JavaScript Frontend Web App
Tutorial Part 1: Building a Minimal App in Seven Steps

An introductory tutorial

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

by Gerd Wagner

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

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

Publication date February 2014
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

Foreword .................................................................................................................. vi
1. Foundations ........................................................................................................ 1
   1. The World Wide Web (WWW) ........................................................................ 1
   2. HTML and XML .............................................................................................. 1
      2.1. XML documents ...................................................................................... 1
      2.2. Unicode and UTF-8 ................................................................................ 1
      2.3. XML namespaces ..................................................................................... 2
      2.4. Correct XML documents ......................................................................... 2
      2.5. The evolution of HTML ........................................................................... 3
      2.6. HTML forms ............................................................................................. 3
   3. JavaScript ........................................................................................................ 4
      3.1. Types and data literals in JavaScript ...................................................... 5
      3.2. Different kinds of objects ........................................................................ 5
      3.3. Arrays ....................................................................................................... 7
      3.4. Associative arrays .................................................................................... 7
      3.5. Defining and instantiating a class ........................................................... 8
      3.6. Enumeration datatypes .......................................................................... 8
      3.7. JavaScript as an object-oriented language ............................................ 8
2. Building a Minimal JavaScript Frontend App in Seven Steps ......................... 10
   1. Step 1 - Set up the Folder Structure ............................................................. 10
   2. Step 2 - Write the Model Code .................................................................... 11
      2.1. Representing the collection of all Book instances .................................. 12
      2.2. Loading all Book instances ................................................................... 12
      2.3. Saving all Book instances ...................................................................... 13
      2.4. Creating a new Book instance ............................................................... 14
      2.5. Updating an existing Book instance ...................................................... 14
      2.6. Deleting an existing Book instance ....................................................... 14
      2.7. Creating test data .................................................................................. 14
      2.8. Clearing all data .................................................................................... 15
   3. Step 3 - Initialize the Application .................................................................. 15
   4. Step 4 - Implement the *List Objects* Use Case ....................................... 15
   5. Step 5 - Implement the *Create Object* Use Case .................................... 16
   6. Step 6 - Implement the *Update Object* Use Case .................................... 17
   7. Step 7 - Implement the *Delete Object* Use Case ...................................... 19
   8. Run the App and Get the Code ................................................................. 20
List of Figures

2.1. The object type Book. ........................................................................................................ 10
2.2. The minimal app's start page index.html. ........................................................................ 11
List of Tables

2.1. An associative array representing a collection of books ........................................ 12
2.2. A collection of book objects represented as a table ........................................ 12
Foreword

This tutorial is Part 1 of our series of five tutorials about engineering a frontend web application with plain JavaScript. It shows how to build such an app with minimal effort, not using any (third-party) framework or library. A frontend web app 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 app is a single-user application, which is not shared with other users.

The minimal version of a JavaScript frontend data management application discussed in this tutorial only includes a minimum of the overall functionality required for a complete app. It takes care of only one object type ("books") and supports the four standard data management operations (Create/Read/Update/Delete), but it needs to be enhanced by styling the user interface with CSS rules, and by adding further important parts of the app's overall functionality:

• Part 2 [validation-tutorial.html]: Handling constraint validation.

• 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.
Chapter 1. Foundations

If you are already familiar with HTML, XML and JavaScript, you can skip this chapter and immediately start developing a minimal front-end web application with JavaScript in the following chapter.

1. The World Wide Web (WWW)

After the Internet had been established in the 1980’ies, Tim Berners-Lee developed the idea and the first infrastructure components of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the web") is based on

• the basic Internet technologies (TCP/IP and DNS),
• the Hypertext Transfer Protocol (HTTP),
• the Hypertext Markup Language (HTML) as well as the Extensible Markup Language (XML), and
• web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

2. HTML and XML

HTML allows to mark up (or describe) the structure of a human-readable web document or web user interface, while XML allows to mark up the structure of all kinds of documents, data files and messages, whether they are human-readable or not. HTML can be based on XML.

2.1. XML documents

XML provides a syntax for expressing structured information in the form of an XML document with elements and their attributes. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or your own user-defined XML format. XML is used for specifying

• document formats, such as XHTML5, the Scalable Vector Graphics (SVG) format or the DocBook format,
• data interchange file formats, such as the Mathematical Markup Language (MathML) or the Universal Business Language (UBL),
• message formats, such as the web service message format SOAP [http://www.w3.org/TR/soap12-part0/]

2.2. Unicode and UTF-8

XML is based on Unicode, which is a platform-independent character set that includes almost all characters from most of the world's script languages including Hindi, Burmese and Gaelic. Each character is assigned a unique integer code in the range between 0 and 1,114,111. For example, the Greek letter # has the code 960, so it can be inserted in an XML document as π (using the XML entity syntax).

Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets.

The default encoding of an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.
Almost all Unicode characters are legal in a well-formed XML document. Illegal characters are the control characters with code 0 through 31, except for the carriage return, line feed and tab. It is therefore dangerous to copy text from another (non-XML) text to an XML document (often, the form feed character creates a problem).

### 2.3. XML namespaces

Generally, namespaces help to avoid name conflicts. They allow to reuse the same (local) name in different namespace contexts.

XML namespaces are identified with the help of a namespace URI (such as the SVG namespace URI "http://www.w3.org/2000/svg"), which is associated with a namespace prefix (such as "svg"). Such a namespace represents a collection of names, both for elements and attributes, and allows namespace-qualified names of the form \textit{prefix:name} (such as "svg:circle" as a namespace-qualified name for SVG circle elements).

A default namespace is declared in the start tag of an element in the following way:

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
```

This example shows the start tag of the HTML root element, in which the XHTML namespace is declared as the default namespace.

The following example shows a namespace declaration for the SVG namespace:

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
...
</head>
<body>
<figure>
  <figcaption>Figure 1: A blue circle</figcaption>
  <svg:svg xmlns:svg="http://www.w3.org/2000/svg">
    <svg:circle cx="100" cy="100" r="50" fill="blue"/>
  </svg:svg>
</figure>
</body>
</html>
```

### 2.4. Correct XML documents

XML defines two syntactic correctness criteria. An XML document must be \textit{well-formed}, and if it is based on a grammar (or schema), then it must also be \textit{valid} against that grammar.

An XML document is called \textit{well-formed}, if it satisfies the following syntactic conditions:

1. There must be exactly one root element.

2. Each element has a start tag and an end tag; however, empty elements can be closed as \texttt{<phone/>} instead of \texttt{<phone></phone>}.

3. Tags don't overlap, e.g. we cannot have

```xml
<author><name>Lee Hong</name></author></name>
```

4. Attribute names are unique within the scope of an element, e.g. the following code is not correct:

```xml
<attachment file="lecture2.html" file="lecture3.html"/>
```

An XML document is called \textit{valid} against a particular grammar (such as a DTD or an XML Schema), if
1. it is well-formed,  
2. and it respects the grammar.

### 2.5. The evolution of HTML

The World-Wide Web Committee (W3C) has developed the following important versions of HTML:

- HTML4 as an SGML-based language (in 1997),
- XHTML 1 as an XML-based version of HTML4 (in 2000),
- (X)HTML5 in competition and cooperation with the WHAT working group led by Ian Hickson and supported by browser vendors (in 2014).

HTML was originally designed as a structure description language, and not as a presentation description language. But HTML4 has a lot of purely presentational elements such as font. XHTML has been taking HTML back to its roots, dropping presentational elements and defining a simple and clear syntax, in support of the goals of:

- device independence,
- accessibility, and
- usability.

We adopt the symbolic equation

$$\text{HTML} = \text{HTML5} = \text{XHTML5}$$

stating that when we say "HTML" or "HTML5", we actually mean XHTML5 because we prefer the clear syntax of XML documents over the liberal and confusing HTML4-style syntax that is also allowed by HTML5.

The following simple example shows the basic code template to be used for any HTML document:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta charset="UTF-8" />
    <title>XHTML5 Template Example</title>
</head>
<body>
<h1>XHTML5 Template Example</h1>
<section>
    <h1>First Section Title</h1>
    ...
</section>
</body>
</html>
```

### 2.6. HTML forms

For user-interactive web applications, the web browser needs to render a user interface. The traditional metaphor for a software application's user interface is that of a form. The special elements for data input, data output and form actions are called form controls. An HTML form is a section of a document consisting of block elements that contain controls and labels on those controls.

Users complete a form by entering text into input fields and by selecting items from choice controls. A completed form is submitted with the help of a submit button. When a user submits a form, it
is sent to a web server either with the HTTP GET method or with the HTTP POST method. The standard encoding for the submission is called *URL-encoded*. It is represented by the Internet media type `application/x-www-form-urlencoded`. In this encoding, spaces become plus signs, and any other reserved characters become encoded as a percent sign and hexadecimal digits, as defined in RFC 1738.

Each control has both an initial value and a current value, both of which are strings. The initial value is specified with the control element's `value` attribute, except for the initial value of a `textarea` element, which is given by its initial contents. The control's current value is first set to the initial value. Thereafter, the control's current value may be modified through user interaction or scripts. When a form is submitted for processing, some controls have their name paired with their current value and these pairs are submitted with the form.

Labels are associated with a control by including the control as a subelement of a `label` element ("implicit labels"), or by giving the control an `id` value and referencing this id in the `for` attribute of the `label` element ("explicit labels"). Notice that implicit labels are, in 2014, still not well supported by CSS libraries and assistive technologies. Therefore, explicit labels seem preferable, despite the fact that they imply quite some overhead and repetitive code.

In the simple user interfaces of our "Getting Started" applications, we only need three types of form controls:

1. **single line input fields** created with an `<input name="..." />` element,
2. **push buttons** created with a `<button type="button">...</button>` element, and
3. **dropdown selection lists** created with a `select` element of the following form:

```html
<select name="...">
  <option value="value1"> option1 </option>
  <option value="value2"> option2 </option>
  ...
</select>
```

An example of an HTML form with implicit labels for creating such a user interface is

```html
<form id="Book">
  <p><label>ISBN: <input name="isbn" /></label></p>
  <p><label>Title: <input name="title" /></label></p>
  <p><label>Year: <input name="year" /></label></p>
  <p><button type="button" id="saveButton">Save</button></p>
</form>
```

### 3. JavaScript

This section provides a brief overview of JavaScript, assuming that the reader is already familiar with basic programming concepts and has some experience with programming, for instance, in PHP, Java or C#.

JavaScript is a programming language that can be used for

1. Enriching a web page by
   - generating browser-specific HTML content or CSS styling,
   - inserting dynamic HTML content,
   - producing special audio-visual effects (animations).
2. Enriching a web user interface by
• implementing advanced user interface components,
• validating user input on the client side,
• automatically pre-filling certain form fields.

3. Implementing a frontend web application with local data storage.

4. Implementing a frontend component for a distributed web application with remote data storage managed by a backend component (server-side program).

3.1. Types and data literals in JavaScript

JavaScript has three primitive data types: string, number and boolean. There are three reference types: object, array and function. Arrays and functions are just special kinds of objects. Types are not declared and not checked since a JavaScript program is not compiled. Type conversion (casting) is performed automatically.

The value of a variable may be

• a data value: either a string, a number, or a boolean,
• an object: either an ordinary object, or an array, or a function,
• one of the following two special data values: undefined or null.

All numeric data values are represented in 64-bit floating point format with an optional exponent (like in the numeric data literal 3.1e10). There is no type distinction between integers and floating point numbers. If an arithmetic expression cannot be evaluated to a number, its value is set to NaN ("not a number").

Like in Java, there are two pre-defined Boolean data literals, true and false, and the Boolean operator symbols are the exclamation mark ! for NOT, the double ampersand && for AND, and the double bar || for OR. When a non-Boolean value is used in a condition, or as an operand of a Boolean expression, it is converted into a Boolean value according to the following rules. The empty string, the (numerical) data literal 0, as well as undefined and null, are mapped to false, and all other values are mapped to true.

For equality and inequality testing, we use the triple equality symbol === and !== instead of the double equality symbol == and !=. Otherwise, for instance, the number 2 would be the same as the string "2", since the condition (2 == "2") evaluates to true in JavaScript.

3.2. Different kinds of objects

JavaScript objects are different from classical OO/UML objects. In particular, they need not instantiate a class. And they can have their own (instance-level) methods in the form of method slots, so they do not only have (ordinary) property slots, but also method slots. In addition they may also have key-value slots. So, they may have three different kinds of slots, while classical objects (called “instance specifications” in UML) only have property slots.

A JavaScript object is essentially a set of name-value-pairs, also called slots, where names can be property names, function names or keys of an associative array. Objects can be created in an ad-hoc manner, using JavaScript's object literal notation, without instantiating a class:

```javascript
var person1 = { lastName:"Smith", firstName:"Tom"};
```

```javascript
var o1 = {}; // an empty object with no slots
```

Whenever the name in a slot is an admissible JavaScript identifier [http://mothereff.in/js-variables], the slot may be either a property slot, a method slot or a key-value slot. Otherwise, if the name is some
other type of string (in particular when it contains any blank space), then the slot represents a \textit{key-value slot}, which is an associative array element, as explained below.

The name in a \textbf{property slot} may denote either

1. a \textbf{data-valued property}, in which case the value is a \textit{data value} or, more generally, a \textit{data-valued expression};

or

2. an \textbf{object-valued property}, in which case the value is an \textit{object reference} or, more generally, an \textit{object-valued expression}.

The name in a \textbf{method slot} denotes a \textit{JavaScript function} (better called \textit{method}), and its value is a function definition text.

Object properties can be accessed in two ways:

1. Using the dot notation (like in C++/Java):
   
   
   
   \texttt{person1.lastName = "Smith"}

2. Using an associative array notation:
   
   
   
   \texttt{person1["lastName"] = "Smith"}

JavaScript objects can be used in many different ways for different purposes. Here are five different use cases for, or possible meanings of, JavaScript objects:

1. A \textbf{record} is a set of property slots like, for instance,

   \begin{verbatim}
   var myRecord = { firstName:"Tom", lastName:"Smith", age:26}
   \end{verbatim}

2. An \textbf{associative array} (or 'hash map') supports look-ups of \textit{values} based on \textit{keys} like, for instance,

   \begin{verbatim}
   var numeral2number = { "one":"1", "two":"2", "three":"3"}
   \end{verbatim}

   which associates the value "1" with the key "one", "2" with "two", etc. A key need not be a valid JavaScript identifier, but can be any kind of string (e.g. it may contain blank spaces).

3. An \textbf{untyped object} does not instantiate a class. It may have property slots and function slots like, for instance,

   \begin{verbatim}
   var person1 = {
       lastName: "Smith",
       firstName: "Tom",
       getInitials: function () {
           return this.firstName.charAt(0) + this.lastName.charAt(0);
       }
   };
   \end{verbatim}

4. A \textbf{namespace} may be defined in the form of an untyped object referenced by a global object variable, the name of which represents a namespace prefix. For instance, the following object variable provides the main namespace of an application based on the Model-View-Controller (MVC) architecture paradigm where we have three subnamespaces corresponding to the three parts of an MVC application:

   \begin{verbatim}
   var myApp = { model:{}, view:{}, ctrl:{} };
   \end{verbatim}

5. A \textbf{typed object} that instantiates a class defined by a JavaScript constructor function \texttt{C} is created with the expression \texttt{o = new C(...)}. The type/class of such a typed object can be retrieved with the introspective expression
3.3. Arrays

An array variable may be initialized with an array literal:

```javascript
var a = [1,2,3];
```

JavaScript arrays can grow dynamically: it is possible to use indexes that are greater than the length of the array. For instance, after the array variable initialization above, the array held by the variable `a` has the length 3, but still we can assign a fifth array element like in

```javascript
a[4] = 7;
```

The contents of an array `a` are processed with the help of a standard `for` loop with a counter variable counting from the first array index 0 to the last array index, which is `a.length-1`:

```javascript
for (var i=0; i < a.length; i++) { ...}
```

Since arrays are special types of objects, we sometimes need a method for finding out if a variable represents an array. We can test, if a variable `a` represents an array by applying the predefined datatype predicate `isArray` as in `Array.isArray(a)`.

For adding a new element to an array, we append it to the array using the `push` operation as in:

```javascript
a.push(newElement);
```

For deleting an element at position `i` from an array `a`, we use the pre-defined array method `splice` as in:

```javascript
a.splice(i, 1);
```

3.4. Associative arrays

An associative array (also called 'hash map') provides a map from keys to their associated values. The keys of an associative array may be string literals that include blank spaces like in:

```javascript
var myTranslation = {
    "my house": "mein Haus",
    "my boat": "mein Boot",
    "my horse": "mein Pferd"
}
```

An associative array is processed with the help of a special loop where we loop over all keys of the associative array using the pre-defined function `Object.keys(a)`, which returns an array of all keys of an associative array `a`. For instance,

```javascript
keys = Object.keys(myTranslation);
for (var i=0; i < keys.length; i++) {
    key = keys[i];
    alert('The translation of '+ key +' is '+ myTranslation[key]);
}
```

For adding a new element to an associative array, we simply create a new key-value entry as in:

```javascript
myTranslation["my car"] = "mein Auto";
```
For deleting an element from an associative array, we can use the pre-defined JavaScript `delete` operator as in:

```javascript
delete myTranslation["my boat"];  
```

### 3.5. Defining and instantiating a class

A class can be defined in two steps. First define the constructor function that defines the properties of the class and assigns them the values of the constructor's parameters:

```javascript
function Person(first, last) {
    this.firstName = first;
    this.lastName = last;
}
```

Next, define the *instance-level methods* of the class as function slots of the prototype object property of the constructor:

```javascript
Person.prototype.getInitials = function () {
    return this.firstName.charAt(0) + this.lastName.charAt(0);
}
```

Finally, *class-level* ("static") *methods* can be defined as function slots of the constructor, as in

```javascript
Person.checkName = function (n) {
    ...
}
```

An instance of a class is created by applying the `new` operator to the constructor:

```javascript
var pers1 = new Person("Tom","Smith");
```

The method `getInitials` is invoked on the Person object `pers1` by using the ‘dot notation’:

```javascript
alert("The initials of the person are: " + pers1.getInitials());
```

### 3.6. Enumeration datatypes

We implement an enumeration datatype in the form of a special object definition:

```javascript
GenderEL = Object.defineProperties( {}, {
    MALE: { value: 1, writable: false},
    FEMALE: { value: 2, writable: false},
    MAX: { value: 2, writable: false}
  });
```

This definition of the special object `GenderEL` allows using the enumeration literals `GenderEL.MALE` and `GenderEL.FEMALE` standing for the integer values 1 and 2. We can then check if an enumeration attribute (such as an attribute `gender` of type `GenderEL`) has an admissible value by testing if its value is not smaller than 1 and not greater than the `MAX` property of the enumeration object (`GenderEL.MAX` in our example).

### 3.7. JavaScript as an object-oriented language

JavaScript is *object-oriented*, but in a different way than classical object-oriented programming languages such as Java and C++. There is no explicit *class* concept, but a more or less equivalent concept of a *constructor*. A constructor definition allows grouping properties and methods that apply to objects created by the constructor. Objects can be created without instantiating a class. Methods can be defined for specific objects independently of any class. An inheritance mechanism is provided.
via the constructor’s built-in prototype property. At run time, properties and methods can be added to, or removed from, any object and class.

## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypertext Transfer Protocol</td>
<td>HTTP is a stateless request/response protocol using human-readable text messages for the communication between web clients and web servers. The main purpose of HTTP has been to allow fetching web documents identified by URLs from a web browser, and invoking the operations of a backend web application program from a HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services.</td>
</tr>
<tr>
<td>Hypertext Markup Language</td>
<td>HTML allows marking up (or describing) the structure of a human-readable web document or web user interface. The XML-based version of HTML, which is called &quot;XHTML5&quot;, provides a simpler and cleaner syntax compared to traditional HTML.</td>
</tr>
<tr>
<td>Extensible Markup Language</td>
<td>XML allows to mark up the structure of all kinds of documents, data files and messages, whether they are human-readable or not. XML is based on Unicode. SVG and MathML are based on XML, and there is an XML-based version of HTML. XML provides a syntax for expressing structured information in the form of an XML document with elements and their attributes. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or user-defined XML formats.</td>
</tr>
</tbody>
</table>
Chapter 2. Building a Minimal JavaScript Frontend App in Seven Steps

In this chapter, we build a minimal frontend web application with JavaScript. The purpose of our example app is to manage information about books. That is, we deal with a single object type: Book, as depicted in Figure 2.1.

Figure 2.1. The object type Book.

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

What do we need for such an information management application? There are four standard use cases, which have to be supported by the application:

1. **Create**: Enter the data of a book that is to be added to the collection of managed books.
2. **Read**: Show a list of all books in the collection of managed books.
3. **Update** the data of a book.

For entering data with the help of the keyboard and the screen of our computer, we can use **HTML forms**, which provide the **user interface** technology for web applications.

For maintaining a collection of data objects, we need a storage technology that allows to keep data objects in persistent records on a secondary storage device, such as a harddisk or a solid state disk. Modern web browsers provide two such technologies: the simpler one is called **Local Storage**, and the more powerful one is called **IndexDB**. For our minimal example app, we use Local Storage.

1. Step 1 - Set up the Folder Structure

In the first step, we set up our folder structure for the application. We pick a name for our app, such as "Public Library", and a corresponding (possibly abbreviated) name for the application folder, such as "publicLibrary". Then we create this folder on our computer's disk and a subfolder "src" for our JavaScript source code files. In this folder, we create the subfolders "model", "view" and "ctrl", following the **Model-View-Controller** paradigm for software application architectures. And finally we create an index.html file for the app's start page, as discussed below. Thus, we end up with the following folder structure:

```
publicLibrary
  src
    ctrl
    model
    view
  index.html
```

The start page of the app loads the Book.js model class file and provides a menu for choosing one of the CRUD data management operations performed by a corresponding page such as, for instance, createBook.html, or for creating test data with the help of the procedure Book.createTestData() in line 17, or clearing all data with Book.clearData() in line 18.
Figure 2.2. The minimal app's start page index.html.

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Frontend App Example</title>
  <script src="src/model/Book.js"></script>
</head>
<body>
<h1>Public Library</h1>
<h2>An Example of a Minimal JavaScript Frontend App</h2>
<p>This app supports the following operations:</p>
<menu>
  <li><a href="listBooks.html"><button type="button">List all books</button></a></li>
  <li><a href="createBook.html"><button type="button">Add a new book</button></a></li>
  <li><a href="updateBook.html"><button type="button">Update a book</button></a></li>
  <li><a href="deleteBook.html"><button type="button">Delete a book</button></a></li>
  <li><button type="button" onclick="Book.clearData()">Clear database</button></li>
  <li><button type="button" onclick="Book.createTestData()">Create test data</button></li>
</menu>
</body>
</html>
```

2. Step 2 - Write the Model Code

In the second step, we write the code of our model class in a specific JavaScript file. In the information design model shown in Figure 2.1 above, there is only one class, representing the object type Book. So, in the folder src/model, we create a file Book.js that initially contains the following code:

```javascript
function Book(slots) {
  this.isbn = slots.isbn;
  this.title = slots.title;
  this.year = slots.year;
};
```

The model class Book is encoded as a JavaScript constructor function with a single slots parameter, which is supposed to be a record object with properties isbn, title and year, representing values for the ISBN, the title and the year attributes of the class Book. Therefore, in the constructor function, the values of the slots properties are assigned to the corresponding attributes whenever a new object is created as an instance of this class.

In addition to defining the model class in the form of a constructor function, we also define the following items in the Book.js file:

1. A class-level property Book.instances representing the collection of all Book instances managed by the application in the form of an associative array.
2. A class-level method Book.loadAllInstances for loading all managed Book instances from the persistent data store.
3. A class-level method Book.saveAllInstances for saving all managed Book instances to the persistent data store.
5. A class-level method Book.updateRow for updating an existing Book instance.
7. A class-level method `Book.createTestData` for creating a few example book records to be used as test data.


### 2.1. Representing the collection of all Book instances

For representing the collection of all Book instances managed by the application, we define and initialize the class-level property `Book.instances` in the following way:

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

So, initially our collection of books is empty. In fact, it's defined as an empty object, since we want to represent it in the form of an associative array (a set of key-value slots, also called 'hashmap') where an ISBN is a key for accessing the corresponding book object (as the value associated with the key). We can visualize the structure of such an associative array in the form of a lookup table, as shown in Table 2.1, “An associative array representing a collection of books”.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>006251587X</td>
<td>{ isbn:&quot;006251587X,&quot; title:&quot;Weaving the Web&quot;, year:2000 }</td>
</tr>
<tr>
<td>0465026567</td>
<td>{ isbn:&quot;0465026567,&quot; title:&quot;Gödel, Escher, Bach&quot;, year:1999 }</td>
</tr>
<tr>
<td>0465030793</td>
<td>{ isbn:&quot;0465030793,&quot; title:&quot;I Am A Strange Loop&quot;, year:2008 }</td>
</tr>
</tbody>
</table>

Notice that the values of this associative array are simple objects corresponding to table rows. Consequently, we could represent them also in a simple table, as shown in Table 2.2, “A collection of book objects represented as a table”.

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>006251587X</td>
<td>Weaving the Web</td>
<td>2000</td>
</tr>
<tr>
<td>0465026567</td>
<td>Gödel, Escher, Bach</td>
<td>1999</td>
</tr>
<tr>
<td>0465030793</td>
<td>I Am A Strange Loop</td>
<td>2008</td>
</tr>
</tbody>
</table>

### 2.2. Loading all Book instances

For persistent data storage, we use `LocalStorage`, which is a HTML5 JavaScript API supported by modern web browsers. Loading the book records from Local Storage involves three steps:

1. Retrieving the book table that has been stored as a large string with the key "bookTable" from Local Storage with the help of the assignment

   ```javascript
   bookTableString = localStorage["bookTable"]; 
   ```

   This retrieval is performed in line 5 of the program listing below.

2. Converting the book table string into a corresponding associative array `bookTable` with book rows as elements, with the help of the built-in procedure `JSON.parse`:

   ```javascript
   bookTable = JSON.parse( bookTableString);
   ```

   This conversion, performed in line 11 of the program listing below, is called `deserialization`. 
3. Converting each row of `bookTable` (representing an untyped record object) into a corresponding object of type `Book` stored as an element of the associative array `Book.instances`, with the help of the procedure `convertRow2Obj` defined as a "static" (class-level) method in the `Book` class:

```javascript
Book.convertRow2Obj = function (bookRow) {
    var book = new Book(bookRow);
    return book;
};
```

Here is the full code of the procedure:

```javascript
Book.loadAllInstances = function () {
    var key = ", keys = [], bookTableString = ", bookTable = {};
    try {
        if (localStorage["bookTable"]) {
            bookTableString = localStorage["bookTable"];  // This line may fail
        }
    } catch (e) {
        alert("Error when reading from Local Storage\n" + e);
    }
    if (bookTableString) {
        bookTable = JSON.parse(bookTableString);
        keys = Object.keys(bookTable);
        console.log(keys.length + " books loaded.");
        for (var i = 0; i < keys.length; i++) {
            key = keys[i];
            Book.instances[key] = Book.convertRow2Obj(bookTable[key]);
        }
    }
};
```

Notice that since an input operation like `localStorage["bookTable"]` may fail, we perform it in a try-catch block, where we can follow up with an error message whenever the input operation fails.

### 2.3. Saving all Book instances

Saving all book objects from the `Book.instances` collection in main memory to Local Storage in secondary memory involves two steps:

1. Converting the associative array `Book.instances` into a string with the help of the predefined JavaScript procedure `JSON.stringify`:

   ```javascript
   bookTableString = JSON.stringify(Book.instances);
   ```

   This conversion is called serialization.

2. Writing the resulting string as the value of the key "bookTable" to Local Storage:

   ```javascript
   localStorage["bookTable"] = bookTableString;
   ```

These two steps are performed in line 5 and in line 6 of the following program listing:

```javascript
Book.saveAllInstances = function () {
    var bookTableString = ", error = false,
    nmrOfBooks = Object.keys(Book.instances).length;
    try {
        bookTableString = JSON.stringify(Book.instances);
        localStorage["bookTable"] = bookTableString;
    } catch (e) {
        alert("Error when writing to Local Storage\n" + e);
    }
};
```
error = true;
}
if (!error) console.log( nmrOfBooks + " books saved.");
};

2.4. Creating a new Book instance

The Book.createRow procedure takes care of creating a new Book instance and adding it to the Book.instances collection:

```javascript
Book.createRow = function (slots) {
    var book = new Book( slots);
    Book.instances[slots.isbn] = book;
    console.log("Book " + slots.isbn + " created!");
};
```

2.5. Updating an existing Book instance

For updating an existing Book instance we first retrieve it from Book.instances, and then re-assign those attributes the value of which has changed:

```javascript
Book.updateRow = function (slots) {
    var book = Book.instances[slots.isbn];
    var year = parseInt( slots.year);
    if (book.title !== slots.title) { book.title = slots.title;}
    if (book.year !== year) { book.year = year;}
    console.log("Book " + slots.isbn + " modified!");
};
```

Notice that in the case of a numeric attribute (such as year), we have to make sure that the value of the corresponding input parameter (y), which is typically obtained from user input via an HTML form, is converted from String to Number with one of the two type conversion functions parseInt or parseFloat.

2.6. Deleting an existing Book instance

A Book instance is deleted from the Book.instances collection by first testing if the associative array has an element with the given key (line 2), and then applying the JavaScript built-in delete operator,. which deletes a slot from an object, or, in our case, an element from an associative array:

```javascript
Book.deleteRow = function (isbn) {
    if (Book.instances[isbn]) {
        console.log("Book " + isbn + " deleted");
        delete Book.instances[isbn];
    } else {
        console.log("There is no book with ISBN " + isbn + " in the database!");
    }
};
```

2.7. Creating test data

For being able to test our code, we may create some test data and save it in our Local Storage database. We can use the following procedure for this:

```javascript
Book.createTestData = function () {
    Book.instances["006251587X"] = new Book({isbn:"006251587X", title:"Weaving the Web", year:2000});
    Book.instances["0465026567"] = new Book({isbn:"0465026567", title:"Gödel, Escher, Bach", year:1999});
    Book.instances["0465030793"] = new Book({isbn:"0465030793", title:"I Am A Strange Loop", year:2008});
    Book.saveAllInstances();
};
```
2.8. Clearing all data

The following procedure clears all data from Local Storage:

```javascript
Book.clearData = function () {
    if (confirm("Do you really want to delete all book data?")) {
        localStorage["bookTable"] = "{}";
    }
};
```

3. Step 3 - Initialize the Application

We initialize the application by defining its namespace and MVC subnamespaces. Namespaces are an important concept in software engineering and many programming languages, including Java and PHP, provide specific support for namespaces, which help grouping related pieces of code and avoiding name conflicts. Since there is no specific support for namespaces in JavaScript, we use special objects for this purpose (we may call them "namespace objects"). First we define a root namespace (object) for our app, and then we define three subnamespaces, one for each of the three parts of the application code: model, view and controller. In the case of our example app, we may use the following code for this:

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

Here, the main namespace is defined to be pl, standing for "Public Library", with the three subnamespaces model, view and ctrl being initially empty objects. We put this code in a separate file initialize.js in the ctrl folder, because such a namespace definition belongs to the controller part of the application code.

4. Step 4 - Implement the List Objects Use Case

This use case corresponds to the "Read" from the four basic data management use cases Create-Read-Update-Delete (CRUD).

The user interface for this use case is provided by the following HTML page containing an HTML table for displaying the book objects. For our example app, this page would be called listBooks.html (in the main folder publicLibrary) and would contain the following HTML code:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta charset="UTF-8" />
    <title>Simple JS Frontend App Example</title>
    <script src="src/ctrl/initialize.js"></script>
    <script src="src/model/Book.js"></script>
    <script src="src/view/listBooks.js"></script>
    <script>
        window.addEventListener( "load", pl.view.listBooks.setupUserInterface);
    </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>
</body>
</html>
```
Notice that this HTML file loads three JavaScript files: the controller file `src/ctrl/initialize.js`, the model file `src/model/Book.js` and the view file `src/view/listBooks.js`. The first two files contain the code for initializing the app and for the model class `Book` as explained above, and the third one, which represents the UI code of the "list books" operation, is developed now. In fact, for this operation, we just need a procedure for setting up the data management context and the UI, called `setupUserInterface`:

```javascript
pl.view.listBooks = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("table#books>tbody");
    var keys=[], key="", row={};
    // load all book objects
    Book.loadAllInstances();
    keys = Object.keys( Book.instances);
    // for each book, create a table row with three cells for the three attributes
    for (var i=0; i < keys.length; i++) {
      key = keys[i];
      row = tableBodyEl.insertRow();
      row.insertCell(-1).textContent = Book.instances[key].isbn;
      row.insertCell(-1).textContent = Book.instances[key].title;
      row.insertCell(-1).textContent = Book.instances[key].year;
    }
  }
};
```

The simple logic of this procedure consists of two steps:

1. Read the collection of all objects from the persistent data store (in line 6).
2. Display each object as a row in a HTML table on the screen (in the loop starting in line 9).

More specifically, the procedure `setupUserInterface` first creates the book objects from the corresponding rows retrieved from Local Storage by invoking `Book.loadAllInstances()` and then creates the view table in a loop over all key-value slots of the associative array `Book.instances` where each value represents a book object. In each step of this loop, a new row is created in the table body element with the help of the JavaScript DOM operation `insertRow()`, and then three cells are created in this row with the help of the DOM operation `insertCell()`: the first one for the isbn property value of the book object, and the second and third ones for its title and year property values. Both `insertRow` and `insertCell` have to be invoked with the argument -1 for making sure that new elements are appended to the list of rows and cells.

5. Step 5 - Implement the Create Object Use Case

For a data management operation with user input, such as the "create object" operation, an HTML page with an HTML form is required as a user interface. The form has a form field for each attribute of the `Book` class. For our example app, this page would be called `createBook.html` (in the app folder `publicLibrary`) and would contain the following HTML code:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
</head>
</html>
```
Building a Minimal JavaScript Frontend App in Seven Steps

1. Build a Minimal JavaScript Frontend App

The view code file `src/view/createBook.js` contains two procedures:

1. **setupUserInterface**
   - Takes care of retrieving the collection of all objects from the persistent data store and setting up an event handler (`handleSaveButtonClickEvent`) on the save button for handling click button events by saving the user input data.

2. **handleSaveButtonClickEvent**
   - Reads the user input data from the form fields and then saves this data by calling the `Book.saveRow` procedure.

```javascript
pl.view.createBook = {
  setupUserInterface: function () {
    var saveButton = document.forms['Book'].commit;
    // load all book objects
    Book.loadAllInstances();
    // Set an event handler for the save/submit button
    saveButton.addEventListener("click",
      pl.view.createBook.handleSaveButtonClickEvent);
    window.addEventListener("beforeunload", function () {
      Book.saveAllInstances();
    });
  },
  handleSaveButtonClickEvent: function () {
    var formEl = document.forms['Book'];
    var slots = {
      isbn: formEl.isbn.value,
      title: formEl.title.value,
      year: formEl.year.value
    };
    Book.createRow(slots);
    formEl.reset();
  }
};
```

6. Step 6 - Implement the **Update Object** Use Case

Again, we have an HTML page for the user interface (`updateBook.html`) and a view code file (`src/view/updateBook.js`). The form for the UI of the "update object" operation has a selection
field for choosing the book to be updated, and a form field for each attribute of the `Book` class. However, the form field for the standard identifier attribute (ISBN) is read-only because we do not allow changing the standard identifier of an existing object.

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Frontend App Example</title>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/updateBook.js"></script>
  <script>
    window.addEventListener("load", pl.view.updateBook.setupUserInterface);
  </script>
</head>
<body>
  <h1>Public Library: Update a book record</h1>
  <form id="Book" action="">
    <p>
      <label>Select book:
        <select name="selectBook">
          <option value=""> --- </option>
        </select>
      </label>
    </p>
    <p><label>ISBN: <input name="isbn" readonly="readonly" /></label></p>
    <p><label>Title: <input name="title" /></label></p>
    <p><label>Year: <input name="year" /></label></p>
    <p><button type="button" name="commit">Save Changes</button></p>
  </form>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>

The `setupUserInterface` procedure now has to set up a selection field by retrieving the collection of all book objects from the persistent data store for populating the select element's option list:

```javascript
pl.view.updateBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        saveButton = formEl.commit,
        selectBookEl = formEl.selectBook;
    var key='', keys=[], book=null, optionEl=null;
    // load all book objects
    Book.loadAllInstances();
    // populate the selection list with books
    keys = Object.keys( Book.instances);
    for (var i=0; i < keys.length; i++) {
      key = keys[i];
      book = Book.instances[key];
      optionEl = document.createElement("option");
      optionEl.text = book.title;
      optionEl.value = book.isbn;
      selectBookEl.add( optionEl, null);
    }
    // when a book is selected, populate the form with the book data
    selectBookEl.addEventListener("change", function () {
      var book=null, key = selectBookEl.value;
      if (key) {
        //
      }
    });
  }
};
```
book = Book.instances[key];
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 = "";
}
};
saveButton.addEventListener("click",
pl.view.updateBook.handleUpdateButtonClickEvent);
window.addEventListener("beforeunload", function () {
Book.saveAllInstances();
});
// save session data
handleUpdateButtonClickEvent: function () {
var formEl = document.forms['Book'];
var slots = { isbn: formEl.isbn.value,
title: formEl.title.value,
year: formEl.year.value
};
Book.updateRow( slots);
formEl.reset();
}
};

7. Step 7 - Implement the Delete Object Use Case

For the "delete object" use case, the UI form just has a selection field for choosing the book to be deleted:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

<meta charset="UTF-8" />
<title>Minimal JS Frontend App Example</title>
<script src="src/ctrl/initialize.js"></script>
<script src="src/model/Book.js"></script>
<script src="src/view/deleteBook.js"></script>
<script>
window.addEventListener("load", pl.view.deleteBook.setupUserInterface);
</script>
</head>
<body>
<h1>Public Library: Delete a book record</h1>
<form id="Book">
<p>
<label>Select book:
<select name="selectBook">
<option value=""></option>
</select>
</label>
</p>
<p><button type="button" name="commit">Delete</button></p>
</form>
<nav><a href="index.html">Back to main menu</a></nav>
```
The view code in src/view/deleteBook.js consists of the following two procedures:

```javascript
pl.view.deleteBook = {
    setupUserInterface: function () {
        var deleteButton = document.forms['Book'].commit;
        var selectEl = document.forms['Book'].selectBook;
        var key = '', keys = [], book = null, optionEl = null;
        // load all book objects
        Book.loadAllInstances();
        keys = Object.keys(Book.instances);
        // populate the selection list with books
        for (var i = 0; i < keys.length; i++) {
            key = keys[i];
            book = Book.instances[key];
            optionEl = document.createElement("option");
            optionEl.text = book.title;
            optionEl.value = book.isbn;
            selectEl.add(optionEl, null);
        }
        deleteButton.addEventListener("click",
            pl.view.deleteBook.handleDeleteButtonClickEvent);
        window.addEventListener("beforeunload", function () {
            Book.saveAllInstances();
        });
    },
    handleDeleteButtonClickEvent: function () {
        var selectEl = document.forms['Book'].selectBook;
        var isbn = selectEl.value;
        if (isbn) {
            Book.deleteRow(isbn);
            selectEl.remove(selectEl.selectedIndex);
        }
    }
};
```

8. Run the App and Get the Code

You can run the minimal app [MinimalApp/] from our server or download it as a ZIP archive file [MinimalApp.zip].