element, we make sure that the form data is only saved (by Book.createRow), 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:

```javascript
pl.view.updateBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit,
        selectBookEl = formEl.selectBook;
    // set up the book selection list
    util.fillWithOptionsFromAssocArray( Book.instances, selectBookEl,
      // when a book is selected, populate the form with its data
      selectBookEl.addEventListener("change", function () {
        var bookKey = selectBookEl.value;
        if (bookKey) {
          book = Book.instances[bookKey];
          formEl.isbn.value = book.isbn;
          formEl.title.value = book.title;
```
formEl.year.value = book.year;
} else {
    formEl.isbn.value = "";
    formEl.title.value = "";
    formEl.year.value = "";
}
});
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);
});
submitButton.addEventListener("click", this.handleSubmitButtonClickEvent);
// neutralize the submit event
formEl.addEventListener( 'submit', function (e) {
    e.preventDefault();
    formEl.reset();
});
window.addEventListener("beforeunload", function () {
    Book.saveAllInstances();
});,

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};
    formEl.title.setCustomValidity( Book.checkTitle( slots.title).message);
    formEl.year.setCustomValidity( Book.checkYear( slots.year).message);
    if (formEl.checkValidity()) {
        Book.updateRow( slots);
    }
}
```

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

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

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