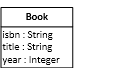
Book library:

Coding the App

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 the following figure.



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.
4. **Delete** a book record.

### Step 1 - Set up the Folder Structure

#### index.html

### 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 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:

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 slotsproperties 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 a map.
2. A class-level method Book.loadAll for loading all managed Book instances from the persistent data store.
3. A class-level method Book.saveAll for saving all managed Book instances to the persistent data store.
4. A class-level method Book.add for creating and storing a new Book record.
5. A class-level method Book.update for updating an existing Book record.
6. A class-level method Book.destroy for deleting a Book instance.
7. A class-level method Book.createTestData for creating a few example book records to be used as test data.
8. A class-level method Book.clearData for clearing the book datastore.

#### 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:

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 a map (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 a map in the form of a lookup table, as shown in Table 1.

##### Table 1: A map representing a collection of books

| **Key** | **Value** |
| --- | --- |
| 006251587X | { isbn:"006251587X," title:"Weaving the Web", year:2000 } |
| 0465026567 | { isbn:"0465026567," title:"Gödel, Escher, Bach", year:1999 } |
| 0465030793 | { isbn:"0465030793," title:"I Am A Strange Loop", year:2008 } |

Notice that the values of this map are simple record objects corresponding to table rows. Consequently, we could represent them also in a simple table, as shown in Table 2.

##### Table 2: A collection of book objects represented as a table

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

#### 2. Loading all Book instances

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

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:

bookTableString = localStorage["bookTable"];

1. This retrieval is performed in line 5 of the program listing below.
2. Converting the book table string into a corresponding map bookTable with book rows as elements, with the help of the built-in function JSON.parse:

bookTable = JSON.parse( bookTableString);

This conversion, performed in line 11 of the program listing below, is called deserialization.

1. Converting each row of bookTable (representing an untyped record object) into a corresponding object of type Book stored as an element of the map Book.instances, with the help of the procedure convertRow2Obj defined as a "static" (class-level) method in the Book class:

Book.**convertRow2Obj** = function (bookRow) {

var book = new Book( bookRow);

return book;

};

#### 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 map Book.instances into a string with the help of the predefined JavaScript function JSON.stringify:

bookTableString = JSON.stringify( Book.instances);

This conversion is called serialization.

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

localStorage["bookTable"] = bookTableString;

#### 4. Creating a new Book instance

The Book.add procedure takes care of creating a new Book instance and adding it to the Book.instancescollection:

Hide   Copy Code

Book.**add** = function (slots) {

var book = new Book( slots);

Book.instances[slots.isbn] = book;

console.log("Book " + slots.isbn + " created!");

};

#### 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.

#### 6. Deleting an existing Book instance

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

#### 7. Creating test data

#### 8. Clearing all data

### 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:

var bl = { model:{}, view:{}, ctrl:{} };

### Step 4 - Implement the List Objects Use Case

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.

The simple logic of this procedure consists of two steps:

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

More specifically, the procedure setupUserInterface first creates the book objects from the corresponding rows retrieved from Local Storage by invoking Book.loadAll() and then creates the view table in a loop over all key-value slots of the map 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.

### Step 5 - Implement the Create Object Use Case

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.

### Step 6 - Implement the Update Object Use Case

### Step 7 - Implement the Delete Object Use Case