JS/Firebase Web App Tutorial Part 4: Managing Unidirectional Associations

Learn how to manage unidirectional associations between object types, such as the associations assigning publishers and authors to books using plain JavaScript and Firebase

By Gerd Wagner and Juan-Francisco Reyes

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

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

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

Published 2022-07-25.

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 4 of our series of six tutorials about model-based development of web applications with plain JavaScript and Firebase. It shows how to build a web app that takes care of the three object types `Book`, `Publisher` and `Author` as well as of two unidirectional associations:

1. the association between the classes `Book` and `Publisher` assigning a publisher to a book,
2. the association between the classes `Book` and `Author` assigning one or more authors to a book.

The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). It extends the example app of Part 2 by adding code for handling the **unidirectional functional** (many-to-one) association between `Book` and `Publisher`, and the **unidirectional non-functional** (many-to-many) association between `Book` and `Author`. The other parts of the tutorial are:

- Part 1: Building a **minimal** app.
- Part 2: Handling **constraint validation**.
- Part 3: Dealing with **enumerations**.
- Part 5: Managing **bidirectional associations**, such as the associations between books and publishers and between books and authors, not only assigning authors and a publisher to a book, but also the other way around, assigning books to authors and to publishers.

- Part 6: Handling **subtype** (inheritance) relationships between object types.

Chapter 1. Reference Properties and Unidirectional Associations

A property defined for an object type, or class, is called a ***reference property*** if its `values` are *references* that reference an object of another, or of the same, type. For instance, the class Committee shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair` below has a reference property `chair`, the values of which are references to objects of type `ClubMember`.

An **association** between object types classifies relationships between objects of those types. For instance, the association *Committee*-**has**-*ClubMember*-**as-chair**, which is visualized as a connection line in the class diagram shown in **Figure 1-2.** An association end with a "dot" below, classifies the relationships FinanceCommittee-**has**-PeterMiller-**as-chair**, RecruitmentCommittee-has-SusanSmith-as-chair and AdvisoryCommittee-has-SarahAnderson-as-chair, where the objects PeterMiller, SusanSmith and SarahAnderson are of type `ClubMember`, and the objects FinanceCommittee, RecruitmentCommittee and AdvisoryCommittee are of type `Committee`. An association as a set of relationships can be represented as a table like so:

Committee-has-ClubMember-as-chair	
Finance Committee	Peter Miller
Recruitment Committee	Susan Smith
Advisory Committee	Sarah Anderson

Reference properties correspond to a special form of associations, namely to **unidirectional** *binary associations*. While a binary association does, in general, not need to be directional, a reference property represents a binary association that is directed from the property's domain class (where it is defined) to its range class.

In general, associations are ***relationship types*** with two or more ***object types*** participating in them. An association between two object types is called ***binary***. In this tutorial we only discuss binary associations. For simplicity, we just say 'association' when we actually mean 'binary association'.

While individual relationships (such as FinanceCommittee-has-PeterMiller-as-chair) are important information items in business communication and in information systems, associations (such as *Committee*-**has**-*ClubMember*-**as-chair**) are important elements of information models. Consequently, software applications have to implement them in a proper way, typically as part of their model layer within a *model-view-controller* (MVC) architecture. Unfortunately, many application development frameworks lack the required support for dealing with associations.

In mathematics, associations have been formalized in an abstract way as sets of uniform tuples, called relations. In Entity-Relationship (ER) modeling, which is the classical information modeling approach in information systems and software engineering, objects are called entities, and associations are called relationship types. The Unified Modeling Language (UML) includes the UML Class Diagram language for information modeling. In UML, object types are called classes, relationship types are called associations, and individual relationships are called "links". These three terminologies are summarized in the following table:

Our preferred term(s)	UML	ER Diagrams	Mathematics
object	object	entity	individual
object type (class)	class	entity type	unary relation
relationship	link	relationship	tuple
association (relationship type)	association	relationship type	relation

Our preferred term(s)	UML	ER Diagrams	Mathematics
functional association		one-to-one, many-to-one or one-to-many relationship type	function

We first discuss reference properties, which implicitly represent unidirectional binary associations in an "association-free" class model (a model without any explicit association element).

1.1. References and Reference Properties

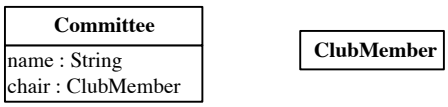
A reference can be either *human-readable* or an *internal object reference*. Human-readable references refer to identifiers that are used in human communication, such as the unique names of astronomical bodies, the ISBN of books and the employee numbers of the employees of a company. Internal object references refer to the computer memory addresses of OOP objects, thus providing an efficient mechanism for accessing objects in the main memory of a computer.

Some languages, like SQL and XML, only support human-readable, but not internal references. In SQL, human-readable references are called *foreign keys*, and the identifiers they refer to are called *primary keys*. In XML, human-readable references are called *ID references* and the corresponding attribute type is `IDREF`.

Objects in an OO program can be referenced either with the help of human-readable references (such as integer codes) or with internal object references, which are preferable for accessing objects efficiently in main memory. Following the XML terminology, we call human-readable references *ID references*. We follow the standard naming convention for ID reference properties where an ID reference property defined in a class A and referencing objects of class B has the name `b_id` using the suffix `_id`. When we store persistent objects in the form of records or table rows, we need to convert internal object references, stored in properties like publisher, to ID references, stored in properties like publisher_id. This conversion is performed as part of the serialization of the object by assigning the standard identifier value of the referenced object to the ID reference property of the referencing object.

In OO languages, a property is defined for an object type, or class, which is its *domain*. The values of a property are either data values from some datatype, in which case the property is called an ***attribute***, or they are object references referencing an object from some class, in which case the property is called a ***reference property***. For instance, the class `Committee` shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair` below has an attribute name with range `String`, and a reference property chair with range `ClubMember`.

Figure 1-1. *A committee has a club member as chair expressed by the reference property* `chair`



Object-oriented programming languages, such as JavaScript, PHP, Java and C#, directly support the concept of *reference properties*, which are properties whose range is not a *datatype* but a *reference type*, or *class*, and whose values are object references to instances of that class.

By default, the multiplicity of a property is 1, which means that the property is ***mandatory*** and ***functional*** (or, in other words, *single-valued*), having ***exactly one*** value, like the property `chair` in class `Committee` shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair`. When a functional property is optional (not mandatory), it has the multiplicity `0..1`, which means that the property's minimum cardinality is 0 and its maximum cardinality is 1.

A reference property can be either ***single-valued*** (*functional*) or ***multi-valued*** (*non-functional*). For instance, the reference property `Committee::chair` shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair` is single-valued, since it assigns a unique club member as chair to a club. An example of a multi-valued reference property is provided by the property Book::authors shown in below.

Normally, the collection value of a multi-valued reference property is a set of references, implying that the order of the references does not matter. In certain cases, however, the order matters and, consequently, the collection value of such a

multi-valued reference property is an *ordered set* of references, typically implemented as a list. Only rarely, the collection value of a multi-valued reference property may be a, possibly ordered, *multi-set* (also called *bag*).

1.2. Referential Integrity

References are important information items in our application's database. However, they are only meaningful, when their referential integrity is maintained by the app. This requires that for any reference, there is a referenced object in the database. Consequently, any reference property p with domain class C and range class D comes with a referential integrity constraint that has to be checked whenever

1. a new object of type `C` is created,
2. the value of `p` is changed for some object of type `C`,
3. an object of type `D` is destroyed.

A referential integrity constraint also implies two change dependencies:

1. An ***object creation dependency***: an object with a reference to another object can only be created after the referenced object has been created.
2. An ***object destruction dependency***: an object that is referenced by another object can only be destroyed after
   a. the referencing object(s) is (are) destroyed first; this approach can be called the *CASCADE deletion policy*, or
   b. the reference in the referencing object is either dropped (the *DROP-REFERENCE deletion policy*) or replaced by another reference.

   For every reference property in our app's model classes, we have to choose, which of these two possible *deletion policies* applies.

In certain cases, we may want to relax this strict regime and allow creating objects that have non-referencing values for an ID reference property, but we do not consider such cases.

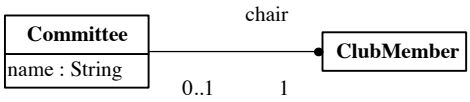
Typically, object creation dependencies are managed in the user interface by not allowing the user to enter a value of an ID reference property, but only to select one from a list of all existing target objects.

1.3. Modeling Reference Properties as Unidirectional Associations

A reference property (such as `chair` in the example shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair` above) can be modeled in a UML class diagram in the form of an ***association end*** owned by its domain class, which is visualized with the help of a small filled circle (also called a "dot"). This requires to connect the domain class and the range class of the reference property with an association line, place an ownership dot at the end of this line at the range class side, and annotate this association end with the property name and with a multiplicity symbol, as shown in **Figure 1-2.** An association end with a "dot" below for the case of our example. In this way we get a ***unidirectional association***, the source class of which is the property's ***domain*** and the ***target*** class of which is the property's ***range***.

The fact that an association end is owned by the class at the other end, as visually expressed by the association end ownership dot at the association end `chair` in the example shown in **Figure 1-2.** An association end with a "dot" below, implies that the association end represents a reference property. In the example of **Figure 1-2.** An association end with a "dot", the represented reference property is `Committee::chair` having ClubMember as range. Such an association, with only one association end ownership dot, is *unidirectional* in the sense that it allows `navigation´ (object access) in one direction only: from the class at the opposite side of the dot (the *source* class) to the class where the dot is placed (the *target* class).

Figure 1-2. *An association end with a "dot"*



Thus, the two diagrams shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair` and Figure **Figure 1-2.** An association end with a "dot" express essentially equivalent models. When a reference property, like chair in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair`, is modeled by an association end with a "dot", then the property's multiplicity is attached to the association end. Since in a design model, all association ends need to have a multiplicity, we also have to define a multiplicity for the other end at the side of the `Committee` class, which represents the inverse of the property. This multiplicity (of the inverse property) is not available in the original property description in the model shown in **Figure 1-1.** A committee has a club member as chair expressed by the reference property `chair`, so it has to be added according to the intended semantics of the association. It can be obtained by answering the question "is it mandatory that any `ClubMember` is the `chair` of a `Committee`?" for finding the minimum cardinality and the question "can a `ClubMember` be the `chair` of more than one `Committee`?" for finding the maximum cardinality.

When the value of a property is a set of values from its range, the property is ***non-functional*** and its multiplicity is either `0..*` or `n..*` where `n > 0`. Instead of `0..*`, which means "neither mandatory nor functional", we can simply write the asterisk symbol `*`. The association shown in **Figure 1-2.** An association end with a "dot" assigns at most one object of type `ClubMember` as chair to an object of type `Committee`. Consequently, it's an example of a functional association.

An overview about the different cases of functionality of an association is provided in the following table:

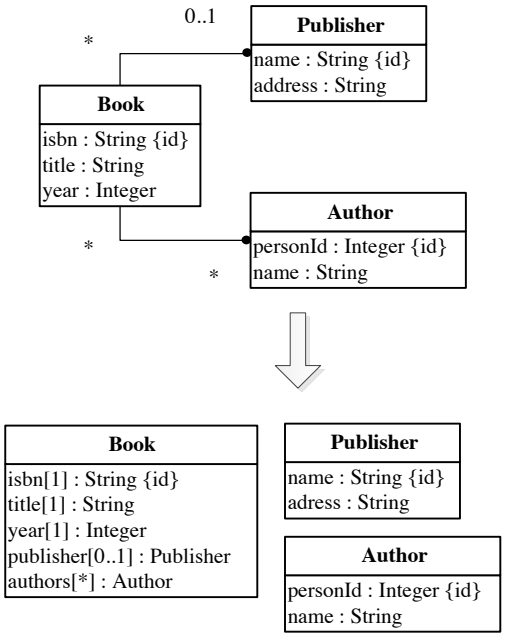
Functionality type	Meaning
one-to-one	both functional and inverse functional
many-to-one	functional
one-to-many	inverse functional
many-to-many	neither functional nor inverse functional

Notice that the directionality and the functionality type of an association are independent of each other. So, a unidirectional association can be either functional (one-to-one or many-to-one), or non-functional (one-to-many or many-to-many).

1.4. Representing Unidirectional Associations as Reference Properties

A unidirectional association between a source and a target class can be represented as a reference property of the source class. This is illustrated in **Figure 1-3.** Representing unidirectional associations as reference properties below for two unidirectional associations: a many-to-one and a many-to-many association. takes care of the three object types Book, Publisher and Author as well as of two unidirectional associations:

Figure 1-3. *Representing unidirectional associations as reference properties*

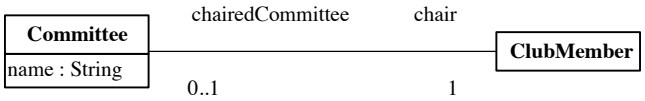


Notice that, in a way, we have eliminated the two explicit associations and replaced them with corresponding reference properties resulting in a class model that can be coded with a classical OOP language in a straightforward way. OOP languages do not support associations as first class citizens. They do not have a language element for defining associations. Consequently, an OOP class design model, which we call ***OO class model***, must not contain any explicit association.

1.5. Adding Directionality to a Non-Directed Association

When we make an information model in the form of a UML class diagram, we typically end up with a model containing one or more associations that do not have any ownership defined for their ends, as, for instance, in **Figure 1-4.** A model of a non-directed association between Committee and ClubMember below. When there is no ownership dot at either end of an association, such as in this example, this means that the model does not specify how the association is to be represented (or realized) with the help of reference properties. Such an association does not have any direction. According to the UML 2.5 specification, the ends of such an association are "owned" by itself, and not by any of the classes participating in it.

Figure 1-4. *A model of a non-directed association between Committee and ClubMember*



An information design model that contains an association without association end ownership dots is acceptable as a relational database design model, but it is incomplete as a design model for OOP languages.

For instance, the model of **Figure 1-4.** A model of a non-directed association between Committee and ClubMember provides a relational database design with two entity tables, `committees` and `clubmembers`, and a separate one-to-one relationship table `committee_has_clubmember_as_chair`. But it does not provide a design for Java classes, since it does not specify how the association is to be implemented with the help of reference properties.

There are three options how to turn an information design model of a non-directed association (without any association end ownership dots) into an information design model where all associations are either unidirectional or bidirectional: we can place an ownership dot at either end or at both ends of the association. Each of these three options defines a different way how to represent, or implement, the association with the help of reference properties. So, for the association shown in **Figure 1-4.** A model of a non-directed association between Committee and ClubMember above, we have the following options:

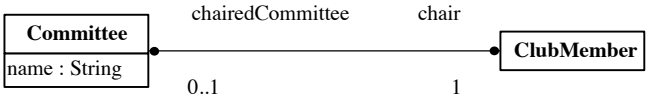
1. Place an ownership dot at the chair association end, leading to the model shown in **Figure 1-2.** An association end with a "dot" above, which can be transformed into the OO class model shown in **Figure 1-1.** A committee has a club member as

chair expressed by the reference property `chair` above.

2. Place an ownership dot at the `chairedCommittee` association end, leading to the completed models shown in **Figure 1-8.** Turning a functional association end into a reference property below.

3. Make the association bidirectional by placing ownership dots at both association ends, as shown in **Figure 1-5.** Modeling a bidirectional association between Committee and ClubMember with the meaning that the association is implemented in a redundant manner by a pair of mutually inverse reference properties `Committee::chair` and `ClubMember::chairedCommittee`, as discussed in the next part of our tutorial.

Figure 1-5. *Modeling a bidirectional association between Committee and ClubMember*



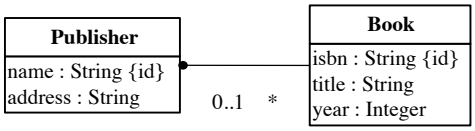
So, whenever we have modeled an association, we have to make a choice, which of its ends represents a reference property and will therefore be marked with an ownership dot. It can be either one, or both. This decision also implies a decision about the *navigability* of the association. When an association end represents a reference property, this implies that it is navigable (via this property).

In the case of a functional association that is not one-to-one, the simplest design is obtained by defining the direction of the association according to its functionality, placing the association end ownership dot at the association end with the multiplicity `0..1` or `1`. For a non-directed one-to-one or many-to-many association, we can choose the direction as we like, that is, we can place the ownership dot at either association end.

1.6. Our Running Example

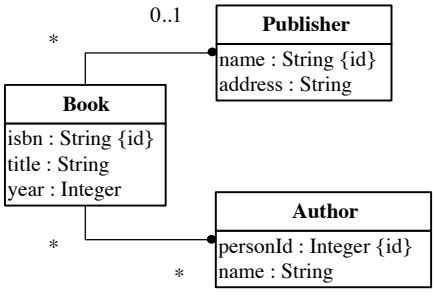
The model shown in **Figure 1-6.** The Publisher-Book information design model with a unidirectional association below (about publishers and books) serves as our running example for a unidirectional functional association. Notice that it contains the unidirectional many-to-one association *Book*-**has**-*Publisher*.

Figure 1-6. *The Publisher-Book information design model with a unidirectional association*



We may also have to deal with a multi-valued reference property representing a unidirectional non-functional association. For instance, the unidirectional many-to-many association between `Book` and `Author` shown in **Figure 1-7.** The Publisher-Book-Author information design model with two unidirectional associations below, models a multi-valued reference property `authors`.

Figure 1-7. *The Publisher-Book-Author information design model with two unidirectional associations*



1.7. Eliminating Unidirectional Associations

Since classical OO programming languages do not support associations as first class citizens, but only classes and reference properties representing unidirectional associations, we have to eliminate all explicit associations from general information design models for obtaining OO class models.

1.7.1 The basic elimination procedure

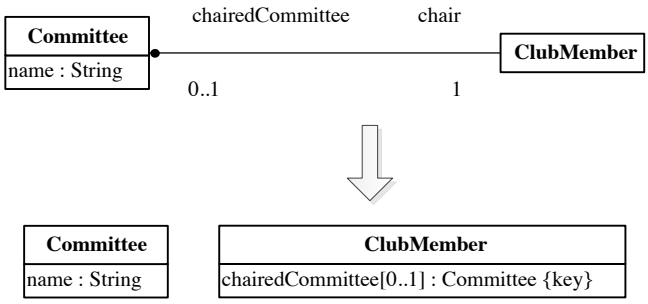
The starting point of our restricted ***association elimination*** procedure is an information design model with various kinds of unidirectional associations, such as the model shown in **Figure 1-6.** The Publisher-Book information design model with a unidirectional association above. If the model still contains any non-directional associations, we first have to turn them into directional ones by making a decision on the ownership of their ends, as discussed in **Section 1.5.** Adding Directionality to a Non-Directed Association.

A unidirectional association connecting a source with a target class is replaced with a corresponding reference property in its source class having

1. the same name as the association end, if there is any, otherwise it is set to the name of the target class (possibly pluralized, if the reference property is multi-valued);
2. the target class as its range;
3. the same multiplicity as the target association end,
4. a uniqueness constraint if the unidirectional association is inverse functional.

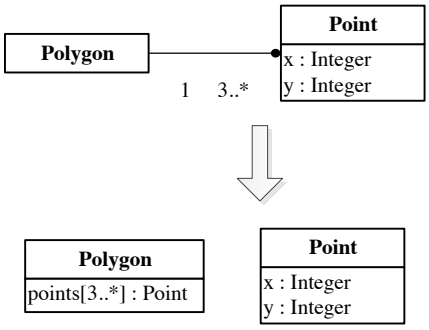
This replacement procedure is illustrated for the case of a unidirectional one-to-one association in **Figure 1-8.** Turning a functional association end into a reference property below, where the uniqueness constraint of the reference property `chairedCommittee` is expressed by the `{key}` property modifier.

Figure 1-8. *Turning a functional association end into a reference property*



For the case of a unidirectional one-to-many association, **Figure 1-9.** Turning a non-functional association end into a multi-valued reference property below provides an illustration of the association elimination procedure. Here, the non-functional association end at the target class Point is turned into a corresponding reference property with name `points` obtained as the pluralized form of the target class name.

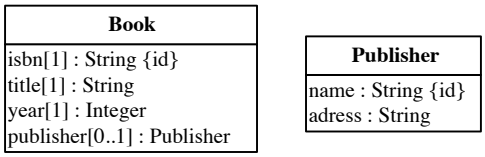
Figure 1-9. *Turning a non-functional association end into a multi-valued reference property*



1.7.2 Eliminating associations from the design model

In the case of our running example, the *Publisher-Book-Author* information design model, we have to replace both unidirectional associations with suitable reference properties. In the first step, we replace the many-to-one association *Book*-**has**-*Publisher* in the model of **Figure 1-6.** The Publisher-Book information design model with a unidirectional association with a functional reference property `publisher` in the class `Book`, resulting in the OO class model shown in **Figure 1-10.** An OO class model for `Publisher` and `Book`.

Figure 1-10. *An OO class model for `Publisher` and `Book`*



Notice that since the target association end of the *Book*-**has**-*Publisher* association has the multiplicity `0..1`, we have to declare the new property publisher as optional by defining its multiplicity to be `0..1`.

The meaning of this OO class model and its reference property `publisher` can be illustrated by a sample data population for the two model classes `Book` and `Publisher` as presented in the following tables:

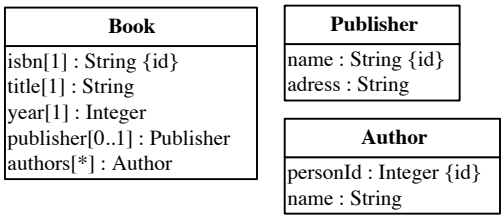
Publisher	
Name	Address
Bantam Books	New York, USA
Basic Books	New York, USA

Book			
ISBN	**Title**	**Year**	**Publisher**
0553345842	The Mind's I	1982	Bantam Books
1463794762	The Critique of Pure Reason	2011	
1928565379	The Critique of Practical Reason	2009	
0465030793	I Am A Strange Loop	2000	Basic Books

Notice that the values of the "Publisher" column of the *Book* table are unique names that reference a row of the *Publisher* table. The "Publisher" column may not have a value for certain rows due to the fact that the corresponding reference property `publisher` is optional.

In the second step, we replace the many-to-many association *Book*-**has**-*Author* in the model of **Figure 1-6.** The Publisher-Book information design model with a unidirectional association with a multi-valued reference property `authors` in the class `Book`, resulting in the OO class model shown in **Figure 1-11.** An OO class model for the classes Book, Publisher and Author.

Figure 1-11. *An OO class model for the classes Book, Publisher and Author*



The meaning of this OO class model and its reference properties `Book::publisher` and `Book::authors` can be illustrated by sample data populations for the three model classes:

Publisher	
Name	Address
Bantam Books	New York, USA
Basic Books	New York, USA

Book				
ISBN	**Title**	**Year**	**Authors**	**Publisher**
0553345842	The Mind's I	1982	1,2	Bantam Books
1463794762	The Critique of Pure Reason	2011	3	
1928565379	The Critique of Practical Reason	2009	3	
0465030793	I Am A Strange Loop	2000	2	Basic Books

Author	
Author ID	Name
1	Daniel Dennett
2	Douglas Hofstadter
3	Immanuel Kant

After the platform-independent OO class model has been completed, one or more platform-specific implementation models, for a choice of specific implementation platforms, can be derived from it. Examples of types of platform-specific implementation models are *JS class models*, *Java Entity class models* and *SQL table models*.

A platform-specific implementation model can still be expressed in the form of a UML class diagram, but it contains only modeling elements that can be directly coded in the chosen platform. Thus, for any platform considered, two guidelines are needed: 1) how to make the platform-specific implementation model, and 2) how to code this model.

Chapter 2. Implementing Associations with JS and Firebase

As we discussed before, Firebase, as a NoSQL database, does not use a relational data model, lacking hence of a database schema, offering thus the possibility of flexible designs. Another advantage of NoSQL databases is the high efficiency and speed, as result of having all data in a single JSON alike object. And finally, NoSQL databases are horizontally scalable, making infrastructure costs and maintenance cheap an agile.

When designing NoSQL data models we should consider that, contrary to SQL databases, NoSQL does not support joins natively, therefore we will have to handle associations during design time. If we join two tables/collections in parallel by querying multiple records/documents in the model and send it merged to the front end layer, we will have an non-tolerable latency that represents an overhead in terms of performance and user experience.

On the other hand, *write transactions* or *read/write transactions*, are optimized for writing data, therefore there is not a way to control concurrency on only *read* operations, and we should live with possible discrepancies generated by changes made by another user while data is read and rendered in the UI.

A plausible solution is ***data duplication***, embedding the target associated record in the source record as well as in its respective collection, and considering this, there are two main techniques that we should consider for NoSQL data modeling:

- **Denormalization**. It consists in replicating data in multiple tables, making data querying easier. On the other hand, denormalization means a considerable increase in data volume, and in the case of Firebase, which is a paid service, this should be considered carefully in the design stage.
- **Aggregation**. It consists of allowing developers to create nested (deep) and complex structures, reducing tables joins and minimizing on-to-one relationships.

In this tutorial we expose a way to handle associations with Firebase by using denormalization and aggregation.

2.1. Read and Write Transactions

As we said before, typically, a front-end web application is a single-user application. However, with Firestore it is possible to create multi-user applications, thanks mainly to its capacity to

a. "listen" to database changes in real-time, as we have seen how to sync database and UI in *Part 3*, and

b. allow simultaneous read and write operations executing *atomic operations* for dealing with *concurrent read and write operations*.

An important issue related to referential integrity and adding and eliminating association are read and write transactions, which deal with ***concurrency control***. In our example, when a book record/document is stored, another user might be updating or deleting associated publishers or authors at the very same time, resulting in a failed operation, that in the best case scenario it could break the app, but in the worst it could happen invisible to the user, hurting referential integrity and generating inconsistent data.

Firestore deals with concurrency control using ***read/write transactions*** and ***write transactions***, which are *atomic operations* that encapsulate read/write operations in a single operation. In an atomic operation either all of the operations succeed, or none of them are completed. If an atomic operation happens when another user modifies the concurrent documents, Firestore retries the transaction, attempting to succeed for data consistency. The two atomic operations that Firestore uses to control concurrency are

- read/write transactions, "*transactions*" in the Firestore's terminology, a combo of a number of read operations followed by a number of write operations, and
- write transaction, "*batched writes*" in the Firestore's terminology, a combo of write operations.

2.1.1 Read/write transactions

A successful read/write transaction can be achieved if:

1. Always a number of reading data operations is followed by writing data operations.
2. If a concurrent user edits a document involved in the atomic operation, the operation runs many times.
3. A transaction fails if the user goes offline.
4. Any information or data should pass out of the transaction, for instance trying to update the UI during the transaction.

2.1.2 Write transactions

A successful write transaction can be achieved if:

1. Only writing data operations are involved.
2. The atomic operation contains not more than 500 operations.

3. A write transaction operation does not fail if the user goes offline.

2.2. Rendering Reference Properties in the User Interface

Another important issue for preserving referential integrity and adding and eliminating association in a web app is the use interface that joins class objects. The widgets used for data input and output in a (CRUD) data management user interface (UI) normally correspond to properties defined in a model class of an app. We have to distinguish between (various types of) *input fields* corresponding to (various kinds of) *attributes*, and *selection widgets* corresponding to enumeration *attributes* or to *reference properties*. Representing reference properties in the UI with an `input` field, instead of `select` control, that prevents users from entering invalid ID references, so it takes care of *referential integrity*.

In general, a ***single-valued reference property*** can be rendered as a single-selection list in the UI, however since each Firestore request is to be paid, we avoid loading an entire table/collection for filling the `option` elements of a selection list by using an `input` field.

A ***multi-valued reference property*** can be rendered as a multiple-selection list in the UI. However, the corresponding multiple-select control of HTML is not really usable as soon as there are many (say, more than 20) different options to choose from because the way it renders the choice is visually too scattered. In the special case of having only a few (say, no more than 7) options, we can also use a checkbox group instead of a multiple-selection list. But for the general case of having in the UI a list containing all associated objects chosen from the reference property's range class, we need to develop a special UI widget that allows to add (and remove) objects to (and from) a list of chosen objects.

Such a ***multi-selection widget*** consists of

1. an HTML input field where the foreign key of the associated object can be entered, and a push button (add) that triggers the association process;
2. invocation of the data retrieval and referential integrity check procedures, that once passed, the new associated object is added to a list from the range class of the multi-valued reference property;
3. a push button (x) for each item of the list of associated objects.

Figure 2-1. *A multi-selection widget showing an input field and an "add" button, as well as a list of associated authors and remove controllers*

Later we will review in detail how the *multi-selection widget* is implemented.

Chapter 3. Implementing Unidirectional Functional Associations with JS and Firebase

The three example apps that we have discussed in previous parts of this tutorial, the *minimal* app, the *validation* app, and the *enumeration* app, have been limited to managing the data of one object type only. A real app, however, has to manage the data of several object types, which are typically related to each other in various ways. In particular, there may be ***associations*** and ***subtype*** (inheritance) relationships between object types. Handling associations and subtype relationships are advanced issues in software application engineering. They are often not sufficiently discussed in text books and not well supported by application development frameworks.

A unidirectional *functional* association is either one-to-one or many-to-one. In both cases such an association is represented, or implemented, with the help of a *single-valued* reference property.

In this chapter, we show

1. how to derive a plain JS class model from an OO class model with single-valued reference properties representing *unidirectional functional associations* in Firestore,

2. how to code the JS class model in the form of plain JavaScript model classes,

3. how to write the view and controller code based on the model code.

3.1. Implementing Single-Valued Reference Properties

When coding a class, the ES6 feature of *function parameter destructuring* allows using a single constructor parameter that is a record with a simplified syntax for defining its fields. We make use of this new feature for obtaining a simplified class definition syntax illustrated by the following example:

```
class Book {
  // using a single record parameter with ES6 function parameter destructuring
  constructor ({isbn, title, publicationDate, publisher_id, ...}) {
    this.isbn = isbn;
    this.title = title;
    this.publicationDate = publicationDate;
    if (publisher_id) this.publisher_id = publisher_id;
    ...
  }
  ...
}
```

When creating a new object, the constructor function needs to have a parameter for allowing to assign a suitable value to the reference property. Notice that the name of a publisher is used as an ID reference, since it is the standard ID of the Publisher class.

3.2. Make a JS Class Model

The starting point for making a JS class model is an OO class model like the one shown in **Figure 1-10.** An OO class model for `Publisher` and `Book`.

We now show how to derive a JS class model from this OO class model in four steps. For each class in the OO class model:

1. Add a «get/set» stereotype to all (non-derived) single-valued properties, implying that they have implicit getters and setters. Recall that in the setter, the corresponding (non-asynchronous) check operation is invoked and the property is only set, if the check does not detect any constraint violation.

2. Create a **check** operation for each (non-derived) property in order to have a central place for implementing ***property constraints***. For a standard ID attribute (such as `Book::isbn`), three or Four check operations are needed:

   - A basic check operation, like `checkIsbn`, for checking all syntactic constraints, but not the *mandatory value* and the *uniqueness constraints*.

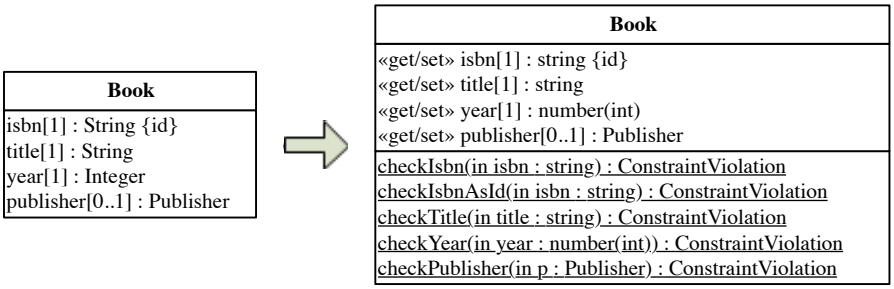
   - A standard ID check (asynchronous) operation, like `checkIsbnAsId`, for checking on Firestore the *mandatory value* and *uniqueness constraints* that are implied by a standard ID attribute.

   - Another standard ID check (asynchronous) operation, like `checkIsbnAsIdRef`, for checking on Firestore the implied *referential integrity constraints* that are implied by a standard ID attribute that refers to an object that already exists; and possibly also a *mandatory value constraint*, if the property is mandatory.

   - If other classes have a reference property that references the class under consideration, add an *ID reference check*

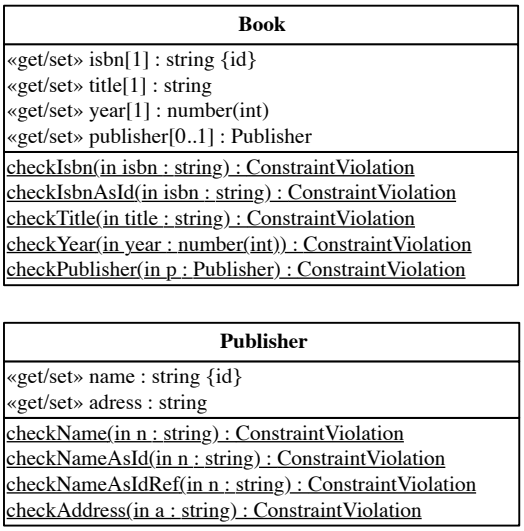
operation for checking the *referential integrity constraint* imposed on *ID reference* (or *foreign key*) attributes. For instance, since the `Book::publisher` property references `Publisher` objects, we need a `checkNameAsIdRef` (asynchronous) operation in the `Publisher` class.

This leads to the following JS class model for `Book`, where the class-level ('static') methods are shown underlined:



We have to perform a similar transformation also for the class `Publisher`. This gives us the complete JS class model derived from the above OO class model, as depicted in the following class diagram.

Figure 3-1. *A JS class model defining the classes Book and Publisher*



3.3. New Issues

Compared to the validation and enumeration apps discussed in Part 2 and Part 3 of our tutorial, we have to deal with a number of new technical issues:

1. In the model code we now have to take care of ***reference properties*** that require

   a. maintaining ***referential integrity***, by invoking (non-asynchronous) checkers from the some setters, and (asynchronous) checkers from some database access operations.

   b. choosing and implementing one of the two possible deletion policies discussed in **Section 1.2.** Referential Integrity for managing the corresponding object destruction dependency in the `destroy` method of the property's range class;

   c. converting JS objects to a Firestore records/documents, or vice versa, creating the serialization functions `toFirestore` and `fromFirestore` using the Firestore method `withConverter()`.

2. In the *user interface ("view")* code we now have to take care of

   a. maintaining ***referential integrity***, by invoking (asynchronous and non-asynchronous) checkers after submitting the form content;

b. showing information about associated objects in the *Retrieve/List* use case;

c. allowing to enter an ID reference (foreign key) of the target class object to associate it with a source class object, in its *Create* and *Update* use cases.

3.4. Code the Model

The JS class model can be directly coded for getting the JS model classes of our app.

3.4.1 Summary

Code each class of the JS class model as a JS class with implicit getters and setters:

1. Code the property check functions in the form of class-level ('static') methods. Take care that all constraints of a property as specified in the JS class model are properly coded in the property checks.
2. Write the code of the serialization functions `toFirestore` and `fromFirestore` within the converter functions, for converting JS objects to a JS Firestore records/documents, or vice versa.
3. Take care of deletion dependencies in the `destroy` method.

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

3.4.2 Code each model class as a JS class

Each class C of the JS class model is coded as a JS class with the same name C and a constructor having a single record parameter, which specifies a field for each (non-derived) property of the class. The range of these properties can be indicated in a comment. In the case of a reference property the range is another model class.

In the constructor body, we assign the fields of the record parameter to corresponding properties. These property assignments invoke the corresponding *setter* methods.

For instance, the `Publisher` class from the JS class model is coded in the following way:

```
class Publisher {
  constructor ({name, address}) {
    this.name = name; // string
    this.address = address; // string
  }
  ...
};
```

Since the setters may throw constraint violation exceptions, the constructor function, and any setter, should be called in a try-catch block where the catch clause takes care of logging suitable error messages.

The Book class from the JS class model is coded in a similar way:

```
class Book {
  constructor ({isbn, title, publicationDate, publisher_id, authorIdRefs}) {
    this.isbn = isbn;
    this.title = title;
    this.publicationDate = publicationDate;
    this.authorIdRefs = authorIdRefs;
    if (publisher_id) this.publisher_id = publisher_id;
  };
  ...
}
```

3.4.3 Code the property checks

Take care that all constraints of a property as specified in the JS class model are properly coded in its check function, as explained in Part 2 of tour tutorial . Recall that constraint violation (or validation error) classes are defined in the module file `lib/errorTypes.mjs`.

For instance, for the `Publisher.checkName` function we obtain the following code:

```
class Publisher {
  ...
  static checkName( n) {
    if (!n) {
      return new NoConstraintViolation(); // not mandatory
    } else {
      if (typeof n !== "string" || n.trim() === "") {
        return new RangeConstraintViolation(
          "The name must be a non-empty string!");
      } else return new NoConstraintViolation();
    }
  };
  ...
}
```

Notice that, since the name attribute is the standard ID attribute of `Publisher`, we only check syntactic constraints in `checkName`, and check the mandatory value and uniqueness constraints in `checkNameAsId`, by querying the table/collection "publishers" and checking if it exists, and invoking `checkName`:

```
static async checkNameAsId( n) {
  let validationResult = Publisher.checkName( n);
  if ((validationResult instanceof NoConstraintViolation)) {
    if (!n) {
      return new MandatoryValueConstraintViolation(
        "A publisher name is required!");
    } else {
      const publisherDocSn = await getDoc( fsDoc( fsDb, "publishers", n));
      if (publisherDocSn.exists()) {
        validationResult = new UniquenessConstraintViolation(
          "There is already a publisher record with this name!");
      } else validationResult = new NoConstraintViolation();
    }
  }
  return validationResult;
};
```

If we have to deal with ID references (foreign keys) in other classes, we need to provide a further check function, called `checkNameAsIdRef`, for checking the referential integrity constraint, again querying the table/collection and checking its existence:

```
static async checkNameAsIdRef( n) {
  let validationResult = Publisher.checkName( n);
  if ((validationResult instanceof NoConstraintViolation) && n) {
    const publisherDocSn = await getDoc( fsDoc( fsDb, "publishers", n));
    if (!publisherDocSn.exists()) {
      validationResult = new ReferentialIntegrityConstraintViolation(
        "There is no publisher record with this name!");
    }
```

```
  }
  return validationResult;
};
```

Again, it queries the table/collection "publishers", with the given name `n`, in conjunction with a check of an existing publisher record with the given name `n`, and then creates a `validationResult` object as an instance of the exception class `ReferentialIntegrityConstraintViolation`.

In `Book.add` and `Book.update` functions, checks are invoked to evaluate referential integrity constraints for the associated Publisher, and then creates a `validationResult` object as an instance of the exception class *ReferentialIntegrityConstraintViolation*. The following code shows how a property checker is invoked in the `Book.add` function, similarly as in the `Book.update` function.

```
Book.add = async function (slots) {
  ..
  try {
    // validate data by creating Book instance
    book = new Book( slots);
    ...
    validationResult = await Publisher.checkNameAsIdRef( book.publisher_id);
    if (!validationResult instanceof NoConstraintViolation) throw validationResult;
    ...
  }
  ...
}
```

3.4.4 Choose and implement a deletion policy

For any reference property, we have to choose and implement one of the two possible deletion policies discussed in **Section 1.2.** Referential Integrity for managing the corresponding object destruction dependency in the `destroy` method of the property's range class. In our case, when deleting a record of a publisher `p`, we have to choose between

1. deleting all records of books published by p (*Existential Dependency*);
2. dropping the reference to p from all books published by p (*Existential Independence*).

Assuming that books do not existentially depend on their publishers, we choose the second option. This is shown in the following code of the `Publisher.destroy` method where a query in the table/collection "books" retrieves all objects of Book type associated with an object of type Publisher of concern, and then the specific property-value slot `publisher_id` is removed with the Firestore methods `update()` and `deleteField()`. Finally the publisher record is dropped with the Firestore method `delete()`. Notice that for preserving reference integrity, both write operations are invoked in a ***write transaction*** operation, explained in **Section 2.1.** Read and Write Transactions.

```
Publisher.destroy = async function (name) {
  const booksCollRef = fsColl( fsDb, "books"),
    q = fsQuery( booksCollRef, where("publisher_id", "==", name)),
    publisherDocRef = fsDoc( fsColl( fsDb, "publishers"), name);
  try {
    const bookQrySns = (await getDocs( q)),
      batch = writeBatch( fsDb); // initiate batch write
    // iterate and delete associations with book records
    await Promise.all( bookQrySns.docs.map( d => {
      batch.update( fsDoc( booksCollRef, d.id), {
        publisher_id: deleteField()
      });
```

```
    }));
    batch.delete( publisherDocRef); // delete publisher record
    batch.commit(); // finish batch write
    console.log(`Publisher record "${name}" deleted!`);
  } catch (e) {
    console.error(`Error deleting publisher record: ${e}`);
  }
};
```

Notice that the deletion of all references from the publisher is performed in an asynchronous scan through all objects of Book type in Cloud Firestore, which is inefficient when there are many of them. It would be much more efficient when each publisher object would hold a list of references to all books published by this publisher. Creating and maintaining such a list would make the association between books and their publisher *bidirectional*.

3.4.5 PublicationDate Checker

Whenever a Book object is instantiated, a setter invokes the `Book.checkPublicationDate` checker that evaluates constraint violations:

```
static checkPublicationDate (publicationDate) {
    const YEAR_FIRST_BOOK = 1459,
      y = new Date( publicationDate).getFullYear();
    if (!publicationDate) {
      return new MandatoryValueConstraintViolation(
        "A value for the publication date must be provided!");
    } else if (!(typeof publicationDate === "string" &&
      /\d{4}-(0\d|1[0-2])-([0-2]\d|3[0-1])/.test( publicationDate) &&
      !isNaN( Date.parse( publicationDate)))) {
      return new PatternConstraintViolation(
        'The publication date is not well formed');
    } else if (y < YEAR_FIRST_BOOK || y > nextYear()) {
      return new IntervalConstraintViolation(
        `The value of year must be between ${YEAR_FIRST_BOOK} and next year!`);
    } else {
      return new NoConstraintViolation();
    }
  };
  set publicationDate( d) {
    let validationResult = Book.checkPublicationDate( d);
    if (validationResult instanceof NoConstraintViolation) this._publicationDate = d;
    else throw validationResult;
  };
```

3.4.6 Code the serialization functions

Also, when the the `Book.converter` function is invoked for reading or writing data we use the Firestore function `Timestamp()` with its method `fromDate()`, and the utility function `date2IsoDateString()` that we implemented in Firestore Timestamp Data Type, in Part 2, like in the following code:

```
Book.converter = {
  toFirestore: function ( book) {
    const data = {
      isbn: book.isbn,
      title: book.title,
      publicationDate: Timestamp.fromDate( new Date( book.publicationDate)),
      authorIdRefs: book.authorIdRefs
```

```
    };
    if (book.publisher_id) data.publisher_id = book.publisher_id;
    return data;
  },
  fromFirestore: function (snapshot, options) {
    const book = snapshot.data( options),
      data = {
      isbn: book.isbn,
      title: book.title,
      publicationDate: date2IsoDateString( book.publicationDate.toDate()),
      authorIdRefs: book.authorIdRefs
    };
    if (book.publisher_id) data.publisher_id = book.publisher_id;
    return new Book( data);
  },
};
```

The serialization function `Publisher.converter` is implemented like this:

```
Publisher.converter = {
  toFirestore: function (publisher) { // setter
    return {
      name: publisher.name,
      address: publisher.address
    };
  },
  fromFirestore: function (snapshot, options) {
    const data = snapshot.data(options);
    return new Publisher( data);
  },
};
```

3.5. Code the View

The user interface (UI) consists of a start page `index.html` that allows navigating to data management UI pages, one for each object type (in our example, `books.html` and `publishers.html`), and one data management UI code file for each object type (in our example, `books.mjs` and `publishers.mjs`). Each data management UI page contains 5 sections: a *Manage* section, like *Manage books*, with a menu for choosing a CRUD use case, and a section for each CRUD use case, like *Retrieve/list all books*, *Create book*, *Update book* and *Delete book*, such that only one of them is displayed at any time (by setting `true` the HTML attribute `hidden` for all others).

Each UI code file for managing the data of an object type O has the following parts (code blocks):

1. Import classes, datatypes and utility procedures.
2. Setup and handle UI Access Control.
3. Declare variables for accessing UI elements.
4. Set up general use-case-independent event listeners, like back buttons and neutralization of form submission.
5. *Retrieve O*: add an event listener for the menu item *Retrieve all* in the Manage UI for creating, and activating
    a. the table view in the Retrieve UI,
    b. the `select` element used to choose the listing order, based in attributes (implemented only for objects type `Book`),

c. the pagination controls to move back and forward (implemented only for objects type `Book`).

6. *Create O*: add event listeners

   a. for the menu item *Create* in the Manage UI,

   b. for responsive constraint validation for ISBN `input` field,

   c. for the *Create* button for creating a new O record.

7. *Update O*: add event listeners

   a. for the menu item *Update* in the Manage UI,

   b. for responsive constraint validation for ISBN `input` field,

   c. for "`blur`" state of ISBN `input` field, retrieving the O record to be updated, filling out the Update UI's fields with the property values, including the widget,

   d. for the `Update` button for updating an existing O record.

8. *Delete O*: add event listeners

   a. for the menu item *Delete* in the Manage UI,

   b. for responsive constraint validation for ISBN `input` field,

   c. for the *Delete* button for deleting an existing O record.

For instance, in `books.mjs`, for managing book data, we have the following first three code blocks:

1. Import classes, datatypes and utility procedures:

```
import { handleAuthentication } from "./accessControl.mjs";
import Book from "../m/Book.mjs";
import Publisher from "../m/Publisher.mjs";
import Author from "../m/Author.mjs";
import { createListFromMap, hideProgressBar, showProgressBar, createMultiSelectionWidget
  from "../../lib/util.mjs";
...
```

2. Declare variables for accessing main UI elements:

```
...
const bookMSectionEl = document.getElementById("Book-M"),
  bookRSectionEl = document.getElementById("Book-R"),
  bookCSectionEl = document.getElementById("Book-C"),
  bookUSectionEl = document.getElementById("Book-U"),
  bookDSectionEl = document.getElementById("Book-D");
...
```

3. Set up general use-case-independent *event listeners*:

```
...
// set up back-to-menu buttons for all use cases
for (const btn of document.querySelectorAll("button.back-to-menu")) {
  btn.addEventListener("click", refreshManageDataUI);
```

```
}
// neutralize the submit event for all use cases
for (const frm of document.querySelectorAll("section > form")) {
  frm.addEventListener("submit", function (e) {
    e.preventDefault();
  });
}
...
```

In `books.html`, there is the following menu for choosing a CRUD operation:

```
...
<section id="Book-M" class="UI-Page" hidden="hidden">
  <h1>Manage Book Data</h1>
  <ul class="menu">
    <li><button type="button" id="RetrieveAndListAll">Retrieve/list all book records</butt
    <li><button type="button" id="Create" disabled="disabled">Create a new book record</bu
    <li><button type="button" id="Update" disabled="disabled">Update a book record</button
    <li><button type="button" id="Delete" disabled="disabled">Delete a book record</button
  </ul>
  <div class="button"><a href="index.html">« Back to Main menu</a></div>
</section>
...
```

The event listener for each of these CRUD buttons sets up the corresponding UI. For instance, for *Create*, we have the following code in `books.mjs`:

```
...
const createFormEl = document.querySelector("section#Book-C > form"),
  createAuthorWidget = createFormEl.querySelector(".MultiSelectionWidget");
await createMultiSelectionWidget (createFormEl, [], "authors",
  "id", "authorId", Author.checkAuthorIdAsIdRef, Author.retrieve);
document.getElementById("Create").addEventListener("click", async function () {
  createFormEl.reset();
  bookMSectionEl.hidden = true;
  bookCSectionEl.hidden = false;
});
...
```

3.5.1 Setting up the Retrieve/List All user interface

In our example, we have only one reference property, `Book::publisher`, which is functional and optional. For showing information about the publisher of a book in the view table of the *Retrieve/list all* user interface, the corresponding cell in the HTML table is filled with the name of the publisher (`publisher_id`), if there is any (in `books.mjs`). The `createBlock()` function invokes the `Book.retrieveBlock` method to retrieve book records/documents from an specific cursor (`startAt`) and order. Missing code is for handling order `select` and `button` elements for paginating the result, which is discussed later:

```
...
async function createBlock (startAt) {
  tableBodyEl.innerHTML = "";
  showProgressBar( "Book-R");
  const bookRecs = await Book.retrieveBlock({"order": order, "cursor": startAt});
  if (bookRecs.length) {
    // set page references for current (cursor) page
```

```
    cursor = bookRecs[0][order];
    // set next startAt page reference, if not next page, assign 'null' value
    nextPageRef = (bookRecs.length < 21) ? null : bookRecs[bookRecs.length - 1][order];
    for (const bookRec of bookRecs) {
      const listEl = createListFromMap( bookRec.authorIdRefs, "name");
      const row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = bookRec.isbn;
      row.insertCell(-1).textContent = bookRec.title;
      row.insertCell(-1).textContent = bookRec.publicationDate;
      row.insertCell(-1).appendChild( listEl);
      row.insertCell(-1).textContent = bookRec.publisher_id;
    }
  }
  hideProgressBar( "Book-R");
}
...
```

For a multi-valued reference property, the table cell would have to be filled with a list of ID of all associated objects. Later we will describe the implementation of how we paginate, order and limit retrieved book records/documents in **Section 5.1.** Paginate, Order and Limit Data while Querying Firestore.

3.5.2 Setting up the Create and Update user interfaces

For allowing to associate objects in the *Create* and *Update* user interfaces, an `input` element is used for entering the document ID. The `input` element is defined in the `books.html` view file:

```
...
<section id="Book-C" class="UI-Page" hidden="hidden">
  <h1>Create a new Book Record</h1>
  <form>
    ...
    <div class="field">
      <label>Publisher ID: <input type="text" name="publisher"/></label>
    </div>
    ...
  </form>
</section>
...
```

Both *Create* and *Update* operations invoke the referential integrity constraints checker using its ID reference (ISBN) on user input via an event handler. Only for the Update operation, whenever a value is entered and the user changes to `blur` (leaves the `input` field), an event handler retrieves the source object (a book) from the database, and renders the form fields, populating them with data of the retrieved object. We have the following code for the *Update* use case in the `books.mjs` view file:

```
...
updateFormEl["isbn"].addEventListener("input", async function () {
  const responseValidation = await Book.checkIsbnAsIdRef( updateFormEl["isbn"].value);
  if (responseValidation) updateFormEl["isbn"].setCustomValidity( responseValidation.messa
  if (!updateFormEl["isbn"].value) {
    updateFormEl.reset();
    updateAuthorWidget.innerHTML = "";
  }
});
updateFormEl["isbn"].addEventListener("blur", async function () {
  if (updateFormEl["isbn"].checkValidity() && updateFormEl["isbn"].value) {
```

```
    const bookRec = await Book.retrieve( updateFormEl["isbn"].value);
    ...
    if (bookRec.publisher_id) updateFormEl["publisher"].value = bookRec.publisher_id;
    ...
  } else {
    updateFormEl.reset();
  }
});
...
```

When the form is submitted, a respective referential integrity checkers are invoked:

```
...
let responseValidation = await Publisher.checkNameAsIdRef( slots.publisher_id);
createFormEl["publisher"].setCustomValidity( responseValidation.message);
...
```

When the user pushes the *Update* button, all form control values, including the value of the `select` field, are copied to a slots record, which is used as the argument for invoking the `update` method after all form fields have been checked for validity, as shown in the following program listing:

```
...
updateFormEl["commit"].addEventListener("click", async function () {
  if (!updateFormEl["isbn"].value) return;
  const addedAuthorsListEl = updateAuthorWidget.children[1],
    slots = {
      isbn: updateFormEl["isbn"].value,
      title: updateFormEl["title"].value,
      publicationDate: updateFormEl["publicationDate"].value,
      publisher_id: updateFormEl["publisher"].value,
    };
  // check all input fields and show error messages
  /* SIMPLIFIED CODE: no before-submit validation of title */
  ...
  const responseValidation = await Publisher.checkNameAsIdRef( slots.publisher_id);
  updateFormEl["publisher"].setCustomValidity( responseValidation.message);
  ...
  // commit the update only if all form field values are valid
  if (updateFormEl.checkValidity()) {
    showProgressBar( "Book-U");
    await Book.update( slots);
    // drop widget content
    updateFormEl.reset();
    updateAuthorWidget.innerHTML = "";
    hideProgressBar( "Book-U");
  }
});
...
```

The code for setting up the *Create* user interface is similar.

Chapter 4. Implementing Unidirectional Non-Functional Associations with JS and Firebase

A unidirectional non-functional association is either *one-to-many* or *many-to-many*. In both cases such an association is represented, or implemented, with the help of a multi-valued reference property.

In this chapter, we show

1. how to derive a JS class model from an OO class model with multi-valued reference properties representing unidirectional non-functional associations in Firestore,
2. how to code the JS class model in the form of JavaScript model classes,
3. how to write the view code based on the model code.

4.1. Implementing Multi-Valued Reference Properties

A multi-valued reference property, such as the property `Book::authors`, allows storing a collection of references to objects of some type, such as `Author` objects. When creating a new object of type `Book`, the constructor function needs to have a parameter for providing a suitable value for this property, being ID references, as shown in the following example:

```
class Book {
  // using a single record parameter with ES6 function parameter destructuring
  constructor ({isbn, title, publicationDate, publisher_id, authorIdRefs}) {
    this.isbn = isbn;
    this.title = title;
    this.publicationDate = publicationDate;
    this.authorIdRefs = authorIdRefs;
    if (publisher_id) this.publisher_id = publisher_id;
  }
  ...
}
```

In JS, a collection-valued reference property can be implemented in two ways:

1. having an array list (a JS array) of object references as its value,
2. having a map as its value, such that the values of the object's standard ID attribute are the keys, which are mapped to internal JS object references.

We prefer using array lists for implementing *set-valued* reference properties since they are fully compatible with the *Array* data type on Firestore, with no need of additional data type conversion.

Additionally, each reference property will be stored in the source object (the Book class object), as a map of the source object (the Author class object) storing the values of the properties `id` and `name`. Being a multi-valued reference property, we express the association of multiple authors as an array like:

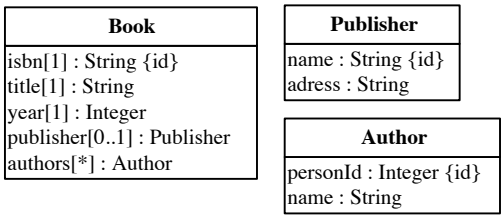
```
[
  {
    "id": 1,
    "name": "Tempkin Curry"
  },
  {
    "id": 2,
    "name": "Sidney Edelman"
  },
  {
    "id": 3,
    "name": "John Taber"
  }
]
```

Our choice has the following advantages:

- ***normalizing*** and ***aggregating*** data will make reading operations more efficient, since when we store a map of the referenced object we are in fact *joining* two tables/collections in the model layer, in the moment a class object is created;
- since read operations happen more frequently than write operations, this ***data duplication*** approach will decrease billing costs in the long term;
- a JS array containing maps, as the described above, can be stored easily without any ***special data transformation*** into Firestore, using the data types *Array* and *Map*;
- maintaining a data structure as such, an *Array* containing *Maps* in Firestore, can only be performed "*adding*" and "*deleting*" array elements, but never "*updating*" them, which forces to use the methods arrayRemove and `arrayUnion` sequentially whenever we need to update a reference property; this design of Firestore aims to ***decrease the possibility of having duplicated elements*** in an array.

4.2. Make a JS Class Model

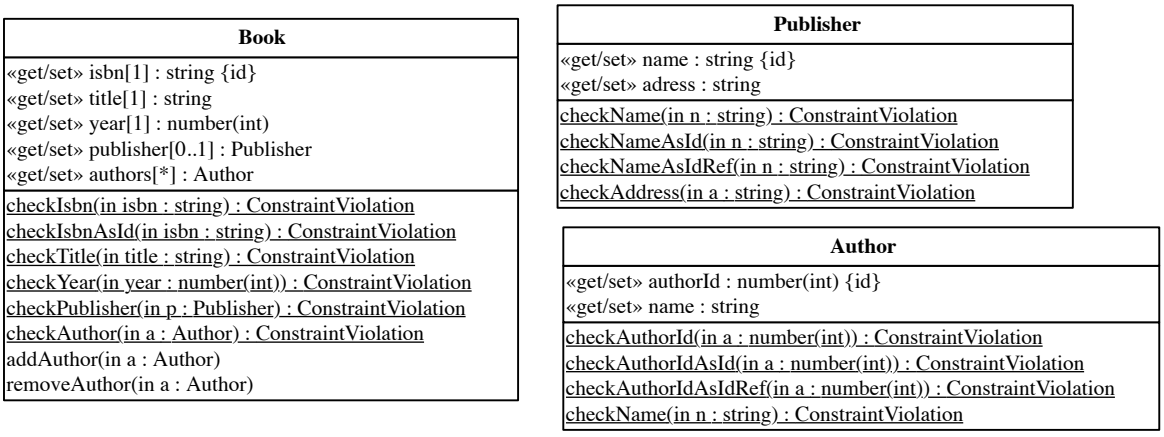
Our starting point for making a JS class model is the following OO class model:



This model contains, in addition to the single-valued reference property `Book::publisher` representing the unidirectional many-to-one association *Book*-**has**-*Publisher*, the multi-valued reference property `Book::authors` representing the unidirectional many-to-many association *Books*-**have**-*Authors*.

For deriving a JS class model from the OO class model we have to follow the same steps as in **Section 1.2.** Referential Integrity and, in addition, we have to take care of multi-valued reference properties, such as `Book::authorIdRefs`, for which we create a class-level check operation, such as `Author.checkAuthorIdAsIdRef`, which is responsible for checking the corresponding referential integrity constraint for the references to be added to the property's collection.

This leads to the following JS class model:



4.3. New issues

Compared to dealing with a functional association, as discussed in the previous chapter, we now have to deal with the following new technical issues:

1. In the *model* code we now have to take care of ***multi-valued reference properties*** that require implementing

   a. normalized data of reference properties stored in the source object as foreign key, for which we use batch write transactions for preserving reference integrity;

   b. maintaining referential integrity, by invoking (non-asynchronous) checkers from the some setters, and (asynchronous) checkers from some database access operations.

   c. an ***add*** and a ***remove*** method, such as `addAuthor` and `removeAuthor`, as well as a ***setter*** method for assigning a set of object references with the help of the add method, possibly converting ID references to object references; all three methods may need to check *cardinality constraints*, if there are any;

   d. a class-level ***check*** operation, such as the `Author.checkAuthorIdAsIdRef` method of the property's range class for checking the property's implicit *referential integrity constraint*;

   e. function for converting *JS objects* to a Firestore records/documents, or vice versa, creating the serialization functions `toFirestore` and `fromFirestore` using the Firestore method `withConverter()`.

2. In the user interface ("*view*") code we now have to take care of

   a. maintaining referential integrity, by invoking (asynchronous and non-asynchronous) checkers after submitting the form content;

   b. showing information about a set of associated objects in the property's column of the table view of the *Retrieve/list all* use case; showing normalized data of the associated target objects as a list, possibly combined with corresponding names; alternatively, HTML lists can be rendered in the property's table data cells;

   c. allowing to select a set of associated objects from a list of all existing instances of the property's range class in the *Create* and *Update* use cases.

   The last issue, allowing to select a set of associated objects from a list of all instances of some class, can, in general, not be solved with the help of an HTML multiple-select form control because of its poor usability and the involved cost of retrieving data from the Firestore database to fill the `option` elements; consequently an alternative ***multi-selection widget*** has to be used.

4.4. Code the Model

Notice that, for simplicity, we do not include the code for all constraint validation checks shown in the JS class model in the code of the example app in `Book.mjs`.

4.4.1 Summary

Code each class of the JS class model as an ES6 class with implicit getters and setters:

1. Code the property checks in the form of class-level ('static') methods. Take care that all constraints of a property as specified in the JS class model are properly coded in the property checks.
2. For each single-valued property, code the specified getter and setter such that in some setters, the corresponding property (non-asynchronous) check is invoked and the property is only set, if the check does not detect any constraint violation; other (asynchronous) checks are invoked on the database operation (*Create* or *Update*), after form submission.
3. For each multi-valued property, code its add and remove operations as (instance-level).
4. Write the code of the serialization functions `toFirestore` and `fromFirestore` within the converter functions, for converting JS objects to a JS Firestore records/documents, or vice versa.
5. Take care of deletion dependencies in the `destroy` method.

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

4.4.2 Code model class as a JS class

In `m/Author.mjs` the model class is coded as a JS class with the same name C and a constructor having a single record parameter, which specifies a field for each (non-derived) property of the class. In the constructor body, we assign the fields of the record parameter to corresponding properties. These property assignments invoke the corresponding setter methods, coded in the following way:

```
class Author {
  // using a single record parameter with ES6 function parameter destructuring
  constructor ({authorId, name}) {
    // assign properties by invoking implicit setters
    this.authorId = authorId;
    this.name = name;
  };
  ...
}
```

4.4.3 Code the add and remove operations

For the multi-valued reference property `Book::authors`, in `m/Book.mjs`, we need to code the operations `addAuthor` and `removeAuthor`. Both operations accept one parameter denoting an author either by ID reference (a map as the author ID). The code of `addAuthor` is as follows:

```
addAuthor( a) {
  this._authorIdRefs.push( a);
};
```

In the `removeAuthor` method, the author reference entry in the array `this._authors` is deleted using the JS `filter()` method:

```
removeAuthor( a) {
  this._authorIdRefs = this._authorIdRefs.filter( d => d.id !== a.id);
};
```

4.4.4 Choose and implement a deletion policy

For the reference property `Book::authors`, we have to choose and implement a deletion policy in the `destroy` method of the `Author` class. We have to choose between

1. deleting all books (co-)authored by the deleted author (reflecting the logic of *Existential Dependency*);
2. dropping from all books (co-)authored by the deleted author the reference to the deleted author (reflecting the logic of *Existential Independence*).

For simplicity, we go for the second option. This is shown in the following code of the static `Author.destroy` method where we use a write transaction. In general, read/write transactions and write transactions assure maintaining reference integrity constrains for both, object creation and object destruction dependencies. In this specific case we use a write transaction to take care of the destruction of an `author` record that represents a many-to-many association *Book*-**has**-*Author*, so we will eliminate multiple associations while preserving reference integrity.

The following procedure exemplifies how to eliminate unidirectional associations with write transactions (batched writes):

- **Create documents/collections references and snapshots involved in the batch write**. A *QuerySnapshot* object is created using the `where()` method with a condition to query all book documents that include (`array-contains`) the parameter `authorRef` (a map) in their multi-valued field `authorIdRefs` (data type Array). Notice here that queries cannot be invoked inside read/write transactions or write transactions. Finally a document reference (`authorRef`) for the author is customized, since queries for maps inside Array fields in Firestore demands that we provide the whole content

of the map.

```
...
Author.destroy = async function (slots) {
  const booksCollRef = fsColl( fsDb, "books"),
    authorsCollRef = fsColl( fsDb, "authors");
  try {
    const authorRef = {id: parseInt( slots.authorId), name: slots.name},
      q = fsQuery( booksCollRef, where("authorIdRefs", "array-contains", authorRef)),
      authorDocRef = fsDoc( authorsCollRef, String( slots.authorId)),
      bookQrySns = (await getDocs( q)),
      ...
```

- **Open batch write and update all Book records associated to Author with a write operation**. With the `batch()` method we initialize a batch write, creating the `batch` object. Then, using a `Promise.all` method asychronously, we iterate the query result (`bookDocSns`) that contains all book records associated to the author to be destroyed. Then we remove author references using the `update()` method in the multi-valued reference property, `authorIdRefs`. Using the `arrayRemove()` method, only an specific object is removed from the Array data type. Notice that the only way to remove or update an specific array element in a Firestore database is using respectively the methods `arrayUnion()` and `arrayRemove()` providing the map with exactly the same values that works a a reference.

```
...
    batch = writeBatch( fsDb); // initiate batch write
  // iterate and delete associations in book records
  await Promise.all( bookQrySns.docs.map( d => {
    const bookDocRef = fsDoc(booksCollRef, d.id);
    batch.update(bookDocRef, {authorIdRefs: arrayRemove( authorRef)});
  }));
...
```

- **Remove Author with a write operation and close batch write**. Using the `delete()` method to remove author record, and then we commit the whole atomic operation with the method `commit()`.

```
...
batch.delete( authorDocRef); // delete author record
batch.commit(); // commit batch write
...
```

Notice that the `batch` object is attached to every part of the amotmic operation. The complete function to destroy an `Author` record with unidirectional associations:

```
Author.destroy = async function (slots) {
  const booksCollRef = fsColl( fsDb, "books"),
    authorsCollRef = fsColl( fsDb, "authors");
  try {
    const authorRef = {id: parseInt( slots.authorId), name: slots.name},
      q = fsQuery( booksCollRef, where("authorIdRefs", "array-contains", authorRef)),
      authorDocRef = fsDoc( authorsCollRef, String( slots.authorId)),
      bookQrySns = (await getDocs( q)),
      batch = writeBatch( fsDb); // initiate batch write
    // iterate and delete associations in book records
    await Promise.all( bookQrySns.docs.map( d => {
      const bookDocRef = fsDoc(booksCollRef, d.id);
      batch.update(bookDocRef, {authorIdRefs: arrayRemove( authorRef)});
    }));
```

```
    batch.delete( authorDocRef); // delete author record
    batch.commit(); // commit batch write
    console.log(`Author record ${slots.authorId} deleted!`);
  } catch (e) {
    console.error(`Error deleting author record: ${e}`);
  }
};
```

4.4.5 Serialization functions

Using the Firestore withConverter() method for reading (`fromFirestore`) and writing (`toFirestore`) data in Firestore.

```
Author.converter = {
  toFirestore: function ( author) {
    return {
      authorId: parseInt( author.authorId),
      name: author.name
    };
  },
  fromFirestore: function (snapshot, options) {
    const data = snapshot.data( options);
    return new Author( data);
  },
};
```

4.5. Code the View

4.5.1 Setting up the Retrieve/List All user interface

For showing information about the authors of a book in the view table of the *Retrieve/List All* user interface, the corresponding cell in the HTML table is filled (in `v/books.mjs`) with a list of the names of all authors with the help of the utility function `createListFromMap`:

```
...
async function createBlock (startAt) {
  tableBodyEl.innerHTML = "";
  showProgressBar( "Book-R");
  const bookRecs = await Book.retrieveBlock({"order": order, "cursor": startAt});
  if (bookRecs.length) {
    // set page references for current (cursor) page
    cursor = bookRecs[0][order];
    // set next startAt page reference, if not next page, assign 'null' value
    nextPageRef = (bookRecs.length < 21) ? null : bookRecs[bookRecs.length - 1][order];
    for (const bookRec of bookRecs) {
      const authListEl = createListFromMap( bookRec.authorIdRefs, "name");
      const row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = bookRec.isbn;
      row.insertCell(-1).textContent = bookRec.title;
      row.insertCell(-1).textContent = bookRec.publicationDate;
      row.insertCell(-1).appendChild( authListEl);
      row.insertCell(-1).textContent = bookRec.publisher_id;
    }
  }
  hideProgressBar( "Book-R");
}
...
```

The utility function `createListFromMap` and its auxiliary function `fillListFromMap` (in `lib/util.mjs`) have the following code:

```
function createListFromMap(et, displayProp) {
  const listEl = document.createElement("ul");
  fillListFromMap( listEl, et, displayProp);
  return listEl;
}
```

```
function fillListFromMap(listEl, et, displayProp) {
  const keys = Object.keys( et);
  // delete old contents
  listEl.innerHTML = "";
  // create list items from object property values
  for (const key of keys) {
    const listItemEl = document.createElement("li");
    listItemEl.textContent = et[key][displayProp];
    listEl.appendChild( listItemEl);
  }
}
```

Later we will describe the implementation of how we paginate, order and limit retrieved book records/documents in **Section 5.1.** Paginate, Order and Limit Data while Querying Firestore.

4.5.2 Selecting associated objects in the Create and Update user interface

Unfortunately, HTML's multiple-select control is not really usable for displaying and allowing to maintain the set of associated authors in realistic use cases where we have several hundreds or thousands of authors, because the way it renders the choice in a large list to be scrolled is visually too scattered, violating general usability requirements. So we have to use a special ***multi-selection widget*** that allows to add (and remove) objects to (and from) a list of associated objects, as discussed in **Section 2.2.** Rendering Reference Properties in the User Interface. In order to show how this widget can replace the multiple-selection list discussed in the previous section, we use it now in the *Create* and *Update* use cases.

For allowing to maintain the set of authors associated with the currently edited book in the *Create* and *Update* use cases, a *multi-selection widget* as shown in the HTML code below, is populated with the instances of the `Author` class.

```
...
<section id="Book-C" class="UI-Page" hidden="hidden">
  <h1>Create a Book Record</h1>
  <form>
    <div class="field">
      <label>Enter ISBN: <input type="text" name="isbn"/></label>
    </div>
    ...
    <div class="MultiSelectionWidget"></div>
    ...
  </form>
</section>
...
```

Unlike *Update*, where the ***multi-selection widget*** is loaded whenever the user selects a book to be modified, in *Create* the widget loaded once, when the user interface is rendered. There is no other main difference between both use cases, so we will describe Update in the lines below.

The *Update* user interface is set up (in a section of `v/books.mjs`) by an event handler whenever the user changes to `blur`,

leaves the `input` field `isbn`, populating the form with data of the book to be updated: the output field `isbn` and the input fields `title` and `publicationDate`, as well as the input field for updating the publisher, are assigned corresponding values from the chosen book, and the the multi-valued property of associated authors, which is set up with the help of the utility function `createMultiSelectionWidget`.

The multi-selection widget used for preserving referential integrity in this app is used in *Create* and *Update* use cases, and in both cases it is invoked. Notice that whenever a new target class object is associated to source target, it is evaluated referential integrity contrainsts before being retrieved, so both functions should be provided as parameters, like in the following code:

```
...
updateFormEl["isbn"].addEventListener("blur", async function () {
  if (updateFormEl["isbn"].checkValidity() && updateFormEl["isbn"].value) {
    const bookRec = await Book.retrieve( updateFormEl["isbn"].value);
    updateFormEl["isbn"].value = bookRec.isbn;
    updateFormEl["title"].value = bookRec.title;
    updateFormEl["publicationDate"].value = bookRec.publicationDate;
    if (bookRec.publisher_id) updateFormEl["publisher"].value = bookRec.publisher_id;
    updateAuthorWidget.innerHTML = "";
    await createMultiSelectionWidget (updateFormEl, bookRec.authorIdRefs,
      "authors", "id", "authorId", Author.checkAuthorIdAsIdRef, Author.retrieve);
  } else {
    updateFormEl.reset();
  }
});
...
```

When the user, after updating some values, finally clicks the *Update* button, all form control values, including the value of the `input` field for assigning a publisher, are copied to corresponding slots in a slots record variable, which is used as the argument for invoking the `Book.update` method after all values have been checked for validity.

Before invoking update, we invoke every involved integrity constraint checker, avoiding invoking checkers of authors to be removed, while a list of ID references to authors to be added, and another list of ID references to authors to be removed, are created (in the `authorIdRefsToAdd` and `authorIdRefsToRemove` slots), from the updates that have been recorded, using the *multi-selection widget*, in the associated authors with "added" and "removed" as values of the corresponding list item's class attribute, as shown in the following program listing:

```
...
// handle Update button click events
updateFormEl["commit"].addEventListener("click", async function () {
  if (!updateFormEl["isbn"].value) return;
  const addedAuthorsListEl = updateAuthorWidget.children[1], // ul
    slots = {
      isbn: updateFormEl["isbn"].value,
      title: updateFormEl["title"].value,
      publicationDate: updateFormEl["publicationDate"].value,
      publisher_id: updateFormEl["publisher"].value,
    };
  // check all input fields and show error messages
  /* SIMPLIFIED CODE: no before-submit validation of title */
  updateFormEl["publicationDate"].setCustomValidity(
    Book.checkPublicationDate( slots.publicationDate).message);
  const responseValidation = await Publisher.checkNameAsIdRef( slots.publisher_id);
  updateFormEl["publisher"].setCustomValidity( responseValidation.message);
  if (addedAuthorsListEl.children.length) {
    // construct authorIdRefs-ToAdd/ToRemove lists
```

```
    const authorIdRefsToAdd=[], authorIdRefsToRemove=[];
    for (const authorItemEl of addedAuthorsListEl.children) {
      if (authorItemEl.classList.contains("added")) {
        const author = JSON.parse(authorItemEl.getAttribute("data-value"));
        const responseValidation = await Author.checkAuthorIdAsIdRef( author.id);
        if (responseValidation.message) {
          updateFormEl["authors"].setCustomValidity( responseValidation.message);
          break;
        } else {
          authorIdRefsToAdd.push( author);
          updateFormEl["authors"].setCustomValidity( "");
        }
      }
      if (authorItemEl.classList.contains("removed")) {
        const author = JSON.parse(authorItemEl.getAttribute("data-value"));
        authorIdRefsToRemove.push( author);
      }
    }
    // if the add/remove list is non-empty, create a corresponding slot
    if (authorIdRefsToRemove.length > 0) {
      slots.authorIdRefsToRemove = authorIdRefsToRemove;
    }
    if (authorIdRefsToAdd.length > 0) {
      slots.authorIdRefsToAdd = authorIdRefsToAdd;
    }
  } else updateFormEl["authors"].setCustomValidity(
    updateFormEl["authors"].value ? "" : "No author selected!");
  // commit the update only if all form field values are valid
  if (updateFormEl.checkValidity()) {
    showProgressBar( "Book-U");
    await Book.update( slots);
    // drop widget content
    updateFormEl.reset();
    updateAuthorWidget.innerHTML = ""; // ul
    hideProgressBar( "Book-U");
  }
});
...
```

Notice how the author reference data in a map is retrieved from the new ("*added*") associated authors and the associated authors to be removed, embedded in the `data-value` attribute in every list item.

You can run the example app from our server and download it as a ZIP archive file.

Chapter 5. Firebase Features

5.1. Paginate, Order and Limit Data while Querying Firestore

In the view layer, the pagination feature that allows browsing a book listing with pages (blocks) of 20 records/documents each is implemented with an order selector element, using a combination of procedures that handle events on user interface elements, and send requests of new data to the model, to later render it in the view. For instance, in the following code we see how the selector element and two button elements (for moving to the "previous" and "next" pages) are handled:

```
/**
 * "Previous" button
 */
```

```
previousBtnEl.addEventListener("click", async function () {
  // locate current page reference in index of page references
  previousPageRef = startAtRefs[startAtRefs.indexOf( cursor) - 1];
  // create new page
  await createBlock( previousPageRef);
  // disable "previous" button if cursor is first page
  if (cursor === startAtRefs[0]) previousBtnEl.disabled = true;
  // enable "next" button if cursor is not last page
  if (cursor !== startAtRefs[startAtRefs.length -1]) nextBtnEl.disabled = false;
});
/**
 *  "Next" button
 */
nextBtnEl.addEventListener("click", async function () {
  await createBlock( nextPageRef);
  // add new page reference if not present in index
  if (!startAtRefs.find( i => i === cursor)) startAtRefs.push( cursor);
  // disable "next" button if cursor is last page
  if (!nextPageRef) nextBtnEl.disabled = true;
  // enable "previous" button if cursor is not first page
  if (cursor !== startAtRefs[0]) previousBtnEl.disabled = false;
});
/**
 * handle order selection events: when an order is selected,
 * populate the list according to the selected order
 */
selectOrderEl.addEventListener("change", async function (e) {
  order = e.target.value;
  startAtRefs = [];
  await createBlock();
  startAtRefs.push( cursor);
  previousBtnEl.disabled = true;
  nextBtnEl.disabled = false;
});
```

On the model layer, the Book.`retrieveBlock` method allows browsing a book listing with pages (blocks) of 20 records/documents each. Notice the use of the orderBy() method in the query, as seen in the following code:

```
Book.retrieveBlock = async function (params) {
  try {
    let booksCollRef = fsColl( fsDb, "books");
    // set limit and order in query
    booksCollRef = fsQuery( booksCollRef, limit( 21));
    if (params.order) booksCollRef = fsQuery( booksCollRef, orderBy( params.order));
    // set pagination "startAt" cursor
    if (params.cursor) {
      if (params.order === "publicationDate")
        booksCollRef = fsQuery( booksCollRef, startAt( Timestamp
          .fromDate( new Date( params.cursor))));
      else booksCollRef = fsQuery( booksCollRef, startAt( params.cursor));
    }
    const bookRecs = (await getDocs( booksCollRef
      .withConverter( Book.converter))).docs.map( d => d.data());
    if (bookRecs.length) {
      console.log(`Block of book records retrieved! (cursor: ${bookRecs[0][params.order]})
    }
    return bookRecs;
```

```
  } catch (e) {
    console.error(`Error retrieving all book records: ${e}`);
  }
};
```

Notice that parameters are passed to invoke the function, defining where the loaded page will start (`params.cursor`), and in which order (`params.order`). Notice that each page request is limited to 21 records/documents, being the first 20 to be showed on the user interface, and the 21th used for defining the cursor of the "next" page.

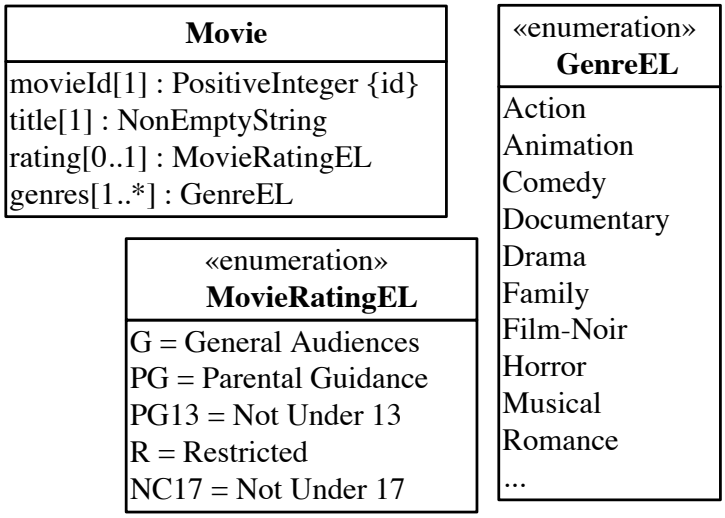
5.2. Points of Attention

We have still included the repetitive code structures (called boilerplate code) in the model layer per class and per property for constraint validation (checks and setters) and per class for the data storage management methods add, update, and destroy. While it is good to write this code a few times for learning app development, you don't want to write it again and again when you work on real projects. For avoiding repetitive boilerplate code, generic forms of these methods are needed, such that they can be reused in all model classes of an app. For instance, the cLASSjs library provides such an approach.

5.3. Practice Project

The purpose of the app to be built is managing information about movies. The app deals with just one object type, `Movie`, and with two enumerations, as depicted in the following class diagram. In the subsequent parts of the tutorial, you will extend this simple app by adding actors and directors as further model classes, and the associations between them.

Figure 5-1. *The object type `Movie` defined together with two enumerations*



First make a list of all the constraints that have been expressed in this model. Then code the app by following the guidance of this tutorial and the Validation Tutorial.

Compared to the practice project of our validation tutorial, two attributes have been added: the optional single-valued enumeration attribute `rating`, and the multi-valued enumeration attribute `genres`.

Following the tutorial, you have to take care of

1. defining the *enumeration data types* `MovieRatingEL` and `GenreEL` with the help of the meta-class `Enumeration`;
2. defining the *single-valued enumeration attribute* `Movie::rating` together with a check and a setter;
3. defining the *multi-valued enumeration attributes* `Movie::genres` together with a check and a setter;
4. extending the methods `Movie.update`, and `Movie.prototype.toString` such that they take care of the added

enumeration attributes.

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

1. adding new table columns in `retrieveAndListAllMovies.html` and suitable form controls (such as *selection lists*, *radio button groups* or *checkbox groups*) in `createMovie.html` and `upateMovie.html`;
2. creating output for the new attributes in the method `v.retrieveAndListAllMovies.setupUserInterface()`;
3. allowing input for the new attributes in the methods `v.createMovie.setupUserInterface()` and `v.upateMovie.setupUserInterface()`.

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

Table 5-1. Table Sample data for movies

Movie ID	Title	Rating	Genres
1	Pulp Fiction	R	Crime, Drama
2	Star Wars	PG	Action, Adventure, Fantasy, Sci-Fi
3	Casablanca	PG	Drama, Film-Noir, Romance, War
4	The Godfather	R	Crime, Drama

Resources

- Transactions and batched writes, Firebase Documentation.
- Paginate data with query cursors, Firebase Documentation.
- Firestore data conversion, Firebase Documentation.
- Perform simple and compound queries in Firestore, Firebase Documentation.