JS/Firebase Web App Tutorial Part 1: Building a Minimal App in Seven Steps

Learn how to build a minimal front-end web application with cloud storage using plain JavaScript and Firebase

By Gerd Wagner and Juan-Francisco Reyes

Warning: This tutorial manuscript may contain errors and may still be incomplete. Please report any issue to Gerd Wagner.

This tutorial is also available in the following formats: PDF.

You may run the example app from our server, or download the code as a ZIP archive file.

Published 2022-05-23.

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 1 of our series of six tutorials about model-based development of front-end web applications with plain JavaScript and the Firebase cloud platform, more specifically Firebase JavaScript/Web SDK version 9, better known as the "modular version". It shows how to build such an app with minimal effort, not using any (third-party) framework or library. While libraries and frameworks may help to increase productivity, they also create black-box dependencies and overhead, and they are not good for ***learning how to do it yourself***.

This tutorial provides theoretically underpinned and example-based learning materials and supports ***learning by doing it yourself***.

A front-end 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. A front-end web app with data storage in the cloud can be a multi-user application, which is shared with other, possibly concurrent, users.

The minimal version of a JS/Firebase front-end 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 (**C**reate/**R**etrieve/**U**pdate/**D**elete), 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. The other parts of the tutorial are:

- Part 2: Handling **constraint validation**.
- Part 3: Dealing with **enumerations**.
- Part 4: Managing **unidirectional associations**, such as the associations between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.
- Part 5: Managing **bidirectional associations**, such as the associations between books and publishers and between books and authors, also assigning books to authors and to publishers.
- Part 6: Handling **subtype** (inheritance) relationships between object types.

Chapter 1. A Quick Tour of the Foundations of Web Apps

1.1. The World Wide Web (WWW)

After the Internet had been established in the 1980'ies, Tim Berners-Lee developed the idea and the first implementation of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the Web") is based on the Internet technologies TCP/IP (the *Internet Protocol*) and DNS (the *Domain Name System*). Initially, the Web consisted of

1. the *Hypertext Transfer Protocol (HTTP)*,
2. the *Hypertext Markup Language (HTML)*, and
3. web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

Later, further important technology components have been added to this set of basic web technologies:

- the page/document style language *Cascading Style Sheets (CSS)* in 1995,
- the web programming language JavaScript in 1995,
- the *Extensible Markup Language (XML)*, as the basis of web formats like SVG and RDF/XML, in 1998,
- the XML-based *Scalable Vector Graphics (SVG)* format in 2001,
- the *Resource Description Framework (RDF)* for knowledge representation on the Web in 2004.

1.2. 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. XML can also be used as the basis for defining a version of HTML that is called *XHTML*.

1.2.1 XML documents

XML provides a syntax for expressing structured information in the form of an XML document with nested *elements* and their *attributes*. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or (private) user-defined XML formats. 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

1.2.2 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 π has the code 960, so it can be inserted in an XML document as `&#960;` 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).

1.2.3 XML namespaces

Generally, namespaces help to avoid name conflicts. They allow to reuse the same (local) name in different namespace contexts. Many computational languages have some form of namespace concept, for instance, Java and PHP.

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 *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:

```
<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 an SVG namespace declaration for an `svg` element embedded in an HTML document:

```
<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>
```

1.2.4 Correct XML documents

XML defines two syntactic correctness criteria. An XML document must be ***well-formed***, and if it is based on a grammar (or schema), then it must also be *valid* with respect to that grammar, or, in other words, satisfy all rules of the grammar.

An XML document is called ***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 <phone/> instead of <phone></phone>.

3. Tags don't overlap. For instance, we cannot have

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

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

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

An XML document is called ***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*.

1.2.5 The evolution of HTML

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

- 1997: ***HTML 4*** as an SGML-based language,
- 2000: ***XHTML 1*** as an XML-based clean-up of HTML 4,
- 2014: *(X)****HTML 5*** in cooperation (and competition) with the WHAT working group supported by browser vendors.

> As the inventor of the Web, Tim Berners-Lee developed a first version of HTML in 1990. A few years later, in 1995, Tim Berners-Lee and Dan Connolly wrote the *HTML 2* standard, which captured the common use of HTML elements at that time. In the following years, HTML has been used and gradually extended by a growing community of early WWW adopters. This evolution of HTML, which has led to a messy set of elements and attributes (called "tag soup"), has been mainly controlled by browser vendors and their competition with each other. The development of XHTML in 2000 was an attempt by the W3C to clean up this mess, but it neglected to advance HTML's functionality towards a richer user interface, which was the focus of the WHAT working group led by Ian Hickson who can be considered as the mastermind and main author of HTML 5 and many of its accompanying JavaScript APIs that made HTML fit for mobile apps.

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

HTML = HTML5 = 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:

```
<!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>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
</head>
<body>
  <h1>XHTML5 Template Example</h1>
  <section><h2>First Section Title</h2>
   ...
  </section>
</body>
</html>
```

Notice that in line 1, the HTML5 document type is declared, such that browsers are instructed to use the HTML5 document object model (DOM). In the `html` start tag in line 2, using the default namespace declaration attribute `xmlns`, the XHTML namespace URI `http://www.w3.org/1999/xhtml` is declared as the default namespace for making sure that browsers, and other tools, understand that all non-qualified element names like `html`, `head`, `body`, etc. are from the XHTML namespace.

Also in the `html` start tag, we set the (default) language for the text content of all elements (here to "en" standing for English) using both the `xml:lang` attribute and the HTML `lang` attribute. This attribute duplication is a small price to pay for having a hybrid document that can be processed both by HTML and by XML tools.

Finally, in line 4, using an (empty) `meta` element with a `charset` attribute, we set the HTML document's character encoding to UTF-8, which is also the default for XML documents.

1.2.6 HTML forms

For user-interactive web applications, the web browser needs to render a user interface (UI). The traditional metaphor for a software application's UI is that of a ***form***. The special elements for data input, data output and user actions are called ***form controls*** or *UI* ***widgets***. In HTML, a `form` element is a section of a web page consisting of block elements that contain form controls and *labels* on those controls.

Users complete a form by entering text into *input fields* and by selecting items from *choice controls*, including dropdown *selection lists*, *radio button groups* and *checkbox groups*. A completed form is submitted with the help of a *submit button*. When a user submits a form, it is normally 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 form 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 child element within 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).

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

1. *single line input fields* created with an `<input name="..." />` element,

2. *single line output fields* created with an `<output name="..." />` element,

3. *push buttons* created with a `<button type="button">...</button>` element, and

4. *dropdown selection lists* created with a `select` element of the following form:

```
<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

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

In an HTML-form-based data management user interface, we have a correspondence between the different kinds of properties defined in the model classes of an app and the form controls used for the input and output of their values. We have to distinguish between various kinds of ***model class attributes***, which are mapped to various kinds of ***form fields***. This mapping is also called ***data binding***.

In general, an attribute of a model class can always be represented in the user interface by a plain `input` control (with the default setting `type="text"`), no matter which datatype has been defined as the range of the attribute in the model class. However, in special cases, other types of `input` controls (for instance, `type="date"`), or other widgets, may be used. For instance, if the attribute's range is an enumeration, a `select` control or, if the number of possible choices is small enough (say, less than 8), a radio button group can be used.

1.3. Styling Web Documents and User Interfaces with CSS

While HTML is used for defining the content structure of a web document or a web user interface, the ***Cascading Style Sheets (CSS)*** language is used for defining the ***presentation style*** of web pages, which means that you use it for telling the browser how you want your HTML (or XML) rendered: using which layout of content elements, which fonts and text styles, which colors, which backgrounds, and which animations. Normally, these settings are made in a separate CSS file that is associated with an HTML file via a special `link` element in the HTML's `head`.

> A first sketch of CSS was proposed in October 1994 by Håkon W. Lie who later became the CTO of the browser vendor Opera. While the official CSS1 standard dates back to December 1996, "most of it was hammered out on a whiteboard in Sophia-Antipolis" by Håkon W. Lie together with Bert Bos in July 1995 (as he explains in an interview).

CSS is based on a form of rules that consist of *selectors*, which select the document element(s) to which a rule applies, and a list of *property-value pairs* that define the styling of the selected element(s) with the help of CSS properties such as `font-`

`size` or `color`. There are two fundamental mechanisms for computing the CSS property values for any page element as a result of applying the given set of CSS rules: *inheritance* and *the cascade*.

The basic element of a CSS layout is a rectangle, also called "box", with an inner content area, an optional border, an optional padding (between content and border) and an optional margin around the border. This structure is defined by the CSS *box model*.

We will not go deeper into CSS in this tutorial, since our focus here is on the logic and functionality of an app, and not so much on its beauty.

1.4. JavaScript - "the assembly language of the Web"

> JavaScript was developed in 10 days in May 1995 by Brendan Eich then working at Netscape, as the HTML scripting language for their browser *Navigator 2* (more about history). Brendan Eich said (at the O'Reilly Fluent conference in San Francisco in April 2015): "I did JavaScript in such a hurry, I never dreamed it would become the assembly language for the Web".

JavaScript is a dynamic functional object-oriented 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 front-end web application with local or remote data storage, as described in the book Building Front-End Web Apps with Plain JavaScript.
4. Implementing a front-end component for a distributed web application with remote data storage managed by a back-end component, which is a server-side program that is traditionally written in a server-side language such as PHP, Java or C#, but can nowadays also be written in JavaScript with NodeJS.
5. Implementing a complete distributed web application where both the front-end and the back-end components are JavaScript programs.

The version of JavaScript that is currently fully supported by modern web browsers is called "ECMAScript 2015", or simply "ES2015", but the following versions, (ES2016, ES2017, ...), are already partially supported by current browsers and back-

end JS environments.

1.4.1 JavaScript as an object-oriented language

JavaScript is *object-oriented*, but in a different way than classical OO programming languages such as Java and C++. In JavaScript, classes, unlike objects and functions, have not been first-class citizens until ES2015 has introduced a `class` syntax. Before ES2015, classes had to be defined by following a code pattern in the form of special JS objects: either as *constructor* functions or as *factory* objects. Notice that when using (the syntactic sugar of) ES2015 `class` declarations, what is really defined internally, is still a constructor function.

However, objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class. This dynamism of JavaScript allows powerful forms of *meta-programming*, such as defining your own concepts of classes and enumerations (and other special datatypes).

1.4.2 Further reading about JavaScript

Good open access books about JavaScript are

- Speaking JavaScript, by Dr. Axel Rauschmayer.
- Eloquent JavaScript, by Marijn Haverbeke.
- Building Front-End Web Apps with Plain JavaScript, by Gerd Wagner

1.5. Accessibility for Web Apps

The recommended approach to providing accessibility for web apps is defined by the *Accessible Rich Internet Applications **(ARIA)*** standard. As summarized by Bryan Garaventa in his article on different forms of accessibility, there are 3 main aspects of accessibility for interactive web technologies: 1) keyboard accessibility, 2) screen reader accessibility, and 3) cognitive accessibility.

Further reading on ARIA:

1. How browsers interact with screen readers, and where ARIA fits in the mix by Bryan Garaventa
2. The Accessibility Tree Training Guide by whatsock.com
3. The ARIA Role Conformance Matrices by whatsock.com
4. Mozilla's ARIA overview article
5. W3C's ARIA Authoring Practices

1.6. Quiz Questions

1.6.1 Question 1: Well-formed XML documents

Which of the following statements represent a requirement for a well-formed XML document? Select one or more:

- ☐ The root element must have a *namespace* attribute.
- ☐ There must be one and only one *top level* element.

- ☐ All non-empty elements must have a *start-tag* and an *end-tag* with matching names.
- ☐ Element names must be *lower case*.

1.6.2 Question 2: Well-formed XML

Which of the following fragments are well-formed XML? Select one or more:

- ☐ This text is bold. and this is italicized and bold.</EM> and this is just italics.
- ☐ <STRONGER>This text is bold. <emph>And this is italicized and bold.</STRONGER> And this is just italics. </emph>
- ☐ <stronger>This text is bold. <emph>And this is italicized and bold.</emph></stronger><emph>And this is just italics. </emph>
- ☐ <emph><STRONGER>This is some text <bold>and this is more text. Here is even more</bold> text.</STRONGER> </emph>

1.6.3 Question 3: Valid XHTML

Which of the following fragments are valid XHTML? Select one or more:

- ☐
```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>My Title</title></head>
  <body><a href="http://www.examples.org"/>Jump!</a></body>
</html>
```
- ☐
```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>My Title</title></head>
  <body><p>Lorem ipsum...</p></body>
</html>
```
- ☐
```
<h:html xmlns:h="http://www.w3.org/1999/xhtml">
  <h:head><h:meta charset="utf-8"></h:head>
  <h:body><h:p>Lorem ipsum...</h:p></h:body>
</h:html>
```
- ☐
```
<h:html xmlns:h="http://www.w3.org/1999/xhtml">
  <h:head><h:title>My Title</h:title></h:head>
  <h:body><h:div>Lorem ipsum...</h:div></h:body>
</h:html>
```
- ☐
```
<h:html xmlns:h="http://www.w3.org/1999/xhtml">
  <head><title>My Title</title></head>
  <body><p>Lorem ipsum...</p></body>
</h:html>
```

1.6.4 Question 4: Valid XHTML5

Which of the following are correct statements about an XHTML5 document? Select one or more:

- ☐ The root element must be <xhtml>.
- ☐ The document must be well-formed.
- ☐ The root element and all its descendant elements must be in the namespace "http://www.w3.org/1999/xhtml".
- ☐ Case does not matter for element names, so both H1 and h1 can be used.

1.6.5 Question 5: HTML forms

Recall that an HTML form is a section of a document consisting of block elements that contain controls and labels on those controls. Which of the following form elements represent correct forms? Select one or more:

- ☐
```
<form>
  <div><label>ISBN: <input name="isbn" /></label></div>
  <div><label>Title: <input name="title" /></label></div>
</form>
```
- ☐
```
<form>
  <div>
    <label for="isbn">ISBN: </label><input id="isbn" name="isbn" />
    <label for="title">title: </label><input id="title" name="title" />
  </div>
</form>
```
- ☐
```
<form>
  <div><label>ISBN: </label><input name="isbn" /></div>
  <div><label>title: </label><input name="title" /></div>
</form>
```
- ☐
```
<form>
  <label>ISBN: <input name="isbn" /></label><br/>
  <label>Title: <input name="title" /></label>
</form>
```
- ☐
```
<form>
  <div><label for="isbn">ISBN: </label><input name="isbn" /></div>
  <div><label for="title">title: </label><input name="title" /></div>
</form>
```

Chapter 2. More on JavaScript

2.1. JavaScript Basics

In this summary we try to take all important points of the classical JavaScript summary by Douglas Crockford into consideration.

2.1.1 Types and data literals in JavaScript

JavaScript has three primitive datatypes: `string`, `number` and `boolean`, and we can test if a variable `v` holds a value of such a type with the help of the JS operator `typeof` as, for instance, in `typeof v === "number"`.

There are five reference types: `Object`, `Array`, `Function`, `Date` and `RegExp`. Arrays, functions, dates and regular expressions are special types of objects, but, conceptually, dates and regular expressions are primitive data values, and happen to be implemented in the form of wrapper objects.

The types of variables, array elements, function parameters and return values are not declared and are normally not checked by JavaScript engines. 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 reference*: either referencing an ordinary object, or an array, function, date, or regular expression;
- the special data value `null`, which is typically used as a default value for initializing an object variable;
- the special data value `undefined`, which is the implicit initial value of all variables that have been declared, but not initialized.

A ***string*** is a sequence of Unicode characters. String literals, like "Hello world!", 'A3F0', or the empty string "", are enclosed in single or double quotes. Two string expressions can be concatenated with the + operator, and checked for equality with the triple equality operator:

```
if (firstName + lastName === "JamesBond") ...
```

The number of characters of a string can be obtained by applying the `length` attribute to a string:

```
console.log("Hello world!".length);  // 12
```

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 explicit type distinction between integers and floating point numbers. If a numeric expression cannot be evaluated to a number, its value is set to `NaN` ("not a number"), which can be tested with the built-in predicate `isNaN(`*expr*`)`.

The built-in function, `Number.isInteger` allows testing if a number is an ***integer***. For making sure that a numeric value is an integer, or that a string representing a number is converted to an integer, one has to apply the built-in function `parseInt`. Similarly, a string representing a decimal number can be converted to this number with `parseFloat`. For converting a number `n` to a string, the best method is using `String(n)`.

There are two predefined ***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 implicitly converted to 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`. This conversion can be performed explicitly with the help of the double negation operation, like in the equality test `!!undefined === false`, which evaluates to `true`.

In addition to strings, numbers and Boolean values, also ***calendar dates*** and times are important types of primitive data values, although they are not implemented as primitive values, but in the form of wrapper objects instantiating `Date`. Notice that `Date` objects do, in fact, not really represent dates, but rather date-time instants represented internally as the number of milliseconds since 1 January, 1970 UTC. For converting the internal value of a `Date` object to a human-readable string, we have several options. The two most important options are using either the standard format of ISO date/time strings of the form "2015-01-27", or localized formats of date/time strings like "27.1.2015" (for simplicity, we have omitted the time part

of the date/time strings in these examples). When `x instanceof Date`, then `x.toISOString()` provides the ISO date/time string, and `x.toLocaleDateString()` provides the localized date/time string. Given any date string `ds`, ISO or localized, `new Date(ds)` creates a corresponding date object.

For testing the ***equality*** (or inequality) of two primitive data vales, always 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.

Assigning an ***empty array literal***, as in `var a = []` is the same as, but more concise than and therefore preferred to, invoking the `Array()` constructor without arguments, as in `var a = new Array()`.

Assigning an ***empty object literal***, as in `var o = {}` is the same as, but more concise than and therefore preferred to, invoking the `Object()` constructor without arguments, as in `var o = new Object()`. Notice, however, that an empty object literal `{}` is not really an empty object, as it contains property slots and method slots inherited from `Object.prototype`. So, a truly empty object (without any slots) has to be created with `null` as prototype, like in `var emptyObject = Object.create(null)`.

A summary of type testing is provided in the following table:

Type	Example values	Test if **x** of type
string	"Hello world!", 'A3F0'	`typeof x === "string"`
boolean	true, false	`typeof x === "boolean"`
(floating point) number	-2.75, 0, 1, 1.0, 3.1e10	`typeof x === "number"`
integer	-2, 0, 1, 250	`Number.isInteger(x)`
Object	{}, {num:3, denom:4}, {isbn:"006251587X," title:"Weaving the Web"}, {"one":1, "two":2, "three":3}	excluding `null`: `x instanceof Object` including `null`: `typeof x === "object"`
Array	[], ["one"], [1,2,3], [1,"one", {}]	`Array.isArray(x)`
Function	`function () { return "one"+1;}`	`typeof x === "function"`
Date	`new Date("2015-01-27")`	`x instanceof Date`
RegExp	`/(\w+)\s(\w+)/`	`x instanceof RegExp`

A summary of type conversions is provided in the following table:

Type	Convert to string	Convert string to type
boolean	`String(x)`	`Boolean(y)`
(floating point) number	`String(x)`	`parseFloat(y)`

Type	Convert to string	Convert string to type
integer	`String(x)`	`parseInt(y)`
Object	`x.toString()` or `JSON.stringify(x)`	`JSON.parse(y)`
Array	`x.toString()` or `JSON.stringify(x)`	`y.split()` or `JSON.parse(y)`
Function	`x.toString()`	`new Function(y)`
Date	`x.toISOString()`	`new Date(y)`
RegExp	`x.toString()`	`new RegExp(y)`

2.1.2 Variable scope

In ES5, there have only been two kinds of scope for variables declared with `var`: the global scope (with `window` as the context object) and function scope, but ***no block scope***. Consequently, declaring a variable with `var` within a code block is confusing and should be avoided. For instance, although this is a frequently used pattern, even by experienced JavaScript programmers, it is a pitfall to declare the counter variable of a `for` loop in the loop, as in

```
function foo() {
  for (var i=0; i < 10; i++) {
    ...  // do something with i
  }
}
```

Instead of obtaining a variable that is scoped to the block defined by the `for` loop, JavaScript is interpreting this code (by means of "hoisting" variable declarations) as:

```
function foo() {
  var i=0;
  for (i=0; i < 10; i++) {
    ...  // do something with i
  }
}
```

Therefore all function-scoped variable declarations (with `var`) should be placed at the beginning of a function. When a variable is to be scoped to a code block, such as to a `for` loop, it has to be declared with the keyword `let`, as in the following example:

```
function foo() {
  for (let i=0; i < 10; i++) {
    ...  // do something with i
  }
}
```

2.1.3 Frozen, or immutable, variables

Whenever a variable is supposed to be immutable (having a frozen value), it should be declared with the keyword `const`:

```
const pi = 3.14159;
```

It is generally recommended that variables be declared with `const` whenever it is clear that their values will never be changed. This helps catching errors and it allows the JS engine to optimize code execution.

2.1.4 Strict Mode

Starting from ES5, we can use strict mode for getting more runtime error checking. For instance, in strict mode, all variables must be declared. An assignment to an undeclared variable throws an exception.

We can turn strict mode on by typing the following statement as the first line in a JavaScript file or inside a `<script>` element:

```
'use strict';
```

It is generally recommended to use strict mode, except when code depends on libraries that are incompatible with strict mode.

2.1.5 Different kinds of objects

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

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

An empty object with no slots is created in the following way:

```
var o1 = Object.create( null);
```

Whenever the name in a slot is an admissible JavaScript identifier, 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 *key-value slot*, which is a map element, as explained below.

The name in a ***property slot*** may denote either

1. a ***data-valued property***, in which case the value is a *data value* or, more generally, a *data-valued expression*;

   or

2. an ***object-valued property***, in which case the value is an *object reference* or, more generally, an *object expression*.

The name in a ***method slot*** denotes a *JS function* (better called *method*), and its value is a *JS function definition expression*.

Object properties can be accessed in two ways:

1. Using the dot notation (like in C++/Java):

```
person1.lastName = "Smith"
```

2. Using a map notation:

```
person1["lastName"] = "Smith"
```

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

1. A ***record*** is a set of property slots like, for instance,

```
var myRecord = {firstName:"Tom", lastName:"Smith", age:26}
```

2. A ***map*** (also called 'associative array', 'dictionary', 'hash map' or 'hash table' in other languages) supports look-ups of *values* based on *keys* like, for instance,

```
var numeral2number = {"one":"1", "two":"2", "three":"3"}
```

   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 ***untyped object*** does not instantiate a class. It may have property slots and method slots like, for instance,

```
var person1 = {
  lastName: "Smith",
  firstName: "Tom",
  getFullName: function () {
    return this.firstName +" "+ this.lastName;
  }
};
```

   Within the body of a method slot of an object, the special variable `this` refers to the object.

4. A ***namespace*** (within the global scope) 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:

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

   A more advanced namespace mechanism, which allows avoiding global scope names for variables, functions and classes, is provided by *ES6 modules*.

5. A ***typed object*** instantiates a class that is defined either by a JavaScript constructor function or by a factory object. See "Defining and using classes".

2.1.6 Array lists

A JS array represents, in fact, the logical data structure of an *array list*, which is a list where each list item can be accessed via an index number (like the elements of an array). Using the term 'array' without saying 'JS array' creates a terminological

ambiguity. But for simplicity, we will sometimes just say 'array' instead of 'JS array'.

A variable may be initialized with a JS *array literal*:

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

Because they are array lists, JS 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 further array elements, and may even create gaps, like in

```
a[3] = 4;
a[5] = 5;
```

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`:

```
for (let 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:

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

For ***appending*** (all elements of) another array `b` to an array `a`, we `push` all the elements of `b` to `a` with the help of the spread operator `...`, like so:

```
a.push( ...b);
```

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

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

For ***searching*** a value `v` in an array `a`, we can use the predefined array method `indexOf`, which returns the position, if found, or -1, otherwise, as in:

```
if (a.indexOf(v) > -1)  ...
```

For ***looping*** over an array `a`, there are two good options: either use a classical `for` loop or an ES6 `for-of` loop. In any case, we can use a classical `for` (counter variable) loop:

```
for (let i=0; i < a.length; i++) {
  console.log( a[i]);
}
```

If no counter variable is needed, however, the best option is using a `for-of` loop (introduced in ES6):

```
for (const elem of a) {
  console.log( elem);
}
```

Notice that in a for-of loop, the looping variable (here: `elem`) can be declared as a *frozen* local variable with `const` whenever it is not re-assigned in the loop body.

Using the array looping method `forEach` is no longer a good idea: it's similar, but inferior to a `for-of` loop since it is syntactically more convoluted and cannot be interrupted with `break`.

For ***cloning*** an array `a`, we can use the array function `slice` in the following way:

```
var clone = a.slice(0);
```

Alternatively, we can use a new technique based on the ES6 spread operator:

```
var clone = [ ...a ];
```

2.1.7 Maps

A map (also called 'hash map', 'associative array' or 'dictionary') provides a mapping from keys to their associated values. Traditionally, before the built-in `Map` object has been added to JS (in ES6), maps have been implemented in the form of plain JS objects where the keys are string literals that may include blank spaces:

```
var emptyMap = Object.create(null); // instead of {}
var myTranslation = {
  "my house": "mein Haus",
  "my boat": "mein Boot",
  "my horse": "mein Pferd"
}
```

Alternatively, a proper map can be constructed with the help of the `Map` constructor:

```
var emptyMap = new Map();
var myTranslation = new Map([
  ["my house", "mein Haus"],
  ["my boat", "mein Boot"],
  ["my horse", "mein Pferd"]
])
```

A traditional map (as a plain JS object) is processed with the help of a loop where we loop over all keys using the predefined function `Object.keys(m)`, which returns an array of all keys of a map `m`. For instance,

```
for (const key of Object.keys( myTranslation)) {
  console.log(`The translation of ${key} is ${myTranslation[key]}`);
}
```

A proper map (i.e. a `Map` object) can be processed with the help of a for...of loop in one of the following ways:

```
// processing both keys and values
for (const [key, value] of myTranslation) {
  console.log(`The translation of ${key} is ${value}`);
}
// processing only keys
for (const key of myTranslation.keys()) {
  console.log(`The translation of ${key} is ${myTranslation.get( key)}`);
}
// processing only values
for (const value of myTranslation.values()) {
  console.log( value)
}
```

For ***adding*** a new entry to a traditional map, we simply associate the new value with its key as in:

```
myTranslation["my car"] = "mein Auto";
```

For ***adding*** a new entry to a proper map, we use the `set` operation:

```
myTranslation.set("my car", "mein Auto");
```

For ***deleting*** an entry from a traditional map, we can use the predefined `delete` operator as in:

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

For ***deleting*** an entry from a proper map, we can use the `Map::delete` method as in:

```
myTranslation.delete("my boat");
```

For ***testing*** if a traditional map contains an entry for a certain key value, such as for testing if the translation map contains an entry for "my bike" we can check the following:

```
if ("my bike" in myTranslation)  ...
```

For ***testing*** if a proper map contains an entry for a certain key value, we can use the Boolean-valued `has` method:

```
if (myTranslation.has("my bike"))  ...
```

For ***cloning*** a traditional map `m`, we can use the composition of `JSON.stringify` and `JSON.parse`. We first serialize `m` to a string representation with JSON.stringify, and then de-serialize the string representation to a map object with JSON.parse:

```
var clone = JSON.parse( JSON.stringify( m));
```

Notice that this method works well if the map contains only simple data values or (possibly nested) arrays/maps containing simple data values. In other cases, e.g. if the map contains `Date` objects, we have to write our own clone method. Alternatively, we could use a new technique based on the ES6 *spread* operator:

```
var clone = { ...m };
```

For ***cloning*** a proper map `m`, we can use the `Map` constructor in the following way:

```
var clone = new Map(m);
```

Since proper maps (defined as instances of `Map`) do not have the overhead of properties inherited from Object.prototype and operations on them, such as adding and deleting entries, are faster, they are preferable to using ordinary objects as maps. Only in cases where it is important to be compatible with older browsers that do not support `Map`, it is justified to use ordinary objects for implementing maps.

2.1.8 Important types of basic data structures

In summary, there are four types of important basic data structures:

1. ***array lists***, such as `["one","two","three"]`, which are special JS objects called 'arrays', but since they are dynamic, they are rather *array lists* as defined in the *Java* programming language.

2. ***records***, which are special JS objects, such as `{firstName:"Tom", lastName:"Smith"}`, as discussed above,

3. ***maps***, which can be realized as ordinary JS objects having only key-value slots, such as `{"one":1, "two":2, "three":3}`, or as `Map` objects, as discussed above,

4. ***entity tables***, like for instance the table shown below, which are special maps where the values are entity records with a standard ID (or *primary key*) slot, such that the keys of the map are the standard IDs of these entity records.

Table 2-1. An example of an entity table 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 our distinction between records, (traditional) maps and entity tables is a purely conceptual distinction, and not a syntactical one. For a JavaScript engine, both `{firstName:"Tom", lastName:"Smith"}` and `{"one":1,"two":2,"three":3}` are just objects. But conceptually, `{firstName:"Tom", lastName:"Smith"}` is a record because `firstName` and `lastName` are intended to denote properties (or fields), while `{"one":1,"two":2,"three":3}` is a map because `"one"` and `"two"` are not intended to denote properties/fields, but are just arbitrary string values used as keys for a map.

Making such conceptual distinctions helps in the logical design of a program, and mapping them to syntactic distinctions, even if they are not interpreted differently, helps to better understand the intended computational meaning of the code and therefore improves its readability.

2.1.9 Procedures, methods and functions

Generally, a (parametrized) *procedure* is like a sub-program that can be called (with a certain number of arguments) any number of times from within a program. Whenever a procedure returns a value, it is called a *function*. In OOP, procedures are called *methods*, if they are defined in the context of a class or of an object.

In JavaScript, procedures are called "functions", no matter if they return a value or not. As shown below in **Figure 2-1.** The built-in JavaScript classes `Object` and `Function`, JS functions are special JS objects, having an optional `name` property and a `length` property providing their number of parameters. If a variable `v` references a function can be tested with

```
if (typeof v === "function") {...}
```

Being JS objects implies that JS functions can be stored in variables, passed as arguments to functions, returned by functions, have properties and can be changed dynamically. Therefore, JS functions are first-class citizens, and JavaScript can be viewed as a functional programming language.

The general form of a JS ***function definition*** is an assignment of a JS *function expression* to a variable:

```
var myMethod = function theNameOfMyMethod( params) {
  ...
}
```

where `params` is a comma-separated list of parameters (or a parameter record), and `theNameOfMyMethod` is optional. When it is omitted, the method/function is ***anonymous***. In any case, JS functions are normally invoked via a variable that references the function. In the above case, this means that the JS function is invoked with `myMethod()`, and not with `theNameOfMyMethod()`. However, a named JS function can be invoked by name from within the function (when the function is recursive). Consequently, a recursive JS function must be named.

Anonymous function expressions are called *lambda expressions* (or shorter *lambdas*) in other programming languages.

As an example of an anonymous function expression being passed as an argument in the invocation of another (higher-order) function, we can take a comparison function being passed to the predefined function `sort` for sorting the elements of an array list. Such a comparison function must return a negative number if its first argument is smaller than its second argument, it must return 0 if both arguments are of the same rank, and it must return a positive number if the second argument is smaller than the first one. In the following example, we sort a list of lists of 2 numbers in lexicographic order:

```
var list = [[1,2],[1,3],[1,1],[2,1]];
list.sort( function (x,y) {
  return x[0] === y[0] ? x[1]-y[1] : x[0]-y[0];
});
```

Alternatively, we can express the anonymous comparison function in the form of an ***arrow function*** expression:

```
list.sort( (x,y) => x[0] === y[0] ? x[1]-y[1] : x[0]-y[0]);
```

A JS ***function declaration*** has the following form:

```
function theNameOfMyMethod( params) {...}
```

It is equivalent to the following named function definition:

```
var theNameOfMyMethod = function theNameOfMyMethod( params) {...}
```

that is, it creates both a function with name `theNameOfMyMethod` and a variable `theNameOfMyMethod` referencing this function.

JS functions can have ***inner functions***. The ***closure*** mechanism allows a JS function using variables (except `this`) from its outer scope, and a function created in a closure remembers the environment in which it was created. In the following example, there is no need to pass the outer scope variable `result` to the inner function via a parameter, as it is readily available:

```
var sum = function (numbers) {
  var result = 0;
  for (const n of numbers) {
    result = result + n;
  }
  return result;
};
console.log( sum([1,2,3,4]));  // 10
```

When a method/function is executed, we can access its arguments within its body by using the built-in `arguments` object, which is "array-like" in the sense that it has indexed elements and a `length` property, and we can iterate over it with a normal `for` loop, but since it's not an instance of `Array`, the JS array methods (such as the `forEach` looping method) cannot be applied to it. The `arguments` object contains an element for each argument passed to the method. This allows defining a method without parameters and invoking it with ***any number of arguments***, like so:

```
var sum = function () {
  var result = 0;
  for (let i=0; i < arguments.length; i++) {
    result = result + arguments[i];
  }
  return result;
};
console.log( sum(0,1,1,2,3,5,8));  // 20
```

A method defined on the prototype of a constructor function, which can be invoked on all objects created with that constructor, such as `Array.prototype.forEach`, where `Array` represents the constructor, has to be invoked with an instance of the class as ***context object*** referenced by the `this` variable (see also the next section on classes). In the following example, the array `numbers` is the context object in the invocation of `forEach`:

```
var numbers = [1,2,3];  // create an instance of Array
numbers.forEach( function (n) {
  console.log( n);
});
```

Whenever such a prototype method is to be invoked not with a context object, but with an object as an ordinary argument, we can do this with the help of ***the JS function `call` method*** that takes an object, on which the method is invoked, as its first parameter, followed by the parameters of the method to be invoked. For instance, we can apply the `forEach` looping method to the array-like object `arguments` in the following way:

```
var sum = function () {
  var result = 0;
  Array.prototype.forEach.call( arguments, function (n) {
    result = result + n;
  });
  return result;
};
```

A two-argument variant of the `Function.prototype.call` method, collecting all arguments of the method to be invoked in an array-like object, is `Function.prototype.apply`. The first argument to both `call` and `apply` becomes `this` inside the function, and the rest are passed through. So, `f.call( x, y, z)` is the same as `f.apply( x, [y, z])`.

Whenever a method defined for a prototype is to be invoked without a context object, or when a method defined in a method slot (in the context) of an object is to be invoked without its context object, we can bind its `this` variable to a given object with the help of ***the JS function bind method*** (`Function.prototype.bind`). This allows creating a shortcut for invoking a method, as in `var querySel = document.querySelector.bind( document)`, which allows to use `querySel` instead of `document.querySelector`.

The option of *immediately invoked JS function expressions* can be used for obtaining a namespace mechanism that is superior to using a plain namespace object, since it can be controlled which variables and methods are globally exposed and which are not. This mechanism is also the basis for JS *module* concepts. In the following example, we define a namespace for the model code part of an app, which exposes some variables and the model classes in the form of constructor functions:

```
myApp.model = function () {
  var appName = "My app's name";
  var someNonExposedVariable = ...;
  function ModelClass1() {...}
  function ModelClass2() {...}
  function someNonExposedMethod(...) {...}
  return {
    appName: appName,
    ModelClass1: ModelClass1,
    ModelClass2: ModelClass2
  }
}();  // immediately invoked
```

2.1.10 Defining and using classes

The concept of a ***class***is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods (as a blueprint) for the objects created with it.

Having a class concept is essential for being able to implement a *data model* in the form of ***model classes*** in a Model-View-Controller (MVC) architecture. However, classes and their inheritance/extension mechanism are over-used in classical OO languages, such as in Java, where all variables and procedures have to be defined in the context of a class and, consequently, classes are not only used for implementing *object types* (or model classes), but also as containers for many other purposes in these languages. This is not the case in JavaScript where we have the freedom to use classes for implementing *object types* only, while keeping method libraries in namespace objects.

Any code pattern for defining classes in JavaScript should satisfy five requirements. First of all, (1) it should allow to define a *class name*, a set of (instance-level) ***properties***, preferably with the option to keep them 'private', a set of (instance-level) ***methods***, and a set of *class-level properties and methods*. It's desirable that properties can be defined with a range/type, and with other meta-data, such as constraints. There should also be two introspection features: (2) an ***is-instance-of predicate***

that can be used for checking if an object is a direct or indirect instance of a class, and (3) an instance-level property for retrieving the ***direct type*** of an object. In addition, it is desirable to have a third introspection feature for retrieving the *direct supertype* of a class. And finally, there should be two inheritance mechanisms: (4) ***property inheritance*** and (5) ***method inheritance***. In addition, it is desirable to have support for *multiple inheritance* and *multiple classifications*, for allowing objects to play several roles at the same time by instantiating several role classes.

There was no explicit class definition syntax in JavaScript before ES6. Different code patterns for defining classes in JavaScript have been proposed and are being used in different frameworks. But they do often not satisfy the five requirements listed above. The two most important approaches for defining classes are:

1. In the form of a ***constructor*** function that achieves method inheritance via the prototype chain and allows to create new instances of a class with the help of the `new` operator. This is the classical approach recommended by Mozilla in their JavaScript Guide. This is also the approach implemented in ES6 with the new `class` definition syntax.

2. In the form of a ***factory*** object that uses the predefined `Object.create` method for creating new instances of a class. In this approach, the prototype chain method inheritance mechanism is replaced by a "copy & append" mechanism. Eric Elliott has argued that factory-based classes are a viable alternative to constructor-based classes in JavaScript (in fact, he even condemns the use of classical inheritance with constructor-based classes, throwing out the baby with the bath water).

When building an app, we can use both kinds of classes, depending on the requirements of the app. Since we often need to define class hierarchies, and not just single classes, we have to make sure, however, that we don't mix these two alternative approaches within the same class hierarchy. While the factory-based approach, as exemplified by mODELcLASSjs, has many advantages, which are summarized in **Table 2-2.** Required and desirable features of JS code patterns for classes, the constructor-based approach enjoys the advantage of higher performance object creation.

Table 2-2. Required and desirable features of JS code patterns for classes

Class feature	Constructor-based approach	Factory-based approach	mODELcLASSjs
Define properties and methods	yes	yes	yes
is-instance-of predicate	yes	yes	yes
direct type property	yes	yes	yes
direct supertype property of classes	no	possibly	yes
Property inheritance	yes	yes	yes
Method inheritance	yes	yes	yes
Multiple inheritance	no	possibly	yes
Multiple classifications	no	possibly	yes
Allow object pools	no	yes	yes

Constructor-based classes

Only in ES6, a user-friendly syntax for constructor-based classes has been introduced. In **Step 1.a)**, a base class

`Person` is defined with two properties, `firstName` and `lastName`, as well as with an (instance-level) method `toString` and a static (class-level) method `checkLastName`:

```
class Person {
    constructor( first, last) {
    this.firstName = first;
    this.lastName = last;
  }
    toString() {
    return this.firstName + " " +
        this.lastName;
  }
  static checkLastName( ln) {
    if (typeof ln !== "string" ||
        ln.trim()==="") {
      console.log("Error: invalid last name!");
    }
  }
}
```

In **Step 1.b)**, class-level ("static") properties are defined:

```
Person.instances = {};
```

Finally, in **Step 2**, a subclass is defined with additional properties and methods that possibly override the corresponding superclass methods:

```
class Student extends Person {
    constructor( first, last, studNo) {
      super.constructor( first, last);
    this.studNo = studNo;
  }
  // method overrides superclass method
    toString() {
    return super.toString() + "(" +
        this.studNo +")";
  }
}
```

In ES5, we can define a base class with a subclass in the form of constructor functions, following a code pattern recommended by Mozilla in their JavaScript Guide, as shown in the following steps.

Step 1.a) First define the constructor function that implicitly defines the properties of the class by assigning them the values of the constructor parameters when a new object is created:

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

```
}
```

Notice that within a constructor, the special variable `this` refers to the new object that is created when the constructor is invoked.

Step 1.b) Next, define the *instance-level methods* of the class as method slots of the object referenced by the constructor's `prototype` property:

```
Person.prototype.toString = function () {
  return this.firstName + " " + this.lastName;
}
```

Step 1.c) *Class-level* ("static") methods can be defined as method slots of the constructor function itself (recall that, since JS functions are objects, they can have slots), as in

```
Person.checkLastName = function (ln) {
  if (typeof ln !== "string" || ln.trim()==="") {
    console.log("Error: invalid last name!");
  }
}
```

Step 1.d) Finally, define class-level ("static") properties as property slots of the constructor function:

```
Person.instances = {};
```

Step 2.a) Define a subclass with additional properties:

```
function Student( first, last, studNo) {
  // invoke superclass constructor
    Person.call( this, first, last);
  // define and assign additional properties
  this.studNo = studNo;
}
```

By invoking the supertype constructor with `Person.call( this, ...)` for any new object created as an instance of the subtype `Student`, and referenced by `this`, we achieve that the property slots created in the supertype constructor (`firstName` and `lastName`) are also created for the subtype instance, along the entire chain of supertypes within a given class hierarchy. In this way we set up a **property inheritance** mechanism that makes sure that the own properties defined for an object on creation include the own properties defined by the supertype constructors.

In **Step 2b)**, we set up a mechanism for **method inheritance** via the constructor's `prototype` property. We assign a new object created from the supertype's `prototype` object to the `prototype` property of the subtype constructor and adjust the prototype's constructor property:

```
// Student inherits from Person
  Student.prototype = Object.create(
      Person.prototype);
// adjust the subtype's constructor property
Student.prototype.constructor = Student;
```

With `Object.create( Person.prototype)` we create a new object with `Person.prototype` as its prototype and without any own property slots. By assigning this object to the `prototype` property of the subclass constructor, we achieve that the methods defined in, and inherited from, the superclass are also available for objects instantiating the subclass. This mechanism of chaining the prototypes takes care of method inheritance.

Step 2c) Define a subclass method that overrides a superclass method:

```
Student.prototype.toString = function () {
  return Person.prototype.toString.call( this) +
      "(" + this.studNo + ")";
};
```

An instance of a constructor-based class is created by applying the `new` operator to the constructor and providing suitable arguments for the constructor parameters:

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

The method `toString` is invoked on the object `pers1` by using the 'dot notation':

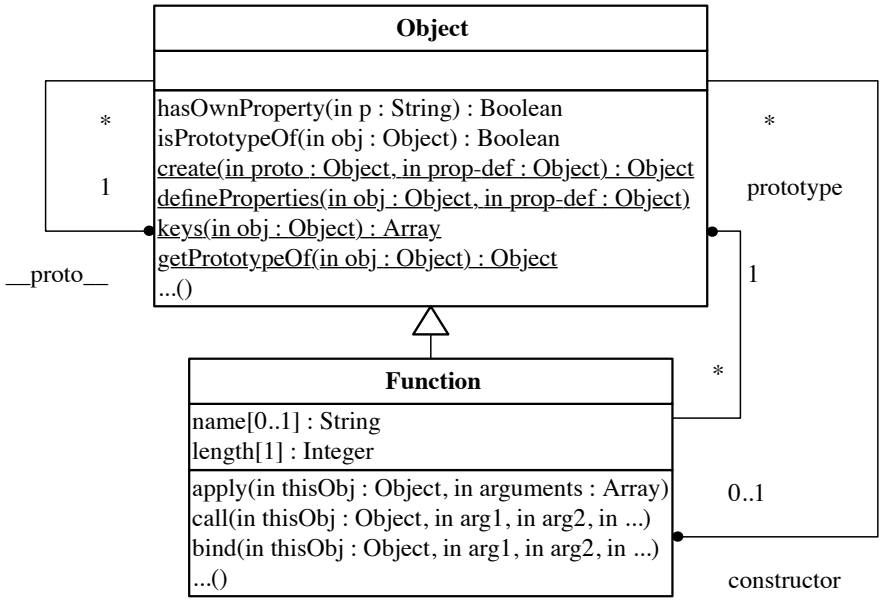
```
alert("The full name of the person is: " + pers1.toString());
```

When an object `o` is created with `o = new C(... )`, where `C` references a named function with name "C", the type (or class) name of `o` can be retrieved with the introspective expression `o.constructor.name`, which returns "C".

In JavaScript, a ***prototype*** object is an object with method slots (and sometimes also property slots) that can be inherited by other objects via JavaScript's method/property slot look-up mechanism. This mechanism follows the ***prototype chain*** defined by the built-in reference property `__proto__` (with a double underscore prefix and suffix) for finding methods or properties. As shown below in **Figure 2-1.** The built-in JavaScript classes `Object` and `Function`, every constructor function has a reference to a prototype object as the value of its reference property `prototype`. When a new object is created with the help of `new`, its `__proto__` property is set to the constructor's `prototype` property.

For instance, after creating a new object with `f = new Foo()`, it holds that `Object.getPrototypeOf(f)`, which is the same as `f.__proto__`, is equal to `Foo.prototype`. Consequently, changes to the slots of `Foo.prototype` affect all objects that were created with `new Foo()`. While every object has a `__proto__` property slot (except `Object`), only objects constructed with `new` have a `constructor` property slot.

Figure 2-1. *The built-in JavaScript classes `Object` and `Function`*



Notice that we can also retrieve the prototype of an object with `Object.getPrototypeOf(o)`, as an alternative to `o.__proto__`.

Factory-based classes

In this approach we define a JS object `Person` (actually representing a class) with a special `create` method that invokes the predefined `Object.create` method for creating objects of type `Person`:

```
var Person = {
  typeName: "Person",
    properties: {
    firstName: {range:"NonEmptyString", label:"First name",
        writable: true, enumerable: true},
    lastName: {range:"NonEmptyString", label:"Last name",
        writable: true, enumerable: true}
  },
    methods: {
    getFullName: function () {
      return this.firstName +" "+ this.lastName;
    }
  },
    create: function (slots) {
    // create object
    var obj =   Object.create( this.methods, this.properties);
    // add special property for *direct type* of object
    Object.defineProperty( obj, "type",
        {value: this, writable: false, enumerable: true});
    // initialize object
    for (prop of Object.keys( slots)) {
      if (prop in this.properties) obj[prop] = slots[prop];
    }
    return obj;
  }
```

```
};
```

Notice that the JS object `Person` actually represents a factory-based class. An instance of such a factory-based class is created by invoking its `create` method:

```
var pers1 = Person.create( {firstName:"Tom", lastName:"Smith"});
```

The method `getFullName` is invoked on the object `pers1` of type `Person` by using the 'dot notation', like in the constructor-based approach:

```
alert("The full name of the person are: " + pers1.getFullName());
```

Notice that each property declaration for an object created with `Object.create` has to include the 'descriptors' `writable: true` and `enumerable: true`, as in lines 5 and 7 of the `Person` object definition above.

In a general approach, like in the mODELcLASSjs library for model-based development, we would not repeatedly define the `create` method in each class definition, but rather have a generic constructor function for defining factory-based classes. Such a factory-based class constructor, like mODELcLASS, would also provide an ***inheritance*** mechanism by merging the own properties and methods with the properties and methods of the superclass. This mechanism is also called *Inheritance by Concatenation*.

2.2. Asynchronous Programming

In programming, we often have the situation that, when calling a possibly time-consuming input/output (I/O) operation (or any long-running operation, e.g., for performing a complex computation), the program execution has to wait for its result being returned before it can go on. Calling such an operation and waiting for its result, while the main program's further execution (and its entire thread) is blocked, represents a *synchronous* operation call. The implied waiting/blocking poses a problem for a JS program that is being executed in a browser thread since during the waiting time the user interface (in a browser tab) would be frozen, which is not acceptable from a usability point of view and therefore not accepted by browsers.

Consequently, in JavaScript, it is not possible to call an I/O operation, e.g., for fetching data from a webpage (with the built-in `XMLHttpRequest` or `fetch` API) or for accessing a remote database (via HTTP request-response messaging) synchronously. These types of operations have to be performed in an ***asynchronous*** (non-blocking) manner, instead.

Asynchronous programming concepts in JavaScript have undergone an evolution from *callbacks* to *promises* to *generators* (coroutines) and, most recently, to *asynchronous procedure calls* with `await` procedure invocation expressions and asynchronous procedure definitions with `async`. Each evolution step has made asynchronous programming a little bit easier for those who have taken the effort to get familiar with it.

Due to this evolution, operations of older JS input/output APIs available in the form of built-in objects, like `XMLHttpRequest` for HTTP messaging or `indexedDB` for object database management, work with callbacks, while newer APIs, like `fetch` for HTTP messaging, work with promises and can also be invoked with `await`.

Callbacks

A simple asynchronous programming approach consists of defining a procedure that is to be executed as soon as the asynchronous operation completes. This allows to continue the program execution after the invocation of the asynchronous operation, however, without assuming that the operation result is available. But how does the execution environment know, which procedure to call after completing the asynchronous operation?

In JS, we can pass a JS function as an argument in the invocation of the asynchronous operation. A *callback* is such a JS

function.

Consider the following example. An external JS file can be dynamically loaded (in the context of an already loaded webpage with associated JS code) by (1) programmatically creating an HTML `script` element DOM object with the file's URL as the value of the script's `src` attribute, and (2) inserting the newly created `script` element after the last child node of the document's `head` element:

```
function loadJsFile( fileURL) {
  const scriptEl = document.createElement("script");
  script.src = fileURL;
  document.head.append( scriptEl);
}
```

When the new script element is inserted into the document's DOM, e.g., with the help of the asynchronous DOM operation `append` (at the end of the `loadJsFile` procedure), the browser will load the JS file and then parse and execute it, which will take some time. Let's assume that we have a JS code file containing the definition of a function `addTwoNumbers` that does what its name says and we first load the file and then invoke the function in the following way:

```
loadJsFile("addTwoNumbers.js");
console.log( addTwoNumbers( 1, 2));
```

This wouldn't work. We would get an error message instead of the sum of 1 and 2, since the intended result of the first statement, the availability of the `addTwoNumbers` function, is not (yet) obtained when the second statement is executed.

We can fix this by adding a callback procedure as a second parameter to the `loadJsFile` procedure and assign it as an event handler of the JS file `load` event :

```
function loadJsFile( fileURL, callback) {
  const scriptEl = document.createElement("script");
  script.src = fileURL;
  script.onload = callback;
  document.head.append( scriptEl);
}
```

Now when calling `loadJsFile` we can provide the code to be executed after loading the "addTwoNumbers.js" file in an anonymous callback function:

```
loadJsFile("addTwoNumbers.js", function () {
  console.log( addTwoNumbers( 1, 2));  // results in 3
]);
```

Since the loading of the JS file can fail, we should better add some error handling for this case by defining an event handler for the `error` event. We can handle possible errors within the *callback* procedure by calling it with an error argument:

```
function loadJsFile( fileURL, callback) {
  const scriptEl = document.createElement("script");
  script.src = fileURL;
  script.onload = callback;
  script.onerror = function () {
      callback( new Error(`Script load error for ${fileURL}`));
```

```
  };
  document.head.append( scriptEl);
}
```

Now we call `loadJsFile` with an anonymous callback function having an `error` parameter:

```
loadJsFile("addTwoNumbers.js", function (error) {
  if (!error) console.log( addTwoNumbers(1,2));  // results in 3
  else console.log( error);
]);
```

Callbacks work well as an asynchronous programming approach in simple cases. But when it is necessary to perform several asynchronous operations in a sequence, one quickly ends up in a "callback hell", a term that refers to the resulting deeply nested code structures that are hard to read and maintain.

Promises

A *promise* (also called *future* in some programming languages, like in Python) is a special object that provides the deferred result of an asynchronous operation to the code that waits for this result. A promise object is initially in the state *pending*. If the asynchronous operation succeeds (in the case when the `resolve` function is called with an argument providing the result value), the promise state is changed from *pending* to *fulfilled*. If it fails (in the case when the `reject` function is called with an argument providing the error), the promise state is changed from *pending* to *rejected*.

An example of a built-in asynchronous operation that returns a promise is `import` for dynamically loading JS code files (and ES6 modules). We can use it instead of the user-defined `loadJsFile` procedure discussed in the previous section for loading the `addTwoNumbers.js` file and subsequently executing code that uses the `addTwoNumbers` function (or reporting an error if the loading failed):

```
import("addTwoNumbers.js")
.then( function () {
  console.log( addTwoNumbers( 1, 2));
})
.catch( function (error) {
  console.log( error);
});
```

This example code shows that on the promise object returned by `import` we can call the predefined functions `then` and `catch`:

`then`
: for continuing the execution only when the `import` operation is completed with a *fulfilled* promise, and

`catch`
: for processing the error result of a *rejected* promise.

The general approach of *asynchronous programming with promises* requires each asynchronous operation to return a promise object that typically provides either a result value, when the promise is fulfilled, or an error value, when the promise is rejected. For user-defined asynchronous procedures, this means that they have to create a promise as their return value, as shown in the promise-valued `loadJsFile` function presented below.

A promise object can be created with the help of the `Promise` constructor by providing an anonymous function expression

as the argument of the `Promise` constructor invocation (with two parameters `resolve` and `reject` representing JS functions). We do this in the following example of a promise-valued `loadJsFile` function, which is a variant of the previously discussed callback-based `loadJsFile` procedure:

```
function loadJsFile( fileURL) {
  return new Promise( function (resolve, reject) {
    const scriptEl = document.createElement("script");
    scriptEl.src = fileURL;
    scriptEl.onload = resolve;
    scriptEl.onerror = function () {
        reject( new Error(`Script load error for ${fileURL}`));
    };
    document.head.append( scriptEl);
  });
}
```

This new version of the asynchronous `loadJsFile` operation is used in the following way:

```
loadJsFile("addTwoNumbers.js")
.then( function () {
  console.log( addTwoNumbers( 1, 2));
})
.catch( function (error) {
  console.log( error);
});
```

We can see that even the syntax of a simple promise-valued function call with `then` and `catch` is more clear than the syntax of a callback-based asynchronous procedure call. This advantage is even more significant when it comes to chaining asynchronous procedure calls, as in the following example where we first sequentially load three JS files and then invoke their functions:

```
loadJsFile("addTwoNumbers.js")
.then( function () {
  return loadJsFile("multiplyBy3.js");})
.then( function () {
  return loadJsFile("decrementBy2.js");})
.then( function () {
  console.log( decrementBy2( multiplyBy3( addTwoNumbers(1,2))));})
.catch( function (error) {
  console.log( error);
});
```

Notice that for executing a sequence of asynchronous operations with `then`, we need to make sure that each `then`-function returns a promise.

As an alternative to the sequential execution of asynchronous operations, we may also execute them **in parallel** with `Promise.all`:

```
Promise.all([ loadJsFile("addTwoNumbers.js"),
  loadJsFile("multiplyBy3.js"),
  loadJsFile("decrementBy2.js")
```

```
])
.then( function () {
  console.log( decrementBy2( multiplyBy3( addTwoNumbers(1,2))));
})
.catch( function (error) {console.log( error);});
```

Unlike `loadJsFile`, which simply completes with a side effect (the loading of JS code), but without a result value being returned, a typical asynchronous operation returns a promise object that provides either a result value, when the promise is fulfilled, or an error value, when the promise is rejected.

Let's consider another example, where we have asynchronous operations with result values. The JS built-in `fetch` operation allows retrieving the contents of a remote resource file via sending HTTP request messages in two steps:

1. In the first step, it returns a promise that resolves with a `response` object as its result value containing the HTTP header information retrieved.

2. Then, invoking the `text()` or the `json()` function on the previously retrieved `response` object returns a promise that resolves to the HTTP response message's body (in the form of a string or a JSON object) when it is retrieved from the remote server.

In such a case, when we chain two or more asynchronous operation calls with result values, each successor call can be expressed as a transformation from the previous result to a new result using arrow functions as shown in line 2 of the following example:

```
fetch("user1.json")
.then( response => response.json())
.then( function (user1) {alert( user1.name);})
.catch( function (error) {console.log( error);});
```

Notice that the text file "user1.json" is assumed to contain a JSON object describing a particular user with a *name* field. This JSON object is retrieved with the arrow function expression in line 2.

Calling asynchronous operations with `await`

When a program with a statement containing an asynchronous procedure call (with `await`) is executed, the program will run up to that statement, call the procedure, and suspend execution until the asynchronous procedure execution completes, which means that if it returns a Promise, it is *settled*. That suspension of execution means that control is returned to the event loop, such that other asynchronous procedures also get a chance to run. If the Promise of the asynchronous procedure execution is fulfilled, the execution of the program is resumed and the value of the `await` expression is that of the fulfilled Promise. If it is rejected, the `await` expression throws the value of the rejected Promise (its error).

When we use `await` for invoking a Promise-valued JS function, we typically do not use Promise chaining with `.then`, because `await` handles the waiting for us. And we can use a regular `try-catch` block instead of a Promise chaining `.catch` clause, as shown in the following example code:

```
try {
  await loadJsFile("addTwoNumbers.js");
  console.log( addTwoNumbers(2,3));
} catch (error) {
  console.log( error);
}
```

Notice that this is the code of an ES6 module. In a normal JS file, `await` can only be used within `async` functions.

When we call several asynchronous procedures in succession with `await`, the code reads in a natural way, similar to the code for calling synchronous procedures:

```
try {
  await loadJsFile("addTwoNumbers.js");
  await loadJsFile("multiplyBy3.js");
  await loadJsFile("decrementBy2.js");
  console.log( decrementBy2( multiplyBy3( addTwoNumbers(2,3))));
} catch (error) {
  console.log( error);
}
```

In an `async` function, we can invoke Promise-valued functions in `await` expressions. Since an `async` function returns a Promise, it can itself be invoked with `await`.

```
async function load3JsFiles() {
  await loadJsFile("addTwoNumbers.js");
  await loadJsFile("multiplyBy3.js");
  await loadJsFile("decrementBy2.js");
}
try {
  await load3JsFiles();
  console.log( decrementBy2( multiplyBy3( addTwoNumbers(2,3))));
} catch (error) {
  console.log( error);
}
```

In the more typical case of asynchronous operation calls with result values, we obtain code like the following await-based version of the above promise-based example of using `fetch`:

```
try {
  const response = await fetch("user1.json");
  const user1 = await response.json();
  alert( user1.name);
} catch (error) {
  console.log( error);
}
```

For more about asynchronous programming techniques, see Promises, async/await and Demystifying Async Programming in Javascript.

2.3. Using ES6 Modules

Normal modules are library code files that explicitly ***export*** those (variable, function and class) names that other modules can use (as implicitly frozen like `const` declarations). A module that is to use items from another module needs to explicitly ***import*** them from that other module using import statements. It is recommended that all JS module files use the file extension "mjs" for indicating that they are different from classical script files.

Web pages can load module files, possibly along with classical script files, with the help of a special type of `script`

element.

The meaning of ES6 modules is based on the following principles:

1. A JS library file can be turned into a module by using "export" for all library items. Other modules can "import" its items.

2. Any ordinary script file that is to use one or more items from a module has itself to be turned into a module ("only modules can use modules"). Since it doesn't export anything, such a module could also be called an "import module".

3. All modules, no matter if they export anything or are just "import modules", are separated from the global scope in the following sense: they have read access to items from the global scope such as DOM objects (like `document`) or other global objects (like `Array`), but they cannot create any names (including objects and functions) in the global scope. This implies, for instance, that a JS function defined in a module cannot be assigned to an `onclick` event handler attribute in an HTML file..

Using modules implies that we can no longer use the global scope for the names of functions/classes, which is a restriction that is considered a good practice in software engineering.

An example of a normal (library) module file is `util.mjs` with the following code:

```
function isNonEmptyString(x) {
  return typeof(x) === "string" && x.trim() !== "";
}
...
export { isNonEmptyString, ... };
```

An example of a module that imports certain items from other modules and then uses them in its own code, and also exports some of its own items is the model class file `Book.mjs` with the following import/export statements:

```
import { isNonEmptyString, ... } from "../../lib/util.mjs";
import { NoConstraintViolation, MandatoryValueConstraintViolation, ... }
    from "../../lib/errorTypes.mjs";
export default function Book( slots) {...}
```

Since this module only exports one class (***Book***), a default export us used, allowing simplified imports.

An example of a module that does not export anything, but only imports certain items, is the view code file `createBook.mjs` with the following import statements:

```
import Book from "../../src/m/Book.mjs";
import { fillSelectWithOptions } from "../../lib/util.mjs";
...
```

An HTML page (here: `createBook.html`) can load such a module with a special type of `script` element:

```
<script src="src/v/createBook.mjs" type="module"></script>
```

Notice that this `script` element's `type` attribute is set to "module".

Alternatively, the code of such a module can be embedded in the HTML page like so:

```
<script type="module">
 import Book from "./src/m/Book.mjs";
 const clearButton = document.getElementById("clearData");
 // Set event handler for the button "clearData"
 clearButton.addEventListener("click", function () {Book.clearData();});
</script>.
```

2.4. Quiz Questions

2.4.1 Question 1: Data values and objects

Which of the following statements about data values and objects in JS are true? Select one or more:

- ☐ `true` is an object.
- ☐ A JS array is a JS object.
- ☐ `false` is a data value.
- ☐ A JS function is a JS object.
- ☐ `1` is a data value.
- ☐ `Infinity` is an object.

2.4.2 Question 2: Evaluating a Boolean expression

What is the value of the Boolean expression `null || !0` ? Select one:

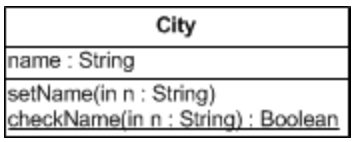
- ☐ true
- ☐ false

2.4.3 Question 3: JavaScript datatypes

Which of the following denote primitive datatypes in JavaScript? Select one or more:

1. double
2. string
3. float
4. int
5. boolean
6. byte
7. number

2.4.4 Question 4: Constructor-based class definition



Which of the following JavaScript fragments correctly defines the constructor-based class `City` shown in the class diagram (either using a constructor function definition or an ES6 class definition)? Hint: notice that `setName` is an instance-level method while `checkName` is a class-level ("static") method. Select one or more:

□
```
function City( n) {
  this.name = n;
  this.setName = function (n) {this.name = n;};
  checkName = function (n) {...}; // returns true or false
}
```

□
```
class City {
  constructor (n) {
    this.setName( n);
  }
  setName( n) {if (City.checkName( n)) this.name = n;}
  static checkName( n) {...}  // returns true or false
}
```

□
```
function City( n) {
  this.setName( n);
  function checkName( n) {...} // returns true or false
}
City.prototype.setName = function (n) {this.name = n;};
```

□
```
function City( n) {
  this.setName( n);
}
City.prototype.setName = function (n) {
  if (City.checkName( n)) this.name = n;
};
City.checkName = function (n) {...};  // returns true or false
```

2.4.5 Question 5: Type coercion

Consider the following JavaScript code:

```
var a = 5;
var b = "7";
var c = a + b;
```

What is the value of the variable `c`? Select one:

□ The string "57"

□ The number 12

□ `undefined`

□ The code will result in an error since you can't use the + operator between two operands of different types.

2.4.6 Question 6: Variable scope

What is the output of the following program?

```
function foo() {
  var i=7;
  for (var i=0; i < 10; i++) {
    ...  // do something
  }
  console.log( i);
};
foo();
```

Answer: _____

Chapter 3. Building Web Apps with Firebase

3.1. Introducing Firebase

Firebase is a platform for creating cloud-based web applications without dealing with the complexity of managing server hardware and server software. From the developer's standpoint, the absence of such complexity means better focusing on web development.

Launched in 2011 by Firebase Inc. and acquired by Google in 2014, Firebase became a popular cloud computing solution among start-ups and businesses that opted for ***Backend-as-a-service (BaaS)*** solutions. Rather than building an ad-hoc server infrastructure, in a BaaS solution, ***APIs*** and ***SDKs*** connect the *frontend* of the apps to *cloud-based backend services*.

Firebase and other *BaaS* providers offer clear benefits for web development:

- ***Speed***: automating most of the backend tasks, so a backend environment can be set up in hours. This time-saving matches with modern agile development philosophy and methods used nowadays by development teams.

- ***Cost***: through a lower learning curve, developers quickly become more efficient, adding that businesses do not need to invest high amounts of money in servers, allowing them to scale their apps as they grow.

Although started as a real-time database, Firebase has evolved as a whole set of services, tools and APIs for mobile (Android and iOS) and web-based applications, aiming to address the entire development life-cycle: ***build***, ***test***, and ***manage***.

This tutorial uses the most essential services for building web applications with plain JavaScript, such as Firebase Authentication, Firestore, Firebase Hosting, and Firebase Functions.

3.1.1 Firestore: a Cloud Database Management System

Firebase provides two database management systems (DBMS): Realtime Database and Firestore. Both offer a NoSQL DBMS as a Service for mobile and web apps. Unlike the older Realtime Database technology, which essentially manages a large JSON tree, Firestore aims at facilitating scalability, mainly through:

- **complex data models**, based on

1. Firestore documents correspond to unique JS objects, representing entity records with possibly complex-valued fields;
2. Firestore collections of documents correspond to object stores or entity tables;

- **better querying options**, allow chain and/or combine filters and sorting in a single query.

Firestore SDKs are for server-side programming code in *Java*, *Python*, *Node.js*, *Ruby*, *PHP*, *Golang*, *.NET*, and *C#* for both DBMSs.

3.1.2 Firebase Hosting

Firebase Hosting is a static web hosting solution for websites and applications built with HTML, CSS, and JavaScript: single-page applications and progressive web apps. Firebase Hosting includes a free Content Delivery Network (CDN), and free SSL/HTTPS, part of Google Cloud Platform. Paired with Cloud Functions and Firestore, we can build microservices and APIs. Firebase Hosting behaviour is highly configurable, allowing URL redirections, URL rewriting, direct HTTP requests to functions or Cloud Run containers (virtualized applications), customization of dynamic links, header configuration, and more.

3.1.3 Firebase Authentication

Firebase Authentication is a user-authentication solution for mobile and web apps. It allows us to use pre-built user interfaces (***FirebaseUI***) or create custom user interfaces for login management and authentication. It handles the most common authentication methods, such as using custom credentials, emails, or federated social media accounts, such as Google, Apple, Facebook, Microsoft, Yahoo, Twitter, GitHub, and more.

3.1.4 Pricing and Billing Concerns

Firebase provides a free plan with limited resources (see the free quota of the Spark Plan). However, these resource limits are sufficient for our tutorials and for creating real-world applications. The paid ('Blaze') plan offers a "Pay as you go" billing model under which we pay what we get, not more and not less. Nevertheless, the Spark Plan includes all Firebase cloud services, such as Authentication, Firestore, Cloud Functions, Hosting, Firebase Machine Learning, Real-Time Database, Storage, and many other features that enable web developers to create web apps, websites, games, mobile apps, etc.

In a real-world development project with Firebase, we should make design decisions only after understanding how billing works; otherwise, we may turn a technically successful project into a financially unsustainable business. In this tutorial, this issue is taken very seriously. We will never get an undesirable outcome in the monthly bill if we use it with the small sample of records presented in this app. Therefore, some portions of code aim to exemplify concepts presented in this educational scenario but never be used in an actual situation.

For instance, consider this snippet to retrieve all documents in a collection:

```
const booksQrySns = await getDocs( collection( db, "books"));
```

If the collection "books" contains just a few records, we will never get close to the limits of the free plan. Still, if the same statement is repeated several times, querying a database with many thousands of records, we should expect a hefty bill from Google at the end of the month.

One way to control the number of Firestore *read* operations is by setting up limits in our queries. When used with pagination, this will provide complete control of our Firestore resource consumption. Additionally, setting daily or monthly spending limits is always a good practice.

Read more about Firebase billing plans and how Firestore is billed.

3.2. Firebase JS SDK version 9, the "Modular Version"

This tutorial uses Firebase JavaScript/Web SDK version 9, better known as the "modular version", rather than version 8, known as the "namespaced version". Currently, Firebase supports both versions, but version 8 will be deprecated soon, and the Firebase encourages new apps' creators to adopt version 9.

This new version released in 2021 takes advantage of:

- **The use of ES6 Modules**, facilitates "tree-shaking" or removing unused code to create smaller and faster web applications. Let's consider that JS began as a language for making small websites and that today it is widely used to create large scale and complex applications. The advent of modules allows developers to separate functionality, simplify dependency management, reuse code and manage the extensibility of code.

- **No side-effect imports**, where we explicitly import the functions used in our app's code. An imported file represents a side-effect import when it includes JS functions that change something different from its own parameters and return value (such as global variables or variables of its outer scope). When we use side-effect imports, we cannot ensure what exactly is being imported, such as in:

  ```
  import "firebase/app";
  ```

  on the other hand, *no side-effect imports* mean that we import individually each function used within a JS module, for instance:

  ```
  import { initializeApp } from "firebase/app";
  ```

- **Smaller libraries**, an enormous reduction in size that improves web performance, particularly in the packages of Firebase Authentication (72% smaller) and Firestore (40% smaller), and up to 84% smaller if we opt for the Firestore Lite JavaScript/Web SDK, a constrained but ultra-light subpackage. In this tutorial series, we will use both the standard and lite versions.

As we can see, the design of Firebase JS/Web SDK version 9 aims to optimize the performance of web apps and increase understandability in our code. When we code using this version, our code will be organized around the functions imported from the SDK libraries. Let's see an example of the use of the "modular" version of the Firebase SDKs for adding Firebase and Firestore to a project:

1. Import the `initializeApp()` function using the ES6 modules specifiers of the core Firebase SDK library and the `getFirestore()` function from the Firestore Web SDK, installed locally using *nmp*.

   ```
   import { initializeApp } from "firebase/app";
   import { getFirestore } from "firebase/firestore";
   ```

2. Initialize a Firebase App object using the `initializeApp()` function and `firebaseConfig`, an object created with the project configuration.

   ```
   const firebaseConfig = {
     // TODO: Replace the following with your web app's Firebase project configuration
   };
   // Initialize a Firebase App object
   initializeApp( firebaseConfig);
   ```

3. Initialize Firestore using the `getFirestore()` function. From now on, the object "`db`" represents the interface to access

our Firestore DB instance on the cloud.

```
// Initialize Firestore interface
const db = getFirestore();
```

Notice that the "modular" version of the Firebase JS SDK has been optimized for using *module bundlers*, such as Webpack or Rollup, therefore it is expected you invoke the Firestore Web SDK from your local version installed using *nmp*. Nevertheless, when we don't want to use *module bundlers* –as in this tutorial– we must import the SDKs from the CDN, like:

```
import { initializeApp } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-app.js";
import { getFirestore, doc, collection, query, setDoc, getDoc } from "https://www.gstatic.
```

3.3. Firestore Database Model

Firestore, or Firestore, is a ***NoSQL database***, and unlike a traditional SQL database (DB), there is no *DB schema* defining a relational structure for all *DB tables* and their records. Instead, like in *object-relational DBs*, a Firestore DB table can have composite attributes such that its records contain composite values.

3.3.1 Database Tables and Records

In the Firestore jargon, a *DB table* is called a "***collection***" (of records), and a *DB record* is called a "***document***". The table's name is called "***collection ID***", and each record has a "***Document ID***", which is typically the *primary key value* of the record (if the table has a non-composite primary key).

Since, unlike relational and object-relational DBs, a Firestore DB is ***schema-free***, one may add any type of Firestore document to a Firestore collection. However, this liberty is hardly used in practice. So, in most cases, a Firestore collection is used as a DB table with a specific (implicit) schema.

Firestore collections are created on the fly, simply by adding a Firestore document to a not yet existing collection, like so:

```
import { doc, setDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore.j
const record = {isbn: "006251587X", title: "Weaving the Web", year: 2000};
await setDoc( doc( db, "books", "006251587X"), record);
```

In this example, the expression `doc( db, "books", "006251587X")` creates a *document reference* to a Firestore document with ID "006251587X" in the collection *books*. If the *books* collection does not yet exist, it will be created *on the fly*.

3.3.2 Data Types

The following data types are supported by Firestore:

Table 3-1. Firestore Data types

Data Type	Example	Note
Text string	`"Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut`	Text of up to 1,048,487 *bytes*, encoded to UTF-8 if we want to be considered by queries.

Data Type	Example	Note
	`labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum."`	
Integer (number)	`12345`	64-bit signed.
Floating-point (number)	`3.1415926535`	64-bit double precision, IEEE 754 (double-precision floating-point).
Boolean	`true` / `false`	
Date and time (timestamp)	`31 May 1999 at 16:46:00 UTC+2`	Firestore returns a *timestamp* object expressed in seconds and nanoseconds, like `Timestamp(seconds=1560523991, nanoseconds=286000000)`.
Map (or **Record**)	`{id: 17, foo: "bar"}`	A Firestore map is a JS record/object, which is a set of name/value pairs.
Array	`[1, "one", {id: 17, foo: "bar"}]`	An array can contain a record, **but not another array**.
Bytes	Any binary data like images or text files.	Binary data of up to 1,048,487 *bytes*, not UTF-8 encoded characters. Consider Firebase Storage for a cloud storage service for large multimedia files.

A Firestore collection is similar to a *JSON array*, while a Firestore document is similar to a *JSON object*. For instance, the following JSON object represents a Firestore document:

```
{
  "isbn": "006251587X",
  "title": "NoSQL Databases",
  "languages": ["en","de","es"],
  "author": {"name":"Peter Hanks",
             "birthDate":"1993-06-17"},
  "reviews": [{title:"Easy and fun to read!", nmrOfStars: 5},
              {title:"Disappointing", nmrOfStars: 1}]
}
```

Notice that the value of the *languages* attribute is an array (list), the value of the *author* attribute is a map (representing a record), and the value of the *reviews* attribute is a record set called *subcollection* in the Firebase jargon.

There are two special Firestore data types: *document references*, like `books/006251587X`, and *geographical points* like `[51.5074, 0.1278]` representing *latitude* and *longitude* values. Both data types are not yet well-supported currently.

Record fields having a set of records as their value

However, while JSON arrays/records can be arbitrarily nested, the use of nested structures within a Firestore document is limited. A Firestore *subcollection* is a collection within a document. Following our example, we add *chapters* to a book record:

```
const chapterDocRef = db.collection("books").document("006251587X")
                        .collection("chapters").document( 1);
chapterDocRef.set({ data })
```

3.4. Important Types of Firestore Objects

This guide aims to clarify the essential types of Firestore objects and how they relate to each other.

3.4.1 References

References are objects that represent the location of a record/document or table/collection in a Firestore database.

- A **document reference** object (DocumentReference), represents the location of a record/document, and is created using the `doc()` method.

  ```
  import { doc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore.js";

  const bookDocRef = doc( db, "books", "006251587X");
  ```

  We can also create document references specifying the path to the record/document as a string for convenience.

  ```
  const bookDocRef = doc( db, "books/006251587X");
  ```

- A **collection reference** object (CollectionReference) represents the location of a table and is created using the `collection()` method.

  ```
  import { collection } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore.

  const booksCollRef = collection( db, "books");
  ```

Whether a Firestore record/document or table/collection already exists in a database or not, they can be "*referenced*" and be used anytime later to **retrieve**, **save** or **listen to** the location in a Firestore database. Notice that creating a reference does not execute any network operation and consequently does not impact your billing due the database has not been queried up to that point.

Although similar, *document references* and *collection references* are two different types of references; hence they have their own properties and methods.

3.4.2 Queries

`query` objects (Query) are also objects that represent a *database query* that we can anytime later retrieve or listen to. As well as references, queries do not execute any network operation. We create `query` objects using the methods `query()` and `where()` is:

```
import { query, where } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore.

const q = query( collection(db, "books"), where("edition", "==", "1"));
```

3.4.3 Snapshots

Snapshots are objects that contain data from either a ***reference*** object or a ***query*** object. We may see a snapshot as a *picture of the data* we receive when retrieving it from the database. The term *snapshot* reminds us that the retrieved values of the queried properties are from when the query has been processed, but these properties' values may have been changed soon after.

Additionally to the record data inside, a snapshot provides several properties and methods that are convenient for knowing how the data or the record change. A snapshot is always invoked asynchronously using `async`/`await`, and always returns a promise.

- **Document snapshot**, in the following example, the `getDoc()` method is used to retrieve a *document snapshot* (DocumentSnapshot) containing data from a single record/document located in a Firestore database.

```
import { doc, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore

const bookDocRef = doc( db, "books", "006251587X"); // document reference
const bookDocSn = await getDoc( bookDocRef); // document snapshot
```

  An attempt to retrieve an inexistent *document snapshot* will return `undefined`.

- **Query snapshot**, the `getDocs()` method is used to retrieve a *query snapshot* (QuerySnapshot) representing the result of a query, and it may contain none, one or many `snapshot` objects (QueryDocumentSnapshot) that may constitute either an entire table/collection in a Firestore database,

```
import { collection, getDocs } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-f

const booksCollRef = collection( db, "books"); // collection reference
const booksQrySn = await getDocs( booksCollRef); // query snapshot
```

  or none, one or many `snapshot` objects (QueryDocumentSnapshot) resulting from a query.

```
import { collection, query, getDocs, where } from "https://www.gstatic.com/firebasejs/9.

const q = query( collection(db, "books"), where("edition", "==", "1")); // query
const booksQrySn = await getDocs( q); // query snapshot
```

  When a *query snapshot* contains none *query snapshot document* objects returns an empty array, and when it has only one *query snapshot document* object returns an array with one element.

- **Query document snapshot**, each snapshot object inside a `QuerySnapshot` is a *query document snapshot* (QueryDocumentSnapshot) containing data retrieved from a table/document in a Firestore database, returned within an array whether it contains one or many. To access each *query document snapshot*, we must iterate the query snapshot object using the `docs` property.

```
for (const bookDocSn of booksQrySn.docs) {
  console.log(bookDocSn.id) // query document snapshot
}
```

  Since *query document snapshots* come from a *query snapshot*, they are always guaranteed to exist.

Document snapshot and *query document snapshot* objects share the same properties and methods, being both the same in practice. This tutorial treat them similarly, calling them indistinctively ***document snapshots*** for convenience.

It may happen that after retrieving any of the previously mentioned snapshots, the referenced original record(s)/document(s) may be deleted by another user however, each individual snapshot will continue existing, although it will be impossible to retrieve data from it.

Later we will know and see the most important methods and properties of every snapshot object working in context, but here there are a few of them very useful:

- **`id`**, a property that returns the Document ID in a table/collection. Very useful when we need to know the foreign key without accessing the record data.
- **`data()`**, a method that extracts the data enclosed in any of the *snapshot objects* described lines above. It returns `undefined` if the data does not exist.
- **`get(field)`**, a method that extracts the data from a specific *field* inside a *snapshot* object.
- **`exists()`**, a method that verifies, through a *document snapshot*, a record, existence in a Firestore database.
- **`empty()`**, a method that verifies if a result from a *query snapshot* is empty, containing no result.

3.4.4 Accessing record data

Finally, the chosen ways to access record data in this tutorial. We present a straightforward way to access every data object and also a simplified form:

- From a single record/document, or *document snapshot*, using the method `data()`.

```
import { doc, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore

const bookDocRef = doc( db, "books", "006251587X"); // document reference
const bookDocSn = await getDoc( bookDocRef); // document snapshot
const bookRec = bookDocSn.data(); // record data
```

  Here is a simplified way to achieve the same.

```
import { doc, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore

const bookRec = (await getDoc( doc(db, "books", "006251587X"))).data();
```

- From a query containing multiple records/documents, or a *query snapshot*, using the Firestore property `docs`, the JS method `map()` and the Firestore method `data()`. In specific circumstances, in this tutorial, we access multiple records/documents data using a `for/of` loop.

```
import { collection, getDocs } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-f

const booksCollRef = collection( db, "books"); // collection reference
const booksQrySn = await getDocs( booksCollRef); // query snapshot
const bookDocSns = booksQrySn.docs; // multiple document snapshots in an array
const bookRecs = bookDocSns.map( d => d.data()); // records in an array
```

  Here is a simplified way to achieve the same.

```
import { collection, getDocs } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-f
```

```
const bookRecs = (await getDocs( collection( db, "books")))docs.map( d => d.data());
```

3.4.5 Naming convention of Firestore objects

This tutorial proposes the following convention for naming types of Firestore objects to keep consistency and readability in the code provided. The examples are based on a "*books*" table/collection.

Table 3-2. Names of the most important Firestore objects

Firestore object	Reference	Snapshot	Record data
Document snapshot (single)	`bookDocRef`	`bookDocSn`	`bookRec`
Collection / query snapshot (multiple)	`booksCollRef`	`booksQrySn`	`bookRecs`
Query / query snapshot (multiple)	`q`	`booksQrySn`	`bookRecs`
Query document snapshot (single)		`bookDocSn` / `bookDocSns`	`bookRec`

3.5. Writing Data to Firestore

Firestore has three operations for writing data to a database.

3.5.1 Create a record with the `setDoc()` method

There are two different ways to use the `setDoc()` method,

- one with automatically generated Document ID,

```
import { doc, setDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore

const bookDocRef = doc( db, "books");
await setDoc( bookDocRef, data);
```

- and another with a specified Document ID.

```
const bookDocRef = doc( db, "books", "006251587X");
await setDoc( bookDocRef, data);
```

In both cases, if the record/document does not exist, a new record/document will be created. If, on the other hand, the record/document already exists, the `setDoc()` method with the `merge` option "*true*" will merge the provided property-value data with the existing record/document, as an *update* operation.

```
const bookDocRef = doc( db, "books", "006251587X");
await setDoc( bookDocRef, { edition: "2" }, { merge: true });
```

Finally, If the record/document exists, the `setDoc()` method without the merge option will overwrite all data.

3.5.2 Create record with the addDoc() method

The addDoc() method creates a new record/document with an automatically generated Document ID, without chance of specifying it.

```
import { collection, addDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-fire

const booksCollRef = collection( db, "books");
await addDoc( booksCollRef, {
  isbn: "006251587X",
  title: "Weaving the Web",
  year: 2000
});
```

3.5.3 Update record with the updateDoc() method

The simple way of the updateDoc() method allows to update some fields of an existing record/document,

```
import { doc, updateDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestor

const bookDocRef = doc( db, "books", "006251587X");
await updateDoc( bookDocRef, { year:2000 }); // update a specific attribute
```

while it can also be used to update record-valued attributes and (array-) list-valued attributes.

```
// update a record-valued attribute
await updateDoc( bookDocRef, {
  "author.name": "Peter Hanks"
});
// add a new language to the list-valued field "languages"
await updateDoc( bookDocRef, {
  languages: arrayUnion("fr")
});
// remove a language from the "languages" array field
await updateDoc( bookDocRef, {
  languages: arrayRemove("fr")
});
```

3.6. Reading Data from Firestore

Firestore has three operations for reading data from a database.

3.6.1 Read a record with the getDoc() method

We show the detailed way to get data from a single record/document,

```
import { doc, collection, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase

const booksCollRef = collection( db, "books");
const bookDocRef = doc( booksCollRef, "006251587X");
const bookDocSn = await getDoc( bookDocRef);
```

```
const bookRec = bookDocSn.data();
```

and a simplified way to achieve the same.

```
const bookDocSn = await getDoc( db, "books", "006251587X");
const bookRec = bookDocSn.data();
```

3.6.2 Read all records in a table with the `getDocs()` method

Notice how the `getDocs()` method relies on the property `docs` to extract all *document snapshots* in the *query snapshot* object.

```
import { doc, collection, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X/firebase

const booksCollRef = collection( db, "books");
const booksQrySn = await getDocs( booksCollRef);
const bookDocSns = booksQrySn.docs;
const bookRecs = bookDocSns.map( d => d.data());
```

Now a simplified way to achieve the same.

```
const booksQrySn = await getDocs( collection( db, "books"));
const bookRecs = bookDocSns.docs.map( d => d.data());
```

3.6.3 Query a table with the `query()`, `where()` and `getDocs()` methods

Retrieve specific records/documents based on attribute filters.

```
import { collection, query, where, getDoc } from "https://www.gstatic.com/firebasejs/9.X.X

const booksCollRef = collection( db, "books");
const q = query( booksCollRef, where("title", "==", "Weaving the Web"));
const booksQrySn = await getDocs( q); // query snapshot
```

Find more advanced ways to query the database with simple and compound queries.

Standard query operators: <, <=, ==, >, >=, !=.

Special query operators:

- `array-contains`
- `array-contains-any`
- `in`
- `not-in`

3.6.4 Data management principles in this tutorial

In the data management approach that we adopt in this tutorial, we are going to use the following principles:

1. We always use entity IDs as Document IDs, and no Firebase auto-IDs.
2. Records/documents are preferably retrieved by their Document IDs (= entity ID) with the `getDoc()` method.
3. For creating a new entity record as a Firestore record/document (with its entity ID as Document ID), we use the Firestore `setDoc()` method.
4. For updating an existing Firestore record/document (representing an entity record), we use the Firestore `updateDoc()` method.

3.7. Quiz Questions

3.7.1 Question 1: Conversion with map

What is the purpose of the *arrow function* as the argument of the `map` function in the following expression? `(await db.collection("books").get()).docs.map( d => d.data())`

For converting a _____ to a _____.

- ☐ Collection object.
- ☐ Document Object
- ☐ QuerySnapshot object
- ☐ Record
- ☐ DocumentSnapshot object

3.7.2 Question 2: Characteristics of Firestore

Which of the following statements apply to Firestore? Select one or more:

- ☐ A Firestore collection has a certain schema (i.e., a list of attributes), implying that only documents that comply with its schema (having values for the given attributes) can be stored in a particular collection.
- ☐ A Firestore database table (representing a set of records) is called a collection.
- ☐ A Firestore collection does not have a schema, implying that any document can be stored in a particular collection. For instance, the document {name:"X", age:13} can be stored in the collection "books".
- ☐ A Firestore database record is called a collection.
- ☐ A Firestore database record is called a document.
- ☐ Firestore supports certain forms of non-elementary attributes with composite data values (lists, records, record lists).
- ☐ A Firestore database table (representing a set of records) is called a document.

3.7.3 Question 3: Document snapshots

What kind of JS object is returned by invoking the Firestore function `db.collection(x).doc(y).get()`? Select one:

- ☐ A QuerySnapshot object.

- ☐ A Query object.
- ☐ A Document object.
- ☐ A DocumentSnapshot object.

3.7.4 Question 4: Firebase JS SDK version 9

What is true about the following import expression? Select one or more:

```
import { initializeApp } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-app.js";
```

- ☐ It is a side-effect import.
- ☐ It is a no side-effect import.
- ☐ Its specifier invokes a function from a CDN origin.
- ☐ Its specifier invokes a function from the locally installed SDK using *npm*.
- ☐ it is optimized for its use with *module bundlers*.

3.7.5 Question 5: Use of the setDoc() method

What happens when we invoke these two expressions when the detailed record ("006251587X") already exists? Select one:

```
const bookDocRef = doc( db, "books", "006251587X");
await setDoc( bookDocRef, data);
```

- ☐ A new record is created with an automatically generated Document ID.
- ☐ The provided property-value data is merged with the existing record/document.
- ☐ The provided property-value data is overwritten with the existing record/document.

Chapter 4. Building a Minimal Web App with Plain JS and Firebase in Seven Steps

This tutorial shows how to build a minimal web application with plain JavaScript and Firestore, Cloud Google's DBMS service. 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 class diagram of **Figure 4-1.** The object type Book.

Figure 4-1. *The object type Book*

Book
isbn : String {id} title : String year : Integer

The **Table 4-1.** A collection of book objects represented as a table shows a sample data population for the model class `Book`:

Table 4-1. 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

What do we need for a data management app? There are four standard use cases, which have to be supported by the app:

1. **Create** a new book record by allowing the user to enter the data of a book that is to be added to the collection of stored book records.
2. **Retrieve** (or *read*) all books from the data store and show them in the form of a list.
3. **Update** the data of a book record.
4. **Delete** a book record.

These four standard data management use cases, and the corresponding operations, are often summarized with the acronym *CRUD*.

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

For maintaining a collection of persistent data objects, we need a storage technology that allows us to keep data objects in persistent records on a secondary storage device, such as a hard-disk or a solid-state disk. For our minimal example app, we will use Google's NoSQL database Cloud Firestore.

4.1. Step 1: Set up the Firebase Project

Before we can create our first plain JavaScript and Firebase web app we need to set up a Firebase environment, divided in 1) a *local environment* on our computer, and 2) a *production environment* on "*the cloud*" (on Google Cloud Platform), that we set up via *Firebase Console*. And as we will see later, what we do in the *local environment* is ***deployed*** in our *production environment*. Something unnoticeable at first sight is that Firebase Console is just *a proxy* for Google Cloud Platform, so everything we do on Firebase Console is mirrored on our GCP account.

We need the following installed on our computer:

1. *A Google account*.
2. Node.js, is an open-source JavaScript runtime environment that allows you to execute programs written in the JavaScript programming language on your browser and on a server environment as a standalone application. Node.js has been written in Chrome's V8 JavaScript engine, the same runtime running on your browser. For installing Node.js use any of the Node.js installers according to your operative system. For installing Node.js via package manager look at this Node.js's official page with the list of all options according to your OS. If you are a Windows user and avoid GUI installers, we advice you to try Scoop for a Unix-alike experience.
3. NPM or NodeJS Package Manager, a free package manager for ***Node.js***, consists of 1) a command-line client (CLI) and 2) the *package registry*, the world's largest database of public and paid-for private packages. NPM is the default package manager for Node.js, and comes together with it; this means that you don't have to install NPM after installing Noje.js on

your computer.

You may be wondering why do we need to setup a Node.js environment if we are going to build a plain JS web app fully running on the browser; however, notice that for using Firebase services via their Firebase SDKs, we intensively use Node.js in the background. Along with this tutorial, we will progressively cover what you need to know about Node.js projects to achieve more advanced things on Firebase.

Consider also that we use WebStorm as our IDE of choice in this tutorial, so we recommend its use. However, any other code editor can be used for completing this tutorial. If you are a student, you can get a free educational license for WebStorm.

Without much ado, let's start to set up our Firebase project:

4.1.1 Setup the production environment

We create our Firebase project:

1. Being logged into your Google Account, go to the Firebase website and click on "*Go to console*".

2. Click on "*Create a project*" or "*add project*" on your Firebase Console home page.

3. Name the project and click on "*Continue*".

4. Disable *Google Analytics*. This "free" user access measurement service, which is overkill for most website owners and comes at the cost of bloating all your pages with Google's tracking code, reducing page load time (bad for SEO), displaying annoying cookie banners (for GDPR-compliance) and tracking your users. Notice that browsers' privacy protection features are increasingly blocking Google Analytics.

   If you need a web analytics service, consider using Plausible Analytics, which is Open Source and works without third-party cookies.

   Then click on "*Create Project*".

5. After a bit of wait, you will see the *Firebase Console* home page, which includes the main menu, access to the project information, the name of your app and a few quick links for getting started. All your Firebase project's services, such as *Cloud Firestore*, *Hosting*, *Storage*, *Cloud Functions*, *Machine Learning* and *Authentication* can be managed by Firebase Console.

4.1.2 Create a Firestore database instance

We now create the Firestore database instance for your Firebase project.

6. Click on "*Build*" in the main menu and then on "*Firestore Database*".

7. Click on "*Create database*".

8. ***Security rules*** are a vital component for setting up security mechanisms for our database. In the Firebase documentation, you can read more about *Firebase Security Rules*. For the moment, choose "*Start in test mode*", and you will have 30 days for defining suitable rules for your app's database, which we will do later in Part 2. Now choose the corresponding "*Cloud server location*" and click on "*Done*".

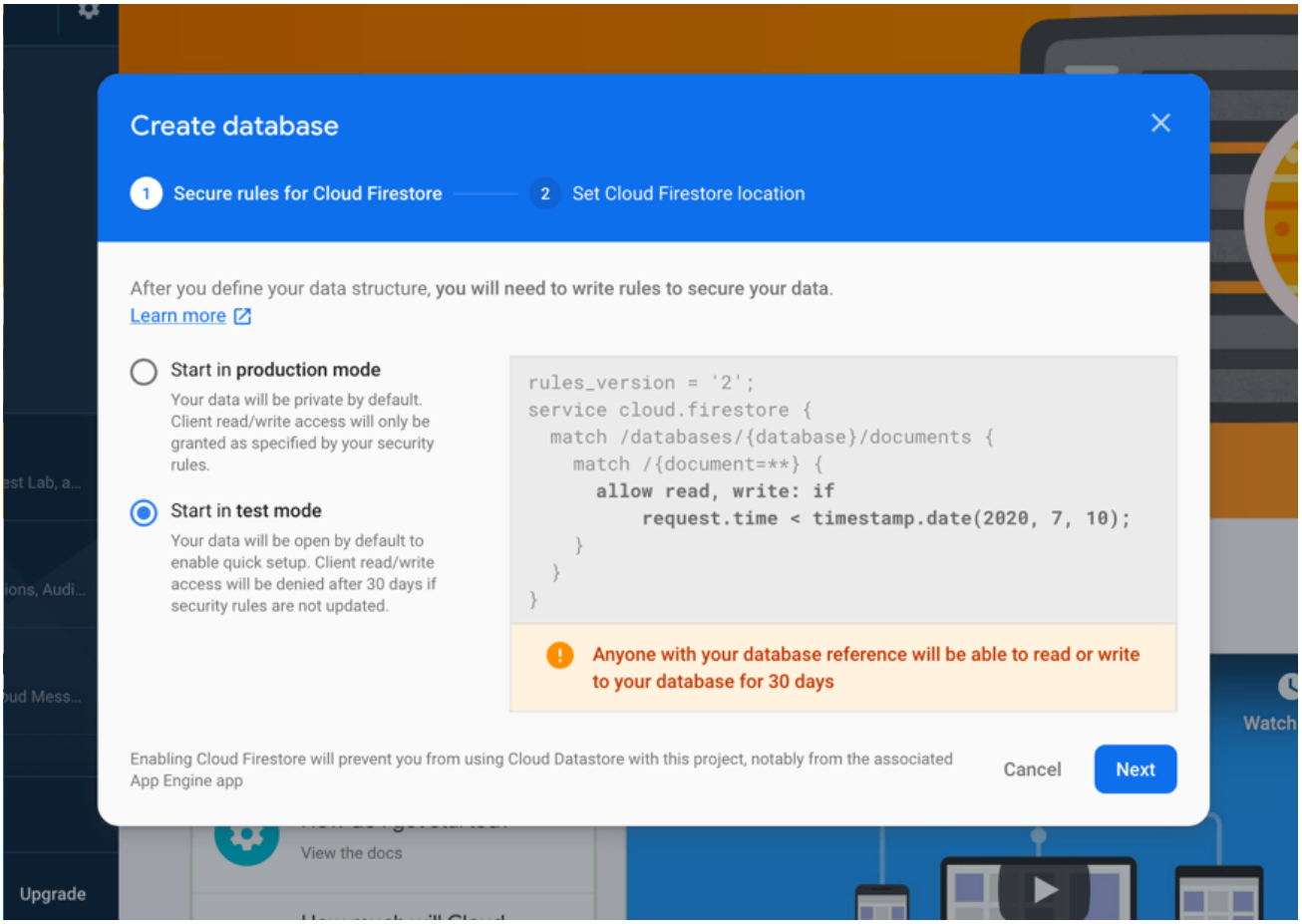


Figure 4-2. *Firestore Security Rules*

9. We will now see the Firestore database console. In the jargon of Firestore, **tables** are called "*collections*", and **records** (table rows) are called "*documents*". We use the platform-independent terminology (*records* and *tables*) along with the Firestore jargon ("documents" and "collections"). For creating your first Firestore collection (database table), click on "*Start collection*", enter the collection name *books*, and then click on "*Next*".

10. Click on "*Add document*" to create your first Firestore document (record). Fill out the `Document ID` and the other three fields: `isbn` (string), `title` (string), `year` (number), as in **Figure 4-3.** Creating the first Firestore *document*/record. Notice that we are assigning the same *ISBN code* to the `Document ID`. After clicking on "*Save*" we see the first Firestore document on the database console.

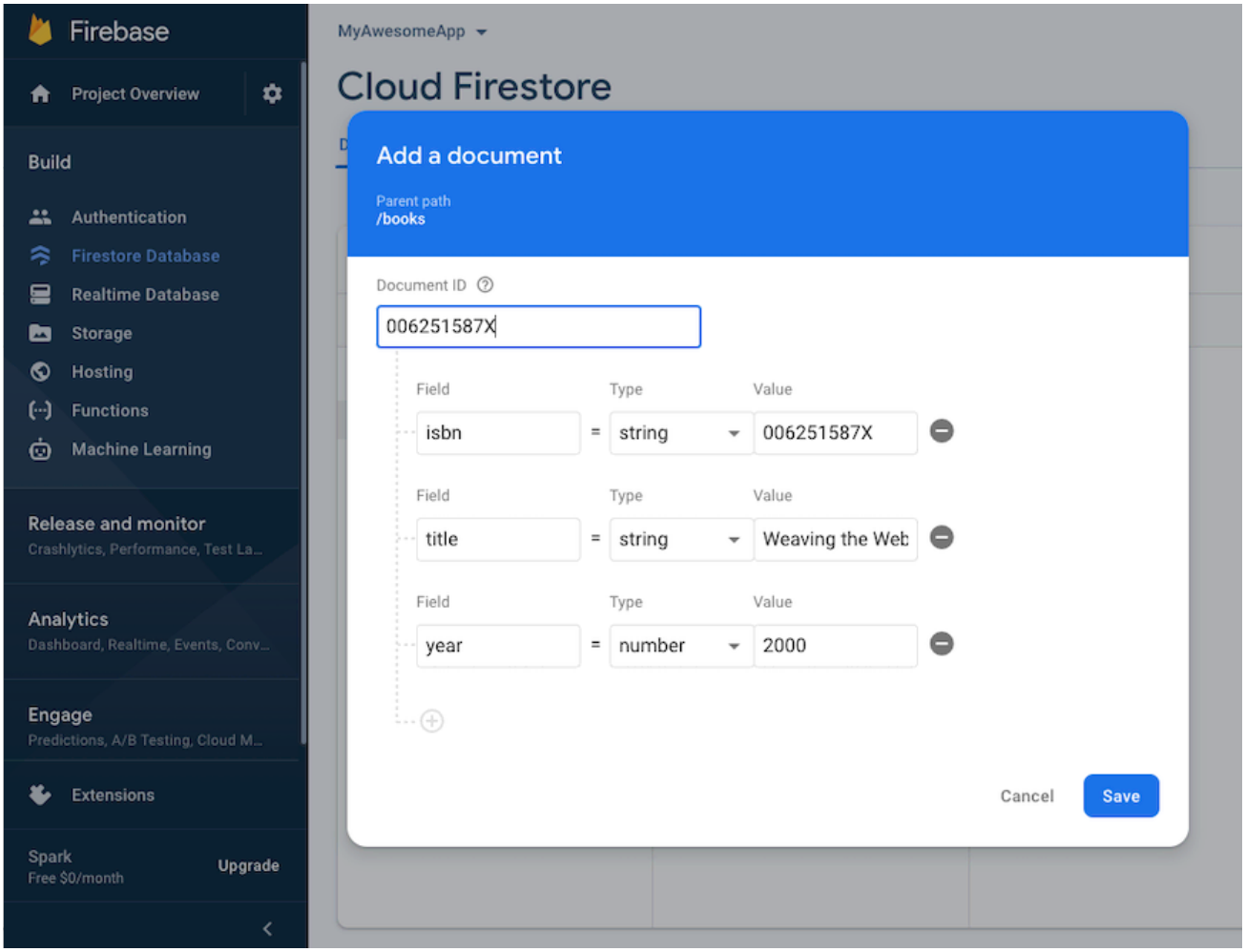


Figure 4-3. *Creating the first Firestore document/record*

4.1.3 Set up a Firebase web app and hosting URL

11. Go to the ***Project Overview*** and click on the "*web*" icon "</>", next to the icons of *iOS*, *Android* and *Unity*.

12. Give your web app a meaningful name, and then make sure to check "*Also set up Firebase Hosting for this app*". An auto-generated name will be given to the website hosting, but you can change it, needing only to make sure it is unique. Remember that here is where you define the free and public URL of your Firebase web app, which always ends with ".web.app". Click on "*Register App*".

13. After waiting while the app is initialized, you will see the Firebase SDK configuration. Save it for later since you need it to initialize the access to your Firebase application. Notice that you can always find it on the Firebase project configuration page.

You have just created your Firebase web app and the production environment where it resides.

4.1.4 Set up the local environment

Now you need to set up your local environment to run and test the web app while you develop, from which you will *deploy* to your production environment located on your Firebase Hosting instance.

14. Download the code of the Minimal App on your computer, and after uncompressing the ZIP file `1-MinimalApp.zip` you will find a folder named `1-MinimalApp`. This folder is the repository of every document of the web app, and it is

also used as your local *Git repo* folder.

15. Create or open a new project on your editor, the `1-MinimalApp` folder the root of your project. Inside, the subfolder `public` is accompanied by files related to the Firebase web app: `firebase.json`, `firebase.indexes.json`, `firebase.rules`, and `package.json`. The subfolder `js` in the public folder for our JavaScript source code files, and along with two subfolders `m`, and `v`, follow the *Model-View-Controller* paradigm for software application architectures. In the subfolder `js` there is a file named `initFirebase.mjs`, an ES6 module file in charge of initializing the web app interface with the Firebase APIs. And finally, there are many HTML files, among them the `index.html` file, the app's start page. Thus, we end up with the following folder structure:

```
1-MinimalApp
  public
    js
      m
      v
      initFirebase.mjs
    index.html
```

Notice that the `js` folder only contains two subfolders `m` and `v` (for model and view), not including a `c` folder since the minimal app doesn't include any controller code.

16. Open a ***terminal*** (e.g., the *Windows Power Shell*) when you are located in the folder `1-MinimalApp`. On WebStorm you can open a terminal window if you click on the tab "Terminal" at the bottom of the editor view. Run the following NPM command to install the latest version of the core Firebase SDK:

```
npm install firebase
```

Finally, click on "*Next*" on the Firebase Console to continue the set-up process.

17. Logging in your Google account by executing the following command:

```
firebase login
```

You can find out which Google account is logged in with the command "`firebase login`".

[Tip] You can log out your Firebase session by executing:

```
firebase logout
```

18. Install the *Firebase CLI* running:

```
npm install -g firebase-tools
```

Notice that you keep your Firebase CLI up-to-date every time you run this command.

19. And then initialize Firebase on your local environment by executing:

```
firebase init
```

20. Now you will go through the Firebase initialization process of your project in your local environment by following a

sequence of questions. When you are prompted with the question "*Which Firebase CLI features do you want to set up for this directory?*" make sure to select the following two options by using the *space bar* (**Figure 4-4.** Through the Firebase project initialization process):

- *Firestore: Deploy rules and create indexes for Firestore*
- *Hosting: Configure and deploy Firebase Hosting sites*

Press *Enter* to continue to the next question.

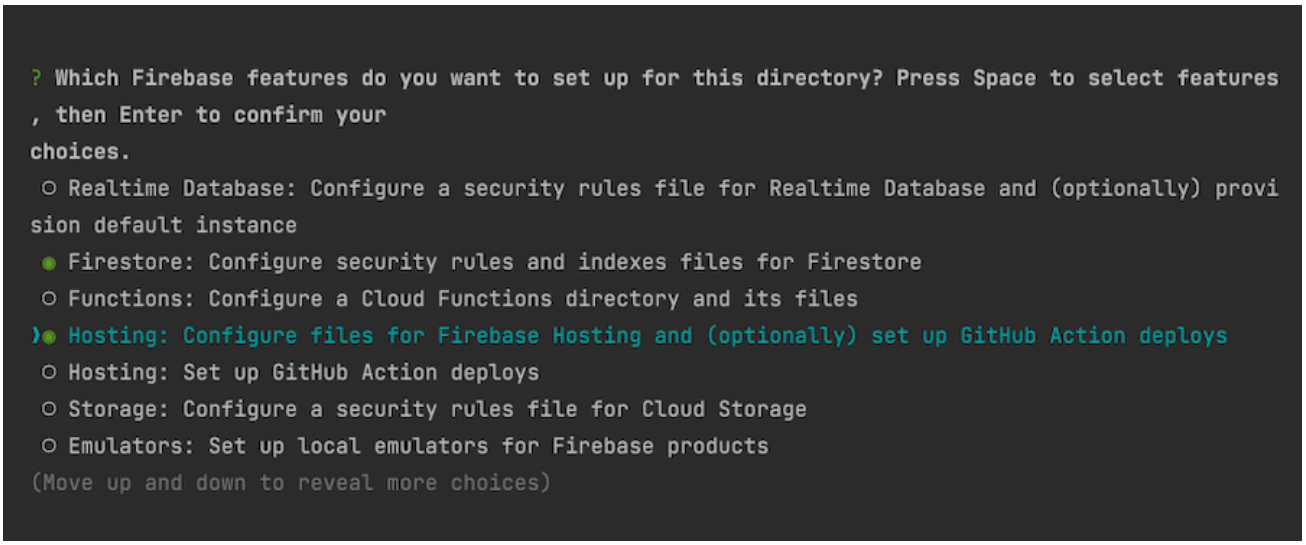


Figure 4-4. *Through the Firebase project initialization process*

21. When prompted for choosing the Firebase project to initialize "*First, let's associate this project directory with a Firebase project*", use the arrow keys and select *"Use an existing project"* and then select the previously created Firebase project: i.e. `"my awesomeweb-xxxxx"`.

    [Tip] If you are not prompted for choosing any Firebase project, delete the hidden file `.fireserc` located in your project folder and start over initializing the project. Optionally, you could run the following command to be prompted to select a Firebase project and assign an alias:

    ```
    firebase use --add
    ```

22. Press `Enter` to name the ***Firestore Security Rules*** file with the default name `firestore.rules`.

23. Press `Enter` to name the ***Firestore Indexes*** file with the default name `firestore.indexes.json`.

24. Press `Enter` to name the public directory with the default name `public`.

25. Press `N` when you are asked if you want to turn your web-based app into a *single-page app*.

26. Optionally, you can link your project to your GitHub repository to automate deployments to Firebase Hosting every time you push changes to your repo. When asked "*Set up automatic builds and deploys with GitHub? (y/N)*" answer accordingly and follow these directions in the official Firebase documentation.

27. Finally you will see the message *"Firebase initialization complete!"*. An `index.html` and a `404.html` files have been created.

28. Go back to the Firebase Console on your web browser, and click on "*Continue to Console*". We now know that Firebase has been initialized and running on our computer.

4.1.5 Running your Firebase web app locally

28. For testing your app locally, run on terminal:

```
firebase serve
```

By default, the local server runs using the port 5000: http://localhost:5000.

If you need to stop your local server, you can press `Ctrl` + `C` on Windows, Linux or Mac.

4.1.6 Firebase Local Emulator Suite (optional)

When we work on web development and run our app on our local environment, we are using the Hosting Emulator in the background, mimicking the actual behaviour of our Firebase Hosting, but on local. The Hosting Emulator is part of the Firebase Local Emulator Suite, allowing developers to build and test apps locally using almost every Firebase service available on the cloud. Firebase Local Emulator includes Cloud Firestore, Realtime Database, Cloud Storage, Authentication, Cloud Functions, and Firebase Hosting.

Start the Local Emulator Suite in your project by executing

```
firebase emulators:start
```

Once running, Firebase Emulator emulates every Firebase service initialized in your project locally. If we start Firebase Emulator, we would see two emulators running, one for Firestore and another for Firebase Hosting, with links to access their Emulator UIs.

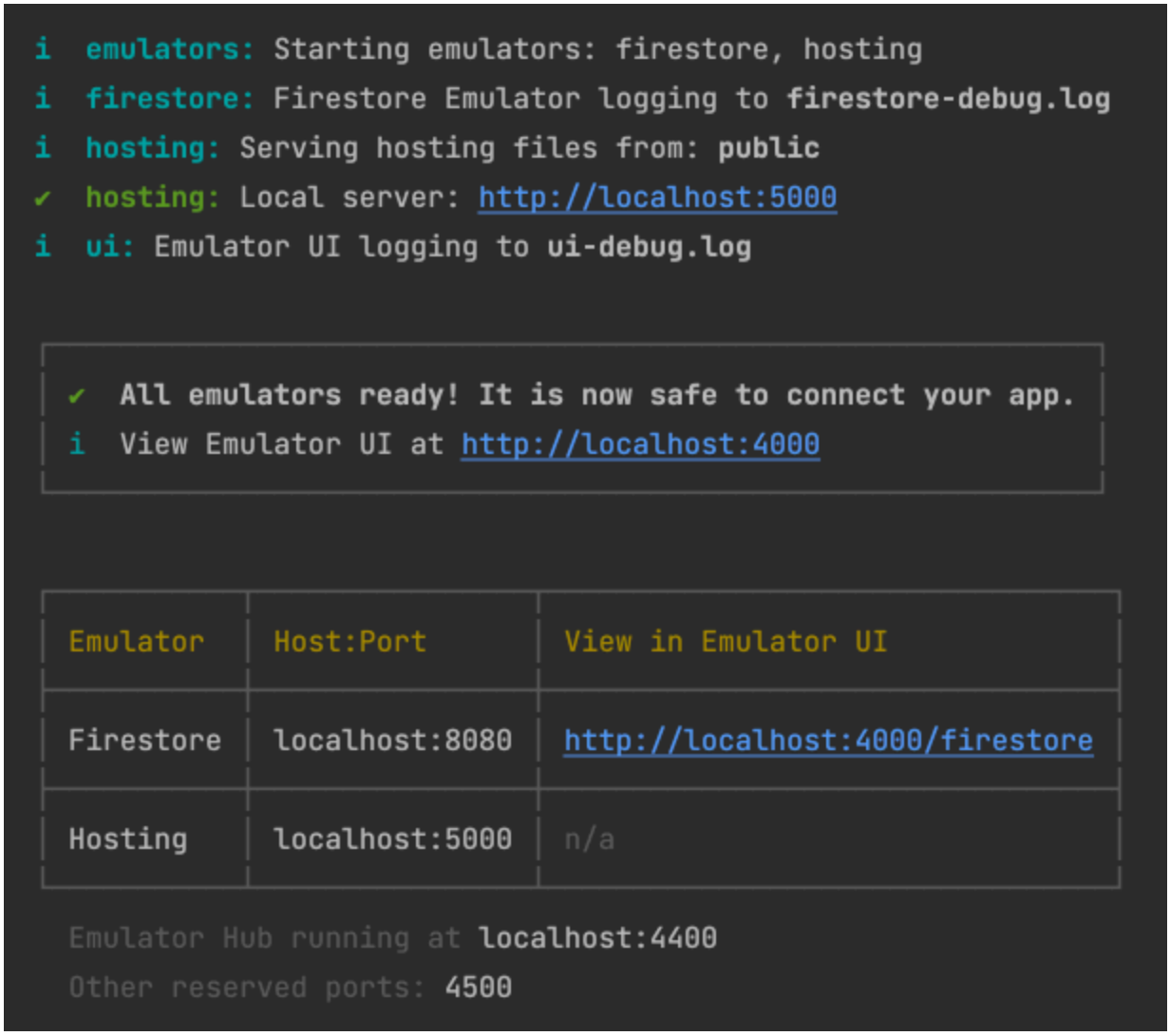


Figure 4-5. *Firebase Local Emulator Suite*

Learn more about the Firebase Local Emulator Suite in the Firebase documentation.

4.1.7 Deploy your app on the production environment

29. For making our app public, we can deploy it to Firebase Hosting by running:

```
firebase deploy
```

Visit the public web app by clicking on the URL provided on the terminal. You might want to know more about testing locally and deploying your app on Firebase.

4.1.8 Defining and testing your first Security Rules

Previously, in the Firestore database setup process, we chose to "*Start in test Mode*" to get started with your database quickly, but now we need to define better Security Rules.

Security Rules in test mode leaves the database ***open to anyone*** on the Internet to read and write/change without restrictions. That is why Firestore forces us to update the Security Rules after ***30 days*** from the day we created the database. To make our web app work beyond that 30-day limitation, you can set up a more advanced Firebase Security Rules configuration.

Notice that the following Security Rules are just meant to continue running our app beyond the 30-day limitation in this educational context, but they do not provide a significaant database security measure in a real-world situation. Hence, remember to strengthen these basic rules to protect an actual web application accordingly. We encourage you to deepen into the Firestore Security Rules by reading more on the subject in the Firebase Documentation.

1. On your IDE (maybe WebStorm), open the file `firestore.rules`.

2. Replace the content of the file with this snippet:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Allow anyone
    match /{document=**} {
      allow read, write; // or allow read, write: if true;
    }
  }
}
```

3. Deploy the project by running:

```
firebase deploy
```

   Notice that the correct workflow for updating Security Rules is first editing the rules on your local version (`firestore.rules`) using your editor and then deploying the project to the cloud in such a way that you are updating both your local rules and the rules on the Firebase Console at the same time. If you edit the rules on the Firebase Console, your local version will be outdated, and you will overwrite the new ones with the outdated version.

4. To ensure the Security Rules have been deployed correctly, you can go to the Firebase Console, go to the Firestore database console, and click on the horizontal tab *"Rules"*. You must see the same snippet recently deployed.

5. For testing the rules, click on the *"Rules playground"* tab.

6. On *"Location"* enter "`/books/Auto-ID`".

7. Click on *"Run"*, and you should see a message in green color saying *"Simulated read allowed"*.

8. Finally, click on "*Publish*" to deploy the new Security Rules:

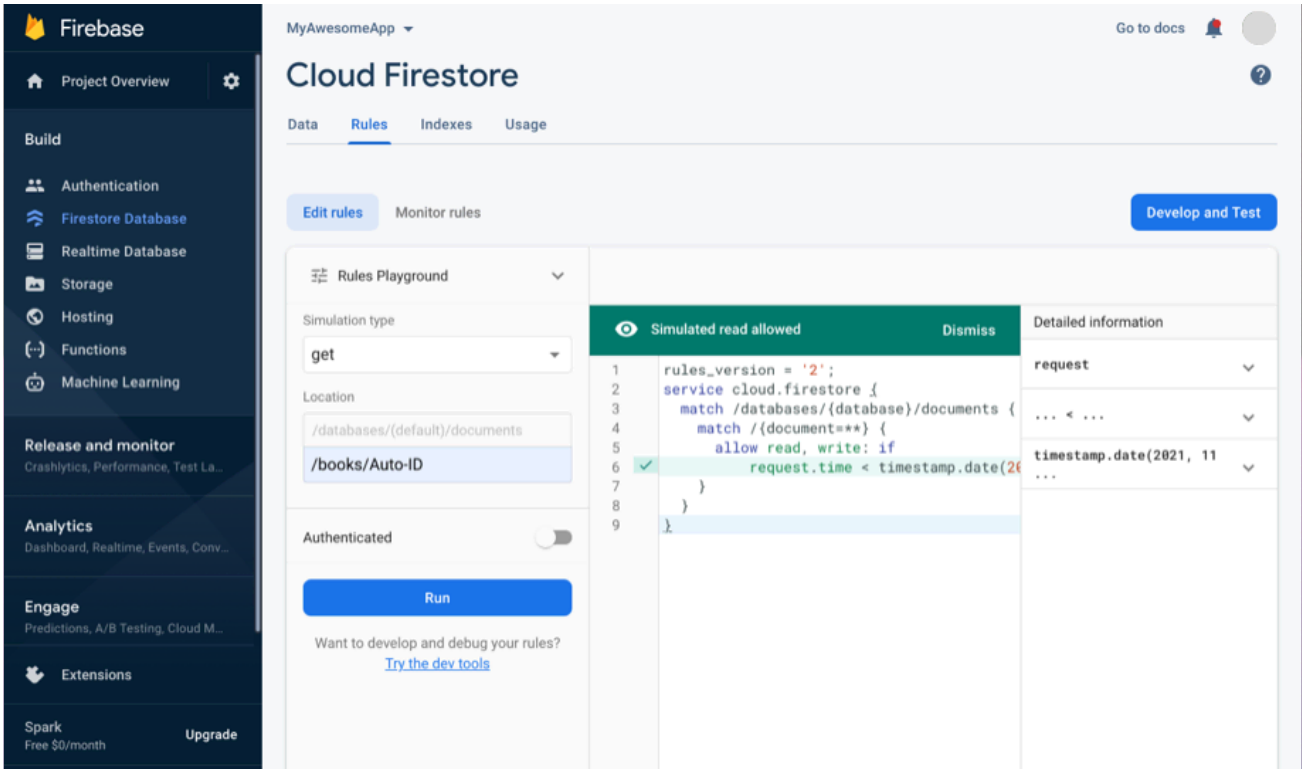


Figure 4-6. *Setting up and Testing Security Rules*

This may be a good time to try out our "Hello World" web app before continuing towards the next step.

4.1.9 Set up Node.js Dependencies

As we discussed before, a plain JavaScript application using the Firebase Web SDK relies totally on **Node.js** behind the scenes, so we need to know about the *Node.js* dependencies that Firebase needs in your local development environment and how to keep them updated and healthy.

In this second step, we locate the `package.json` file, which contains essential metadata that configures and describes a Node.js project, as a Firebase project is, defining attributes that NPM uses to install dependencies, run scripts, identify the application's entry point, and others that determine how our application interacts and runs. It is thus of crucial importance to understand the role of `package.json` in the JavaScript ecosystem.

On your editor, open the `package.json` file, and pay attention to the attribute "*dependencies*" in the JSON object located inside:

```
{
  "name": "js-firebase-minimal-app",
  "version": "1.1.0",
  "description": "Part 1: Learn how to build a Minimal Web App using JS and Firebase",
  "homepage": "https://minimalapp-ea662.web.app/tutorial",
  "keywords": [
    "javascript",
    "firebase",
    "firestore",
    "crud",
    "firebase SDK version 9",
    "minimal"
```

```
  ],
  ...
  "dependencies": {
    "firebase": "^X.X.X",
    "firebase-firestore-lite": "^X.X.X",
    "firebase-auth": "^X.X.X"
  },
  ...
}
```

We see three Firebase SDK libraries and their corresponding versions. That means we need those libraries installed in our local environment to run the Minimal App. Run the following command to install the defined dependencies. WebStorm highlights the dependencies not found in the local environment, facilitating to know that we still need to fix something.

```
npm install
```

Then the folder "`node_modules`" is generated with all the dependencies.

Whenever you want to reset all the dependencies and ensure a healthy local environment, you can safely delete the `node_modules` directory and run `npm install` again.

Since Firebase Hosting is in fact a Node.js environment, whenever we deploy our app, the Node.js dependencies are not likewise deployed, but the package.json file is read to set up in the production environment every dependence appropriately defined.

Firebase is permanently updated; therefore, sometimes, we need to update our `package.json` file with the latest versions of the defined dependencies. Use the following command to update the `package.json` file along with your dependencies.

```
npm i firebase
```

Learn about all the `package.json` attributes with the official npm guide.

Note for WebStorm users

When writing JavaScript code with Firebase SDK libraries on *WebStorm* we may see functions, methods or objects as errors. If that happens, just delete the node_modules folder and run `npm install` again.

Optionally, only for seeing correctly formatted *Firebase Rules* files (`firestore.rules`), also on `Settings/Preferences` go to `Plugins` and search for "`Firebase`" on the search bar. You will find a *third-party* plugin named `Firebase Rules Syntax Highlighter`, select it and install it.

4.1.10 Initialize Firebase

In the second step, we initialize an interface for our Firestore instance. We start by creating an ES6 module file named `initFirebase.mjs` located in the root of the `js` folder, which first statements import the functions we need from the JavaScript versions of the Firebase SDK libraries. We always initialize Firestore through Firebase, so we import the

`initializeApp()` function from the core Firebase SDK library (`firebase-app.js`), and the `getFirestore()` function from the Firestore Lite Web SDK (`firebase-firestore-lite.js`), introduced from Firebase Web SDK version 9, and designed to improve performance of the most basic read/write operations.

```
import { initializeApp } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-app.js";
import { getFirestore } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore-
```

To start using Firestore we initialize a Firebase App instance for our app, using the values taken from the web app's Firebase project configuration page and pass them as a parameter in a variable named `firebaseConfig` to invoke the `initializeApp()` function. The Firebase App instance object is from now on available in our session, containing configuration information that is consumed across other Firebase services, such as Firestore or Firebase Authentication.

```
// TODO: Replace the following with your web app's Firebase project configuration
const firebaseConfig = {
  apiKey: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  authDomain: "minimalapp-XXXX.firebaseapp.com",
  projectId: "minimalapp-XXXX",
  appId: "1:XXXXXXXXXXXX:web:XXXXXXXXXXXXXXXXXXXXXX"
};
// Initialize a Firebase App object
initializeApp( firebaseConfig);
```

Once the Firebase App instance has been initialized, we can initialize Cloud Firestore using the `getFirestore()` function to create the ***fsDb*** object that works now as an interface to our Firestore DB instance.

```
// Initialize Firestore interface
const fsDb = getFirestore();
```

Finally, the ***fsDb*** object is exported and becomes available to other procedures in the minimal app.

```
export { fsDb };
```

4.1.11 Changing names of Firestore SDK's functions

You may have noticed that formerly we named the *Firebase database* instance object as "`db`", but now we have named it "`fsDb`" since we find this name more meaningful and straightforward once we use it in our code. Additionally, we find three names of the original Firestore SDK JS version 9 functions are too generic, violating good practice conventions, and may turn our code confusing, so we propose to rename them like:

Table 4-2. Change of Firebase SDK's function names

Original Firestore name	Rename
`doc`	`fsDoc`
`collection`	`fsColl`
`query`	`fsQuery`

The name changes happen when the functions are imported from the Firestore SDK library.

```
import { collection as fsColl, doc as fsDoc, query as fsQuery } from "https://www.gstatic.
```

4.2. Step 2: Write the Model Code

In the third step, we write the code of our model class and save it in a specific model class file. In an MVC app, the model code is the most critical part of the app, and it's also the basis for writing the view and controller code. Large parts of the view and controller code could be automatically generated from the model code, and many MVC frameworks provide this kind of code generation.

In the information design model shown in **Figure 2-1.** The built-in JavaScript classes `Object` and `Function`, there is only one class, representing the object type `Book`. So, in the folder `js/m`, we create a file `Book.mjs`.

As `Book.mjs` is a ES6 module, its initial statements import the ***fsDb*** object to interface our Firestore instance, and the functions from the Firestore Lite Web SDK library that later are invoked to write/read operations using instances of the model class `Book`.

```
import { fsDb } from "../initialize.mjs";
import { collection as fsColl, deleteDoc, doc as fsDoc, getDoc, getDocs, setDoc, updateDoc
  from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore-lite.js";
```

The model class `Book` is coded as a JavaScript class with a constructor function defining 1) a single constructor parameter in the form of a record using ES6 function parameter destructuring, and 2) the attributes `isbn`, `title` and `year`.

```
class Book {
  constructor({isbn, title, year}) {
    this.isbn = isbn;
    this.title = title;
    this.year = year;
  }
}
```

In addition to defining the model class, we also define the following items in the `Book.mjs` file:

1. A class-level method `Book.retrieve` for loading only one book record/document.
2. A class-level method `Book.retrieveAll` for loading all managed book records/documents.
3. A class-level method `Book.add` for creating a new book record/document.
4. A class-level method `Book.update` for updating an existing book record/document.
5. A class-level method `Book.destroy` for deleting a book record/document.
6. A class-level method `Book.createTestData` for creating a few example book records/documents to be used as test data.
7. A class-level method `Book.clearData` for clearing the book table/collection.

4.2.1 Creating a new `Book` record

Since database access operations can fail, we always call them in a `try-catch` block to follow up with an error message whenever the input operation fails.

The following `Book.add` procedure takes care of creating a new `book` record/document and adding it to the Firebase table/collection "books":

```
Book.add = async function (slots) {
  const booksCollRef = fsColl( fsDb, "books"),
    bookDocRef = fsDoc (booksCollRef, slots.isbn);
  slots.year = parseInt( slots.year);  // convert from string to integer
  try {
    await setDoc( bookDocRef, slots);
    console.log(`Book record ${slots.isbn} created.`);
  } catch( e) {
    console.error(`Error when adding book record: ${e}`);
  }
};
```

4.2.2 Retrieving a `Book` record

For retrieving a book record/document we use the `Book.retrieve()` procedure that is called with a parameter `isbn`:

```
Book.retrieve = async function (isbn) {
  let bookDocSn = null;
  try {
    const bookDocRef = fsDoc( fsDb, "books", isbn);
    bookDocSn = await getDoc( bookDocRef);
  } catch( e) {
    console.error(`Error when retrieving book record: ${e}`);
    return null;
  }
  const bookRec = bookDocSn.data();
  return bookRec;
};
```

4.2.3 Retrieving all `Book` records

For retrieving the book records/documents from the Firestore "books" table collection, we use the `Book.retrieveAll` procedure:

```
Book.retrieveAll = async function () {
  let booksQrySn = null;
  try {
    const booksCollRef = fsColl( fsDb, "books");
    booksQrySn = await getDocs( booksCollRef);
  } catch( e) {
    console.error(`Error when retrieving book records: ${e}`);
    return null;
  }
  const bookDocs = booksQrySn.docs,
    bookRecs = bookDocs.map( d => d.data());
  console.log(`${bookRecs.length} book records retrieved.`);
```

```
  return bookRecs;
};
```

4.2.4 Updating a Book record

For updating an existing book record/document, we first retrieve it from the Firestore "books" table/collection, and then re-assign those attributes the value of which has changed. Here is the full code of the procedure:

```
Book.update = async function (slots) {
  const updSlots = {};
  // retrieve up-to-date book record
  const bookRec = await Book.retrieve( slots.isbn);
  // convert from string to integer
  if (slots.year) slots.year = parseInt( slots.year);
  // update only those slots that have changed
  if (bookRec.title !== slots.title) updSlots.title = slots.title;
  if (bookRec.year !== slots.year) updSlots.year = slots.year;
  if (Object.keys( updSlots).length > 0) {
    try {
      const bookDocRef = fsDoc( fsDb, "books", slots.isbn);
      await updateDoc( bookDocRef, updSlots);
      console.log(`Book record ${slots.isbn} modified.`);
    } catch( e) {
      console.error(`Error when updating book record: ${e}`);
    }
  }
};
```

Notice that since the `updSlots` map may contain a variable number of property-value slots, we need to test if it's not empty by converting the map to an array of keys with `Object.keys`.

4.2.5 Deleting a Book record

A book record/document is deleted from the Firestore "books" table/collection using the `Book.destroy` procedure:

```
Book.destroy = async function (isbn) {
  try {
    await deleteDoc( fsDoc( fsDb, "books", isbn));
    console.log(`Book record ${isbn} deleted.`);
  } catch( e) {
    console.error(`Error when deleting book record: ${e}`);
  }
};
```

4.2.6 Creating test data

To test our code, we may create some test data and save it in our Firestore DB. We first create an array of book records. Then, to use the `Promise.all` function, we *map* each book record, "bookRec", to an `Book.add()` procedure invocation expression of the following form:

```
await Promise.all( bookRecs.map( d => Book.add( d)));
```

Notice that `Promise.all` allows invoking a list of asynchronous operations, which are not executed sequentially but simultaneously.

```
Book.generateTestData = async function () {
  let bookRecs = [
    {
      isbn: "006251587X",
      title: "Weaving the Web",
      year: 2000},
    {
      isbn: "0465026567",
      title: "Gödel, Escher, Bach",
      year: 1999
    },
    {
      isbn: "0465030793",
      title: "I Am A Strange Loop",
      year: 2008
    }
  ];
  // save all book records
  await Promise.all( bookRecs.map( d => Book.add( d)));
  console.log(`${Object.keys( bookRecs).length} books saved.`);
};
```

4.2.7 Clearing all data

The following two-part procedure clears all data from our Firestore "books" table/collection:

1. First, a JS array (list) of all book records is retrieved from the Firestore DB using the `Book.retrieveAll()` procedure:

   ```
   const bookRecords = await Book.retrieveAll();
   ```

2. All records/documents in the `books` table/collection are then deleted individually, invoking asynchronously the `Book.destroy()` procedure:

   ```
   await Promise.all( bookRecs.map( b => Book.destroy( b.isbn)));
   ```

Here is the full code of the procedure:

```
Book.clearData = async function () {
  if (confirm("Do you really want to delete all book records?")) {
    // retrieve all book documents from Firestore
    const bookRecs = await Book.retrieveAll();
    // delete all documents
    await Promise.all( bookRecs.map( b => Book.destroy( b.isbn)));
    // ... and then report that they have been deleted
    console.log(`${Object.values( bookRecs).length} books deleted.`);
  }
};
```

4.3. Step 3: Write the Start Page

In the start page HTML file of the app, `index.html`, besides adding event listeners for the buttons to generate and clear test data in the DB with the help of the procedure `Book.createTestData()`, and for clearing all data with `Book.clearData()`, we load the model class `Book` from the model class file `Book.mjs`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8"/>
  <title>Minimal Web App with JS and Firebase</title>
  <meta name="description" content="A minimal effort web app with plain JS + Firebase."/>
  <link rel="icon" href="favicon.ico"/>
  <script type="module">
    import Book from "./js/m/Book.mjs";

    window.addEventListener("load", function () {
      const clearButton = document.getElementById("clearData"),
        generateTestDataButtons = document.querySelectorAll("button.generateTestData");
      // set event handlers for the buttons "clearData" and "generateTestData"
      clearButton.addEventListener("click", Book.clearData);
      for (const btn of generateTestDataButtons) {
        btn.addEventListener("click", Book.generateTestData);
      }
    });
  </script>
</head>
<body>
...
</body>
</html>
```

The start page provides a menu for choosing one of the *CRUD* data management use cases. Each use case is performed by a corresponding page such as, for instance, `createBook.html`.

```
<body>
<main>
  <h1>Minimal App – Public Library</h1>
  <div class="subheading">A Minimal effort Web App built with Plain JS and Firebase</div>
  <p>This app supports the following operations:</p>
  <menu>
    <li><a href="createBook.html">Create</a> a new book record</li>
    <li><a href="retrieveAndListAllBooks.html">Retrieve</a> and list all book records</li>
    <li><a href="updateBook.html">Update</a> a book record</li>
    <li><a href="deleteBook.html">Delete</a> a book record</li>
    <li style="margin-top: 1em">
      <button id="clearData" type="button">Clear database</button>
    </li>
    <li>
      <button class="generateTestData" type="button">Generate test data</button>
    </li>
  </menu>
```

```
</main>
</body>
```

4.4. Step 4: Implement the Create Use Case

For our example app, the user interface page for the *CRUD* use case *Create* is called `createBook.html`, located in the `1-MinimalApp` folder. In its `<head>` element, it loads the view code file `createBook.mjs`, which sets up the *Create* user interface:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
  <meta charset="UTF-8"/>
  <title>Minimal Web App with JS and Firebase: Create</title>
  <link rel="icon" href="favicon.ico"/>
  <script src="js/v/createBook.mjs" type="module"></script>
</head>
<body>
...
</body>
<html>
```

For a data management use case with user input, such as "Create", an HTML form is required as a user interface. The form typically has a labelled `<input>` field for each attribute of the model class:

```
<body>
<header>
  <h1>Create a new book record</h1>
</header>
<main>
  <form id="Book">
    <div><label>ISBN: <input name="isbn"></label></div>
    <div><label>Title: <input name="title"></label></div>
    <div><label>Year: <input name="year"></label></div>
    <div>
      <button name="commit" type="button">Create</button>
    </div>
  </form>
</main>
<footer>
  <a href="index.html">« Back to main menu</a>
</footer>
</body>
```

The view code file `js/v/createBook.mjs` contains three statements:

1. import statements for the model class `Book`.
2. Variables declaration to access UI elements, for the `<form>` element, and the "*create button*" to save the user input data.
3. An `addEventListener` attached to click events on the "*create button*" takes the user input data from the input fields and saves this data by calling the `Book.add()` procedure. Finally, it clears the form.

```
import Book from "../m/Book.mjs";

const formEl = document.forms["Book"],
  createButton = formEl["commit"];

createButton.addEventListener("click", async function () {
  const slots = {
    isbn: formEl["isbn"].value,
    title: formEl["title"].value,
    year: formEl["year"].value
  };
  await Book.add( slots);
  formEl.reset();
});
```

4.5. Step 5: Implement the Retrieve/List All Use Case

The user interface for the *CRUD* use case *Retrieve* consists of an HTML table for displaying the data of all model objects. For our example app, this page is called `retrieveAndListAllBooks.html`, located in the main folder `1-MinimalApp`, and it contains the following code in its `<head>` element:

```
<head>
  <meta charset="UTF-8"/>
  <title>Minimal Web App with JS and Firebase: Retrieve and List</title>
  <link rel="icon" href="favicon.ico"/>
  <script src="js/v/retrieveAndListAllBooks.mjs" type="module"></script>
</head>
```

We load the view code file (here: `retrieveAndListAllBooks.mjs`). This is the pattern we use for all four *CRUD* use cases.

```
<body>
<header>
  <h1>Retrieve and list all book records</h1>
</header>
<main>
  <table id="books">
    <thead>
    <tr>
      <th>ISBN</th>
      <th>Title</th>
      <th>Year</th>
    </tr>
    </thead>
    <tbody></tbody>
  </table>
</main>
<br>
<footer>
  <a href="index.html">« Back to main menu</a>
</footer>
</body>
```

In the view code we first import the model class `Book`, then invoke the `retrieveAll()` procedure to retrieve all book records data from Firestore and then fill the table by creating a table row for each book object:

```
import Book from "../m/Book.mjs";

const bookRecords = await Book.retrieveAll();

const tableBodyEl = document.querySelector("table#books>tbody");

// for each book, create a table row with a cell for each attribute
for (const bookRec of bookRecords) {
  const row = tableBodyEl.insertRow();
  row.insertCell().textContent = bookRec.isbn;
  row.insertCell().textContent = bookRec.title;
  row.insertCell().textContent = bookRec.year;
}
```

More specifically, this procedure creates the view table in a loop over all array objects retrieved from the `Book.retrieveAll()` procedure. 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 made 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 to ensure that new elements are appended to the list of rows and cells.

4.6. Step 6: Implement the Update Use Case

Also for the *Update* use case, we have an HTML page for the user interface (`updateBook.html`) and a view code file (`js/v/updateBook.mjs`). The HTML form for the UI of the "update book" operation has a selection field for choosing the book to be updated, an `<output>` field for the standard identifier attribute `isbn`, and an `<input>` field for each attribute of the model class *Book* that it is filled with its respective value that can be updated with a new value. Notice that by using an `<output>` field for the standard identifier attribute, we do not allow changing the standard identifier of an existing object.

```
<main>
  <form id="Book">
    <div>
      <label>Select book:
        <select name="selectBook">
          <option value=""> ---</option>
        </select>
      </label>
    </div>
    <div><label>ISBN: <output name="isbn"></output></label></div>
    <div><label>Title: <input name="title"></label></div>
    <div><label>Year: <input name="year"></label></div>
    <div>
      <button name="commit" type="button">Update</button>
    </div>
  </form>
</main>
```

Notice that we include a kind of empty `<option>` element, with a value of `""` and a display text of `---`, as a default choice

in the `selectBook` selection list element. So, by default, the `value` of the `selectBook` form control is empty, requiring the user to choose one of the available options for filling the form.

The view code populates the `select` element's option list by loading the collection of all book objects from the Firestore "books" table/collection using the `Book.retrieveAll()` function, and then creating an `option` element for each book object.

```
const bookRecords = await Book.retrieveAll();

const formEl = document.forms["Book"],
  updateButton = formEl["commit"],
  selectBookEl = formEl["selectBook"];

// fill select with options
for (const bookRec of bookRecords) {
  const optionEl = document.createElement("option");
  optionEl.text = bookRec.title;
  optionEl.value = bookRec.isbn;
  selectBookEl.add( optionEl, null);
}
```

A book selection event is caught via a listener for `change` events on the `select` element. When a book is selected, the form is filled with its data retrieved using the `Book.retrieve()` function:

```
selectBookEl.addEventListener("change", async function () {
  const isbn = selectBookEl.value;
  if (isbn) {
    // retrieve up-to-date book record
    const bookRec = await Book.retrieve( isbn);
    formEl["isbn"].value = bookRec.isbn;
    formEl["title"].value = bookRec.title;
    formEl["year"].value = bookRec.year;
  } else {
    formEl.reset();
  }
});
```

When the save button is activated, a `slots` record is created from the form field values and used as the argument for calling `Book.update`:

```
updateButton.addEventListener("click", async function () {
  const slots = {
    isbn: formEl["isbn"].value,
    title: formEl["title"].value,
    year: formEl["year"].value
  },
    bookIdRef = selectBookEl.value;
  if (!bookIdRef) return;
  await Book.update( slots);
  // update the selection list option element
  selectBookEl.options[selectBookEl.selectedIndex].text = slots.title;
  formEl.reset();
```

```
});
```

4.7. Step 7: Implement the Delete Use Case

The user interface for the *Delete* use case just has a `<select>` field for choosing the book to be deleted:

```
<main>
  <form id="Book">
    <div>
      <label>Select book:
        <select name="selectBook">
          <option value=""> ---</option>
        </select>
      </label>
    </div>
    <div>
      <button name="commit" type="button">Delete</button>
    </div>
  </form>
</main>
```

Like in the *Update* case, the view code in `js/v/deleteBook.mjs` loads the book data into main memory, populates the book selection list and adds some event listeners. The event handler for *Delete* button click events has the following code:

```
const bookRecords = await Book.retrieveAll();

const formEl = document.forms["Book"],
  deleteButton = formEl["commit"],
  selectBookEl = formEl["selectBook"];

for (const bookRec of bookRecords) {
  const optionEl = document.createElement("option");
  optionEl.text = bookRec.title;
  optionEl.value = bookRec.isbn;
  selectBookEl.add( optionEl, null);
}

deleteButton.addEventListener("click", async function () {
  const isbn = selectBookEl.value;
  if (!isbn) return;
  if (confirm("Do you really want to delete this book record?")) {
    await Book.destroy( isbn);
    // remove deleted book from select options
    selectBookEl.remove( selectBookEl.selectedIndex);
  }
});
```

You can run the minimal app from our server or download the code as a ZIP archive file.

4.8. Points of Attention

4.8.1 Styling the User Interface

For simplicity, we have used raw HTML with minimal CSS styling. But a user interface (UI) should be appealing. So, the code of this app should be extended by adding suitable CSS style rules.

Today, the UI pages of a web app have to be adaptive (frequently called "responsive") for being rendered on different devices with different screen sizes and resolutions. The main issue of an adaptive UI is to have a *fluid layout*, in addition to proper viewport settings. Whenever images are used in a UI, we also need an approach for adaptive bitmap images: serving images in smaller/larger sizes for smaller/large screens (and in higher resolutions for high-resolution screens), while preferring scalable SVG images for diagrams and artwork. In addition, we may decrease the font-size of headings and suppress unimportant content items on smaller screens.

For our purposes and keeping things simple, we have customized the adaptive web page design defined by the HTML5 Boilerplate project (more precisely, the minimal "responsive" configuration available on www.initializr.com). It consists of an HTML template file and two CSS files: the browser style normalization file `normalize.css` and a `main.css`, which contains the HTML5 Boilerplate style and our customizations. Consequently, we use a new `css` subfolder containing these two CSS files:

```
1-MinimalApp-with-CSS
  public
    css
      main.css
      normalize.css
    js
      m
      v
    index.html
```

We define our own styles for `<table>`, `<menu>` and `<form>` elements, in `main.css`. Concerning the styling of HTML forms, we define a simple style for implicitly labeled form control elements.

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

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

Since the styling of user interfaces is not our primary concern, we do not discuss its details and leave it to our readers to take a closer look. You can run the CSS-styled minimal app from our code or download its code as a ZIP archive file.

4.8.2 Catching invalid data

The app discussed in this chapter is limited to support the minimum functionality of a data management app, and it does not prevent users from entering invalid data into the app's database. In Part 2 of this tutorial, we show how to express integrity constraints in a model class and how to perform data validation both in the model/storage code of the app and in the user interface code.

4.8.3 Boilerplate code

Another issue with the do-it-yourself code of this example app is the *boilerplate code* needed per model class for the data storage management methods `add`, `retrieve`, `update`, and `destroy`. While it is good to write this code a few times for learning app development, you don't want to write it again and again later when you work on real projects. In *our mODELcLASSjs tutorial*, we present an approach to putting these methods in a generic form in a meta-class, such that they can be reused in all model classes of an app.

4.8.4 Serialization and de-serialization

Serializing a property's value means converting it to a suitable string value. For standard data types, such as numbers, a standard serialization is provided by the predefined conversion function `String`, which doesn't have to be used in many cases since the JS engine performs the serialization automatically.

When a string value, like "13" or "yes", represents the value of a non-string-valued attribute, it has to be *de-serialized*, that is, converted to the range type of the attribute, before it is assigned to the attribute. This is the situation, for instance, when a user has entered a value in a form input field for an integer-valued attribute. The values of form fields are always of type `string`. Consequently, the value of a form input field for an integer-valued attribute has to be converted (de-serialized) to an integer using the predefined conversion function `parseInt`.

For instance, in our example app, we have the integer-valued attribute `year`. When the user has entered a value for this attribute in a corresponding form field, in the *Create* or *Update* user interface, the form field holds a string value, which has to be converted to an integer in an assignment like the following:

```
this.year = parseInt( formEl.year.value);
```

One important question is: where should we take care of de-serialization: in the "view" (before the value is passed to the "model" layer), or in the "model"? Since attribute range types are a business concern, and the business logic of an app is supposed to be encapsulated in the "model", de-serialization should be performed in the "model" layer, and not in the "view".

4.8.5 Synchronizing views with the model

When more than one user uses an app at the same time, we have to take care of somehow synchronizing the possibly concurrent read/write actions of users such that users always have current data in their "views" and are prevented from interfering with each other. This is a complicated problem, which is attacked differently by different approaches. It has been mainly investigated for multi-user database management systems and large enterprise applications built on top of them.

The original MVC proposal included a data binding mechanism for automated one-way model-to-view synchronization (updating the model's views whenever a change in the model data occurs). We didn't take care of this in our minimal app because a front-end app with local storage doesn't have multiple concurrent users. However, we can create a (somewhat artificial) situation that illustrates the issue:

1. Open the *Update* UI page of the minimal app twice (for instance, by opening `updateLearningUnit.html` twice), such that you get two browser tabs rendering the same page.

2. Select the same learning unit on both tabs, such that you see its data in the *Update* view.

3. Change one data item of this learning unit on one of the tabs and save your change.

4. When you go to the other tab, you still see the old data value, while you may have expected it to be automatically updated.

A mechanism for automatically updating all views of a model object whenever a change in its property values occurs is provided by the *observer pattern* that treats any view as an observer of its model object. Applying the observer pattern requires that

- model objects can have a multi-valued reference property like *observers*, which holds a set of references to view objects;
- a *notify* method can be invoked on view objects by the model object whenever one of its property values is changed; and

- the *notify* method defined for view objects refreshes the user interface.

Notice, however, that the general model-view synchronization problem is not solved by automatically updating all (other users') views of a model object whenever a change in its data occurs because this would only help if the users of these views didn't make themselves any modification of the data item concerned, meanwhile. Otherwise, their changed data value would be overwritten by the automated refresh, and they may not even notice this, which is not acceptable in terms of usability.

4.8.6 Architectural separation of concerns

From an architectural point of view, it is important to keep the app's model classes independent of

1. the user interface (UI) code because it should be possible to reuse the same model classes with different UI technologies;

2. the storage management code because it should be possible to reuse the same model classes with different storage technologies.

In this tutorial, we have kept the model class `Book` independent of the UI code since it does not contain any references to UI elements, nor does it invoke any view method. However, for simplicity, we didn't keep it independent of storage management code since we have included the method definitions for add, update, destroy, etc., which invoke the storage management methods of JavaScrpt's `localStorage` API. Therefore, the separation of concerns is incomplete in our minimal example app.

We show in our mODELcLASSjs tutorial how to achieve a complete separation of concerns by defining abstract storage management methods in a special storage manager class, which is complemented by libraries of concrete storage management methods for specific storage technologies, called *storage adapters*.

4.8.7 404 Pages

Whenever a Firebase project is set up, a `404.html` document is generated in the root folder, generally named "public". The primary purpose of this web page is to attend to a crucial -but often unattended- issue of the user experience on web apps and sites: address errors when a web page is not found within a web server. This may happen for several reasons, like a web page being renamed, moved to another folder, or simply because it doesn't exist anymore. Nevertheless, the consequence will always be that the user will be unable to find specific content or resource that previously existed. Web apps and sites are dynamic entities that change along with their life, so handling this issue is a good practice. Another goal of 404 pages is to turn a potential negative user experience into a positive one by providing the necessary information that guides the user to find the new location of the seek resource. 404 pages contain links to help users exit successfully from the error page.

4.9. Quiz Questions

4.9.1 Question 1: Properties/methods of a model class

Which of the following are properties or methods of a model class Book? Select one or more:

- □ `Book.retrieveAll`
- □ `Book.update`
- □ `Book.destroy`
- □ `Book.save`
- □ `Book.retrieve`
- □ `Book.load`

□ `Book.instances`

□ `Book.add`

4.9.2 Question 2: Using the output element

In which CRUD use case does the user interface include an HTML output element? Select one or more:

□ Retrieve/list all

□ Update

□ Delete

□ Create

4.9.3 Question 3: Entity tables

Which of the following tables represent entity tables for a model class Book? Select one or more:

□

Key	Value
1	`[ "006251587X", "Weaving the Web", 2000 ]`
2	`[ "0465026567", "Gödel, Escher, Bach", 1999 ]`
3	`[ "0465030793", "I Am A Strange Loop", 2008 ]`

□

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

□

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 }`

□

Key	Value
006251587X	`[ "006251587X", "Weaving the Web", 2000 ]`
0465026567	`[ "0465026567", "Gödel, Escher, Bach", 1999 ]`
0465030793	`[ "0465030793", "I Am A Strange Loop", 2008 ]`

4.10. Practice Projects

4.10.1 Managing information about movies

The purpose of the app to be developed is managing information about movies. Like in the book data management app discussed in the tutorial, you will use Firestore as the cloud database management system.

The app deals with just one object type: `Movie`, as depicted in the **Figure 4-7.** The object type `Movie`. In the subsequent parts of the tutorial, you will extend this simple app by adding integrity constraints, enumeration attributes, further model classes for actors and directors, and the associations between them.

Notice that `releaseDate` is an attribute with range `Date`, so you need to find out how to display, and support user input of, calendar dates.

Figure 4-7. *The object type* `Movie`

Movie
movieId : Integer title : String releaseDate : Date

For developing the app, simply follow the sequence of seven steps described in the tutorial:

1. Step 1 - Set up the Firebase Project
2. Step 2 - Write the Model Code
3. Step 3 - Initialize the Application
4. Step 4 - Implement the *List Objects* Use Case
5. Step 5 - Implement the *Create Object* Use Case
6. Step 6 - Implement the Update Object Use Case
7. Step 7 - Implement the Delete Object Use Case

You can use the following sample data for testing:

Table 4-3. Sample data

Movie ID	Title	Release date
1	Pulp Fiction	1994-05-12
2	Star Wars	1977-05-25
3	Casablanca	1943-01-23
4	The Godfather	1972-03-15

Make sure that

1. your HTML pages comply with the XML syntax of HTML5,
2. international characters are supported by using UTF-8 encoding for all HTML files,
3. your JavaScript code complies with our Coding Guidelines and is checked with JSHint (for instance, instead of the unsafe equality test with "==", always the strict equality test with "===" has to be used).

Chapter 5. Adding Access Control to the Minimal App with Firebase

As nearly every app requires to handle permissions to grant access to certain resources for specific users while simultaneously restricting access for everyone else, we will extend our tutorial of the minimal app described before to show how you can build a simple but effective *access control* handling solution.

On the one hand, we will use Firebase Authentication, which provides a backend solution to authenticate users to mobile and web apps through a Firebase JS SDK library and ready-made UI libraries. On the other hand, we are going to grant and restrict access to the four HTML pages of the CRUD data management use cases already implemented by manipulating the DOM of our start page for enabling and disabling the items of the main menu, likewise as giving access for signing up and signing in the web app.

Firebase Authentication supports user authentication using several methods, such as email and password, phone numbers, popular federated social media and identity providers like Google, Apple, Facebook, Twitter, GitHub, Microsoft, Yahoo and

more. In this tutorial, we have chosen to use the traditional authentication method based on an email and password.

You can run the minimal app with access control from our server or download the code as a ZIP archive file.

5.1. Access Control

In the context of computer systems, whereas ***Access*** is the capability of users to conduct specific tasks in an app, device, or network, ***Access Control*** is a selective *grant* or *restriction* of *access* to data, resources or features, and it consists of two components: ***Authentication*** and ***Authorization***.

Authentication vs authorization: as an example in the context of web development, while users authenticate whom they are by using a password-based method, the web app authorizes individually specific administrative *access* to perform database access operations, such as create, update, delete or retrieve and view a record or file. Usually, *authorization* follows *authentication*, being users forced to prove their identities as genuine before a system grants them *access*.

5.1.1 Authentication

A ***user account*** establishes a relationship between a *user identity* and an app, device, or network, and *Authentication* is the process of validating that users are whom they say to be, their *user identity*, for granting them access control. *Authentication* aims to prove user identity.

According to Schneider, user authentication solutions can be categorized by:

1. **Something you know**: a password, personal identification number (PIN), challenge-response, etc.
2. **Something you have**: security token device/app, certificate, ID card, etc.
3. **Something you are**: DNA sequence, face/voice/retinal pattern, fingerprint, etc.

Credential

A ***credential*** is a digital document, object, or data structure that associates a user identity to a proof of authenticity. An example of a credential may be an email and password combination.

5.1.2 Authorization

Authorization is the process of verifying what specific data/resources/features a particular user has access to. Authorization aims to provide correct access.

User roles

User Roles are permissions defined that control *access* to data/resources/features according to authorization policies.

5.2. Using Firebase for Access Control

5.2.1 User Authentication Status

There are different types of authentication status the user has while using any app with Firebase Authentication:

- **Anonymous**: a visitor who is "*signed in*" as *anonymous* in the background (unnoticeable) by the app. It is the opposite to *registered*, having the user not provided (yet) *email* and *password*. The status before being upgraded to anonymous is "visitor", and the upgrade happens programmatically when the visitor uses the app for the first time.
- **Registered**: a user who has provided *email* and *password*, is already registered in the app, and can be:

- With non-verified email.
- With verified email.

5.2.2 Design of our Access Control Handling Solution

The design of our *access control* handling solution with Firebase Authentication includes the following features:

- Use of the Firebase's *email and password* authentication method.
- The *access control policies* grant access to user with authenticated status *registered with a verified email* to change/write/read operations and restrict access to *registered with non-verified email* and *anonymous* users to read-only operations.
- Rather than relying on role-based access control, the solution will use the different *user authentication statuses* to perform DOM operations on the main menu to implement *access control policies* to grant/restrict access to the database management operations.
- Protect the database with Firestore Security Rules that correspond with the defined access control policies, limiting hence bypassing the front-end layer to get access directly to the database maliciously.
- Individual pages allow users to *sign up* and *sign in* to the app using email and password.
- Registered user must verify email for granting full access to change/write operations. A verification link should be sent after the sign-up process, and once clicked on it, the user is upgraded to full access.
- User's password can be changed if it is forgotten it. An email address must be submitted, and the app sends an email with a link that the requested change is confirmed once clicked on it.
- A ***sign out*** solution to handle when a *registered* user leaves the session, returning to the user authentication status *anonymous*.
- A ***UI that* adapts to *changes of the different* user authentication *statuses***, allows users to sign up/in/out of the app. The UI design should respond to user *interactions with redirections*, *messages* or *restrictions*.

5.2.3 Set up the Folder Structure and add access control files

The MVC folder structure of our minimal app with access control extends the structure of the minimal app by adding five ES6 module files and four HTML files:

- five ES6 module files: **accessControl.mjs**, **actionHandler.mjs**, **resetPassword.mjs**, **signIn.mjs**, **signUp.mjs**, **initFirebase.mjs**, and
- four HTML files: **actionHandler.html**, **resetPassword.html**, **signIn.html**, **signUp.html**.

Thus, we get the following folder structure:

```
1-MinimalApp-with-access-control
  public
    js
      m
        Book.mjs
      v
```

```
    accessControl.mjs
    actionHandler.mjs
    createBook.mjs
    deleteBook.mjs
    resetPassword.mjs
    retrieveAndListAllBooks.mjs
    signIn.mjs
    signUp.mjs
    update.mjs
  initFirebase.mjs
404.html
actionHandler.html
createBook.html
credits.html
deleteBook.html
favicon.ico
index.html
machine-build.svg
resetPassword.html
retrieveAndListAllBooks.html
signIn.html
signUp.html
update.html
```

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

5.3. Step 1: Initialize Firebase Authentication

5.3.1 Enable Firebase Authentication

Go to the Firebase Console and click on the "*Authentication*" option in the main menu. Select the tab ***Sign-in Method*** and enable the option "`Email/Password`" and the option "`Anonymous`".

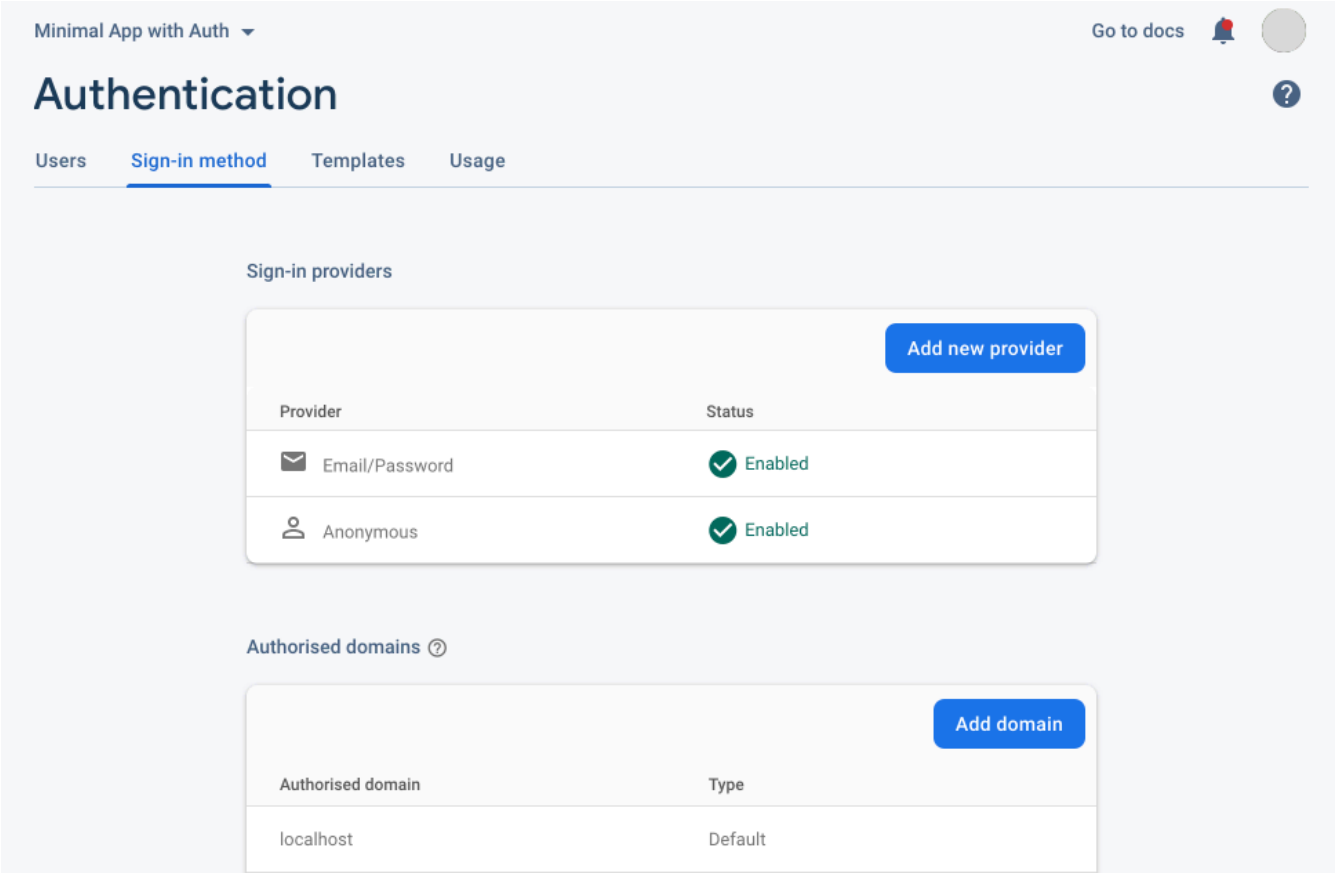


Figure 5-1. *Enabling sign-in providers in Firebase*

5.3.2 Initialize Firebase Authentication

In the file `initFirebase.mjs` we add the Firebase user authentication interface, creating an interface to our authentication instance in the "`auth`" object, which later is exported to be consumed by other ES6 modules that are part of the access control handling solution:

```
import { initializeApp, getApp, getApps } from "https://www.gstatic.com/firebasejs/9.X.X/f
import { getFirestore } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-firestore-
import { getAuth } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-auth.js";

// TODO: Replace the following with your web app's Firebase project configuration
const config = {
  apiKey: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  authDomain: "minimalapp-XXXX.firebaseapp.com",
  projectId: "minimalapp-XXXX",
  appId: "1:XXXXXXXXXXX:web:XXXXXXXXXXXXXXXXXXXXX"
};
// Initialize a Firebase App object only if not already initialized
const app = (!getApps().length) ? initializeApp( config ) : getApp();
// Initialize Firebase Authentication
const auth = getAuth( app);
// Initialize Firestore interface
const fsDb = getFirestore();

export { auth, fsDb };
```

Notice that this time we initialize the Firebase App instance differently, storing it as an object in a variable named "`app`" and later being given as a parameter to the `getAuth()` function to create the authentication instance in the "`auth`" object. Contrary to how we initialize Firestore, we cannot initialize Firebase Authentication without using the Firebase App instance object. For such purpose, we evaluate first whether there is already an initialized Firebase app instance using the `getApps()` function, and, if not, initialize it with the `initializeApp()` function. If there is already an initialized Firebase app instance, we use the getApp() function to get it and store it in the variable "`app`".

```
const app = (!getApps().length) ? initializeApp( config ) : getApp();
...
const auth = getAuth( app);
```

5.4. Step 2: Prepare UI for Authentication and Authorization

According to the design of our access control handling solution, once the user is authenticated, the proposed implementation deals mainly with two issues on the user interface:

1. one for handling the **main menu**, granting and restricting access to pages of the four CRUD data management use cases,
2. and another for handling how to **sign up**, **sign in**, and **sign out** the app.

5.4.1 Prepare the Main Menu for Granting or Restricting Access

The menu items (links and buttons) on the start page may have any of the following three states:

1. ***Disabled***, for menu items to access change/write operations if the user status is *anonymous*.
2. ***Enabled***, for menu items to access either
   - change/write/read operations if a logged-in user is *registered* with a verified email,
   - or read operations if a user is *anonymous*.
3. ***Enabled but unavailabe***, for menu items to access change/write operations if the user is registered with non-verified email. The menu item is enabled, but a redirection back to the start page is triggered after clicking on it.

For setting up the default state of the menu items we perform different DOM operations on UI elements. For instance, the `<a>` elements are disabled using CSS rules, using the "*disabled*" `class`, and the `<button>` elements are disabled adding the `disabled` attribute. The whole setup can be seen on the resulting in this HTML code:

```
<ul role="menubar">
  <li role="menuitem">
    <a href="createBook.html" class="disabled">Create</a> a new book record
  </li>
  <li role="menuitem">
    <a href="retrieveAndListAllBooks.html">Retrieve</a> and list all book records
  </li>
  <li role="menuitem">
    <a href="updateBook.html" class="disabled">Update</a> a book record
  </li>
  <li role="menuitem">
    <a href="deleteBook.html" class="disabled">Delete</a> a book record
  </li>
  <li role="menuitem">
```

```
    <button class="generateTestData" type="button" disabled="disabled">Generate test data<
  </li>
  <li role="menuitem">
    <button id="clearData" type="button" disabled="disabled">Clear test data</button>
  </li>
</ul>
```

Notice that the only "*non-disabled*" menu item by default is `"Retrieve and list all book records"` since we want to allow unlimited access to the only read operation, whether the user status is *anonymous* or not.

We also use CSS rules: the "*disabled*" `class`, removes click events from the `<a>` elements using the property `pointer-events` set to `none` and changes their appearance by reducing its `opacity` and setting the `cursor` to `default`, which removes the "hand" icon when hovering "*clickable*" UI elements:

```
a.disabled {
  opacity: 0.4;
  pointer-events: none;
  cursor: default;
}
```

5.4.2 Login Management Area

The main UI element that accompanies the four pages of the CRUD use cases in our app is a ***login management area*** that adapts its behavior and appearance according to changes in the user authentication status. This area, located in the `header` section of each page, gives access to links and buttons that allow users to ***sign up*** (page `signUp.html`), ***sign in*** (page `signIn.html`), and ***sign out***. User messages are also shown in this area, sometimes inviting to do something, others providing information about the current authentication status.

```
<header>
  <div id="login-management"></div>
  <h1>Minimal App with Authentication - Public Library</h1>
</header>
```

5.4.3 Error Messages from Firebase Authentication

Error messages are meant to provide feedback to users about what they did wrong and how to fix mistakes. The design of our access control handling solution relies very much on user interaction elements. Since many things may go wrong while interacting with such forms, buttons, verification emails/links, etc., we need to handle error messages returned from our Firebase Authentication instance accordingly. To show error messages, every page view related to our access control handling solution includes a `div` element exclusively for displaying error messages.

```
<header>
  <div id="message" hidden="hidden"></div>
  <h1>...</h1>
  ...
</header>
```

5.5. Step 3: Implement the Access Control Handling Solution

Our implementation of the *access control handling solution* resides in the view code file `js/v/accessControl.mjs`, and it is divided in the following five procedures:

1. **`handleAuthentication()`** handles Firebase Authentication SDK functionalities to identify the *user authentication status* and invoke the correct authorization via the `handleAuthorization()` procedure.

2. **`handleAuthorization()`** handles the *authorization policies* to correctly *grant*/*restrict* access to the four CRUD use cases by DOM operations on the UI via the `createSignInAndSignUpUI()` and `createSignOutUI()` helper functions.

3. **`createSignInAndSignUpUI()`** renders the UI elements and interactivity for accessing the *sign-up* and *sign-in* pages.

4. **`createSignOutUI()`** renders the UI elements and interactivity for *signing out* the app.

5. **`handleSignOut()`** handles the end of the user session in the app.

The first step is importing the `auth` instance object from the Firebase initialization file `initFirebase.mjs`, and required functions from the Firebase Authentication SDK library.

```
import { auth } from "../initialize.mjs";
import { onAuthStateChanged, signInAnonymously, signOut } from "https://www.gstatic.com/fi
```

5.5.1 Handle Authentication

In the `handleAuthentication()` procedure resides the logic that encompasses the authentication capabilities of the Firebase Authentication SDK library with the *access control policies* to authorize access to the CRUD use case pages. For handling authentication we evaluate the different *user authentication statuses* for invoking accordingly the `handleAuthorization()` procedure, which grants and restricts access to the CRUD use case pages.

It starts by evaluating the user authentication status using the `onAuthStateChanged()` function, which returns a user instance object examined by its `isAnonymous` property. We know if the user is signed in as anonymous or registered throughout this property. However, if the user instance object returns null, there is no active user session, and we use the `signInAnonymously()` method to upgrade the user from visitor to anonymous, signing in an anonymous user session in the app.

For each evaluated *user authentication status* we invoke the `handleAuthorization()` procedure with different parameters for handling authorization. The first parameter is a string value that describes the *user authentication status*.

```
function handleAuthentication() {
  // get current page value
  const currentPage = window.location.pathname;
  try {
    // evaluate user authentication status
    onAuthStateChanged( auth, async function (user) {
      // if status is "anonymous" or "registered"
      if (user) {
        if (user.isAnonymous) { // if user is "anonymous"
          handleAuthorization( "Anonymous", currentPage);
        } else { // if status is "registered"
          if (!user.emailVerified) { // if email address is not verified
            handleAuthorization( "Registered with non-verified email", currentPage, user.e
          } else { // if email address is verified
            handleAuthorization( "Registered with verified email", currentPage, user.email
          }
        }
      }
```

```
      else signInAnonymously( auth); // otherwise, upgrade to "anonymous"
    });
  } catch (e) {
    console.error(`Error with user authentication: ${e}`);
  }
}
```

The second parameter used with the `handleAuthentication()` procedure is the "*current page*" in which the user is located, using the `location` interface and its property `pathname`,

```
const currentPage = window.location.pathname;
```

and the third -and optional– parameter is the user's email address, get from using the `email` property on the `user` instance object.

5.5.2 Handle authorization

The `handleAuthorization()` procedure coordinates every UI behaviour orchestrating DOM operations either in the login management area or main menu items, using a `switch`/`case` statement based on the three possible passed *user authentication status*es. We present the basic structure of this procedure:

```
function handleAuthorization( userStatus, currentPage, email) {
  ...
  switch (userStatus) {
    case "Anonymous":
      ...
    case "Registered with non-verified email":
      ...
    case "Registered with verified email":
      ...
  }
}
```

Before handling any possible case of *user authentication status*, we declare variables for accessing the *login management area*, a list of "*authorized pages*" that *anonymous* users can access without authentication, and another list containing the two forms of how the property `pathname` of the `location` interface recognizes the "start page". Notice that the `authorizedPages` variable concatenates both lists:

```
function handleAuthorization( userStatus, currentPage, email) {
  // declare variables for current page and for accessing UI elements
  const divLoginMgmtEl = document.getElementById("login-management"),
    startPage = ["/","/index.html"],
    authorizedPages = startPage.concat(["/retrieveAndListAllBooks.html", "/credits.html"])
  switch (userStatus) {
    ...
```

If case is "*Anonymous*"

First, we check if the current page is in the list of *authorized pages*, and if not, the user is redirected to the *sign-up* page (`signUp.html`), or else we show in the *login management area* the links to the `signUp.html` and `signIn.html` pages, using the `createSignInAndSignUpUI()` helper function:

```
...
case "Anonymous":
  // if user is not authorized to current page, restrict access & redirect to sign up page
  if (!authorizedPages.includes( currentPage)) window.location.pathname = "/signUp.html";
  else divLoginMgmtEl.appendChild( createSignInAndSignUpUI());
  console.log(`Authenticated as "${userStatus}"`);
  break;
...
```

If case is "*Registered with non-verified email*"

First, we check if the current page is in the list of *authorized pages*, and if not, the user is redirected to the start page (`index.html`), or else we invoke the `createSignOutUI()` helper function that creates a "Sign out" button element in the *login management area*. This time two parameters are passed to the function, the user's email address and a boolean that, if present with the value `true`, renders a message inviting the user to verify the registered email address:

```
...
case "Registered with non-verified email":
  // if user is not authorized to current page, restrict access & redirect to start page
  if (!authorizedPages.includes( currentPage)) window.location.pathname = "/index.html";
  else divLoginMgmtEl.appendChild( createSignOutUI( email, true));
  console.log(`Authenticated as "${userStatus}" (${email})`);
  break;
...
```

If case is "*Registered with verified email*"

First, we declare variables for accessing UI elements on the main menu located on the *start page* (`index.html`). After checking if we are located on it, we enable links by removing the CSS class "disabled" and buttons by setting to false the "disabled" attribute, authorizing as a result full access to create/write operations in the pages of the four database access operations. Then we invoke the `createSignOutUI()` helper function that creates a "Sign out" button element in the *login management area*. This time only one parameter is passed to the function, the user's email address:

```
...
case "Registered with verified email":
  // if current page is start page grant access to the four database operations
  if (startPage.includes( currentPage)) {
    // declare variables for accessing UI elements
    const clearDataBtn = document.getElementById("clearData"),
      generateDataBtns = document.querySelectorAll(".generateTestData"),
      disabledEls = document.querySelectorAll(".disabled");
    // perform DOM operations to enable menu items
    for (const el of disabledEls) el.classList.remove("disabled");
    clearDataBtn.disabled = false;
    for (const btn of generateDataBtns) btn.disabled = false;
  }
  divLoginMgmtEl.appendChild( createSignOutUI( email));
  console.log(`Authenticated as "${userStatus}" (${email})`);
  break;
...
```

5.5.3 Helper Functions for Rendering the UI

The two functions in charge of manipulating UI elements related to authentication and authorization *are* `createSignInAndSignUpUI()` *and* `createSignOutUI()`.

Notice the use of the JavaScript method `createDocumentFragment()`, that creates a DOM node object separated from the main DOM tree, over which we perform DOM operations without affecting it, being consequently more computationally efficient. When the fragment is appended to the main DOM tree, it disappears in just one operation:

```
function createSignInAndSignUpUI() {
  const fragment = document.createDocumentFragment(),
    linkSignUpEl = document.createElement("a"),
    linkSignInEl = document.createElement("a"),
    text = document.createTextNode(" o ");
  linkSignUpEl.href = "signUp.html";
  linkSignInEl.href = "signIn.html";
  linkSignUpEl.textContent = "Sign up";
  linkSignInEl.textContent = "Sign in";
  fragment.appendChild( linkSignUpEl);
  fragment.appendChild( text);
  fragment.appendChild( linkSignInEl);
  return fragment;
}
```

The `createSignOutUI()` helper function receives two parameters, one is the user's email address as string, and an optional boolean that, if `true`, it renders a message inviting the user to verify the registered email address. Notice the event listener added to the button that invokes the `handleSignOut()` function while the DOM operations happen:

```
function createSignOutUI( email, invitation) {
  const fragment = document.createDocumentFragment(),
    divEl = document.createElement("div"),
    buttonEl = document.createElement("button");
  if (invitation) {
    const divEl = document.createElement("div");
    divEl.textContent = "Check your email for instructions to verify your account " +
      "and authorize access to operations";
    fragment.appendChild( divEl);
  }
  buttonEl.type = "button";
  buttonEl.innerText = "Sign Out";
  buttonEl.addEventListener("click", handleSignOut);
  divEl.innerText = `${email} `;
  divEl.appendChild( buttonEl);
  fragment.appendChild( divEl);
  return fragment;
}
```

5.5.4 The Sign out Procedure

This procedure is part of the user authentication status handler, and it ends the user session with the Firebase Authentication SDK `signOut()` function, redirecting afterwards the user to the start page.

```
async function handleSignOut() {
  try {
```

```
    signOut( auth);
    window.location.pathname = "/index.html";
  } catch (e) {
    console.error(`${e.constructor.name}: ${e.message}`);
  }
}
```

5.6. Step 4: Implement Sign up and Sign in

The user interfaces for either *signing up* or *signing in* the app reside respectively in the HTML pages `signUp.html` and `signIn.html`. They have allocated a similar number of interactive elements, such as 1) a `<form>` element, 2) two `<input>` to capture accordingly "***email***" and "***password***" with type attributes `email` and `password`, and 3) and a `<button>` element that invokes its corresponding authentication procedure. We present the HTML form located in the file `signUp.html`:

```
<head>
  ...
  <script type="module" src="js/v/signUp.mjs" ></script>
  ...
</head>
<body>
  ...
  <form id="Auth">
    <div>
      <label>Email: <input name="email" type="email"/></label>
    </div>
    <div>
      <label>Password: <input name="password" type="password" placeholder="6+ characters"/
    </div>
    <div>
      <button type="button" name="signUp">Sign up</button>
    </div>
  </form>
  ...
</body>
```

However, in the user interface for signing in the app located in the HTML file `signIn.html` you may see an additional element, a link labelled as "Forgot password?" to the HTML file `resetPassword.html`, offering the choice to reset the password in case the user forgot it. In the next step, we will discuss this issue in details.

```
<head>
  ...
  <script type="module" src="js/v/signIn.mjs" ></script>
  ...
</head>
<body>
  ...
  <form id="Auth">
    <div>
      <label>Email: <input name="email" type="email"/></label>
    </div>
    <div>
```

```
      <label>Password: <input name="password" type="password"/></label>
    </div>
    <div>
      <button type="button" name="signIn">Sign in</button>
    </div>
  </form>
  <p>
    Or <a href="signUp.html" title="Sign in">sign up</a> to create your account.
  </p>
  <p><a href="resetPassword.html" title="Reset password">Forgot password?</a></p>
  ...
</body>
```

Both `signUp.html` and `signIn.html` invoke, respectively, view code from the ES6 modules `signUp.mjs` and `signIn.mjs` to add authentication behaviour to the user interface.

5.6.1 Sign up View Code

In simple, the *sign-up* view code upgrades an *anonymous* user to *registered*, using the *email* and *password* passed from the HTML form, in four steps:

1. We generate an authentication credential object using the `createUserWithEmailAndPassword()` method, providing it as parameters the Firebase Athentication interface object `auth`, and the entered email and password.

2. We create a `user` reference in our project's instance of Firebase Authentication using the user method of the authentication credential object.

3. At this point, the user account has been created in our project's instance of Firebase Authentication, being the current status of the user *registered with non-verified email*. However, to complete the *sign up* process, we send a verification email that contains a link including a *unique activation code*. The verification email is generated and sent from our Firebase's project instance, using the `sendEmailVerification()` method and the `user` reference described in step 2.

4. We display an alert dialog with a reminder message encouraging us to verify the provided email to complete the *sign up* process.

Notice that we do all this in a `try`/`catch` block, and if Firebase Authentication API returns an error message, we "catch it" and display it accordingly on the `div` element created for such purpose.

```
import { auth } from "../initFirebase.mjs";
import { createUserWithEmailAndPassword, sendEmailVerification } from "https://www.gstatic

const formEl = document.forms["Auth"],
  signUpBtn = formEl["signUp"];

// manage sign up event
signUpBtn.addEventListener("click", async function () {
  const email = formEl["email"].value,
    password = formEl["password"].value;
  if (email && password) {
    try {
      // create account and get credential by providing email and password
      // user is signed in automatically if the account is created successfully
```

```
      const userCredential = await createUserWithEmailAndPassword( auth, email, password);
      // get user reference from Firebase
      const userRef = userCredential.user;
      // send verification email
      await sendEmailVerification( userRef);
      console.log (`User ${email} became "Registered"`);
      alert (`Account created ${email}. Check your email for instructions to verify this a
      window.location.pathname = "/index.html"; // redirect user to start page
    } catch (e) {
      const divMsgEl = document.getElementById("message");
      divMsgEl.textContent = e.message;
      divMsgEl.hidden = false;
    }
  }
});
```

5.6.2 Sign in View Code

This procedure uses the `signInWithEmailAndPassword()` method to *sign in* to the app with email and password. If the *sign-in* process is successful, the user is redirected to the start page. Likewise, the *sign-up* process error messages are handled accordingly in this view code:

```
import { auth } from "../initFirebase.mjs";
import { signInWithEmailAndPassword } from "https://www.gstatic.com/firebasejs/9.X.X/fireb

const formEl = document.forms["Auth"],
  signInBtn = formEl["signIn"];

signInBtn.addEventListener("click", async function () {
  const email = formEl["email"].value,
    password = formEl["password"].value;
  if (email && password) {
    try {
      // sign in user using email + password
      await signInWithEmailAndPassword( auth, email, password);
      window.location.pathname = "/index.html"; // redirect user to start page
    } catch (e) {
      const divMsgEl = document.getElementById("message");
      divMsgEl.textContent = e.message;
      divMsgEl.hidden = false;
    }
  }
});
```

5.7. Step 5: Implement User Authentication Action Handlers

Both user authentication procedures, *sign-up* and *sign-in*, reviewed in the previous step, involve a ***confirmation email*** sent to the user for completing

1. ***email address verification***, when signing up the app, and
2. ***password reset***, if the user forgot it, at the moment of signing to in the app.

The confirmation emails contain a unique and self-generated link that recipients open to handle and complete the user management action on another web page, called now ***email action handler page***.

By default, Firebase hosts generic templates for user authentication action handlers, but we choose to create our pages for custom email action handler and confirmation email. Firebase also uses user authentication action handlers to change the accounts' primary email addresses, but we won't cover such because it is pretty similar to the others covered in this tutorial.

5.7.1 Email Action Handler Templates

Go to the Firebase Console and click on the "*Authentication*" option in the main menu. Select the tab "*Templates*" and

1. edit the fields that have been filled out automatically by Firebase, such as ***Sender name***, ***From***, ***Reply to***, and ***Subject***, with the content of your choice. Notice that Firebase prevents customizing the field **Message** to evade abusive behaviour, such as spam; if you want to customize the email message, you need to handle the flow outside Firebase cloud services. In this same screen, you can make individual changes in the templates for "Email address verification" and "Password reset", and afterwards,
2. being on edit mode in any template, click on "*Customize action URL*", below the "***Action URL***" field, and refactor the URL by replacing the ending part "`__/auth/action`" with the file name of the action handler page, resulting into something similar to "`https://xxxxx.firebaseapp.com/actionHandler.html`". Notice that the *Action URL* value is shared across all templates, so if you change it in one action handler/template, the same value is updated in all the other action handlers/templates.

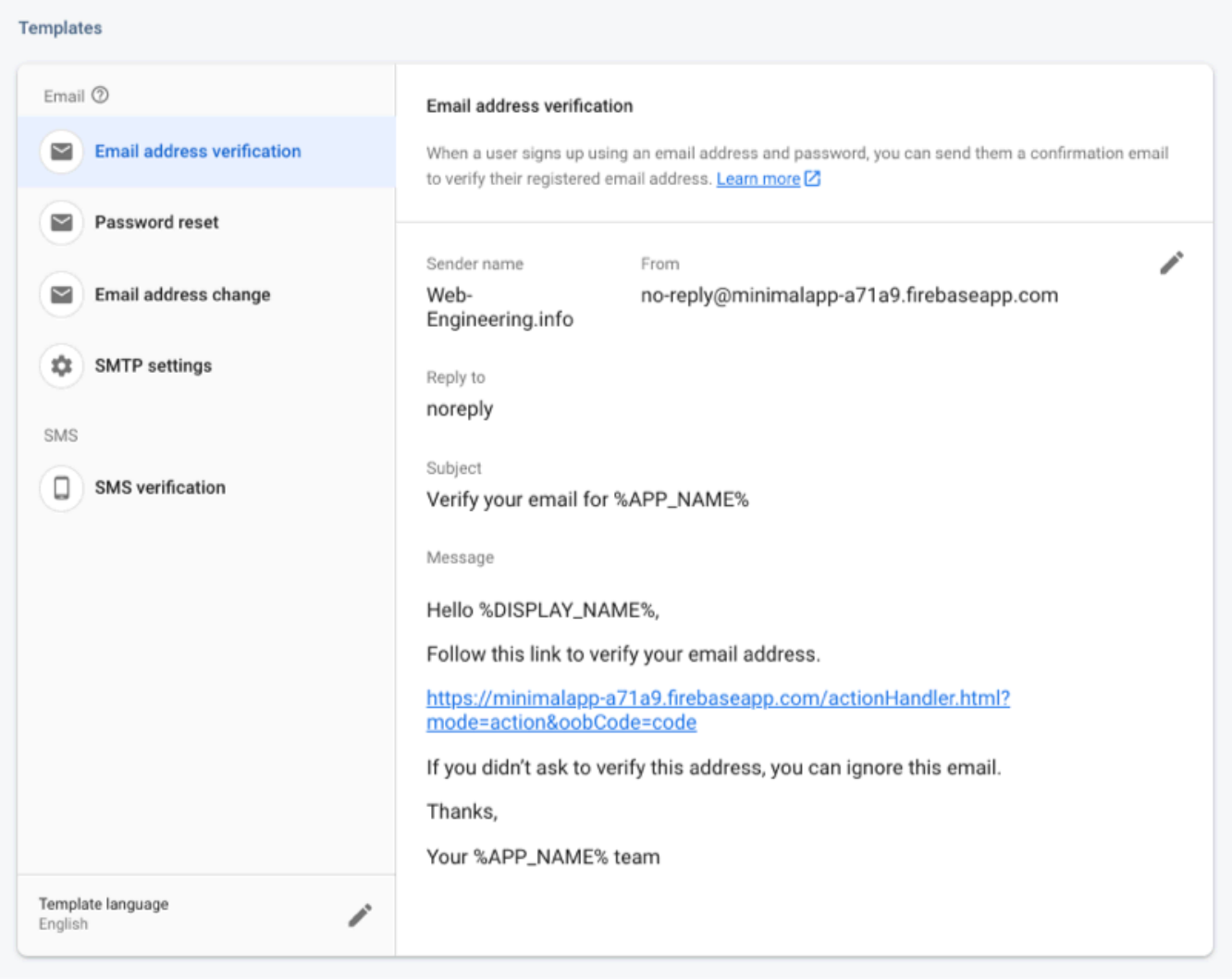


Figure 5-2. *Email action handler templates*

5.7.2 Email Action Handler Page

The *email action handler page* is common for each user management action handler, so we create an HTML file named `actionHandler.html`, and within we allocate a `<section>` element for each planned action; the first `<section>` element deals with *email address verification*, and the second with *password reset*. Both `<section>` elements are *hidden* by default, and contain a basic structure that defines the page's layout, and after being filled out with content by the view code it is "*unhidden*". Notice that the heading element `<h1>` is also present empty and "*hidden*" by default:

```
<header>
  <div id="message" hidden="hidden"></div>
  <div>
    <div><h1></h1></div>
  </div>
</header>
<main>
  <div>
    <section hidden="hidden">
      <p></p>
    </section>
    <section hidden="hidden">
      <p></p>
      ...
    </section>
  </div>
</main>
```

When Firebase generates the link attached in the *confirmation email* to complete the action, several query parameters are added to the ***action handler URL***, so the first step is parsing from the URL the two parameters needed to complete the user authentication actions: 1) "*mode*" (stored in a variable `mode`), and 2) "*oobCode*" (stored in the variable `actionCode`). Notice that the oobCode has a self generated unique code, invalidated by Firebase once it is used.

The following is an example of an *action handler URL* with parameters generated by Firebase and attached to *confirmation emails*:

```
https://xxxxx.firebaseapp.com/actionHandler.html?mode=verifyEmail&oobCode=sxMPvK72F3gNuEd4
```

Notice the optional query parameters present in the action handler URL, such as ***apiKey*** and ***lang***, included for convenience if you need your Firebase project's API Key and/or plan to provide localized email action handler pages.

In the view code module file `actionHandler.mjs` we initialize individual variables for accessing each section in the page, `sectionVeriEmailEl` and `sectionRstPswEl`, the first section element for the action handler for *email address verification*, and the second for the action handler for *password reset*.

To handle the required actions we use a `switch/case` statement using the "`mode`" variable value, to invoke either the procedures `handleVerifyEmail()` or `handleResetPassword()` with the section element and the unique action code as parameters:

```
const mode = getParameterByName("mode");
const actionCode = getParameterByName("oobCode");
```

```
const [sectionVeriEmailEl, sectionRstPswEl]
  = document.querySelectorAll("main>div>section");

switch (mode) {
  case "verifyEmail":
    sectionVeriEmailEl.hidden = false;
    await handleVerifyEmail( sectionVeriEmailEl, actionCode);
    break;
  case "resetPassword":
    sectionRstPswEl.hidden = false;
    await handleResetPassword( sectionRstPswEl, actionCode);
    break;
}
```

For getting parameter values from the URL we use the `getParameterByName()` function.

```
function getParameterByName( parameter) {
  const urlParams = new URLSearchParams( location.search);
  return urlParams.get( parameter);
}
```

5.7.3 Verify Email Address

The view code for handling *email address verification* is centralized in the `handleVerifyEmail() procedure` within the `actionHandler.html` file. We start by declaring variables for accessing the heading element h1 and the p element (`h1El` and `pEL`), in which we add text content to communicate the result of this action handler. Later, in a `try/catch` statement we use the `actionCode` variable value to verify if the action code is valid using the `verifyPasswordResetCode()` function. Although this Firebase Authentication SDK's function is not documented for use beyond validating action codes for password resetting, its use is safe in this context. Afterwards, ensuring the action code is valid, we use the `applyActionCode()` function with confidence. The remaining code deals with DOM operations for embedding text in the web page ,whether the *email address verification* is successful or failed. Notice that if the verification process fails, the error message coming from our Firebase Authentication instance is displayed in the `div` element reserved for handling user messages.

```
async function handleVerifyEmail( sectionVeriEmailEl, actionCode) {
  const h1El = document.querySelector("h1"),
    pEl = sectionVeriEmailEl.querySelector("p");
  let email = null;
  try {
    email = await verifyPasswordResetCode( auth, actionCode);
    await applyActionCode( auth, actionCode);
    h1El.textContent = "Your email address has been verified";
    const bEl = document.createElement("b");
    bEl.textContent = email;
    pEl.innerText = "Now this account can use any data management operation: ";
    pEl.appendChild( bEl);
  } catch (e) {
    h1El.textContent = "Invalid or expired link.";
    pEl.textContent = "Your email address has not been verified.";
    const divMsgEl = document.getElementById("message");
    divMsgEl.textContent = e.message;
    divMsgEl.hidden = false;
```

```
  }
}
```

5.7.4 Reset Password

The flow for *password reset* starts when the user intentionally clicks on the link *"Forgot password?"* located in the Sign-in page `signUp.html`, which leads to the `forgotPassword.html` page. This web page has a `<form>` element containing an <input> field in which the user enters an email address to the confirmation email is being sent. Finally, a `<button>` element is in charge of submitting the form content to the view code.

```
<header>
  <div id="message" hidden="hidden"></div>
  <h1>Reset your password</h1>
  <p>
    Have you forgotten your password? No problem! Just enter your email address and click
  </p>
</header>
<main>
<form id="Password">
  <div>
    <label>Email: <input name="email" type="email"/></label>
  </div>
  <div>
    <button name="commit" type="button">Reset your password</button>
  </div>
</form>
</main>
```

The view code in the `passwordReset.mjs` ES6 module initialize variables for accessing UI elements, such as the form and button elements. An event listener is attached to the button, and whenever the user clicks on it, a *confirmation email* is being sent to the user using the `sendPasswordResetEmail()` function using the entered email address as parameter. Finally, an alert is showed instructing the user to check their email in order to create the new password; and the user is redirected to the start page:

```
import { auth } from "../initFirebase.mjs";
import { sendPasswordResetEmail } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-

const formEl = document.forms["Password"],
  resetBtn = formEl["commit"];

resetBtn.addEventListener("click", async function () {
  const email = formEl["email"].value;
  if (email) {
    try {
      await sendPasswordResetEmail( auth, email);
      alert(`Check your email "${email} and confirm this request to create a new password.
      window.location.pathname = "/index.html";
    } catch (e) {
      const divMsgEl = document.getElementById("message");
      divMsgEl.textContent = e.message;
      divMsgEl.hidden = false;
    }
```

```
  }
});
```

As well as for the email address verification action, once the user clicks on the attached link, the action handler URL leads the user to the *email action handler page* `actionHandler.html`. Its corresponding view code is located in the `actionHandler.mjs` file, whereby evaluates the action handler URL parameter "mode" and, if its value is "`resetPassword`" then "unhides" the corresponding section element and invokes the `handleResetPassword()` procedure with the value of action code as parameter:

```
switch (mode) {
  case "verifyEmail":
    ...
  case "resetPassword":
    sectionRstPswEl.hidden = false;
    await handleResetPassword( sectionRstPswEl, actionCode);
    break;
}
```

On the action handler page `actionHandler.html`, the second <section> element contains an <input> element where the new password can be entered:

```
<main>
  <section hidden="hidden">
    ...
  </section>
  <section hidden="hidden">
    <p></p>
    <form id="Password">
      <div>
        <label>New password: <input name="password" type="password"/></label>
      </div>
      <div>
        <button type="button" name="commit">Save</button>
      </div>
    </form>
  </section>
  ...
</main>
```

The `handleResetPassword()` procedure is very similar to the previously described `handleVerifyEmail()`, but additionally, after verifying the email address is valid, the Firebase Authentication SDK's function `confirmPasswordReset()` resets the new password using the action code. Finally, using the email address and reset password, the user is signed in automatically and unnoticed using the `signInWithEmailAndPassword()` function.

```
async function handleResetPassword( sectionRstPswEl, actionCode) {
  const h1El = document.querySelector("h1"),
    pEl = sectionRstPswEl.querySelector("p"),
    formEl = document.forms["Password"];
  try {
    const email = await verifyPasswordResetCode( auth, actionCode);
    h1El.textContent = "Reset password";
```

```
    const bEl = document.createElement("b");
    bEl.textContent = email;
    pEl.innerText = "For: ";
    pEl.appendChild( bEl);
    const saveButton = formEl["commit"];
    saveButton.addEventListener("click", async function () {
      const newPassword = formEl["password"].value;
      if (newPassword) {
        await confirmPasswordReset( auth, actionCode, newPassword);
        alert(`Your password has been update! You will be automatically signed in with you
        await signInWithEmailAndPassword( auth, email, newPassword);
        window.location.pathname = "/index.html"; // redirect user to start page
      }
    });
  } catch (e) {
    formEl.hidden = true;
    h1El.textContent = "Invalid or expired link.";
    pEl.textContent = "Your password cannot be reset.";
    const divMsgEl = document.getElementById("message");
    divMsgEl.textContent = e.message;
    divMsgEl.hidden = false;
  }
}
```

5.8. Step 6: Configure Security Rules

Although our access control handling solution for the minimal app manages authorization on the user interface, we also need to restrict the direct access to our Firestore database instance to protect it against any direct query. The following ***Firestore Security Rules*** also have been created following the *authorization policies* reviewed previously for the four CRUD use cases.

In the following Firestore Security Rules we are allowing to ***change/write*** only if the user authentication status is "Registered with verified email", and to ***read*** if the user authentication status is either "Anonymous" or "Registered with non-verified email", so those users are authorized to **Retrieve** records/documents.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /books/{docID} {
      allow write: if request.auth.token.email_verified == true;
      allow read: if request.auth != null;
    }
  }
}
```

`match /books/{docID}` is a wildcard to define the scope of the rule to every document in the table/collection "`book`".

As we said before, for updating the Firestore Security Rules for our Cloud Firestore database instance, we advise you to use the file `firestore.rules`, and then ***deploy*** the whole project with "`firebase deploy`". The deployment of a new version of security rules overwrites the previous rules located on the cloud. Notice that "`firebase deploy`" overwrites any editing made over the security rules on the Firebase Console.

5.9. Points of Attention

5.9.1 Use of FirebaseUI and Firebase UI Auth

Described as a set of open-source UI libraries for Firebase, FirebaseUI is built over the Firebase SDK libraries and provides a simple way to connect UI components to Firebase databases, allowing frontend views to update in real-time as data changes on the backend. On the other hand, FirebaseUI Auth, built over of the Firebase Authentication SDK library, provides complete "*drop-in*" authentication patterns for mobile apps and websites that allow developers to customize easily (usually) complex UI workflows for different *sign in* and *sign up* methods, such as *email and password*, *phone numbers*, and the most popular federated identity providers like Google, Facebook, Twitter, GitHub, Apple, Microsoft, Yahoo, and others under industry standards like OAuth 2.0 and OpenID Connect.

5.9.2 Separating model and view in our access control handling solution

Although access control components (user accounts, roles, credentials, policies, etc.) should be defined on the model layer of an app, we have chosen to keep it on the view layer for simplicity. However, separating view code from model code in our access control handling solution is totally possible. For instance, Firebase Authentication allows to manage user accounts on the cloud, and it is feasible to add a "user" model class to instantiate "user" objects.

5.9.3 Extending to a role-based access control

Authentication can be extended with role based

5.9.4 Adding action handler for changing the accounts' primary email

Firebase also uses user authentication action handlers for changing the accounts' primary email, but we won't cover such due it is quite similar to the others covered in this tutorial.

5.10. Quiz Questions

5.10.1 Question 1: Access Control definition

Chose the correct options in the following statement:

Access Control consists of two components: authentication and authorization. And while _____ aims to prove user identity in _____, _____ aims to provide correct access data/resources/features to _____.

- ☐ Authentication.
- ☐ Authorization.
- ☐ An authenticated user.
- ☐ An authenticated app, device or network.

5.10.2 Question 2: Email Action Handlers

Which of the following statements apply to Email Action Handlers in Firestore? Select two:

- ☐ The email action handlers provided by Firebase Authentication are email address verification, password reset, and email address change.
- ☐ The email action handlers provided by Firebase Authentication are email address verification and password reset.
- ☐ The templates of email action handlers are good starting points for customizing users' confirmation actions through emails but are not flexible enough for customizing the whole email message within Firebase infrastructure. This design responds to the need of evading abusive behavior, therefore full customization may be achieved using your own

infrastructure.

- ☐ The templates of email action handlers provide complete customization of users' confirmation actions through emails, with no need for your own infrastructure. This design responds to the capabilities that Firestore provides to prevent abusive behavior, therefore full customization may be achieved using only Firebase infrastructure.

5.10.3 Question 3: Access Control Handling Solution

According to the Access Control Handling Solution proposed in the tutorial, which features in conjunction grant or restrict access to the database management operations. Select three:

- ☐ Firebase Authentication statuses.
- ☐ DOM operations on the main menu.
- ☐ Firebase Security Rules.
- ☐ Email Action Handlers.
- ☐ Cloud Firestore.

Appendix A. Appendix: "Hello World" Web App with Firebase and Firestore

5.0.1 Step 1. Setup Firebase Services and Assets

Set up a Firebase project, including a new Firebase web app and a Firestore database instance.

5.0.1 Step 2. Create a Record in a Firestore Database

Create a *collection* named "***greetings***" and then a few records/documents. Make sure the document has a `Document ID` (*1*, *2*, *3*...), and a *field* "***greeting***" (*string*) filled the *value* with "***Hello World***".

You might want to add more records with greetings in different languages.

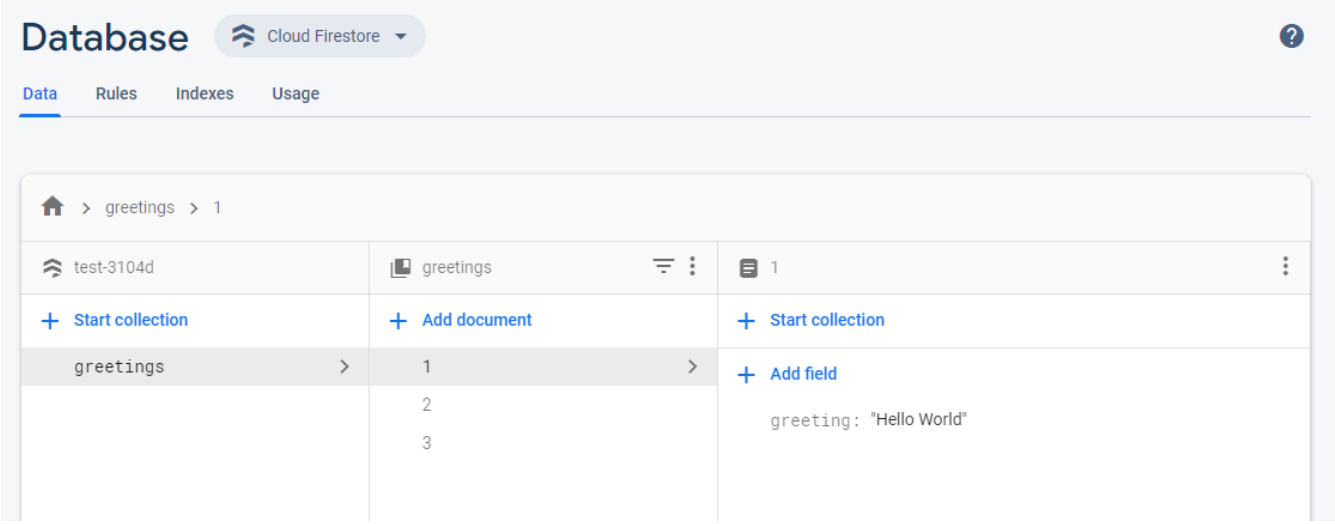


Figure A-1. *Creating a Firestore document for the 'Hello World' App*

This small project requires the creation of an *HTML* document (`index.html`) and a *JS* document (`scripts.mjs`).

5.0.1 Step 3. Define the HTML User Interface

In the <head> section of the *HTML* file invoke the JavaScript file as a module, and in the <body> section create an empty unordered list <ul>.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8">
  <script type="module" src="scripts.mjs"></script>
</head>
<body>
  <h1>Hello World App with Firebase + JS</h1>
  <ul></ul>
</body>
</html>
```

5.0.1 Step 4. Initialize Firebase and Firestore interfaces

The first statement in the JS file `scripts.mjs` imports required functions from the core Firebase SDK and Firestore Lite Web SDK libraries, and from the second statement, your web app is initialized using the Firebase project configuration page.

```
import { initializeApp } from "https://www.gstatic.com/firebasejs/9.X.X/firebase-app.js";
import { getFirestore, collection as fsColl, getDocs } from "https://www.gstatic.com/fireb

// TODO: Replace the following with your app's Firebase project configuration
// Set firebase App configuration values
const firebaseConfig = {
  apiKey: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  authDomain: "minimalapp-XXXX.firebaseapp.com",
  projectId: "minimalapp-XXXX",
  appId: "1:XXXXXXXXXXX:web:XXXXXXXXXXXXXXXXXXXXXX"
};
// Initialize Firebase interface
initializeApp( firebaseConfig);
// Initialize Firestore interface
const db = getFirestore();
```

5.0.1 Step 5. Add Behavior with JavaScript

Retrieve all greeting records/documents from the Firestore table "*greetings*", and show their greeting text in the list with the following procedure:

```
// retrieve and list all books records/documents
const ulEl = document.querySelector("body>ul");
const greetingsCollRef = fsColl( db, "greetings");
let greetingsQrySn = null;
try {  // get a query snapshot object
  greetingsQrySn = await getDocs( greetingsCollRef);
  // get the retrieved collection of documents from the snapshot query object
  const greetingDocSns = greetingsQrySn.docs;
  // convert Firestore documents to ordinary JS records/objects
```

```
  const greetingRecords = greetingDocSns.map( d => d.data());
  // create the greetings list items
  for (const g of greetingRecords) {
    const liEl = document.createElement("li");
    liEl.innerHTML += g.greeting;
    ulEl.appendChild( liEl);
  }
} catch( e) {
  console.error(`Error when retrieving greetings: ${e}`);
}
```

5.0.1 Step 5. Run the app

Run on your terminal `firebase serve`, and on your browser, go to `localhost:5000` for getting the list of greetings.

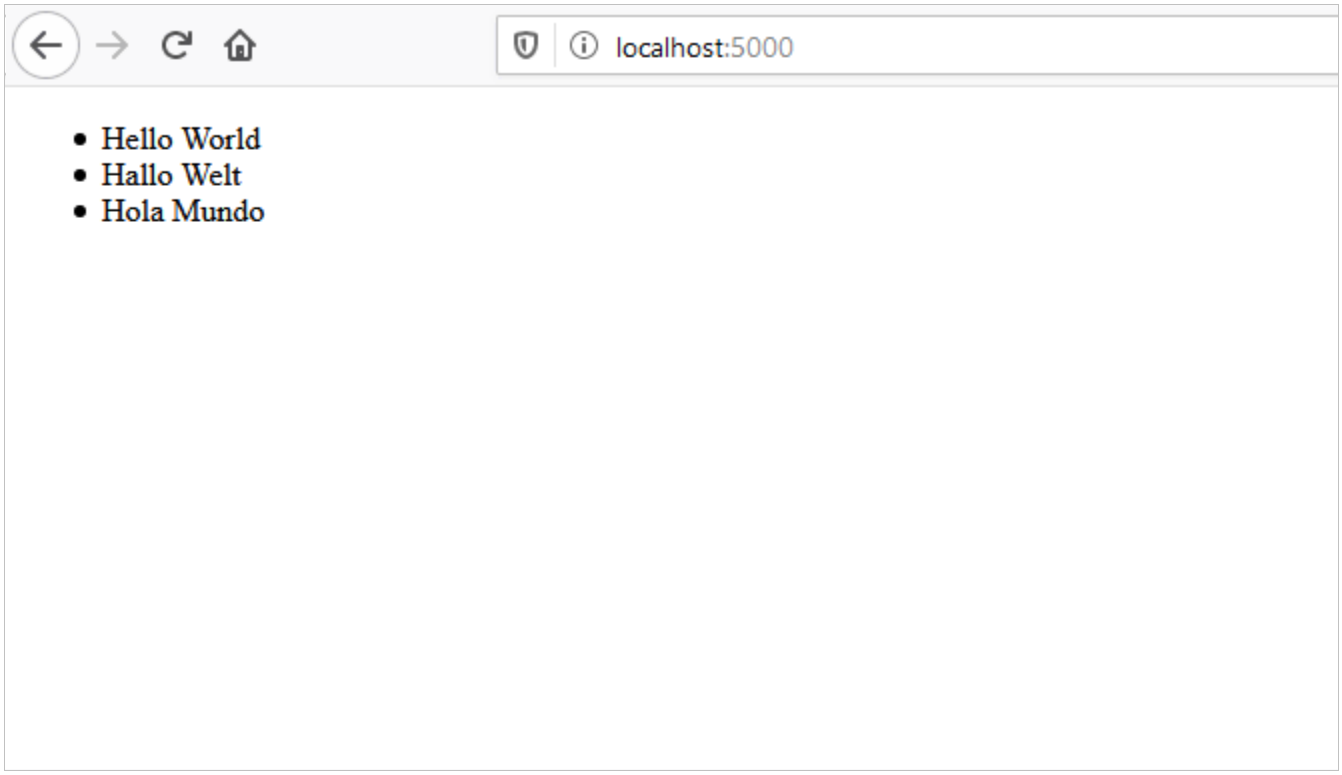


Figure A-2. *The "Hello World" Web App App*

This first simple example shows the basic elements of a JS+Firebase Web App. In the following sections we go into further details.

Glossary

C

CRUD

> CRUD is an acronym for ***C**reate*, ***R**ead/Retrieve*, ***U**pdate*, ***D**elete*, which denote the four basic data management operations to be performed by any software application.

Cascading Style Sheets

CSS is used for defining the presentation style of web pages by telling the browser how to render their HTML (or XML) contents: using which layout of content elements, which fonts and text styles, which colors, which backgrounds, and which animations. Normally, these settings are made in a separate CSS file that is associated with an HTML file via a special link element in the HTML's head element.

Cloud Firestore

Cloud Firestore, or just Firestore, is a *NoSQL Database as a Service*, schema-free, part of *Google Firebase* to develop mobile and web apps.

Collection

In Firestore database a *collection* is a database *table*.

D

Document

In Firestore database a *document* is a database *row*, or *record*.

Document ID

In Firestore database a *Document ID* is a unique record identifier within a database. Typically is the primary key.

Document Object Model

The DOM is an abstract API for retrieving and modifying nodes and elements of HTML or XML documents. All web programming languages have DOM bindings that realize the DOM.

DocumentSnapshot

In Firestore a *DocumentSnapshot* contains data read from a *document* in your database.

Domain Name System

The DNS translates user-friendly domain names to IP addresses that allow to locate a host computer on the Internet.

E

ECMAScript

A standard for JavaScript defined by the industry organization "Ecma International".

Extensible Markup Language

XML allows to mark up the structure of all kinds of documents, data files and messages in a machine-readable way. XML may also be human-readable, if the tag names used are self-explaining. 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.

F

Firebase

> Firebase is a cloud platform for software development, provided with a suite of services to create applications without dealing with the complexity of managing server hardware, code, security and architecture.

Firebase Authentication

> It is a user authentication solution for applications developed on Firebase, on both frontend (with UI libraries) and backend (SDKs). It supports multiple ways to sign in and sign up users, such as passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.

Firebase CLI

> The Firebase CLI is a set of tools for managing, viewing, and deploying to Firebase projects.

Firebase Security Rules

> Firestore Security rules is a tool that allows developers to manage authorization, giving control for the precise access we want for our users and data validation to improve business logic in your apps.

Firebase Local Emulator Suite

> The Firebase Local Emulator Suite is a set of tools that allow developers to build and test apps locally using Firestore, Realtime Database, Cloud Storage, Authentication, Cloud Functions, and Firebase Hosting.

H

Hypertext Markup Language

> 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 "XHTML5", provides a simpler and cleaner syntax compared to traditional HTML.

Hypertext Transfer Protocol

> HTTP is a stateless request/response protocol based on the Internet technologies TCP/IP and DNS, 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 back-end web application program from an HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services.

I

IANA

> IANA stands for *Internet Assigned Numbers Authority*, which is a subsidiary of ICANN responsible for names and numbers used by Internet protocols.

ICANN

> ICANN stands for *Internet Corporation of Assigned Names and Numbers*, which is an international nonprofit organization that maintains the domain name system.

IndexedDB

A JavaScript API for indexed data storage managed by browsers. Indexing allows high-performance searching. Like many SQL DBMS, IndexedDB supports database transactions.

I18N

A set of best practices that help to adapt products to any target language and culture. It deals with multiple character sets, units of measure, keyboard layouts, time and date formats, and text directions.

J

JSON

JSON stands for *JavaScript Object Notation*, which is a data-interchange format following the JavaScript syntax for object literals. Many programming languages support JSON as a light-weight alternative to XML.

M

MathML

An open standard for representing mathematical expressions, either in data interchange or for rendering them within webpages.

MIME

A MIME type (also called "media type" or "content type") is a keyword string sent along with a file for indicating its content type. For example, a sound file might be labeled `audio/ogg`, or an image file `image/png`.

***M**odel- **V**iew-**C**ontroller*

MVC is a general architecture metaphor emphasizing the principle of separation of concerns, mainly between the model and the view, and considering the model as the most fundamental part of an app. In MVC frameworks, "M", "V" and "C" are defined in different ways. Often the term “model” refers to the app’s data sources, while the “view” denotes the app’s code for the user interface, which is based on CSS-styled HTML forms and DOM events, and the “controller” typically denotes the (glue) code that is in charge of mediating between the *view* and the *model*.

N

Nested Object (maps)

In Firestore a Nested Object is called a *map* and host complex structure within a document.

O

Object Constraint Language

The OCL is a formal logic language for expressing integrity constraints, mainly in UML class models. It also allows defining derivation expressions for defining derived properties, and defining preconditions and postconditions for operations, in a class model.

Object-Oriented Programming

OOP is a programming paradigm based on the concepts of *objects* and *classes* instantiated by objects. Classes are like blueprints for objects: they define their *properties* and the *methods/functions* that can be applied to them. A higher-level characteristic of OOP is *inheritance* in class hierarchies: a subclass inherits the features (properties, methods and constraints) of its superclass.

Web Ontology Language

OWL is formal logic language for knowledge representation on the Web. It allows defining vocabularies (mainly classes with properties) and supports expressing many types of integrity constraints on them. OWL is the basis for performing automated inferences, such as checking the consistency of an OWL vocabulary. Vocabularies, or data models, defined in the form of UML class models can be converted to OWL vocabularies and then checked for consistency.

P

Portable Network Graphics

PNG is an open (non-proprietary) graphics file format that supports lossless data compression.

Polyfill

A polyfill is a piece of JavaScript code for emulating a standard JavaScript method in a browser, which does not support the method.

Q

QuerySnapshot

In Firestore *QuerySnapshot* contains none, one or multiple *DocumentSnapshot* objects that represente the results of a query.

R

References

In Firestore a reference is an internal object that points to a location in a database, and can be used later to retrieve or save data from or to Firestore. Creating a reference does not have any cost on the monthly billing.

Resource Description Framework

RDF is a W3C language for representing machine-readable propositional information on the Web.

S

Standard Generalized Markup Language

SGML is an ISO specification for defining markup languages. HTML4 has been defined with SGML. XML is a simplified successor of SGML. HTML5 is no longer SGML-based and has its own parsing rules.

Scalable Vector Graphics

SVG is a 2D vector image format based on XML. SVG can be styled with CSS and made interactive using JavaScript. HTML5 allows direct embedding of SVG content in an HTML document.

Slot

A slot is a name-value pair. In an object of an object-oriented program (for instance, in a Java object), a slot normally is a property-value pair. But in a JavaScript object, a slot may also consist of a method name and a method body or it may be a key-value pair of a map.

Subcollection

In Firestore, a subcollection is a collection within a document, allowing the parent document to create nested and hierarchical structure.

U

Unicode

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 π has the code 960. Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets.

XML is based on Unicode. Consequently, the Greek letter π (with code 960) can be inserted in an XML document as π using the XML entity syntax. The default encoding of Unicode characters in an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.

Uniform Resource Identifier

A URI is either a *Uniform Resource Locator (URL)* or a *Uniform Resource Name (URN)*.

Uniform Resource Locator

A URL is a resource name that contains a web address for locating the resource on the Web.

Unified Modeling Language

The UML is an industry standard that defines a set of modeling languages for making various kinds of models and diagrams in support of object-oriented problem analysis and software design. Its core languages are *Class Diagrams* for information/data modeling, and *Sequence Diagrams*, *Activity Diagrams* and *State Diagrams* (or *State Charts*) for process/behavior modeling.

Uniform Resource Name

A URN refers to a resource without specifying its location.

User Agent

A user agent is a front-end web client program such as a web browser.

W

WebM

WebM is an open (royalty-free) web video format supported by Google Chrome and Mozilla Firefox, but not by Microsoft Internet Explorer and Apple Safari.

Web Hypertext Application Technology Working Group

The *WHATWG* was established in 2004 by former employees of Apple, Mozilla, and Opera who have been unhappy with the slow progress of web technology standardization due to W3C's choice to focus on the standardization of XHTML2. Led by Ian Hickson, they developed HTML5 and related JavaScript APIs in competition and collaboration with the W3C.

World Wide Web

The WWW (or, simply, "the Web") is a huge client-server network based on HTTP, HTML and XML, where web browsers (and other 'user agents'), acting as HTTP clients, access web server programs, acting as HTTP servers.

World Wide Web Consortium

The *W3C* is an international organization in charge of developing and maintaining web standards.

X

XML HTTP Request

The XML HTTP Request (XHR) API allows a JavaScript program to exchange HTTP messages with back-end programs. It can be used for retrieveing/submitting information from/to a back-end program without submitting HTML forms. XHR-based approaches have been subsumed under the acronym "AJAX" in the past.

Resourses

- Firebase API Reference.
- Firebase Guides, step-by-step guides.
- Cloud Firestore, documentation on this NoSQL cloud database for web development.
- Node.js installation guide by Microsoft: Set up a NodeJS development environment under Windows.
- Official Firebase Youtube Channel: Firebase, videos with tutorials and step-by-step guides.
- Firebase Developers, an updated publication on Medium about Firebase, written for developers.