JS/Firebase Web App Tutorial Part 2: Adding Constraint Validation

Learn how to build a front-end web application with constraint validation 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.

Copyright © 2020-2022 Gerd Wagner and Juan-Francisco Reyes.

This tutorial article, along with any associated source code, is licensed under The Code Project Open License (CPOL), implying that the associated code is provided "as-is", can be modified to create derivative works, can be redistributed, and can be used in commercial applications, but the article must not be distributed or republished without the author's consent.

Published 2022-06-24.

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 2 of our series of six tutorials about model-based development of front-end web applications with plain JavaScript and Firebase. It shows how to build a single-class front-end web application with constraint validation using plain JavaScript and Firebase, and no third-party framework or library. While libraries and frameworks may help to increase productivity, they also create black-box dependencies and overhead, and they are not good for learning how to do it yourself.

A front-end web application can be provided by any web server, but it is executed on the user's computer device (smartphone, tablet or notebook), and not on the remote web server. Typically, but not necessarily, a front-end web application is a single-user application, which is not shared with other users.

The *minimal* JavaScript and Firebase app that we have discussed in the first part of this 6-part tutorial has been limited to support the minimum functionality of a data management app only. However, it did not take care of preventing users from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a JavaScript *model class*, and how to perform constraint validation both in the model/storage code of the app and in the user interface built with HTML5.

The simple form of a JavaScript and Firebase data management application presented in this tutorial takes care of only one object type ("books") for which it supports the four standard data management operations (**C**reate/**R**etrieve/**U**pdate/**D**elete). It extends the minimal app discussed in the Minimal App Tutorial by adding ***constraint validation*** (and some CSS styling), but it needs to be enhanced by adding further important parts of the app's overall functionality. The other parts of the tutorial are:

- Part 1: Building a **minimal**.
- 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. Integrity Constraints and Data Validation

1.1. Introduction

For detecting non-admissible and inconsistent data and for preventing such data to be added to an application's database, we need to define suitable ***integrity constraints*** that can be used by the application's ***data validation*** mechanisms for catching these cases of flawed data. Integrity constraints are logical conditions that must be satisfied by the data entered by a user and stored in the application's database.

For instance, if an application is managing data about persons including their birth dates and their death dates, then we must make sure that for any person record with a death date, this date is not before that person's birth date.

Since *integrity maintenance* is fundamental in database management, the *data definition language* part of the *relational database language SQL* supports the definition of integrity constraints in various forms. On the other hand, however, there is hardly any support for integrity constraints and data validation in common programming languages such as PHP, Java, C# or JavaScript. It is therefore important to take a systematic approach to constraint validation in web application engineering, like choosing an application development framework that provides sufficient support for it.

Unfortunately, many web application development frameworks do not provide sufficient support for defining integrity constraints and performing data validation. Integrity constraints should be defined in one (central) place in an app, and then be used for configuring the user interface and for validating data in different parts of the app, such as in the user interface and in the database. In terms of usability, the goals should be:

1. To prevent the user from entering invalid data in the user interface (UI) by limiting the input options, if possible.

2. To detect and reject invalid user input as early as possible by performing constraint validation in the UI for those UI widgets where invalid user input cannot be prevented by limiting the input options.

3. To prevent that invalid data pollutes the app's main memory state and persistent database state by performing constraint validation also in the model layer and in the database.

HTML5 provides support for validating user input in an HTML-forms-based user interface (UI). Here, the goal is to provide immediate feedback to the user whenever invalid data has been entered into a form field. This UI mechanism of ***responsive validation*** is an important feature of modern web applications. In traditional web applications, the back-end component validates the data and returns the validation results in the form of a set of error messages to the front-end. Only then, often several seconds later, and in the hard-to-digest form of a bulk message, does the user get the validation feedback.

1.2. Integrity Constraints

Integrity constraints (or simply *constraints*) are logical conditions on the data of an app. They may take many different forms. The most important type of constraints, ***property constraints***, define conditions on the admissible property values of an object. They are defined for an object type (or class) such that they apply to all objects of that type. We concentrate on the most important cases of property constraints:

String Length Constraints

require that the length of a string value for an attribute is less than a certain maximum number, or greater than a minimum number.

Mandatory Value Constraints

require that a property must have a value. For instance, a person must have a name, so the name attribute must not be empty.

Range Constraints

require that an attribute must have a value from the value space of the type that has been defined as its range. For instance, an integer attribute must not have the value "aaa".

Interval Constraints

require that the value of a numeric attribute must be in a specific interval.

Pattern Constraints

require that a string attribute's value must match a certain pattern defined by a regular expression.

Cardinality Constraints

apply to multi-valued properties, only, and require that the cardinality of a multi-valued property's value set is not less than a given minimum cardinality or not greater than a given maximum cardinality.

Uniqueness Constraints (also called 'Key Constraints')

require that a property's value is unique among all instances of the given object type.

Referential Integrity Constraints

require that the values of a reference property refer to an existing object in the range of the reference property.

Frozen Value Constraints

require that the value of a property must not be changed after it has been assigned initially.

The visual language of UML class diagrams supports defining integrity constraints either in a special way for special cases (like with predefined keywords), or, in the general case, with the help of *invariants*, which are conditions expressed either in plain English or in the *Object Constraint Language (OCL)* and shown in a special type of rectangle attached to the model element concerned. We use UML class diagrams for modeling constraints in *design models* that are independent of a specific programming language or technology platform.

1.2.1 OCL

The *Object Constraint Language* (OCL) was defined in 1997 as a formal logic language for expressing integrity constraints in UML version 1.1. Later, it was extended for allowing to define also (1) derivation expressions for defining derived properties, and (2) preconditions and postconditions for operations, in a class model.

UML class diagrams provide special support for expressing multiplicity (or cardinality) constraints. This type of constraint allows to specify a lower multiplicity (minimum cardinality) or an upper multiplicity (maximum cardinality), or both, for a property or an association end. In UML, this takes the form of a multiplicity expression `l..u` where the lower multiplicity `l` is a non-negative integer and the upper multiplicity `u` is either a positive integer not smaller than `l` or the special value `*` standing for *unbounded*. For showing property multiplicity (or cardinality) constrains in a class diagram, multiplicity expressions are enclosed in brackets and appended to the property name, as shown in the `Person` class rectangle below.

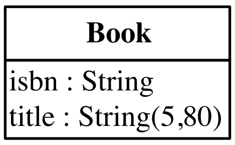
In the following sections, we discuss the different types of property constraints listed above in more detail. We also show how to express some of them in computational languages such as *UML* class diagrams, *SQL* table creation statements, *JavaScript* model class definitions, or the annotation-based languages *Java Bean Validation* annotations and *ASP.NET Data Annotations*.

Any systematic approach to constraint validation also requires to define a set of error (or 'exception') classes, including one for each of the standard property constraints listed above.

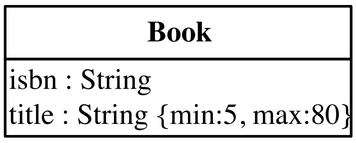
1.2.2 String Length Constraints

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

UML does not define any special way of expressing string length constraints in class diagrams. Of course, we always have the option to use an *invariant* for expressing any kind of constraint, but it seems preferable to use a simpler form of expressing these property constraints. One option is to append a maximum length, or both a minimum and a maximum length, in parenthesis to the datatype name, like so

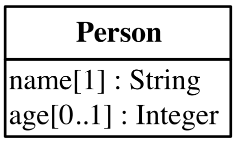


Another option is to use min/max constraint keywords in the property modifier list:



1.2.3 Mandatory Value Constraints

A *mandatory value constraint* requires that a property must have a value. This can be expressed in a UML class diagram with the help of a multiplicity constraint expression where the lower multiplicity is 1. For a single-valued property, this would result in the multiplicity expression `1..1`, or the simplified expression `1`, appended to the property name in brackets. For example, the following class diagram defines a mandatory value constraint for the property `name`:



Whenever a class rectangle does not show a multiplicity expression for a property, the property is mandatory (and single-valued), that is, the multiplicity expression `1` is the default for properties.

In an SQL table creation statement, a mandatory value constraint is expressed in a table column definition by appending the key phrase `NOT NULL` to the column definition as in the following example:

```
CREATE TABLE persons(
  name  VARCHAR(30) NOT NULL,
  age   INTEGER
)
```

According to this table definition, any row of the `persons` table must have a value in the column `name`, but not necessarily in the column `age`.

In JavaScript, we can code a mandatory value constraint by a class-level check function that tests if the provided argument evaluates to a value, as illustrated in the following example:

```
Person.checkName = function (n) {
  if (n === undefined) {
    return "A name must be provided!"; // constraint violation error message
  } else return "";  // no constraint violation
};
```

With Java Bean Validation, a mandatory property like `name` is annotated with `NotNull` in the following way:

```
@Entity
public class Person {
  @NotNull
  private String name;
  private int age;
}
```

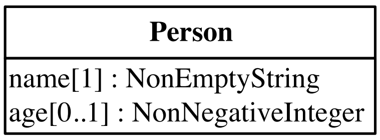
The equivalent ASP.NET Data Annotation is `Required` as shown in

```
public class Person {
  [Required]
  public string name { get; set; }
  public int age { get; set; }
}
```

1.2.4 Range Constraints

A range constraint requires that a property must have a value from the value space of the type that has been defined as its range. This is implicitly expressed by defining a type for a property as its range. For instance, the attribute `age` defined for the object type `Person` in the class diagram above has the range `Integer`, so it must not have a value like "aaa", which does not denote an integer. However, it may have values like -13 or 321, which also do not make sense as the age of a person. In a similar way, since its range is `String`, the attribute `name` may have the value "" (the empty string), which is a valid string that does not make sense as a name.

We can avoid allowing negative integers like -13 as age values, and the empty string as a name, by assigning more specific datatypes as range to these attributes, such as `NonNegativeInteger` to `age`, and `NonEmptyString` to `name`. Notice that such more specific datatypes are neither predefined in SQL nor in common programming languages, so we have to implement them either in the form of user-defined types, as supported in SQL-99 database management systems such as PostgreSQL, or by using suitable additional constraints such as *interval constraints*, which are discussed in the next section. In a UML class diagram, we can simply define `NonNegativeInteger` and `NonEmptyString` as custom datatypes and then use them in the definition of a property, as illustrated in the following diagram:



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

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

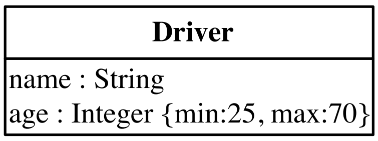
This check function detects and reports a constraint violation if the given value for the `name` property is not of type "string" or is an empty string.

In a Java EE web app, for declaring empty strings as non-admissible user input we must set the context parameter `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true` in the web deployment descriptor file `web.xml`.

In ASP.NET, empty strings are non-admissible by default.

1.2.5 Interval Constraints

An interval constraint requires that an attribute's value must be in a specific interval, which is specified by a minimum value or a maximum value, or both. Such a constraint can be defined for any attribute having an ordered type, but normally we define them only for numeric datatypes or calendar datatypes. For instance, we may want to define an interval constraint requiring that the `age` attribute value must be in the interval [25,70]. In a class diagram, we can define such a constraint by using the property modifiers `min` and `max`, as shown for the `age` attribute of the `Driver` class in the following diagram.



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

```
CREATE TABLE drivers(
  name  VARCHAR NOT NULL,
  age   INTEGER CHECK (age >= 25 AND age <= 70)
)
```

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

```
Driver.checkAge = function (a) {
  if (a < 25 || a > 70) {
    return "Age must be between 25 and 70!";
  } else return "";
};
```

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

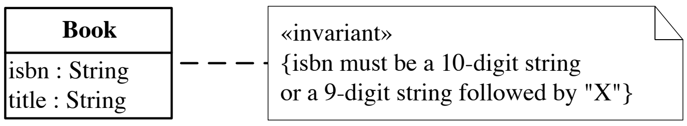
```
@Entity
public class Driver {
  @NotNull
  private String name;
  @Min(25) @Max(70)
  private int age;
}
```

The equivalent ASP.NET Data Annotation is `Range(25,70)` as shown in

```
public class Driver{
  [Required]
  public string name { get; set; }
  [Range(25,70)]
  public int age { get; set; }
}
```

1.2.6 Pattern Constraints

A pattern constraint requires that a string attribute's value must match a certain pattern, typically defined by a *regular expression*. For instance, for the object type `Book` we define an `isbn` attribute with the datatype `String` as its range and add a pattern constraint requiring that the `isbn` attribute value must be a 10-digit string or a 9-digit string followed by "X" to the `Book` class rectangle shown in the following diagram.


Book
isbn : String
title : String
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}

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

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

The ~ (tilde) symbol denotes the regular expression matching predicate and the regular expression `^\d{9}(\d|X)$` follows the syntax of the POSIX standard (see, e.g. the PostgreSQL documentation).

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

```
Person.checkIsbn = function (id) {
  if (!/\b\d{9}(\d|X)\b/.test( id)) {
    return "The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!";
  } else return "";
};
```

In Java EE Bean Validation, this pattern constraint for `isbn` is expressed with the annotation `Pattern` in the following way:

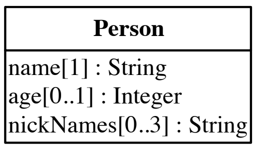
```
@Entity
public class Book {
@NotNull
  @Pattern(regexp="^\\(\d{9}(\d|X))$")
  private String isbn;
  @NotNull
  private String title;
}
```

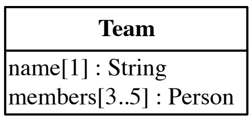
The equivalent ASP.NET Data Annotation is `RegularExpression` as shown in

```
public class Book{
  [Required]
  [RegularExpression(@"^(\d{9}(\d|X))$")]
  public string isbn { get; set; }
  public string title { get; set; }
}
```

1.2.7 Cardinality Constraints

A cardinality constraint requires that the cardinality of a multi-valued property's value set is not less than a given ***minimum cardinality*** or not greater than a given ***maximum cardinality***. In UML, cardinality constraints are called ***multiplicity constraints***, and minimum and maximum cardinalities are expressed with the lower bound and the upper bound of the multiplicity expression, as shown in the following diagram, which contains two examples of properties with cardinality constraints.





The attribute definition `nickNames[0..3]` in the class `Person` specifies a minimum cardinality of 0 and a maximum cardinality of 3, with the meaning that a person may have no nickname or at most 3 nicknames. The reference property definition `members[3..5]` in the class `Team` specifies a minimum cardinality of 3 and a maximum cardinality of 5, with the meaning that a team must have at least 3 and at most 5 members.

It's not obvious how cardinality constraints could be checked in an SQL database, as there is no explicit concept of cardinality constraints in SQL, and the generic form of constraint expressions in SQL, assertions, are not supported by available DBMSs. However, it seems that the best way to implement a minimum (or maximum) cardinality constraint is an on-delete (or on-insert) trigger that tests the number of rows with the same reference as the deleted (or inserted) row.

In JavaScript, we can code a cardinality constraint validation for a multi-valued property by testing the size of the property's value set, as illustrated in the following example:

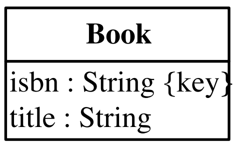
```
Person.checkNickNames = function (nickNames) {
  if (nickNames.length > 3) {
    return "There must be no more than 3 nicknames!";
  } else return "";
};
```

With Java Bean Validation annotations, we can specify

```
@Size( max=3)
List<String> nickNames
@Size( min=3, max=5)
List<Person> members
```

1.2.8 Uniqueness Constraints

A *uniqueness constraint* (or *key constraint*) requires that a property's value (or the value list of a list of properties in the case of a composite key constraint) is unique among all instances of the given object type. For instance, in a UML class diagram with the object type `Book` we can define the `isbn` attribute to be *unique*, or, in other words, a *key*, by appending the (user-defined) property modifier keyword `key` in curly braces to the attribute's definition in the `Book` class rectangle shown in the following diagram.



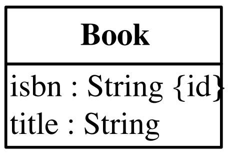
In an SQL table creation statement, a uniqueness constraint is expressed by appending the keyword `UNIQUE` to the column definition as in the following example:

```
CREATE TABLE books(
  isbn   VARCHAR(10) NOT NULL UNIQUE,
  title  VARCHAR(50) NOT NULL
)
```

In JavaScript, we can code this uniqueness constraint by a check function that tests if there is already a book with the given `isbn` value in the `books` table of the app's database.

1.2.9 Standard Identifiers (Primary Keys)

A unique attribute (or a composite key) can be declared to be the standard identifier for objects of a given type, if it is mandatory (or if all attributes of the composite key are mandatory). We can indicate this in a UML class diagram with the help of the property modifier `id` appended to the declaration of the attribute `isbn` as shown in the following diagram.



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

Often, practitioners do not recommended using a composite key as a standard ID, since composite identifiers are more difficult to handle and not always supported by tools. Whenever an object type does not have a key attribute, but only a composite key, it may therefore be preferable to add an artificial standard ID attribute (also called *surrogate ID*) to the object type. However, each additional surrogate ID has a price: it creates some cognitive and computational overhead. Consequently, in the case of a simple composite key, it may be preferable not to add a surrogate ID, but use the composite key as the standard ID.

There is also an argument against using any real attribute, such as the `isbn` attribute, for a standard ID. The argument points to the risk that the values even of natural ID attributes like `isbn` may have to be changed during the life time of a business object, and any such change would require an unmanageable effort to change also all corresponding ID references. However, the business semantics of natural ID attributes implies that they are frozen. Thus, the need of a value change can only occur in the case of a data input error. But such a case is normally detected early in the life time of the object concerned, and at this stage the change of all corresponding ID references is still manageable.

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

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

In object-oriented programming languages, like JavaScript and Java, we cannot code a standard ID declaration, because this would have to be part of the metadata of a class definition, and there is no support for such metadata. However, we should still check the implied mandatory value and uniqueness constraints.

1.2.10 Referential Integrity Constraints

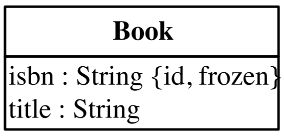
A referential integrity constraint requires that the values of a reference property refer to an object that exists in the population of the property's range class. Since we do not deal with reference properties in this chapter, we postpone the discussion of referential integrity constraints to Part 4 of our tutorial.

1.2.11 Frozen and Read-Only Value Constraints

A frozen value constraint defined for a property requires that the value of this property must not be changed after it has been assigned. This includes the special case of ***read-only value constraints*** on mandatory properties that are initialized at object creation time.

Typical examples of properties with a frozen value constraint are standard identifier attributes and event properties. In the case of events, the semantic principle that the past cannot be changed prohibits that the property values of events can be changed. In the case of a standard identifier attribute we may want to prevent users from changing the ID of an object since this requires that all references to this object using the old ID value are changed as well, which may be difficult to achieve (even though SQL provides special support for such ID changes by means of its `ON UPDATE CASCADE` clause for the change management of foreign keys).

The following diagram shows how to define a frozen value constraint for the `isbn` attribute:



In Java, a *read-only* value constraint can be enforced by declaring the property to be `final`. In JavaScript, a *read-only* property slot can be implemented as in the following example:

```
Object.defineProperty( obj, "teamSize", {value: 5, writable: false, enumerable: true})
```

where the property slot `obj.teamSize` is made unwritable. An entire object `obj` can be frozen with `Object.freeze( obj)`.

We can implement a frozen value constraint for a property in the property's setter method like so:

```
Book.prototype.setIsbn = function (i) {
  if (this.isbn === undefined) this.isbn = i;
  else console.log("Attempt to re-assign a frozen property!");
}
```

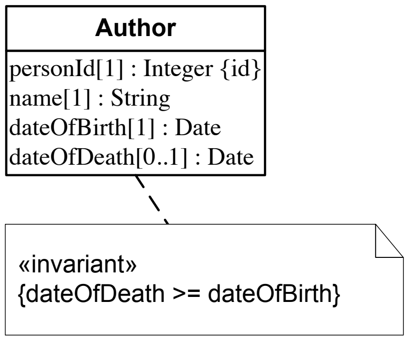
1.2.12 Beyond property constraints

So far, we have only discussed how to define and check *property constraints*. However, in certain cases there may be also integrity constraints that do not just depend on the value of a particular property, but rather on

1. the values of several properties of a particular object (object-level constraints),
2. the value of a property before and its value after a change attempt (dynamic constraints),
3. the set of all instances of a particular object type (type-level constraints),
4. the set of all instances of several object types.

In a class model, property constraints can be expressed within the property declaration line in a class rectangle (typically with keywords, such as `id`, `max`, etc.). For expressing more complex constraints, such as object-level or type-level constraints, we can attach an *invariant* declaration box to the class rectangle(s) concerned and express the constraint either in (unambiguous) English or in the *Object Constraint Language (OCL)*. A simple example of an object-level constraint expressed as an OCL invariant is shown in **Figure 1-1.** An example of an object-level constraint.

Figure 1-1. *An example of an object-level constraint*



A general approach for implementing *object-level constraint validation* consists of taking the following steps:

1. Choose a fixed name for an object-level constraint validation function, such as `validate`.
2. For any class that needs object-level constraint validation, define a `validate` function returning either a `ConstraintViolation` or a `NoConstraintViolation` object.
3. Call this function, if it exists, for the given model class,
    1. in the UI/view, on form submission;
    2. in the model class, before save, both in the `create` and in the `update` method.

1.3. Responsive Validation

This problem is well-known from classical web applications where the front-end component submits the user input data via HTML form submission to a back-end component running on a remote web server. Only this back-end component validates the data and returns the validation results in the form of a set of error messages to the front-end. Only then, often several seconds later, and in the hard-to-digest form of a bulk message, does the user get the validation feedback. This approach is no longer considered acceptable today. Rather, in a *responsive validation* approach, the user should get immediate validation feedback on each single data input. Technically, this can be achieved with the help of event handlers for the user interface events `input` or `change`.

Responsive validation requires a data validation mechanism in the user interface (UI), such as the HTML5 form validation API, which essentially provides new types of `input` fields (such as `number` or `date`), a set of new attributes for form control elements and new JS methods for the purpose of supporting responsive validation performed by the browser. Since using the new validation attributes (like `required`, `min`, `max` and `pattern`) implies defining constraints in the UI, they are not really useful in a general approach where constraints are only checked, but not defined, in the UI.

Consequently, we only use two methods of the HTML5 form validation API for validating constraints in the HTML-forms-based user interface of our app. The first of them, `setCustomValidity`, allows to mark a form field as either valid or invalid by assigning either an empty string or a non-empty (constraint violation) message string.

The second method, `checkValidity`, is invoked on a form before user input data is committed or saved (for instance with a form submission). It tests, if all fields have a valid value. For having the browser automatically displaying any constraint violation messages, we need to have a `submit` event, even if we don't really submit the form, but just use a `save` button.

See also this Mozilla tutorial for more about the HTML5 form validation API.

1.4. Constraint Validation in MVC Applications

Integrity constraints should be defined in the model classes of an MVC app since they are part of the business semantics of a model class (representing a business object type). However, a more difficult question is where to perform data validation? In the database? In the model classes? In the controller? Or in the user interface ("view")? Or in all of them?

A relational database management system (DBMS) performs data validation whenever there is an attempt to change data in the database, provided that all relevant integrity constraints have been defined in the database. This is essential since we want to avoid, under all circumstances, that invalid data enters the database. However, it requires that we somehow duplicate the code of each integrity constraint, because we want to have it also in the model class to which the constraint belongs.

Also, if the DBMS would be the only application component that validates the data, this would create a latency, and hence usability, problem in distributed applications because the user would not get immediate feedback on invalid input data. Consequently, data validation needs to start in the user interface (UI).

However, it is not sufficient to perform data validation in the UI. We also need to do it in the model classes, and in the database, for making sure that no flawed data enters the application's persistent data store. This creates the problem of how to maintain the constraint definitions in one place (the model), but use them in two or three other places (at least in the model classes and in the UI code, and possibly also in the database).We call this the ***multiple validation problem***. This problem can be solved in different ways. For instance:

1. Define the constraints in a declarative language (such as *Java Bean Validation Annotations* or *ASP.NET Data Annotations*) and generate the back-end/model and front-end/UI validation code both in a back-end application programming language such as Java or C#, and in JavaScript.

2. Keep your validation functions in the (PHP, Java, C# etc.) model classes on the back-end, and invoke them from the JavaScript UI code via XHR. This approach can only be used for specific validations, since it implies the penalty of an additional HTTP communication latency for each validation invoked in this way.

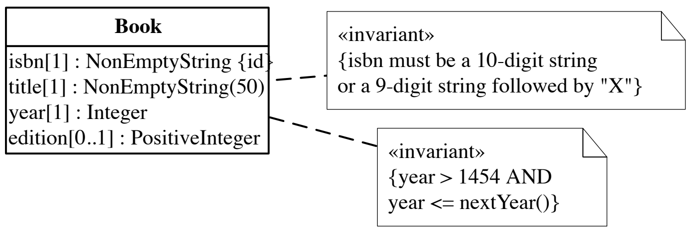
3. Use JavaScript as your back-end application programming language (such as with NodeJS), then you can code your validation functions in your JavaScript model classes on the back-end and execute them both before committing changes on the back-end and on user input and form submission in the UI on the front-end side.

The simplest, and most responsive, solution is the third one, using only JavaScript both for the back-end and front-end components of a web app.

1.5. Adding Constraints to a Design Model

We again consider the book data management problem that was considered in Part 1 of this tutorial. But now we also consider the ***data integrity rules*** (or 'business rules') that govern the management of book data. These integrity rules, or *constraints*, can be expressed in a UML class diagram as shown in **Figure 1-2.** A design model defining the object type `Book` with two invariants below.

Figure 1-2. *A design model defining the object type* `Book` *with two invariants*


Book
isbn[1] : NonEmptyString {id}
title[1] : NonEmptyString(50)
year[1] : Integer
edition[0..1] : PositiveInteger
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}
«invariant»
{year > 1454 AND
year <= nextYear()}

In this model, the following constraints have been expressed:

1. Due to the fact that the `isbn` attribute is declared to be the *standard identifier* of `Book`, it is ***mandatory*** and ***unique***.

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

3. The `title` attribute is ***mandatory***, as indicated by its multiplicity expression [1], and has a ***string length constraint*** requiring its values to have at most 50 characters.

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

Notice that the `edition` attribute is not mandatory, but *optional*, as indicated by its multiplicity expression [0..1]. In addition to the constraints described in this list, there are the implicit range constraints defined by assigning the datatype `NonEmptyString` as range to `isbn` and `title`, `Integer` to `year`, and `PositiveInteger` to `edition`. In our plain JavaScript approach, all these property constraints are coded in the model class within property-specific *check* functions.

The meaning of the design model can be illustrated by a sample data population respecting all constraints:

Table 1-1. Sample data for `Book`

ISBN	Title	Year	Edition
006251587X	Weaving the Web	2000	3
0465026567	Gödel, Escher, Bach	1999	2
0465030793	I Am A Strange Loop	2008	

1.6. Summary

1. Constraints are logical conditions on the data of an app. The simplest, and most important, types of constraints are property constraints and object-level constraints.

2. Constraints should be defined in the model classes of an MVC app, since they are part of their business semantics.

3. Constraints should be checked in various places of an MVC app: in the UI/view code, in model classes, and possibly in the database.

4. Software applications that include CRUD data management need to perform two kinds of bi-directional object-to-string type conversions:

    1. Between the model and the UI: converting model object property values to UI widget values, and, the other way around, converting input widget values to property values. Typically, widgets are form fields that have string values.

    2. Between the model and the datastore: converting model objects to storage data sets (called serialization), and, the other way around, converting storage data sets to model objects (called de-serialization). This involves converting property values to storage data values, and, the other way around, converting storage data values to property values. Typically, datastores are either JavaScript's local storage or IndexedDB, or SQL databases, and objects have to be mapped to some form of table rows. In the case of an SQL database, this is called "Object-Relational Mapping" (ORM), and in the case of Firebase the objects are mapped through JSON objects.

5. Do not perform any string-to-property-value conversion in the UI code. Rather, this is the business of the model code.

6. For being able to observe how an app works, or, if it does not work, where it fails, it is essential to log all critical application events, such as data retrieval, save and delete events, at least in the JavaScript console.

7. Responsive validation means that the user, while typing, gets immediate validation feedback on each input (keystroke), and when requesting to save the new data.

Chapter 2. Constraint Validation in a JS and Firebase Web App

The minimal web application built with Plain JavaScript and Firebase that we have discussed in the Minimal App Tutorial has been limited to support the minimum functionality of a data management app only. For instance, it did not take care of preventing the user from entering invalid data into the app's database. In this chapter, we show how to express integrity constraints in a JavaScript *model class*, and how to perform constraint validation both in the *model* part of the app and in the user interface built with HTML5. Additionally, we show how to express integrity constraints using the Firebase Security Rules language, as an additional validation layer on the back-end layer, our Firestore database instance.

We show how to perform responsive validation with the HTML5 form validation API. Since using the new HTML5 `input` field types and validation attributes (like `required`, `min`, `max` and `pattern`) implies defining constraints in the UI, they are not really useful in a best-practice approach where constraints are only checked, but not defined, in the UI.

Consequently, we will not use the new HTML5 features for defining constraints in the UI, but only use two methods of the HTML5 form validation API:

1. `setCustomValidity`, which allows to mark a form field as either valid or invalid by assigning either an empty string or a non-empty (constraint violation) message string;
2. `checkValidity`, which is invoked on a form before user input data is committed or saved (for instance with a form submission); it tests, if all fields have a valid value.

In the case of two special kinds of attributes, having *calendar dates* or *colors* as values, it is desirable to use the new UI widgets defined by HTML5 for picking a date or picking a color (with corresponding `input` field types).

2.1. New Issues

Compared to the JS/Firebase Minimal Web App discussed in the JS/Firebase Minimal App Tutorial we have to deal with a number of new issues:

1. In the *model* code we have to add for every property of a class
    1. a **check** function that can be invoked for validating the constraints defined for the property,
    2. *get/set* methods, such that the *check* function is invoked in the *set* method.
2. In the *user interface ("view")* code we have to take care of
    1. **responsive validation** on user input for providing immediate feedback to the user,
    2. validation on form submission for preventing the submission of flawed data to the model layer.

    As opposed to the JS/LocalStorage validation app, **ID constraints can no longer be checked on user input**, because their check now requires invoking an *asynchronous* database access operation, which may take too much time. Consequently, ID constraints can only be checked *on submit* (when the user submits the form data by clicking the *Save* button) and after that, in the model code, *before save*.

    To improve the break-down of the view code, we introduce a utility method (in `lib/util.js`) that fills an HTML `select` form control with `option` elements which content is retrieved from the collection "books". This method is used for setting up the user interface both in the *Update Book* and the *Delete Book* use cases.
3. As a namespace approach (for avoiding name conflicts), we will now use ES6 **modules**, instead of a global namespace object with subnamespace objects, like `pl = {m:{}, v:{}, c:{}}`.
4. For *object-to-storage* mapping, we use the Firestore feature of **converter** functions, transforming typed JS objects to JS/Firestore records, and vice versa.

Checking the constraints in the user interface (UI) on user input is important for providing immediate feedback to the user. But it is not safe enough to perform constraint validation only in the UI, because this could be circumvented in a distributed web application where the UI runs in the web browser of a front-end device while a back-end component manages the application's data on a remote web server. Consequently, we need multiple constraint validation, first in the UI *on input* (or *on change*) and *on form submission*, and subsequently in the model layer before saving/sending data to the persistent data store. In an application based on a DBMS we may also use a third round of validation by using the validation mechanisms of the DBMS. This redundancy is necessary when the application's database is shared with other apps.

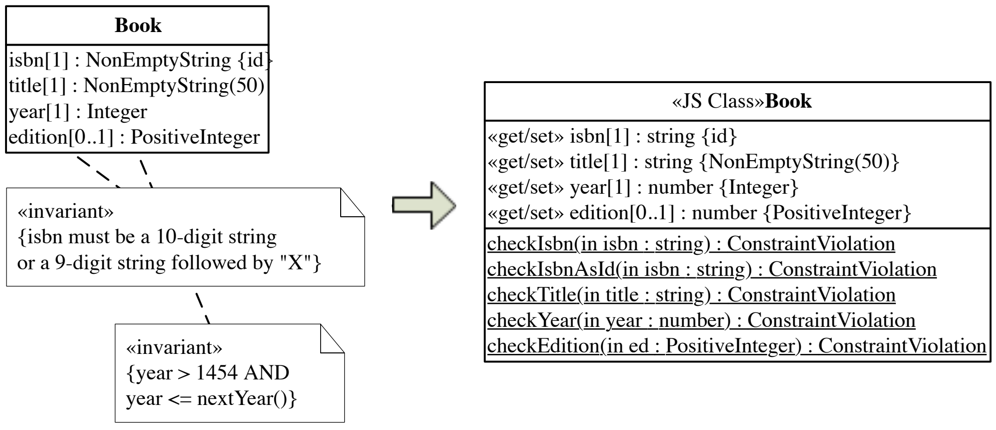
Our proposed solution to this *multiple validation problem* is keeping the constraint validation code in special *check* functions in the model classes, and invoking these functions on the UI, on user input, and on form submission; as well as in the Create and Update use cases of the model class via invoking the setters and getters.

Notice that *referential integrity* constraints (and other relationship constraints) may also be violated through a *delete* operation. Still, in our single-class example we don't have to consider this.

2.2. Make a JavaScript Class Model

Using the information design model shown in **Figure 2-1.** From an information design model to a JS class model as the starting point, we make a *JS class model*, essentially by decorating properties with a «get/set» stereotype, implying that they have implicit getters and setters, and by adding (class-level) check methods:

Figure 2-1. *From an information design model to a JS class model*


Book
isbn[1] : NonEmptyString {id}
title[1] : NonEmptyString(50)
year[1] : Integer
edition[0..1] : PositiveInteger
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}
«invariant»
{year > 1454 AND
year <= nextYear()}
«JS Class»Book
«get/set» isbn[1] : string {id}
«get/set» title[1] : string {NonEmptyString(50)}
«get/set» year[1] : number {Integer}
«get/set» edition[0..1] : number {PositiveInteger}
checkIsbn(in isbn : string) : ConstraintViolation
checkIsbnAsId(in isbn : string) : ConstraintViolation
checkTitle(in title : string) : ConstraintViolation
checkYear(in year : number) : ConstraintViolation
checkEdition(in ed : PositiveInteger) : ConstraintViolation

In the *JavaScript* class model we perform the following steps:

1. Create a ***check*** operation for each (non-derived) property in order to have a central place for implementing all the constraints that have been defined for a property in the design model. For a standard identifier attribute, such as `Book::isbn`, two check operations are needed:

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

   b. An extended check operation, such as `checkIsbnAsId`, for checking, in addition to the basic constraints, the *mandatory value* and *uniqueness* constraints that are required for a standard identifier attribute.

   The `checkIsbn` operation is invoked on user input for the `isbn` form field in the *create book* form, and also in the `setIsbn` method, used for testing if a value satisfies the syntactic constraints defined for an ISBN. Likewise, the `checkIsbnAsId` is used only when the create*book* form is submitted, invoked *asynchronously*.

2. Create **setter** and **getter** operations for each (non-derived) ***single-valued*** property. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.

The implicit getters and setters implied by the «get/set» stereotype are methods for getting and setting the value of a property *p* which allow keeping the simple syntax of getting its value with *v* = o.*p*, and setting it with o.*p* = *expr*. They require to define another, internal, property (like `_p`) for storing the value of *p* because the name "p" does not refer to a normal property, but rather to a pair of get/set methods.

The main reason for using setters is the need to always check constraints before setting a property.

2.3. Set up the Folder Structure and Add Some Library Files

The MVC folder structure of our validation app extends the structure of the minimal app with access control by adding three subfolders:

- a `css` folder containing CSS style files for styling the user interface pages of the app;
- a `lib` folder containing the generic code libraries `errorTypes.mjs` and `util.mjs`;
- a `test-data` folder containing a JSON file `books.json` with all `Book` data for generating test data.

Thus, we get the following folder structure:

```
2-ValidationApp
  public
    css
      main.css
      normalize.min.css
    js
      m
        Book.mjs
      v
        accessControl.mjs
        actionHandler.mjs
        createBook.mjs
        deleteBook.mjs
        resetPassword.mjs
        retrieveAndListAllBooks.mjs
        signIn.mjs
        signUp.mjs
        updateBook.mjs
      initFirebase.mjs
    lib
      errorTypes.mjs
      util.mjs
    test-data
      books.json
    404.html
    actionHandler.html
    apple-touch-icon.png
    createBook.html
    credits.html
    deleteBook.html
    favicon.ico
    favicon.svg
    forgotPassword.html
    google-touch-icon.html
    index.html
    manifest.json
    mask-icon.svg
    retrieveAndListAllBooks.html
    signIn.html
    signUp.html
    updateBook.html
```

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

2.3.1 Provide general utility functions and JavaScript fixes in library files

We add two module files to the `lib` folder:

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

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

2.3.2 Provide an external test data source

Unlike the minimal app which only generates a small data sample for data purposes, this app generates 50 book records, loaded from the JSON file `books.json`, located in the folder `test-data`.

2.3.3 Create a start page

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

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

The first CSS file (normalize.css) is a widely used collection of style normalization rules making browsers render all HTML elements more consistently. The second file (main.css) contains the specific styles of the app's user interface (UI) pages.

Since the app's start page does not provide much UI functionality, but only a few navigation links and two buttons, only a few lines of code are needed for setting up the buttons' event listeners. This is taken care of in an embedded `script` element of type `module`:

```
<script type="module">
  import { handleAuthentication } from "./js/v/accessControl.mjs";
  import Book from "./js/m/Book.mjs";

  handleAuthentication();
  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>
```

Notice how the Book class is loaded by importing the module `Book.mjs` from the `js/m` folder, as well the `handleAuthentication()` procedure from the module `handleAuthentication.mjs`, later invoked to handle access control on the start page.

2.4. Initialize Firebase

In the second step, as well as we did in the minimal app with access control, we initialize our Firestore instance. For which we create 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 Firebase SDK libraries:

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

We initialize the Firebase App first storing it as an object in a variable named "`app`" using the values taken from the web app's Firebase project configuration page, evaluating whether there is already an initialized Firebase app instance using the `getApps()` function, and, if not, with the `initializeApp()` function. In the case there is already an initialized Firebase app instance, we use the getApp() function. We use the "app" object by being given as parameter to the `getAuth()` function in order to create the authentication instance in the "`auth`" object:

```
// TODO: Replace the following with your web app's Firebase project configuration
const config = {
  apiKey: "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  authDomain: "minimalapp-XXXX.firebaseapp.com",
  projectId: "minimalapp-XXXX",
  appId: "1:XXXXXXXXXXXX:web:XXXXXXXXXXXXXXXXXXXXXXX"
};

// Initialize a Firebase App object only if not already initialized
const app = (!getApps().length) ? initializeApp( config ) : getApp();
// Initialize Firebase Authentication
const auth = getAuth( app);
...
```

Once the Firebase App instance has been initialized, we can initialize Cloud Firestore using the `getFirestore()` function to create the ***fsDb*** object, which 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 our app.

```
export { fsDb };
```

2.5. Write the Model Code

The JavaScript class model shown on the right hand side in **Figure 2-1.** From an information design model to a JS class model can be coded step by step to get the code of the model layer of our JavaScript and Firebase web app. These steps are summarized in the following section.

2.5.1 Summary

We want to check if a new property value satisfies all constraints of a property whenever the value of a property is set. A best practice for making sure that new values are validated before assigned is using a setter method for assigning a property, and invoking a check function in the setter. We use JavaScript's implicit getters and setters in combination with an internal property name (like `_isbn`).

2.5.2 Code the model class as a constructor function

The model class `Book` is coded as a JS class with a constructor function having a single constructor parameter in the form of a record using ES6 function parameter destructuring:

```
class Book {
  // using a single record parameter with ES6 function parameter destructuring
  constructor ({isbn, title, year, edition}) {
    // assign properties by invoking implicit setters
    this.isbn = isbn;  // string
    this.title = title;  // string
    this.year = year;  // number (int)
    // edition is an optional property
    if (edition) this.edition = edition;  // optional number (int)
  };
  ...
}
```

In the constructor body, we assign the values of the constructor parameters to the corresponding class properties. It is helpful to indicate the range of a property in a comment. This requires to map the platform-independent datatypes of the information design model to the corresponding implicit JavaScript datatypes according to the following table.

Table 2-1. Datatype mapping

Platform-independent datatype	JavaScript datatype	SQL
String	string	CHAR(*n*) or VARCHAR(*n*)
Integer	number (int)	INTEGER
Decimal	number (float)	REAL, DOUBLE PRECISION or DECIMAL(*p*,*s*)
Boolean	boolean	BOOLEAN
Date	Date	DATE

Since the setters may throw constraint violation errors, the model class `Book` constructor, and any setter, should be called in a try-catch block where the catch clause takes care of processing errors (at least logging suitable error messages).

2.5.3 Code the property checks

Code the property check functions in the form of class-level ('static') methods. In JavaScript, this means to define them as method slots of the constructor, as in `checkIsbn` (recall that a constructor is a JS object, since in JavaScript, functions are objects, and as an object, it can have slots).

Take care that all constraints of a property as specified in the class model are properly coded in its check function. This concerns, in particular, the *mandatory value* and *uniqueness* constraints implied by the *standard identifier* declaration (with `{id}`), and the *mandatory value* constraints for all properties with multiplicity 1, which is the default when no multiplicity is shown. If any constraint is violated, an error object instantiating one of the error classes listed above in and defined in the file `errorTypes.mjs` is returned.

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

```
static checkIsbn( isbn) {
  if (!isbn) return new NoConstraintViolation();
  else if (typeof( isbn) !== "string" || isbn.trim() === "") {
    return new RangeConstraintViolation("The ISBN must be a non-empty string!");
  } else if (!(/\b\d{9}(\d|X)\b/.test( isbn))) {
    return new PatternConstraintViolation(
      'The ISBN must be a 10-digit string or a 9-digit string followed by "X"!');
  } else {
    return new NoConstraintViolation();
  }
}
```

Notice that, since `isbn` is the standard identifier attribute of `Book`, we only check the syntactic constraints in `checkIsbn`, but we check the *mandatory value* and *uniqueness* constraints in `checkIsbnAsId`, which itself first invokes `checkIsbn` and then queries the Firestore database to check if the ISBN already exists:

```
static async checkIsbnAsId( isbn) {
  let validationResult = Book.checkIsbn( isbn);
  if ((validationResult instanceof NoConstraintViolation)) {
    if (!isbn) {
      validationResult = new MandatoryValueConstraintViolation(
        "A value for the ISBN must be provided!");
    } else {
      const bookDocSn = await getDoc( fsDoc( fsDb, "books", isbn));
      if (bookDocSn.exists) {
        validationResult = new UniquenessConstraintViolation(
           "There is already a book record with this ISBN!");
      } else {
        validationResult = new NoConstraintViolation();
      }
    }
  }
  return validationResult;
}
```

Notice that all check functions should be able to deal both with proper data values (that are from the attribute's range type) and also with string values that are supposed to represent proper data values, but have not yet been converted to the attribute's range type. We take this approach for avoiding datatype conversions in the user interface ("view") code. Notice that all data entered by a user in an HTML form field is of type String and must be converted before its validity can be checked and it can be assigned to the corresponding property. It is preferable to perform these type conversions in the model code, and not in the user interface code.

For instance, in our example app, we have the integer-valued attribute `year`. When the user has entered a value for this attribute in a corresponding form field, in the *Create* or *Update* user interface, the form field holds a string value. This value is first passed to `checkYear` method and then to the `Book.add` or `Book.update` methods, where this string value is converted to an integer and assigned to the `year` attribute.

2.5.4 Code the property getters and setters

For each property, we define implicit getters and setters using the predefined JS keywords `get` and `set`:

Code the setter operations as (instance-level) methods. In the setter, the corresponding check function is invoked and the property is only set, if the check does not detect any constraint violation. Otherwise, the *constraint violation* error object returned by the check function is thrown. For instance, the `get isbn` and `set isbn` operations are coded in the following way:

```
get isbn() {
  return this._isbn;
}
set isbn(n) {
  const validationResult = Book.checkIsbn(n);
  if (validationResult instanceof NoConstraintViolation) {
    this._isbn = n;
  } else {
    throw validationResult;
  }
}
```

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

2.5.5 Object-to-Storage Mapping

Firestore supports an Object-to-Storage mapping from typed objects to JS/Firestore records and, vice versa, from JS/Firestore records to typed objects, by defining two conversion functions:

- `toFirestore`, used every time a write operation is invoked, converting a typed JS object (instantiating a model class) into a JS record.
- `fromFirestore`, used every time a read operation is invoked, converting a JS record into a typed JS object instantiating a specific model class.

These two functions have to be assembled in an object like `Book.converter`:

```
Book.converter = {
  toFirestore: function (book) {
    const data = {
      isbn: book.isbn,
      title: book.title,
      year: parseInt( book.year)
    };
    if (book.edition) data.edition = parseInt( book.edition);
    return data;
  },
  fromFirestore: function (snapshot, options) {
    const data = snapshot.data( options);
    return new Book( data);
  }
};
```

Such a converter can be applied to a Firestore collection reference with the help of the built-in method `withConverter`:

```
const bookDocRef = fsDoc( fsDb, "books", book.isbn).withConverter( Book.converter);
```

Whenever a Firestore operation is performed, the data converter function will act accordingly, distinguishing between write operations (`toFirestore`) and read operations (`fromFirestore`).

2.5.6 Data Management Operations

In addition to defining the model class in the form of a constructor function with property definitions, checks, setters and getters, we also need to define the following data management operations as class-level methods of the model class:

- `Book.add` for creating a new Book instance and adding it to the collection of all Book instances.
- `Book.update` for updating an existing Book instance.
- `Book.destroy` for deleting a Book instance from Firestore.
- `Book.retrieve` for retrieving a Book instance.
- `Book.retrieveAll` for retrieving all Book instances from Firestore.

- `Book.generateTestData` for creating sample book records to be used as test data.
- `Book.clearData` for clearing the book data store.

Notice that for the change operations `add` (create), we need to implement an asynchronous invocation of checkIsbnAsId, working as an all-or-nothing policy, whenever there is a constraint violation for the isbn property. When a constraint violation is detected in this checker, the object creation attempt fails, and instead a constraint violation error message is created. Otherwise, the new book object is added to *Firestore* and a status message is created, as shown in the following program listing:

```
Book.add = async function (slots) {
  let book = null;
  try {
    // validate data by creating Book instance
    book = new Book( slots);
    // invoke asynchronous ID/uniqueness check
    let validationResult = await Book.checkIsbnAsId( book.isbn);
    if (!validationResult instanceof NoConstraintViolation) throw validationResult;
  } catch (e) {
    console.error(`${e.constructor.name}: ${e.message}`);
    book = null;
  }
  if (book) {
    try {
      const bookDocRef = fsDoc( fsDb, "books", book.isbn).withConverter( Book.converter);
      await setDoc( bookDocRef, book);
      console.log(`Book record "${book.isbn}" created!`);
    } catch (e) {
      console.error(`${e.constructor.name}: ${e.message} + ${e}`);
    }
  }
};
```

When an object of a model class is to be updated, we first retrieve a book record from Firestore to use it as a reference to check what properties has been updated, and only if a property has changed the update is performed on the database, and we report this in a status log.

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

When a constraint violation is detected in one of the checkers invoked in `Book.update`, the object update attempt fails, and instead the error message of the constraint violation object thrown by the checker and caught in the `update` method is shown, and the record is not updated. Otherwise, a status message is created, as shown in the following program listing:

```
Book.update = async function (slots) {
  let noConstraintViolated = true,
    validationResult = null,
    bookBeforeUpdate = null;
  const bookDocRef = fsDoc( fsDb, "books", slots.isbn).withConverter( Book.converter),
    updatedSlots = {};
  try {
    // retrieve up-to-date book record
    const bookDocSn = await getDoc( bookDocRef);
    bookBeforeUpdate = bookDocSn.data();
  } catch (e) {
    console.error(`${e.constructor.name}: ${e.message}`);
  }
  try {
    if (bookBeforeUpdate.title !== slots.title) {
      validationResult = Book.checkTitle( slots.title);
      if (validationResult instanceof NoConstraintViolation) updatedSlots.title = slots.ti
      else throw validationResult;
    }
    if (bookBeforeUpdate.year !== parseInt( slots.year)) {
      validationResult = Book.checkYear( slots.year);
      if (validationResult instanceof NoConstraintViolation) updatedSlots.year = parseInt(
      else throw validationResult;
    }
    if (slots.edition && bookBeforeUpdate.edition !== parseInt( slots.edition)) {
      // slots.edition has a non-empty value that is different from the old value
      validationResult = Book.checkEdition( slots.edition);
      if (validationResult instanceof NoConstraintViolation) updatedSlots.edition = parseI
      else throw validationResult;
    } else if (!slots.edition && bookBeforeUpdate.edition) {
      // slots.edition has an empty value while the old value was not empty
      updatedSlots.edition = await updateDoc( bookDocRef, {edition: deleteField()});
    }
  } catch (e) {
    noConstraintViolated = false;
    console.error(`${e.constructor.name}: ${e.message}`);
  }
  if (noConstraintViolated) {
    const updatedProperties = Object.keys(updatedSlots);
    if (updatedProperties.length) {
      await updateDoc(bookDocRef, updatedSlots);
      console.log(`Property(ies) "${updatedProperties.toString()}" modified for book recor
    } else {
      console.log(`No property value changed for book record "${slots.isbn}"!`);
    }
  }
};
```

Notice that optional properties, like `edition`, need to be treated in a special way. If the user doesn't enter any value for them in a *Create* or *Update* user interface, the form field's value is the empty string `""`. In the case of an optional property, this means that the property is not assigned a value in the *add* use case, or that it is *unset* if it has had a value in the *update* use case. This is different from the case of a mandatory property, where the empty string value obtained from an empty form field may or may not be an admissible value. Notice the use of the Firestore method `deleteField()` used in combination with `updateDoc`, for deleting an specific field during the *Update* operation.

When an object of a model class is to be deleted, the destroy procedure is invoked.

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

Whenever an individual object of a model class is to be retrieved, we can use this simplified version of the function

Book.retrieve:

```
Book.retrieve = async function( isbn) {
  try {
    const bookRec = (await getDoc( fsDoc(fsDb, "books", isbn)
      .withConverter( Book.converter))).data();
    console.log(`Book record "${bookRec.isbn}" retrieved.`);
    return bookRec;
  } catch (e) {
    console.error(`Error retrieving book record: ${e}`);
  }
};
```

And when all records in a Firestore collection are to be retrieved, we can again use a simplified version of function Book.retrieveAll. However this time it is introduced the option to order the resulting records, using the `orderBy()` method. Later we will see on the view code the implementation of a HTML select element to trigger this function with the parameter order, being for instance isbn, title and year.

```
Book.retrieveAll = async function (order) {
  if (!order) order = "isbn";
  const booksCollRef = fsColl( fsDb, "books"),
    q = fsQuery( booksCollRef, orderBy( order));
  try {
    const bookRecs = (await getDocs( q.withConverter( Book.converter))).docs.map( d => d.d
    console.log(`${bookRecs.length} book records retrieved ${order ? "ordered by " + order
    return bookRecs;
  } catch (e) {
    console.error(`Error retrieving book records: ${e}`);
  }
};
```

For testing the app, 50 book records can be generated by, first retrieving data stored in JSON format in an external file `books.json`, using the *Fetch API*. Then the retrieved data is parsed as *JSON* using the method `json()`. Finally, every record is stored to Firestore using JS promises in parallel, using the `Promise.all` method.

```
Book.generateTestData = async function() {
  try {
    console.log("Generating test data...");
    const response = await fetch( "../../test-data/books.json");
    const bookRecs = await response.json();
    await Promise.all( bookRecs.map( d => Book.add( d)));
    console.log(`${bookRecs.length} books saved.`);
  } catch (e) {
    console.error(`${e.constructor.name}: ${e.message}`);
  }
};
```

Inversely, whenever we need to empty the Book collection, we use function `Book.clearData`. First all the book records in the collection books are retrieved from Firestore, and then iteratively and asynchronously we use the `Promise.all` method to delete every record, this time reusing the `Book.destroy()` function.

```
Book.clearData = async function() {
  if (confirm("Do you really want to delete all book records?")) {
    try {
      console.log("Clearing test data...");
      const booksCollRef = fsColl( fsDb, "books");
      const booksQrySn = (await getDocs( booksCollRef));
      await Promise.all( booksQrySn.docs.map( d => Book.destroy( d.id)))
      console.log(`${booksQrySn.docs.length} books deleted.`);
    } catch (e) {
      console.error(`${e.constructor.name}: ${e.message}`);
    }
  }
};
```

2.6. Write the View Code

The user interface (UI) consists of a start page `index.html` that allows the user choosing one of the data management operations by navigating to the corresponding UI page such as `retrieveAndListAllBooks.html` or `createBook.html` in the app folder. The start page `index.html` has been discussed in . It sets up two buttons for clearing the app's database by invoking the procedure `Book.clearData()` and for creating sample data by invoking the procedure `Book.generateTestData()` from the buttons' `click` event listeners.

Each data management UI page loads the same basic CSS and JavaScript files like the start page `index.html` discussed above. In addition, it loads a use-case-specific view code file `js/v/`*useCase*`.mjs`.

For setting up the user interfaces of the data management use cases, we have to distinguish the case of "Retrieve/List All" from the other ones (Create, Update, Delete). While the latter ones require using an HTML form and attaching event handlers to form controls, in the case of "Retrieve/List All" we only have to render a table displaying all books, as in the case of the Minimal App discussed in *Part 1* of this tutorial.

For the *Create*, *Update* and *Delete* use cases, we need to:

1. import the `Book` class and the `showProgressBar` and `showProgressBar` utils,

```
import Book from "../m/Book.mjs";
import { showProgressBar, hideProgressBar } from "../../lib/util.mjs";
```

2. define variables for accessing the UI form element and the save/delete button,

```
const formEl = document.forms["Book"],
  createButton = formEl["commit"],
  progressEl = document.querySelector("progress");
```

Then a group of event listeners are added for responsive validation on form field input events:

```
// add event listeners for responsive validation
formEl["isbn"].addEventListener("input", function () {
  // do not yet check the ID constraint, only before commit
  formEl["isbn"].setCustomValidity( Book.checkIsbn( formEl["isbn"].value).message);
});
formEl["title"].addEventListener("input", function () {
  formEl["title"].setCustomValidity( Book.checkTitle( formEl["title"].value).message);
});
formEl["year"].addEventListener("input", function () {
  formEl["year"].setCustomValidity( Book.checkYear( formEl["year"].value).message);
});
formEl["edition"].addEventListener("input", function () {
  formEl["edition"].setCustomValidity( Book.checkEdition( formEl["edition"].value).message
});
```

Notice that for each input field we add a listener for `input` events, such that on any user input a validation check is performed because `input` events are created by user input actions such as typing. We use the predefined function `setCustomValidity` from the HTML5 form validation API for having our property check functions invoked on the current value of the form field and returning an error message in the case of a constraint violation. So, whenever the string represented by the expression `(Book.checkIsbn( slots.isbn.value)).message` is empty, everything is fine. Otherwise, if it represents an error message, the browser indicates the constraint violation to the user by rendering a red outline for the form field concerned (due to our CSS rule for the `:invalid` pseudo class).

While the validation on user input enhances the usability of the UI by providing immediate feedback to the user, validation on form data submission is even more important for catching invalid data. Therefore, multiple property checks are invoked again with the help of `setCustomValidity`, as shown in the following program listing:

```
createButton.addEventListener("click", async function () {
  const formEl = document.forms["Book"],
    slots = {
    isbn: formEl["isbn"].value,
    title: formEl["title"].value,
    year: formEl["year"].value
  };
  // check constraints and set error messages
  showProgressBar( progressEl);
  formEl["isbn"].setCustomValidity(( await Book.checkIsbnAsId( slots.isbn)).message);
  formEl["title"].setCustomValidity( Book.checkTitle( slots.title).message);
  formEl["year"].setCustomValidity( Book.checkYear( slots.year).message);
  if (formEl["edition"].value) {
    slots.edition = formEl["edition"].value;
    formEl["edition"].setCustomValidity( Book.checkEdition( slots.edition).message);
  }
  if (formEl.checkValidity()) {
    await Book.add( slots);
    formEl.reset();
  }
  hideProgressBar( progressEl);
});
// neutralize the submit event
formEl.addEventListener("submit", function (e) {
  e.preventDefault();
});
```

By invoking `checkValidity()` on the form element, we make sure that the form data is only saved (by `Book.add`), if there is no constraint violation. After the property checks are executed on an invalid form, the browser takes control and tests if the predefined property `validity` has an error flag for any form field. In our approach, since we use `setCustomValidity`, the `validity.customError` would be true. If this is the case, the custom constraint violation message will be displayed (in a bubble) and the `submit` event will be suppressed.

The `showProgressBar` function is used to show or hide a HTML `<progress>` element while the invoked asynchronous function is happening. This visual cue is used to give feedback about the interaction on the submit event, communicating that our app is in control now processing the order. Its counterpart, `hideProgressBar`, hides the progress element after the operation finishes.

In the UI of the use case *Update*, which is handled in `v/updateBook.mjs`, we do not have an `input`, but rather an `output` field for the standard identifier attribute `isbn`, since it is not supposed to be modifiable. Consequently, we don't need to validate any user input for it.

The implementation on the view code start by retrieving all book records from the Firestore collection, and then initializing the form element with its values, the save button and the book selector, which is a `select` element inside the form.

```
const bookRecords = await Book.retrieveAll();
...
const formEl = document.forms["Book"],
  saveButton = formEl.commit,
  selectBookEl = formEl.selectBook;
```

Additionally, we need to set up a selection list (in the form of an HTML `select` element) with the retrieved book records, allowing the user to select a book record in the first step, before its data can be modified. This requires to add a `change` event listener on the `select` element such that the fields of the UI can be filled with the data of the selected object. Finally, all custom validation errors messages are deleted with setCustomValidity(""), in case they were set before.

```
fillSelectWithOptions(bookRecords, selectBookEl, "isbn", "title");
// when a book is selected, populate the form with its data
selectBookEl.addEventListener("change", async function () {
  const bookKey = selectBookEl.value;
  if (bookKey) {
    // retrieve up-to-date book record
    const bookRecord = await Book.retrieve( bookKey);
    for (const a of ["isbn", "title", "year", "edition"]) {
      formEl[a].value = bookRecord[a] !== undefined ? bookRecord[a] : "";
      // delete custom validation error message which may have been set before
      formEl[a].setCustomValidity("");
    }
  } else {
    formEl.reset();
  }
});
```

There is no need to set up responsive validation for the standard identifier attribute `isbn`, but for all other form fields, as shown above for the *Create* use case.

```
formEl["title"].addEventListener("input", function () {
  formEl["title"].setCustomValidity(
    Book.checkTitle( slots.title).message);
});
formEl["year"].addEventListener("input", function () {
  formEl["year"].setCustomValidity(
    Book.checkYear( slots.year).message);
});
if (formEl["edition"].value) {
  formEl["edition"].addEventListener("input", function () {
    formEl["edition"].setCustomValidity(
      Book.checkEdition(slots.edition).message);
  });
}
```

The last step in the UI is handling the submit event, when the user clicks on the "Update" button, which consists in five steps:

1. initialize UI elements and the book key value (`bookIdRef`),
2. prepare the slots object,
3. invoke the constraint violations checkers and set error messages on the UI, and
4. Invoke the update function with the slots object.

5. Neutralize the submit event.

```
updateButton.addEventListener("click", function () {
  const formEl = document.forms["Book"],
    selectBookEl = formEl["selectBook"],
    bookIdRef = selectBookEl.value;
  if (!bookIdRef) return;
  const slots = {
    isbn: formEl["isbn"].value,
    title: formEl["title"].value,
    year: formEl["year"].value
  };
  // set error messages in case of constraint violations
  formEl["title"].addEventListener("input", function () {
    formEl["title"].setCustomValidity(
      Book.checkTitle( slots.title).message);
  });
  formEl["year"].addEventListener("input", function () {
    formEl["year"].setCustomValidity(
      Book.checkYear( slots.year).message);
  });
  if (formEl["edition"].value) {
    formEl["edition"].addEventListener("input", function () {
      formEl["edition"].setCustomValidity(
        Book.checkEdition(slots.edition).message);
    });
  }
  if (formEl.checkValidity()) {
    Book.update( slots);
    // update the selection list option
    selectBookEl.options[selectBookEl.selectedIndex].text = slots.title;
    formEl.reset();
  }
});
// neutralize the submit event
formEl.addEventListener("submit", function (e) {
  e.preventDefault();
});
```

The logic for setting up the UI for the *Delete* use case is similar. We only need to take care that the object to be deleted can be selected by providing a selection list, like in the *Update* use case. No validation is needed for the *Delete* use case.

2.7. Integrity Constraints with Firestore Security Rules

Recall that a complete data validation approach for a multi-user app with backend data storage requires a third (and last) round of validation in the backend database system. In the case of a JS/Firebase app, this means by using *Firebase Security Rules* for running the property checks defined in the app's model code. The Security Rules code is centralized in the back-end, consequently its check cannot be avoided by circumventing the front-end code.

2.7.1 Basic match/allow pattern

For defining Security Rules we follow this simple pattern:

1. declare `service` (Firestore, in this case),
2. `match` a database path, and

3. apply *conditions* to `allow` access to data in that path, using specific `methods`.

```
rules_version = '2';
service cloud.firestore {
  match <path> {
    allow <methods>: <condition>;
  }
}
```

2.7.2 Methods in the *allow* clause

Each `allow` statement includes at least one method to give access for incoming requests, for reading or writing data, and a ***convenience*** method can be used to allow access to ***standard*** methods, as shown in the following table:

Table 2-2. Convenience and standard methods for the allow clause in Security Rules

Convenience method	Standard method	Description
`read`	`get`	Read a single document
	`list`	Read queries and collections
`write`	`create`	Write a new document
	`update`	Write an existing document
	`delete`	Delete data

Being conditions optional, it is possible to write `allow` statements as simple as

```
allow read: false;
allow write: false;
```

or

```
allow read: true;
allow write: true;
```

Notice however that such simplicity may either compromise security or restrict undesirable access to our Firebase project, so for achieving ***granularity*** in our restrictions we can use conditions.

2.7.3 Conditions

Firebase Security Rules can be written with a syntax similar to JavaScript. For defining conditions web developers use

- **wildcard** variables, used to define specificity in a match path
  - for any book record/document in the books table/collection, or

    ```
    match /books/{book}/{document=**}
    ```

  - for any record in the books table/collection, and its subcollections

```
match /{path=**}/songs/{song}
```

- **request** objects, represent the incoming request, with the following attributes: `request.auth`, `request.path`, `request.query.resource`, `request.time`, and `request.writeFields`.
- **resource** objects, represent an available recourse (collection or document).

Example 1: this condition allows to create a new document if the *user is authenticated with verified email*:

```
allow create: if request.auth.token.email_verified == true
```

Example 2: this condition allows to create a new document if its *ISBN* field is equal to "`0279872631`".

```
allow write: if request.resource.data.isbn == "0279872631"
```

A well written condition has the following components: `allow` clause, method, `if` statement, affected object and attribute, and `value` against the affected object is compared to.

2.7.4 Creating Security Rules step by step

Going through each line in the following Security Rules file, we found

- line 1: the latest version of the Firebase Security Rules language is ***v2***, this statement goes always on the first line.
- line 2: defines the Firebase service Firestore.
- line 3: the `{database}` clause indicates that these rules apply to all databases in Firestore.
- line 4: defines a rule for every record/document within the "books" table/collection, even its subcollections.
- line 5: defines a rule that allows to create, update and delete records/documents if the *user is registered with verified email*.
- line 6: defines a rule that allows to get individual documents and queries or collections, even if the user is not authenticated (*anonymous*).

```
1 rules_version = '2';
2 service cloud.firestore {
3   match /databases/{database}/documents {
4     match /books/{document=**} {
5       allow write: if request.auth.token.email_verified == true;
6       allow read: if request.auth != null;
7     }
8   }
9 }
```

2.7.5 Functions

In the quest for a granular control to secure a web app, rules may become very complex, and functions are the way to reuse code in our Security Rules. Even though Security rules are written in a *similar-to-JavaScript DSL* (domain-specific language) we must know its limitations:

1. can contain a unique `return` statement.
2. can invoke other functions, but their recursive capacities are limited.

3. can use let to define variables, but must contain a return statement.

In this example Security Rules, the function **checkIsbnAsId** is invoked with the isbn value obtained from the request object:

```
service cloud.firestore {
  match /databases/{database}/documents {
    function checkIsbnAsId(isbn) {
      return !(exists(/databases/$(database)/documents/books/$(isbn)));
    }
    match /books/{document=**} {
      allow create: if request.auth.token.email_verified == true &&
        checkIsbnAsId(request.resource.data.isbn) == true;
    }
  }
}
```

2.7.6 Security Rules for the Validation App

For implementing backend data validation with Security Rules, we have to copy the check functions from the app's model classes and rewrite them using the Security Rules language, which has a syntax that is similar to the syntax of JS. The main components of Security Rules for the validation app are:

1. four functions to check ISBN, year and edition attributes: **checkIsbn** uses Firebase Security Rules regular expressions to check how `ISBN` is well formed, and **checkIsbnAsId** edition

   - `checkIsbn` uses Firebase Security Rules regular expressions to check if `ISBN` is well formed, and

     ```
     function checkIsbn( isbn) {
       return isbn.matches('^[0-9]+[0-9X]$')
              && isbn != null;
     }
     ```

   - `checkIsbnAsId` checks if another document on the books collection exists with same `ISBN`,

     ```
     function checkIsbnAsId( isbn) {
       return !(exists(/databases/$(database)/documents/books/$(isbn)));
     }
     ```

   - `checkYear` checks all constraints for the attribute `year`: min, max and integer, and

     ```
     function checkYear( year) {
       return (timestamp.date( year,1,1).toMillis() < request.time.toMillis()
               && year > 1459
               && year is int
               && year != null);
     }
     ```

   - `checkEdition` checks if the `edition` attribute is present, and if it is integer,

     ```
     function checkEdition( edition) {
       return (edition is int || !("edition" in request.resource.data));
     }
     ```

2. one `allow` statement with the ***convenience method*** `read`, for giving access to read to any anonymous user,

   ```
   allow read: if request.auth != null;
   ```

3. one `allow` statement with the *standard method* **create**, for giving access to write to *registered users with verified email*, in the process the `ISBN`, `year` and `edition` attributes are checked,

   ```
   allow create: if request.auth.token.email_verified == true
                 && checkIsbnAsId( request.resource.data.isbn) == true
                 && checkIsbn( request.resource.data.isbn) == true
                 && request.resource.data.title != null
                 && checkYear( request.resource.data.year) == true
                 && checkEdition( request.resource.data.edition);
   ```

4. one `allow` statement with the *standard method* **update**, for giving access to write to *registered users with verified email*, and since the incoming data is mean to *update* a document we use the clause `diff` to compare incoming data (`request.resource.data`) with stored data (`resource.data`); but the update may only happen on the attributes `title`, `year` and `edition`, and not on `ISBN`, which never should be updated

   ```
   allow update: if request.auth.token.email_verified == true
                    && (request.resource.data.diff( resource.data).affectedKeys()
                     .hasOnly(['title', 'year', 'edition']))
                    && request.resource.data.year != null ?
                      checkYear( request.resource.data.year) : true
                    && request.resource.data.edition != null ?
                      checkEdition( request.resource.data.edition) : true;
   ```

5. one `allow` statement with the *standard method* **delete**, for giving access to delete to *authenticated users with verified email*.

   ```
   allow delete: if request.auth.token.email_verified == true;
   ```

These Security Rules are located in the file `2-ValidatioApp/firestore.rules`.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    /** VALIDATION FUNCTIONS **/
    // check how ISBN is formed: a 10-digit string or a 9-digit string followed by "X"
    function checkIsbn( isbn) {
      return isbn.matches('^[0-9]+[0-9X]$')
             && isbn != null;
    }
    // check if exist document with same ISBN
    function checkIsbnAsId( isbn) {
      return !(exists(/databases/$(database)/documents/books/$(isbn)));
    }
    // check all constraints for year: min, max and integer
    function checkYear( year) {
      return (timestamp.date( year,1,1).toMillis() < request.time.toMillis()
             && year > 1459
             && year is int
             && year != null);
    }
```

```
    // if present, check if it is integer
    function checkEdition( edition) {
      return (edition is int || !("edition" in request.resource.data));
    }
    /** VALIDATION RULES **/
    match /{books}/{document=**} {
      /** RULES FOR allow read WITH CONVENIENCE METHOD - LOW GRANULARITY **/
      /** NO authentication required **/
      allow read: if request.auth != null;

      /** RULES FOR allow write WITH STANDARD METHODS - HIGH GRANULARITY **/
      /** authentication required **/
      //validate when create new book record
      allow create: if request.auth.token.email_verified == true
                    && checkIsbnAsId( request.resource.data.isbn) == true
                    && checkIsbn( request.resource.data.isbn) == true
                    && request.resource.data.title != null
                    && checkYear( request.resource.data.year) == true
                    && checkEdition( request.resource.data.edition);

      // validate when update book record
      allow update: if request.auth.token.email_verified == true
                    && (request.resource.data.diff( resource.data).affectedKeys()
                     .hasOnly(['title', 'year', 'edition']))
                    && request.resource.data.year != null ?
                        checkYear( request.resource.data.year) : true
                    && request.resource.data.edition != null ?
                        checkEdition( request.resource.data.edition) : true;

      // validate when delete book record
      allow delete: if request.auth.token.email_verified == true;
    }
  }
}
```

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

2.7.7 Attribute-level access control

More fine-grained access control can be achieved by specifying access control rules for specific fields.

2.7.8 Testing security rules in Firestore

While writing Security Rules we need to test our rules in real time, and deploying our rules to Firebase every time we change them is not an efficient approach. The advised tool is Firebase Emulator, which allows running a Firestore database locally.

2.7.9 Using regular expressions

Sometimes using a regular expression is the only way to define a validations constraint. Firebase uses RE2, a Regular Expression software created by Google.

2.8. Possible Variations and Extensions

2.8.1 Adding an object-level validation function

When object-level validation (across two or more properties) is required for a model class, we can add a custom validation function `validate` to it, such that object-level validation can be performed before save by invoking `validate` on the object concerned. For instance, for expressing the constraint defined in the class model shown in **Figure 1-1.** An example of an object-level constraint, we define the following validation function:

```
Author.prototype.validate = function () {
  if (this.dateOfDeath && this.dateOfDeath < this.dateOfBirth) {
    throw new ConstraintViolation("The dateOfDeath must be after the dateOfBirth!");
  }
};
```

When a `validate` function has been defined for a model class, it can be invoked in the create and update methods. For instance,

```
Author.add = function (slots) {
  var author = null;
  try {
    author = new Author( slots);
    author.validate();
  } catch (e) {
    console.log( e.constructor.name +": "+ e.message);
  }
};
```

2.8.2 Handling success and error messages

We have implemented a good solution for handling error messages throughout this tutorial, showing them contextually in each input field. However, we should consider extending this solution to show success messages on the user interface when the user completes an action successfully, for instance, for any CRUD use case or in the authentication workflow. Each app in this series of tutorials handles success messages, but only on the JavaScript console.

2.9. Points of Attention

2.9.1 Boilerplate code

An issue with the do-it-yourself code of this example app is the *boilerplate code* needed

1. per model class for the storage management methods `add`, `update`, `destroy`, etc.;

2. per model class and property for getters, setters and validation checks.

While it is good to write this code a few times for learning app development, you don't want to write it again and again later when you work on real projects. In our mODELcLASSjs tutorial, we present an approach how to put these methods in a generic form in a meta-class, such that they can be reused in all model classes of an app.

2.9.2 Configuring the UI for preventing invalid user input

Many of the new HTML5 input field types (like `number`, `tel`, `email`, `url`, `date` (together with `datetime-local`, `time` and `month`) or `color`) are intended to allow web browsers rendering corresponding `input` elements in the form of UI widgets (like a *date* picker or a `color picker`) that limit the user's input options such that only valid input is possible. In terms of usability, it's preferable to prevent users from entering invalid data instead of allowing to enter it and only then checking its validity and reporting errors.

Input fields for decimal number input should not be defined like

```
<input type="number" name="..." />
```

but rather like

```
<input type="text" inputmode="decimal" name="..." />
```

because this provides for a better user experience on mobile phones.

2.9.3 Improving the user experience by showing helpful auto-complete suggestions

While browsers have heuristics for showing auto-complete suggestions, you cannot rely on them, and should better add the autocomplete attribute with a suitable value. For instance, in iOS Safari, setting the input type to "tel" does only show auto-complete suggestions if autocomplete="tel" is added.

HTML5 defines more than 50 possible values for the autocomplete attribute. So, you have to make an effort looking up the one that best suits your purposes.

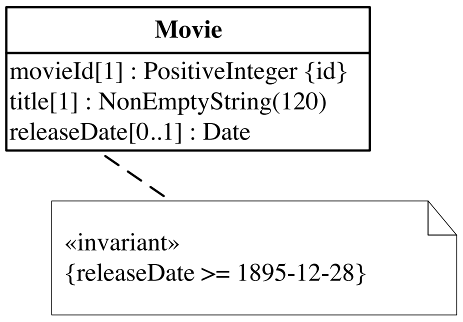
You can also create your own custom auto-complete functionality with datalist.

2.10. Practice Project

The purpose of the app to be built is managing information about movies. Like in the book data management app discussed in the tutorial, storing all the data in the Firestore database.

The app deals with just one object type: `Movie`, as depicted in **Figure 2-2.** The object type `Movie` defined with several constraints below. In the subsequent parts of the tutorial, you will extend this simple app by adding enumeration-valued attributes, as well as actors and directors as further model classes, and the associations between them.

Figure 2-2. *The object type* `Movie` *defined with several constraints*


Movie
movieId[1] : PositiveInteger {id}
title[1] : NonEmptyString(120)
releaseDate[0..1] : Date
«invariant»
{releaseDate >= 1895-12-28}

In this model, the following constraints have been expressed:

1. Due to the fact that the `movieId` attribute is declared to be the *standard identifier* of `Movie`, as expressed by the property annotation `{id}` shown after the property range, it is *mandatory* and *unique*.

2. The `title` attribute is *mandatory*, as indicated by its multiplicity expression [1], and has a *string length constraint* requiring its values to have at most 120 characters.

3. The `releaseDate` attribute has an *interval constraint*: it must be greater than or equal to 1895-12-28.

Notice that the `releaseDate` attribute is not mandatory, but *optional*, as indicated by its multiplicity expression [0..1]. In addition to the constraints described in this list, there are the implicit range constraints defined by assigning the datatype `PositiveInteger` to `movieId`, `NonEmptyString` to `title`, and `Date` to `releaseDate`. In our plain JavaScript approach, all these property constraints are coded in the model class within property-specific *check* functions.

Following the tutorial, you have to take care of

1. adding for every property a *check* function that validates the constraints defined for the property, and a *setter* method that

invokes the check function and is to be used for setting the value of the property,

2. *performing validation* before any data is saved in the `Movie.add` and `Movie.update` methods.

in the *model* code of your app, while In the *user interface* ("view") code you have to take care of

1. styling the user interface with CSS rules (by integrating a CSS library such as Yahoo's Pure CSS),
2. ***validation on user input*** for providing immediate feedback to the user,
3. ***validation on form submission*** for preventing the submission of invalid data.

You can use the following sample data for testing your app:

Table 2-3. 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

Chapter 3. Firebase Features

3.1. Firestore Timestamp Data Type

Firestore has a *Timestamp* data type, the value of which is represented as seconds and fractions of seconds at nanosecond resolution. A timestamp looks like:

```
seconds: 1022803200, nanoseconds: 276147000
```

But in the Firebase console it is shown encoded in ISO format like:

```
9 July 1971 at 02:00:00 UTC+2
```

The utility function `date2IsoDateString` (in `lib/util.mjs`) has the following code. Notice that we use the Firestore method `toDate()` for converting the timestamp object coming from FIrestore to a JavaScript `Date` object, and then to human-readable format string:

```
function date2IsoDateString (timeStamp) {
  let dateObj = timeStamp.toDate();
  let  y = dateObj.getFullYear(),
    m = "" + (dateObj.getMonth() + 1),
    d = "" + dateObj.getDate();
  if (m.length < 2) m = "0" + m;
  if (d.length < 2) d = "0" + d;
  return [y, m, d].join("-");
}
```

Use this function for your convenience in the practice project.

Resources

- Secure data in Firestore (official Firebase documentation by Google).
- Firebase Security Rules (official Firebase documentation by Google).
- Firebase Security Rules Regular Expressions (official Firebase documentation by Google).
- Test your Firestore Security Rules with Firebase Emulator (official Firebase documentation by Google).
- A List of Firestore Security Rules for Your Firebase Project.
- RE2 Regular Expressions Generator by Olaf Neumann.
- RE2 Regular Expressions Tester by Steve Domin.