JS/Firebase Web App Tutorial Part 5: Managing Bidirectional Associations

Learn how to manage bidirectional associations between object types, such as the association assigning authors to their books as well as books to their authors 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 5 of our series of six tutorials about model-based development of front-end web applications with plain JavaScript and Firebase. It shows how to build a web app that takes care of the object types `Author`, `Publisher` and `Book` as well as the bidirectional associations between `Book` and `Author` and between `Book` and `Publisher`.

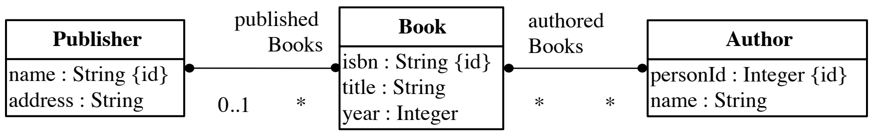
The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). It extends the example app of part 3 by adding code for handling ***derived inverse reference properties***. The other parts of the tutorial are:

- Part 1: Building a **minimal** app.
- Part 2: Handling **constraint validation**.
- Part 3: Dealing with **enumerations**.
- Part 4: Managing **unidirectional associations**, such as the associations between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.
- Part 6: Handling **subtype** (inheritance) relationships between object types.

Chapter 1. Bidirectional Associations

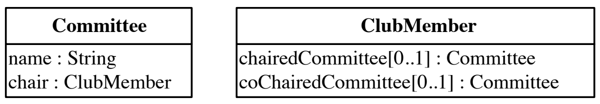
In OO modeling and programming, a bidirectional association is an association that is represented as a pair of mutually inverse reference properties, which allow `navigation´ (object access) in both directions. The model shown in **Figure 1-1.** The Publisher-Book-Author information design model with two bidirectional associations below (about publishers, books and their authors) serves as our running example. Notice that it contains two bidirectional associations, as indicated by the ownership dots at both association ends.

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

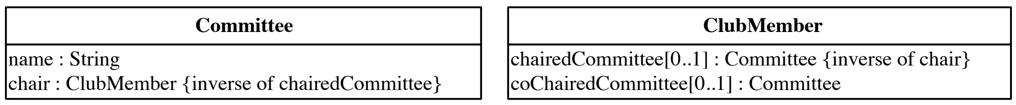


1.1. Inverse Reference Properties

For being able to easily retrieve the committees that are chaired or co-chaired by a club member, we add two reference properties to our *Committee-ClubMember* example model: the property of a club member to be the chair of a committee (`ClubMember::chairedCommittee`) and the property of a club member to be the co-chair of a committee (`ClubMember::coChairedCommittee`). We assume that any club member may chair or co-chair at most one committee (where the disjunction is non-exclusive). So, we get the following model:


Committee
name : String
chair : ClubMember
ClubMember
chairedCommittee[0..1] : Committee
coChairedCommittee[0..1] : Committee

Notice that there is a close correspondence between the two reference properties `Committee::chair` and `ClubMember::chairedCommittee`. They are the inverse of each other: when the club member Tom is the chair of the budget committee, expressed by the tuple ("*budget committee*", "*Tom*"), then the budget committee is the committee chaired by the club member Tom, expressed by the inverse tuple ("*Tom*", "*budget committee*"). For expressing this inverse correspondence in the diagram, we append an inverse property constraint, `inverse of chair`, in curly braces to the declaration of the property `ClubMember::chairedCommittee`, and a similar one to the property `Committee::chair`, as shown in the following diagram:


Committee
name : String
chair : ClubMember {inverse of chairedCommittee}
ClubMember
chairedCommittee[0..1] : Committee {inverse of chair}
coChairedCommittee[0..1] : Committee

Using the reference path notation of OOP languages, with c referencing a `Committee` object, we obtain the equation:

Equation 1.1.

c.chair.chairedCommittee = c

Or, the other way around, with m referencing a ClubMember object, we obtain the equation:

Equation 1.2.

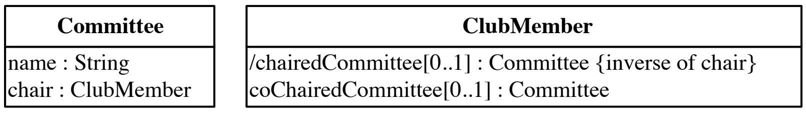
m.chairedCommittee.chair = m

Notice that when a property *p2* is the inverse of a property *p1*, this implies that, the other way around, *p1* is the inverse of *p2*. Therefore, when we declare the property `ClubMember::chairedCommittee` to be the inverse of `Committee::chair`, then, implicitly, `Committee::chair` is the inverse of `ClubMember::chairedCommittee`. We therefore call `Committee::chair` and `ClubMember::chairedCommittee` a pair of mutually inverse reference properties. Having such a pair in a model implies redundancy because each of the two involved reference properties can be derived from the other by inversion. This type of redundancy implies data storage overhead and update overhead, which is the price to pay for the bidirectional navigability that supports efficient object access in both directions.

In general, a bidirectional association between the classes `A` and `B` is represented by two reference properties `A::bbb` and `B::aaa` such that for any object `a1` instantiating `A`, it holds that

1. `a1.bbb.aaa` = `a1` if both `A::bbb` and `B::aaa` are single-valued,
2. `a1.bbb.aaa` contains `a1` if `A::bbb` is single-valued and `B::aaa` is multi-valued,
3. for any `b1` from `a1.bbb`, `b1.aaa` = `a1` if `A::bbb` is multi-valued and `B::aaa` is single-valued,
4. for any `b1` from `a1.bbb`, `b1.aaa` contains `a1` if both `A::bbb` and `B::aaa` are multi-valued.

For maintaining the duplicate information of a mutually inverse reference property pair, it is common to treat one of the two involved properties as the *master*, and the other one as the *slave*, and take this distinction into consideration in the code of the change methods (such as the property setters) of the affected model classes. We indicate the slave of an inverse reference property pair in a model diagram by declaring the slave property to be a ***derived*** property using the UML notation of a slash (/) as a prefix of the property name as shown in the following diagram:


Committee
name : String
chair : ClubMember
ClubMember
/chairedCommittee[0..1] : Committee {inverse of chair}
coChairedCommittee[0..1] : Committee

The property `chairedCommittee` in `ClubMember` is now derived (as indicated by its slash prefix). Its annotation `{inverse of chair}` defines a derivation rule according to which it is derived by inverting the property `Committee::chair`.

There are two ways how to realize the derivation of a property: it may be *derived* on *read* via a read-time computation of its value, or it may be *derived* on *update* via an update-time computation performed whenever one of the variables in the derivation expression (typically, another property) changes its value. The latter case corresponds to a *materialized view* in a database. While a reference property that is derived on read may not guarantee efficient navigation, because the on-read computation may create unacceptable latencies, a reference property that is derived on update does provide efficient navigation.

When we designate an inverse reference property as derived by prefixing its name with a slash (/), we indicate that it is derived on update. For instance, the property `/chairedCommittee` in the example above is derived on update from the property `chair`.

In the case of a derived reference property, we have to deal with ***life-cycle dependencies*** between the affected model classes requiring special change management mechanisms based on the functionality type of the represented association (either *one-to-one*, *many-to-one* or *many-to-many*).

In our example of the derived inverse reference property `ClubMember::chairedCommittee`, which is single-valued and optional, this means that

1. whenever a new committee object is created (with a mandatory `chair` assignment), the corresponding `ClubMember::chairedCommittee` property has to be assigned accordingly;

2. whenever the `chair` property is updated (that is, a new chair is assigned to a committee), the corresponding `ClubMember::chairedCommittee` property has to be unset for the club member who was the previous chair and set for the one being the new chair;

3. whenever a committee object is destroyed, the corresponding `ClubMember::chairedCommittee` property has to be unset.

In the case of a derived inverse reference property that is multi-valued while its inverse base property is single-valued (like `Publisher::publishedBooks` in **Figure 1-2.** The OO class model with two pairs of mutually inverse reference properties below being derived from `Book::publisher`), the life cycle dependencies imply that

1. whenever a new 'base object' (such as a book) is created, the corresponding inverse property has to be updated by adding a reference to the new base object to its value set (like adding a reference to the new book object to `Publisher::publishedBooks`);

2. whenever the base property is updated (e.g., a new publisher is assigned to a book), the corresponding inverse property (in our example, `Publisher::publishedBooks`) has to be updated as well by removing the old object reference from its value set and adding the new one;

3. whenever a base object (such as a book) is destroyed, the corresponding inverse property has to be updated by removing the reference to the base object from its value set (like removing a reference to the book object to be destroyed from `Publisher::publishedBooks`).

Notice that from a purely computational point of view, we are free to choose either of the two mutually inverse reference properties (like `Book::authors` and `Author::authoredBooks`) to be the master. However, in many cases, associations represent asymmetrical ontological existence dependencies that dictate which of the two mutually inverse reference properties is the master. For instance, the authorship association between the classes `Book` and `Author` represents an existential dependency of books on their authors. A book existentially depends on its author(s), while an author does not existentially depend on any of her books. Consequently, the corresponding object lifecycle dependency between Book and Author implies that their bidirectional association is maintained by maintaining `Author` references in `Book::authors` as the natural choice of master property, while `Author::authoredBooks` is the slave property, which is derived from `Book::authors`.

1.2. Making an OO Class Model

Since classical OO programming languages do not support explicit associations as first class citizens, but only classes with reference properties representing implicit associations, we have to eliminate all explicit associations for obtaining an OO class model.

1.2.1 The basic procedure

The starting point of our ***association elimination*** procedure is an information design model with various kinds of unidirectional and bidirectional associations, such as the model shown in **Figure 1-1.** The Publisher-Book-Author information design model with two bidirectional associations above. If the model still contains any non-directed associations, we first have to turn them into directed ones by making a decision on the ownership of their ends, which is typically based on navigability requirements.

Notice that both associations in the *Publisher-Book-Author* information design model, *publisher-publishedBooks* and *authoredBooks-authors* (or *Authorship*), are bidirectional as indicated by the ownership dots at both association ends. For eliminating all explicit associations from an information design model, we have to perform the following steps:

1. ***Eliminate unidirectional associations***, connecting a source with a target class, by replacing them with a reference property in the source class such that the target class is its range.
2. ***Eliminate bidirectional associations*** by replacing them with a pair of mutually inverse reference properties.

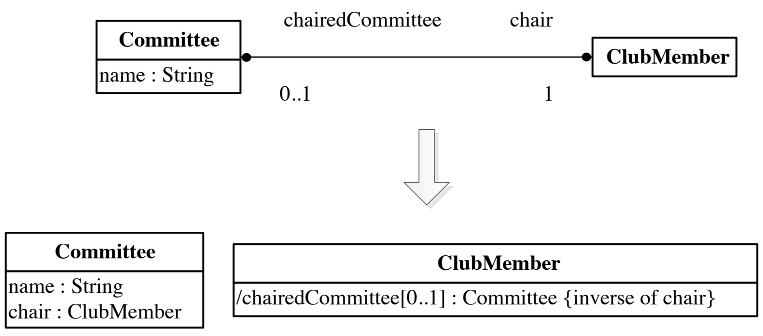
1.2.2 How to eliminate unidirectional associations

A unidirectional association connecting a source with a target class is replaced with a corresponding reference property in its source class having the target class as its range. Its multiplicity is the same as the multiplicity of the target association end. Its name is the name of 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).

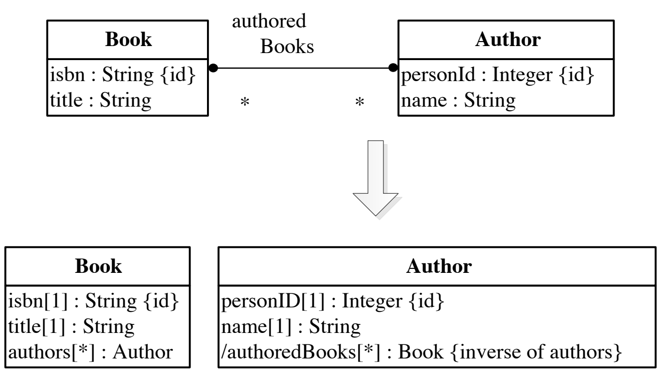
1.2.3 How to eliminate bidirectional associations

A bidirectional association, such as the authorship association between the classes `Book` and `Author` in the model shown in **Figure 1-1.** The Publisher-Book-Author information design model with two bidirectional associations above, is replaced with a pair of mutually inverse reference properties, such as `Book::authors` and `Author::authoredBooks`. Since both reference properties represent the same information (the same set of binary relationships), it's an option to consider one of them being the "master" and the other one the "slave", which is derived from the master. We discuss the two cases of a one-to-one and a many-to-many association

1. In the case of a bidirectional one-to-one association, this leads to a pair of mutually inverse single-valued reference properties, one in each of the two associated classes. Since both of them represent essentially the same information (the same collection of links/relationships), one has to choose which of them is considered the master property, such that the other one is the slave property, which is derived from the master property by inversion. In the class diagram, the slave property is designated as a derived property that is automatically updated whenever 1) a new master object is created, 2) the master reference property is updated, or 3) a master object is destroyed. This transformation is illustrated with the following example:



2. A bidirectional many-to-many association is mapped to a pair of mutually inverse multi-valued reference properties, one in each of the two classes participating in the association. Again, in one of the two classes, the multi-valued reference property representing the (inverse) association is designated as a *derived property* that is automatically updated whenever the corresponding property in the other class (where the association is maintained) is updated. This transformation is illustrated with the following example:



1.2.4 The resulting OO class model

After replacing both bidirectional associations with reference properties, we obtain the OO class model shown in **Figure 1-2.** The OO class model with two pairs of mutually inverse reference properties.

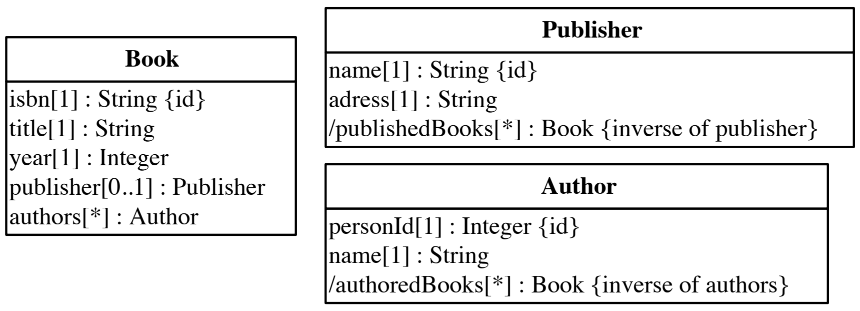


Figure 1-2. *The OO class model with two pairs of mutually inverse reference properties*

Since books are entities that existentially depend on authors and possibly on publishers, and not the other way around, it's natural to maintain the master references in book objects, and consider the inverse references in publisher and author objects as derived (or 'slave') data. Therefore, we define `publishedBooks` and `authoredBooks` as derived inverse reference properties, which is indicated by their slash prefix in the OO class model.

The meaning of this OO class model can be illustrated by a sample data population for the three model classes involved:

Publisher		
Name	**Address**	**Published books**
Bantam Books	New York, USA	0553345842
Basic Books	New York, USA	0465030793

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	Authored books
1	Daniel Dennett	0553345842
2	Douglas Hofstadter	0553345842, 0465030793
3	Immanuel Kant	1463794762, 1928565379

Notice how Book records reference Publisher and Author records, and, vice versa, Publisher and Author records reference Book records.

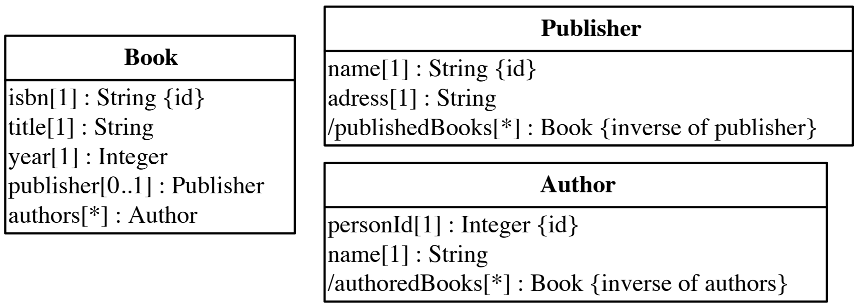
Chapter 2. Implementing Bidirectional Associations with Plain JS and Firebase

In this chapter, we show

1. how to derive a JS class model from an OO class model with ***derived inverse reference properties***,
2. how to code the JS class model in the form of JS model classes,
3. how to write the view and controller code based on the model code.

2.1. Make a JavaScript Class Model

The starting point for making our JS class model is an OO class model with derived inverse reference properties like the one discussed above, which we present here again, for convenience:

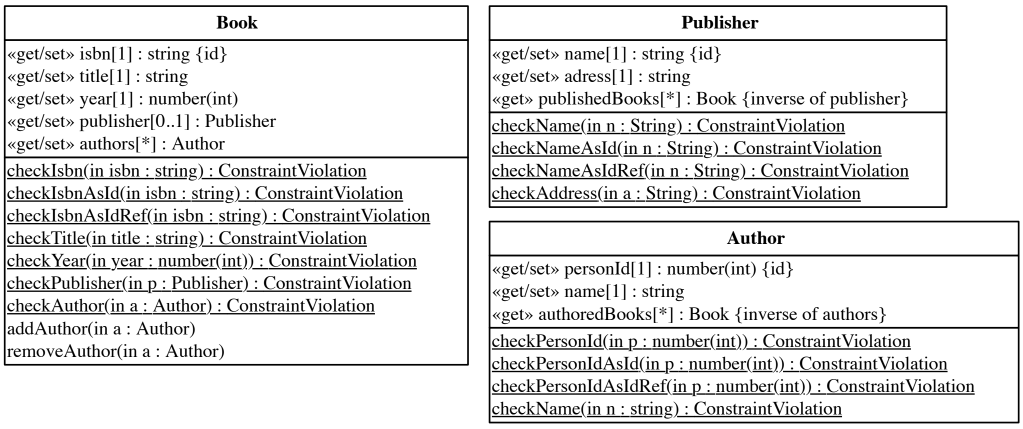


Notice that the model contains two derived inverse reference properties: `Publisher::/publishedBooks` and `Author::/authoredBooks`. Each of them is linked to a master property, from which it is derived. Consequently, each of them represents a pair of mutually inverse reference properties corresponding to a bidirectional association.

Compared to making JS class models with unidirectional associations, the only new issue is:

1. Add a «get» stereotype to all derived inverse reference properties, implying that they have an implicit getter, but no setter. They are programatically set whenever their inverse master reference property is updated.

This concerns the two derived inverse reference properties `Publisher::/publishedBooks` and `Author::/authoredBooks`. Thus, we get the following JavaScript class model:


Book
«get/set» isbn[1] : string {id}
«get/set» title[1] : string
«get/set» year[1] : number(int)
«get/set» publisher[0..1] : Publisher
«get/set» authors[*] : Author
checkIsbn(in isbn : string) : ConstraintViolation
checkIsbnAsId(in isbn : string) : ConstraintViolation
checkIsbnAsIdRef(in isbn : string) : ConstraintViolation
checkTitle(in title : string) : ConstraintViolation
checkYear(in year : number(int)) : ConstraintViolation
checkPublisher(in p : Publisher) : ConstraintViolation
checkAuthor(in a : Author) : ConstraintViolation
addAuthor(in a : Author)
removeAuthor(in a : Author)
Publisher
«get/set» name[1] : string {id}
«get/set» adress[1] : string
«get» publishedBooks[*] : Book {inverse of publisher}
checkName(in n : String) : ConstraintViolation
checkNameAsId(in n : String) : ConstraintViolation
checkNameAsIdRef(in n : String) : ConstraintViolation
checkAddress(in a : String) : ConstraintViolation
Author
«get/set» personId[1] : number(int) {id}
«get/set» name[1] : string
«get» authoredBooks[*] : Book {inverse of authors}
checkPersonId(in p : number(int)) : ConstraintViolation
checkPersonIdAsId(in p : number(int)) : ConstraintViolation
checkPersonIdAsIdRef(in p : number(int)) : ConstraintViolation
checkName(in n : string) : ConstraintViolation

2.2. Write the Model Code

The JS class model can be directly coded for getting the code of the model layer of our bidirectional association app.

2.2.1 New issues

Compared to the unidirectional association app, we have to deal with a number of new technical issues:

1. We define the derived inverse reference properties, like `Publisher::/publishedBooks`, without a check operation and without a set operation.

2. We also have to take care of maintaining the derived inverse reference properties by maintaining the derived (sets of) inverse references that form the (collection) value of a derived inverse reference property. This requires in particular that

   a. whenever the value of a ***single-valued*** master reference property is ***initialized or updated*** (such as assigning a reference to a `Publisher` instance `p` to `b.publisher` for a **Book** instance b), an inverse reference has to be assigned (or added) to the corresponding value (set) of the derived inverse reference property (such as adding `b` to `p.publishedBooks`); when the value of the master reference property is updated and the derived inverse reference property is multi-valued, then the obsolete inverse reference to the previous value of the single-valued master reference property has to be deleted;

   b. whenever the value of an optional ***single-valued*** master reference property is ***unset*** (e.g. by assigning `null` to `b.publisher` for a `Book` instance b), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `p.publishedBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

   c. whenever a reference is ***added*** to the value of a ***multi-valued*** master reference property (such as adding an `Author` reference a to `b.authors` for a `Book` instance b), an inverse reference has to be assigned or added to the corresponding value of the derived inverse reference property (such as adding `b` to `a.authoredBooks`);

   d. whenever a reference is ***removed*** from the value of a ***multi-valued*** master reference property (such as removing a reference to an `Author` instance `a` from `b.authors` for a `Book` instance `b`), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `a.authoredBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

   e. whenever an object with a single reference or with multiple references as the value of a master reference property is ***destroyed*** (e.g., when a `Book` instance `b` with a single reference `b.publisher` to a `Publisher` instance `p` is destroyed), the derived inverse references have to be removed first (e.g., by removing `b` from `p.publishedBooks`).

   Notice that when a new object is created with a single reference or with multiple references as the value of a master reference property (e.g., a new `Book` instance b with a single reference `b.publisher`), the `Book.add` method will take care of creating the derived inverse references.

2.2.2 Coding 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:
   a. In some setters, the corresponding property (non-asynchronous) check is invoked and the property is only set/unset, if the check does not detect any constraint violation; other (asynchronous) checks are invoked on the database operation (*Create* or *Update*), after form submission.
   b. **If the concerned property is the inverse of a derived reference property (representing a bidirectional association), make sure to assigns/unsets the corresponding inverse reference to/from (the collection value of) the inverse property.**
3. For each multi-valued property, code its add and remove operations, as well as the specified get/set operations:
   a. Code the add/remove operations as (instance-level) methods.
   b. Code the setter such that it invokes the add operation for each item of the collection to be assigned.
   c. **If the concerned property is the inverse of a derived reference property (representing a bidirectional association), make sure to assign/unset (or add/remove) the corresponding inverse reference to/from (the collection value of) the inverse property.**
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, with the derived properties publishedBooks and authoredBooks not included since their information is redundant (they are derived from the publisher and authors properties of `books`).
5. Take care of deletion dependencies in the destroy method. **Make sure that when an object with a single reference (or with multiple references) as the value of a master reference property is destroyed, all referenced objects are destroyed as well or their (derived) inverse references are unset (or removed) first**.

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

2.2.3 Code each class of the JS class model

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

```
class Publisher {
  // using a single record parameter with ES6 function parameter destructuring
  constructor ({name, address}) {
    // assign properties by invoking implicit setters
    this.name = name;
    this.address = address;
  };
  ...
  get name() {...}
  static checkName( n) {...}
  static checkNameAsId( n) {...}
  static checkNameAsIdRef( n) {...}
  set name( n) {...}
  get address() {...}
  static checkAddress( a) {...}
  set address( a) {...}
  get publishedBooks() {...}
}
```

Notice that the (derived) multi-valued reference property `publishedBooks` has no setter method and is not assigned in the constructor function because it is a read-only property that is assigned implicitly when its inverse master reference property `Book::publisher` is assigned.

2.2.4 Code Create use case that handles derived inverse references properties

Any reference property that is coupled to a derived inverse reference property (implementing a bidirectional association), now also needs to assign (or add/remove) inverse references to (or from) the corresponding (collection) value of the inverse reference property. An example of such a single-valued and multi-valued references in a batch write transaction in the `Book` class for *Create*:

```
Book.add = async function (slots) {
  let book = null, validationResult = null;
  try {
    // validate data by creating Book instance
    book = await new Book( slots);
    // invoke asynchronous ID/uniqueness check
    validationResult = await Book.checkIsbnAsId( book.isbn);
    if (!validationResult instanceof NoConstraintViolation) throw validationResult;
    validationResult = await Publisher.checkNameAsIdRef( book.publisher_id);
    if (!validationResult instanceof NoConstraintViolation) throw validationResult;
    for (const a of book.authorIdRefs) {
      const validationResult = await Author.checkAuthorIdAsIdRef( String(a.id));
      if (!validationResult instanceof NoConstraintViolation) {
        throw validationResult;
      }
    }
  }
  ...
  if (book) {
    const bookDocRef = fsDoc( fsDb, "books", book.isbn)
        .withConverter( Book.converter),
      publishersCollRef = fsColl( fsDb, "publishers")
        .withConverter( Publisher.converter),
      authorsCollRef = fsColl( fsDb, "authors")
        .withConverter( Author.converter);
    const bookInverseRef = {isbn: book.isbn, title: book.title};
    try {
      const batch = writeBatch( fsDb); // initiate batch write object
      await batch.set( bookDocRef, book); // create book record (master)
      // iterate ID references (foreign keys) of slave class objects (authors) and
      // create derived inverse reference properties to master class object (book)
      // Author::authoredBooks
      await Promise.all( book.authorIdRefs.map( a => {
        const authorDocRef = fsDoc( authorsCollRef, String( a.id));
        batch.update( authorDocRef, {authoredBooks: arrayUnion( bookInverseRef)});
      }));
      if (book.publisher_id) {
        // create derived inverse reference properties between slave class objects
        // (publisher) with master class object (book) Publisher::publishedBooks
        const publisherDocRef = fsDoc( publishersCollRef, book.publisher_id);
        batch.update( publisherDocRef, {publishedBooks: arrayUnion( bookInverseRef)});
      }
      batch.commit(); // commit batch write
      console.log(`Book record "${book.isbn}" created!`);
    } catch (e) {
      console.error(`${e.constructor.name}: ${e.message}`);
    }
  }
};
```

2.2.5 Code Update use case that handles derived inverse references properties

For the *Update* case, several things happen sequentially. In the first block we initialize variables, and, in a `try/catch` block, we retrieve the data of the up-to-date book record/document (`bookBeforeUpdate`):

```
Book.update = async function ({isbn, title, publicationDate,
    publisher_id, authorIdRefsToAdd, authorIdRefsToRemove}) {
  let validationResult = null,
    bookBeforeUpdate = null;
  const bookDocRef = fsDoc( fsDb, "books", isbn).withConverter( Book.converter),
    updatedSlots = {};
  try {
    // retrieve up-to-date book record
    bookBeforeUpdate = (await getDoc( bookDocRef)).data();
  } catch (e) {
    console.error(`${e.constructor.name}: ${e.message}`);
  }
  ...
```

In the next block, while evaluating changes in each `Book` object property, we build the object `updatedSlots` with updates coming in the slots object from the view layer. Likewise, the add/remove "instance-level" methods are invoked to update the multi-valued property, as is seen in the following code:

```
if (bookBeforeUpdate) {
  if (bookBeforeUpdate.title !== title) updatedSlots.title = title;
    if (bookBeforeUpdate.publicationDate !== publicationDate)
      updatedSlots.publicationDate = Timestamp.fromDate(new Date(
        publicationDate));
  if (publisher_id && bookBeforeUpdate.publisher_id !== publisher_id) {
    updatedSlots.publisher_id = publisher_id;
  } else if (!publisher_id && bookBeforeUpdate.publisher_id !== undefined) {
    updatedSlots.publisher_id = deleteField();
  }
  if (authorIdRefsToAdd) for (const authorIdRef of authorIdRefsToAdd)
    bookBeforeUpdate.addAuthor( authorIdRef);
  if (authorIdRefsToRemove) for (const authorIdRef of authorIdRefsToRemove)
    bookBeforeUpdate.removeAuthor( authorIdRef);
  if (authorIdRefsToAdd || authorIdRefsToRemove)
    updatedSlots.authorIdRefs = bookBeforeUpdate.authorIdRefs;
  ...
}
```

If the object `updatedSlots` brings updates, then the updates of the master class object (book) are processed while reference integrity constraints are invoked. This process involves multiple write operations, that are performed in a batch write transaction. All this happens in a try/catch block, in order to catch errors, and interrupt the transaction process to preserve reference integrity of the object of concern.

1. In the first part, we initialize variables to access both slave class objects through a reference of the table/collection in Firestore (`authorsCollRef` and `pubsCollRef`). Also, we initialize map objects defining inverse references properties, one containing the book title before update (`inverseRefBefore`), and the other containing the updated book title (`inverseRefAfter`). These references will be used as derived inverse reference properties for updating slave class objects if the book title has changed:

```
...
const updatedProperties = Object.keys(updatedSlots);
if (updatedProperties.length) {
  try {
    const authorsCollRef = fsColl( fsDb, "authors")
        .withConverter( Author.converter),
      pubsCollRef = fsColl( fsDb, "publishers")
        .withConverter( Publisher.converter);
    // initialize inverse ID references, before update and after update
    const inverseRefBefore = {isbn: isbn, title: bookBeforeUpdate.title};
    const inverseRefAfter = {isbn: isbn, title: title};
    ...
```

2. in the next part, the batch write transaction is initiated after invoking reference integrity checkers, and then consuming the author reference properties to "add" and to "remove" from the up-to-date book, given as parameters to the `Book.update()` function.

   Firestore does not provide a method to "update" a Map data type within an Array, so we need to remove the reference property (a Map) allocated in an array, and then add a new reference property (another Map), therefore each reference property to be removed uses the method `arrayRemove()` and the reference properties to be added use the method `arrayUnion()`. In both cases the map references discussed lines above are used, since `arrayRemove()` and `arrayUnion()` need the entire value of the Map to be able to add or remove values. Notice that we only invoke the reference integrity checkers for the new (added) slave class objects as in the code showed below:

```
...
const batch = writeBatch( fsDb);
// check constraint violation for publication date
if (updatedSlots.publicationDate) {
  validationResult = Book.checkPublicationDate( publicationDate);
  if (!validationResult instanceof NoConstraintViolation) throw validationResult;
}
// check constraint violation for publisher
if (updatedSlots.publisher_id) {
  validationResult = await Publisher.checkNameAsIdRef( publisher_id);
  if (!validationResult instanceof NoConstraintViolation) throw validationResult;
}
// remove old derived inverse references properties from slave
// objects (authors) Author::authoredBooks
if (authorIdRefsToRemove) {
  await Promise.all(authorIdRefsToRemove.map( a => {
    const authorDocRef = fsDoc(authorsCollRef, String( a.id));
    batch.update(authorDocRef, {authoredBooks: arrayRemove( inverseRefBefore)});
  }));
}
// add new derived inverse references properties from slave objects
// (authors) Author::authoredBooks, while checking constraint violations
if (authorIdRefsToAdd) {
  await Promise.all(authorIdRefsToAdd.map( async a => {
    const authorDocRef = fsDoc(authorsCollRef, String( a.id));
    validationResult = await Author.checkAuthorIdAsIdRef( a.id);
    if (!validationResult instanceof NoConstraintViolation) throw validationResult;
    batch.update(authorDocRef, {authoredBooks: arrayUnion( inverseRefAfter)});
  }));
}
...
```

3. Since book titles are used in the derived inverse reference properties (`Author::authoredBooks` and `Publisher::/publishedBooks`), we need to update each reference properties in the already existing slave class objects (authors). For which we filter every not changed (already existing) ID reference, and we update them again using the `arrayRemove()` and `arrayUnion()` methods. Notice that we don't invoke any reference integrity constraint checker with the already existing slave class objects:

```
...
// if title changes, update title in ID references (array of maps) in
// unchanged author objects
if (updatedSlots.title) {
  const NoChangedAuthorIdRefs = authorIdRefsToAdd ?
    bookBeforeUpdate.authorIdRefs.filter(d => !authorIdRefsToAdd.includes(d))
    : bookBeforeUpdate.authorIdRefs;
  await Promise.all(NoChangedAuthorIdRefs.map( a => {
    const authorDocRef = fsDoc(authorsCollRef, String( a.id));
    batch.update(authorDocRef, {authoredBooks: arrayRemove( inverseRefBefore)});
  }));
  await Promise.all(NoChangedAuthorIdRefs.map( a => {
    const authorDocRef = fsDoc(authorsCollRef, String( a.id));
    batch.update(authorDocRef, {authoredBooks: arrayUnion( inverseRefAfter)});
  }));
}
if (!updatedSlots.publisher_id && updatedSlots.title) {
  // update derived inverse references property if publisher didn't change,
  // but title changed
  const pubBeforeUpdtDocRef = fsDoc(pubsCollRef, bookBeforeUpdate.publisher_id);
  batch.update(pubBeforeUpdtDocRef, {publishedBooks: arrayRemove( inverseRefBefore)});
  batch.update(pubBeforeUpdtDocRef, {publishedBooks: arrayUnion( inverseRefAfter)});
}
...
```

4. In this part we deal with updating the derived inverse references property in slave objects (publisher), first evaluating the content to know which kind of update we should process, for instance if the `publisher_id` value is a string that means that we are creating or updating the new reference property, but if we the value is an object that means that the value has been deleted:

```
...
// update derived inverse references property in slave objects (publisher)
// Publisher::/publishedBooks, being created or deleted
if (typeof updatedSlots.publisher_id === "string") { // has been created/updated (set)
  if (bookBeforeUpdate.publisher_id) {
    const pubBeforeUpdtDocRef = fsDoc(pubsCollRef, bookBeforeUpdate.publisher_id);
    batch.update(pubBeforeUpdtDocRef, {publishedBooks: arrayRemove(inverseRefBefore)});
  }
  const publisherDocRef = fsDoc(pubsCollRef, updatedSlots.publisher_id);
  batch.update(publisherDocRef, {publishedBooks: arrayUnion( inverseRefAfter)});
}
if (typeof updatedSlots.publisher_id === "object") { // has been deleted (unset)
  const pubBeforeUpdtDocRef = fsDoc(pubsCollRef, bookBeforeUpdate.publisher_id);
  batch.update(pubBeforeUpdtDocRef, {publishedBooks: arrayRemove( inverseRefBefore)});
}
...
```

5. Finally, we update the master class object, book, and we commit the batch write, executing every write operation in the transaction as a single atomic operation:

```
// update book object (master)
batch.update(bookDocRef, updatedSlots);
batch.commit(); // commit batch write
```

2.2.6 Take care of deletion dependencies

When a Book instance b, with a single reference b.publisher to a Publisher instance p and multiple references b.authors to Author instances, is destroyed, depending on the chosen deletion policy (1) CASCADE or (2) DROP-REFERENCES, (1) the dependent Publisher instance and Author instances have to be deleted first or (2) the derived inverse references have to be removed first (e.g., by removing b from p.publishedBooks). We assume Existential Independence for both associated object types and, consequently, implement the DROP-REFERENCES policy, again in a batch write transaction:

```
Book.destroy = async function (isbn) {
  const bookDocRef = fsDoc( fsDb, "books", isbn)
      .withConverter( Book.converter),
    publishersCollRef = fsColl( fsDb, "publishers")
      .withConverter( Publisher.converter),
    authorsCollRef = fsColl( fsDb, "authors")
      .withConverter( Author.converter);
  try {
    const bookRec = (await getDoc( bookDocRef
      .withConverter( Book.converter))).data();
    const inverseRef = {isbn: bookRec.isbn, title: bookRec.title};
    const batch = writeBatch( fsDb); // initiate batch write object
    // delete derived inverse reference property, Authors::/authoredBooks
    await Promise.all( bookRec.authorIdRefs.map( aId => {
      const authorDocRef = fsDoc( authorsCollRef, String( aId.id));
      batch.update( authorDocRef, {authoredBooks: arrayRemove( inverseRef)});
    }));
    if (bookRec.publisher_id) {
      // create derived inverse reference property, Publisher::/publishedBooks
      const publisherDocRef = fsDoc( publishersCollRef, bookRec.publisher_id);
      batch.update( publisherDocRef, {publishedBooks: arrayRemove( inverseRef)});
    }
    batch.delete( bookDocRef); // create book record (master)
    batch.commit(); // commit batch write
    console.log(`Book record "${isbn}" deleted!`);
  } catch (e) {
    console.error(`Error deleting book record: ${e}`);
  }
};
```

2.3. Exploit Inverse Reference Properties in the User Interface

In the UI code we can now exploit the inverse reference properties for more efficiently creating a list of inversely associated objects in the Retrieve/List All use case. For instance, we can more efficiently create a list of all published books for each publisher. However, we do not allow updating the set of inversely associated objects in the update object use case (e.g., updating the set of published books in the update publisher use case). Rather, such an update has to be done via updating the master objects (in our example, the books) concerned.

2.3.1 Show published books in Retrieve/List All publishers

For showing information about published books in the Retrieve/List All publishers use case, we can now exploit the derived inverse reference property publishedBooks:

```
const tableBodyEl = pubRSectionEl.querySelector("table > tbody")
document.getElementById("RetrieveAndListAll").addEventListener("click", async function ()
  tableBodyEl.innerHTML = "";
  pubMSectionEl.hidden = true;
  pubRSectionEl.hidden = false;
  showProgressBar( "Publisher-R");
  const publisherRecs = await Publisher.retrieveAll();
  for (const publisher of publisherRecs) {
    const row = tableBodyEl.insertRow();
    row.insertCell().textContent = publisher.name;
    row.insertCell().textContent = publisher.address;
    // create list of books published by this publisher
    if (publisher.publishedBooks && publisher.publishedBooks.length) {
      const listEl = createListFromMap(publisher.publishedBooks, "title");
      row.insertCell().appendChild(listEl);
    }
  }
  hideProgressBar( "Publisher-R");
});
```

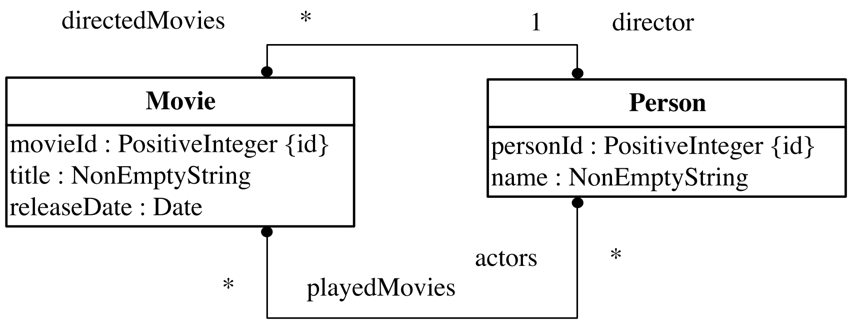
2.4. Points of Attention

1. it is not possible to execute query collections within tarnsactions. https://stackoverflow.com/questions/50071700/can-we-not-query-collections-inside-transactions
2. We can use data validation for reinforcing referential integrity, while executing atomic operations. https://firebase.google.com/docs/firestore/manage-data/transactions#data_validation_for_atomic_operations

2.5. Practice Project

This project is based on the information design model below. The app from the previous assignment is to be extended by adding derived inverse reference properties for implementing the bidirectional associations. This is achieved by adding the multi-valued reference properties directedMovies and playedMovies to the model class Person, both with range Movie.

Figure 2-1. *Two bidirectional associations between Movie and Person.*



This project includes the following tasks:

1. Make an OO design model derived from the given information design model.
2. Make a JavaScript class model derived from the OO class model.

3. Code your JS class model, following the guidelines of the tutorial.

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

Table 2-1. Movies

Movie ID	Title	Release date	Director	Actors
1	Pulp Fiction	1994-05-12	1	5, 6
2	Star Wars	1977-05-25	2	7, 8
3	Inglourious Basterds	2009-05-20	1	9, 1
4	The Godfather	1972-03-15	4	11, 12

Table 2-2. People

Person ID	Name	Directed movies	Played movies
1	Quentin Tarantino	1, 3	3
2	George Lucas	2	
4	Francis Ford Coppola	4	
5	Uma Thurman		1
6	John Travolta		1
7	Ewan McGregor		2
8	Natalie Portman		2
9	Brad Pitt		3
11	Marlon Brando		4
12	Al Pacino		4

Make sure that your pages comply with the XML syntax of HTML5, and that your JavaScript code complies with our Coding Guidelines and is checked with JSHint.