Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Notes	Weekly
Resources	Graded work
Policies	Standards
Professors & addendum	Code examples

Data associations or relations… *document reference* how-to

This document enables the reader to begin learning how to design and implement *document references* as a data organization scheme in a MongoDB database.

Before continuting, ensure that you have read, analyzed, and understood the content in the data associations or relations introduction document.

Data organization and design

For this example, assume the following:

- One entity is a “company” that sells products
- The other entity is a “product” sold by a company

One *company* will sell zero or more *products*. From the perspective of a *company* item, it has a *to-many* association or relation with *product* items.

One *product* is sold by only one specific *company*. From the perspective of a *product* item, it has a *to-one* association or relation with a *company* item.

Database work

Obviously, a MongoDB database is needed. For testing purposes, we will assume that your computer already has a *folder* (maybe named `db-local`) to hold databases.

Start the database engine, and then start the Compass tool.

In an existing database, or in a new database (maybe named `testing`), create two (empty) collections, probably named:

- companies
- products

Next, we'll generate and load data into these collections.

Generating and loading sample data

For this how-to document, we will use the Mockaroo service to generate data. You may need a Mockaroo account and be signed in to complete some of these tasks.

The following is a six step procedure:

1. Generate some MongoDB identifiers
2. Create a Mockaroo "dataset"
3. Generate some "company" data
4. Import the generated "company" data into the database
5. Generate some "product" data
6. Import the generated "product" data into the database

Here are the details:

1. Generate some MongoDB identifiers

For this task, we will need some MongoDB object (document) identifiers for the *companies*.

Navigate to the Mockaroo page that enables you to create a new schema. The new schema will have only one field. The name does not matter, but the type will be "MongoDB ObjectID". Choose to generate a reasonable number of rows, maybe around 50. Choose the CSV format, and do NOT include a header or BOM. Download / save.

2. Create a Mockaroo "dataset"

Navigate to the Mockaroo "DATASETS" page. Choose to upload a new dataset, maybe named `MongoDB-ObjectIDs-Company` or whatever enables you to understand its purpose and content. The data will be from the file downloaded/saved in step 1 above.

3. Generate some “company” data

Navigate to the Mockaroo page that enables you to create a new schema. The new schema will have as many field names as needed to define a company. Choose to generate the *same* number of rows (as JSON, but not in an array wrapper) generated for the MongoDB identifiers in step 1 above.

In addition, it *must* include:

- Field Name `_id`
- Type `Dataset Column`
- Options…
  Choose the name of the dataset from step 2 above
  Choose `sequential`

Download / save. Notice that each row now includes a MongoDB identifier.

4. Import the generated “company” data into the MongoDB database collection

Use the Compass tool, and navigate to the “companies” collection that was created above in the Database work section.

Import the data generated in step 3 above, and test.

5. Generate some “product” data

Navigate to the Mockaroo page that enables you to create a new schema. The new schema will have as many field names as needed to define a product. Choose to generate *more* rows than was done for “companies”. Maybe generate about 200 or 300 rows (again, as JSON, but not in an array wrapper).

This time, do *not* include an `_id` field.

However, it *must* include a field that holds the document reference:

- Field Name `companyId.$oid`
- Type `Dataset Column`
- Options…
  Choose the name of the dataset from step 2 above
  Choose `random`

Download / save. Notice that each row now includes a `companyId` field with a value that is a MongoDB identifier for the `company` object/document that it is associated or related to.

6. Import the generated “product” data

into the MongoDB database collection

Use the Compass tool, and navigate to the “products” collection that was created above in the Database work section.

Import the data generated in step 5 above, and test.

Web API work

For this how-to document, we will write a DEN (database, Express.js, Node.js) web API.

> *Note that this was implemented in a recent code example, webapi-data-assoc-doc-ref*

The following is a five step procedure:

1. Create the web API or use a template
2. Write a “company” Mongoose schema
3. Write a “product” Mongoose schema
4. At a minimum, write “get” methods for “company” and “product”

Then, to show that associated or related data can be fetched:

1. Modify the “get one” method for product

Here are the details:

1. Create the app or use a template

Create a new Node.js and Express.js app from scratch, and add all the bits needed to create a simple working web API that uses a database.

Alternatively, use a template or code example (e.g. `webapiv7`) to do the same.

2. Write a “company” Mongoose schema

In a new source code file (maybe named `msc-company.js`), write a Mongoose schema that describes a “company”. For example, the code below describes a simple “company”, but yours will likely have more fields:

```
// Setup
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

```
// Entity schema
var companySchema = new Schema({
  name: String,
  country: String,
  ceo: String
});

// Make schema available to the application
module.exports = companySchema;
```

In `manager.js`, add statements (in various places) that will enable you to use this schema:

```
// Use the schema
const companySchema = require('./msc-company');

// Declare the model variable
let Company;

// Initialize the model
Company = db.model("companies", companySchema, "companies");
```

3. Write a "product" Mongoose schema

In a new source code file (maybe named `msc-product.js`), write a Mongoose schema that describes a "product". For example, the code below describes a simple "product", but yours will likely have more fields.

Notice the Mongoose syntax for the document reference. The value of the `ref` property MUST match the string used for the model name in the model initializer statement (above) in `manager.js`.

```
// Setup
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// Entity schema
var productSchema = new Schema({
  name: String,
  yearReleased: Number,
  // to-one with company, as a document reference
  companyId: { type: Schema.Types.ObjectId, ref: 'companies' }
});

// Make schema available to the application
```

```
module.exports = productSchema;
```

In `manager.js`, add statements (in various places) that will enable you to use this schema:

```
// Use the schema
const productSchema = require('./msc-product');

// Declare the model variable
let Product;

// Initialize the model
Product = db.model("products", productSchema, "products");
```

4. At a minimum, write “get” methods for “company” and “product”

Following the well-known pattern, add “get all” and “get one” methods, in both `server.js` and `manager.js`. For both entities.

Test with Postman.

5. Modify the “get one” method for “product”

In step 4 above, you wrote a standard “get one” method for product. Notice that it probably returned something like this:

```
{
  "_id": "5e5e967dfebfae0f659875c1",
  "name": "ASPERGILLUS FUMIGATUS",
  "yearReleased": 1994,
  "companyId": "5e5e66effc13ae3c5f000031"
}
```

Instead of the MongoDB object identifier value of the `companyId` property, we want the *full and complete “product” object*.

We use the Mongoose `populate()` method (doc link here). Its simplest usage or syntax accepts an arguement that’s a field name string:

```
// Find one specific document
Product.findById(itemId)
  .populate('companyId')
  .exec((error, item) => {
  if (error) {
```

```
    // Find/match is not found
    return reject(error.message);
  }
  // Check for an item
  if (item) {
    // Found, one object will be returned
    return resolve(item);
  } else {
    return reject('Not found');
  }
}
```

Notice that it probably returned something like this:

```
{
  "_id": "5e5e967dfebfae0f659875c1",
  "name": "ASPERGILLUS FUMIGATUS",
  "yearReleased": 1994,
  "companyId": {
    "_id": "5e5e66effc13ae3c5f000031",
    "name": "Mylan Institutional Inc.",
    "country": "China",
    "ceo": "Florina Liddall"
  }
}
```

Nice.

Happy coding!