Mongoose Relationships Tutorial

NoSQL databases like MongoDB work differently than the older and more established Relational Databases like MySQL, Oracle, Microsoft SQL, and so on. Relationships in the traditional sense don't really exist in MongoDB like they do in MySQL. In this tutorial we'll take a look at how you can work with related data, even though it is not explicitly enforced by MongoDB. We'll have a look at Reference Based Relationships (Normalization) as well as Embedded Documents Relationships (Denormalization).

Reference Based Relationships (Normalization)

In this approach let's say we have two collections. One will be for publishers and another will be for games. So first we'll have a publisher object like so.

```
let publisher = {
    companyName: 'Nintendo'
}
```

Then, we will have another collection to represent a game. So in the object here, we have a game that references the id of a publisher document.

```
let game = {
    publisher: 'id'
}
```

This is the reference approach. It feels similar to how things might be done in a relational database, but there is a difference. In MongoDB, this relationship is *not* enforced – unlike a relational database that enforces data integrity across relationships. Even though the game document has a reference to a publisher document via an id, in MongoDB there is no actual relationship between these two documents.

Embedded Documents Relationships (Denormalization)

The other approach to relationships is to embed a related document inside of another document. For example we can embed a publisher document inside of a game document.

```
let game = {
    publisher: {
        companyName: 'Nintendo'
    }
}
```

So which approach should you use? Well, Normalization is really a relational database type approach. If you are going to be focusing strictly on Normalization, a relational database might be the better option. MongoDB doesn't support server-side foreign key relationships, normalization is often discouraged. It is more common to embed a child object within a parent objects if possible, since this increases performance and makes foreign keys unnecessary.

Normalization -> Better Consistency

- Requires additional queries
- Provides Consistency

Denormalization -> Better Performance

- Can use a single query for related documents
- Consistency can degrade over time

You may also use a combination of these two approaches in your application, but in general, you will embed a child object within a parent object if possible.

Referencing A Document in another Document

We will start with this code below to get things started.

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/mongo-games')
    .then(() => console.log('Now connected to MongoDB!'))
    .catch(err => console.error('Something went wrong', err));

const Publisher = mongoose.model('Publisher', new mongoose.Sche
    companyName: String,
    firstParty: Boolean,
    website: String
}));

const Game = mongoose.model('Game', new mongoose.Schema({
    title: String,
}));

async function createPublisher(companyName, firstParty, website
    const publisher = new Publisher({
        companyName,
        firstParty,
        website
    });

    const result = await publisher.save();
    console.log(result);
}

async function createGame(title, publisher) {
    const game = new Game({
        title,
        publisher
    });

    const result = await game.save();
    console.log(result);
}

async function listGames() {
    const games = await Game
        .find()
        .select('title');
    console.log(games);
}

createPublisher('Nintendo', true, 'https://www.nintendo.com/');
```

So first off here, we create a Publisher in the database. The publisher is Nintendo, it is true that they are a first party publisher, and the website address is provided. Also note, once the publisher is inserted into the database we are provided with a unique id for that document: `5b2bdc233e939402b41d90bf`

```
mongo-crud $node index.js
Now connected to MongoDB!
{ _id: 5b2bdc233e939402b41d90bf,
  companyName: 'Nintendo',  firstParty: true,
  website: 'https://www.nintendo.com/',
  __v: 0 }
```

Now that we have the unique id for this particular publisher, we can insert a new game into the database while specifying a publisher by using the unique id of `5b2bdc233e939402b41d90bf` as the second argument here.

```
createGame('Super Smash Bros', '5b2bdc233e939402b41d90bf')
```

Now our goal was to create a new game in the database which is associated with a publisher. It looks like all we got is the game title for output here, the publisher appears to be missing.

```
mongo-crud $node index.js
Now connected to MongoDB!
{ _id: 5b2bdca63c61aa362c25dc7b,
  title: 'Super Smash Bros',  __v: 0 }
```

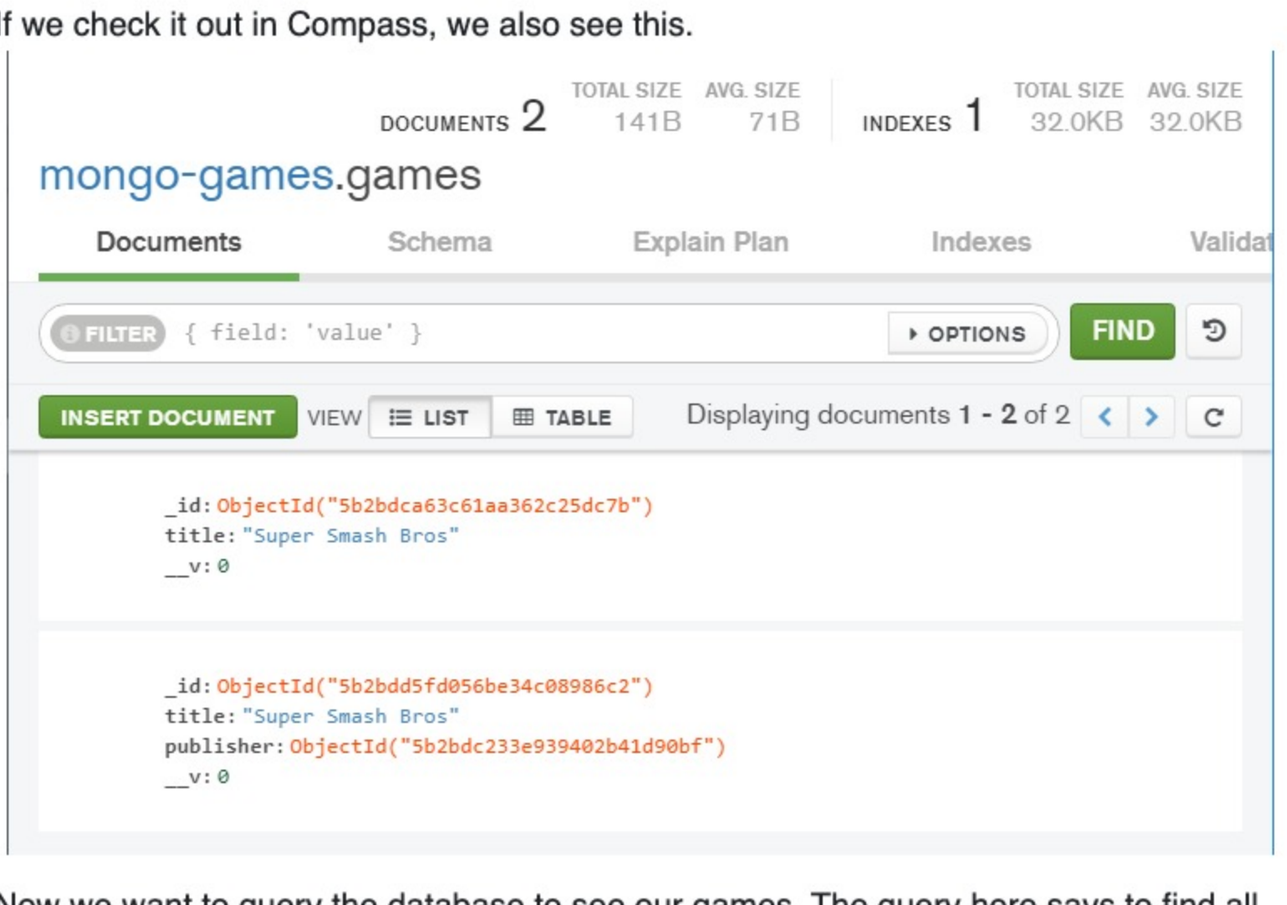
We can fix this by modifying our Game model to include a publisher with it's type set to mongoose.Schema.Types.ObjectId like so.

```
const Game = mongoose.model('Game', new mongoose.Schema({
    title: String,
    publisher: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Publisher'
    }
}));
```

Now when we insert a game into the database, we see both the title and publisher are displayed.

```
mongo-crud $node index.js
Now connected to MongoDB!
{ _id: 5b2bdd5fd056be34c08986c2,
  title: 'Super Smash Bros',  publisher: 5b2bdc233e939402b41d90bf,
  __v: 0 }
```

If we check it out in Compass, we also see this.



Now we want to query the database to see our games. The query here says to find all games and select their title.

```
async function listGames() {
    const games = await Game
        .find()
        .select('title');
    console.log(games);
}

listGames();
```

We see the two games we inserted already.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b2bdca63c61aa362c25dc7b, title: 'Super Smash Bros' },
  { _id: 5b2bdd5fd056be34c08986c2, title: 'Super Smash Bros' } ]
```

We can also select the publisher by simply including it in the select portion of the query.

```
async function listGames() {
    const games = await Game
        .find()
        .select('title publisher');
    console.log(games);
}

listGames();
```

Now when running the program, we see the first game which has no publisher, and also the second game which is associated with a publisher thanks to our code updates just above. Note that the publisher is simply just the unique id of `5b2bdc233e939402b41d90bf`.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b2bdca63c61aa362c25dc7b, title: 'Super Smash Bros' },
  { _id: 5b2bdd5fd056be34c08986c2,    title: 'Super Smash Bros',
    publisher: 5b2bdc233e939402b41d90bf } ]
```

So, it would be better to see the actual data of the publisher rather than just the identifying id. We can do that with the populate() method.

```
async function listGames() {
    const games = await Game
        .find()
        .populate('publisher')
        .select('title publisher');
    console.log(games);
}

listGames();
```

Ah ha! Now take a look at the data we get back.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b2bdca63c61aa362c25dc7b, title: 'Super Smash Bros' },
  { _id: 5b2bdd5fd056be34c08986c2,    title: 'Super Smash Bros',
    publisher:
     { _id: 5b2bdc233e939402b41d90bf,       companyName: 'Nintendo',
       firstParty: true,
       website: 'https://www.nintendo.com/',       __v: 0 } } ]
```

So let's say you only want to see the company name of the publisher but not all the other associated data of the publisher. Great, just update your populate() call like so.

```
async function listGames() {
    const games = await Game
        .find()
        .populate('publisher', 'companyName')
        .select('title publisher');
    console.log(games);
}

listGames();
```

Now we get what we want.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b2bdca63c61aa362c25dc7b, title: 'Super Smash Bros' },
  { _id: 5b2bdd5fd056be34c08986c2,    title: 'Super Smash Bros',
    publisher: { _id: 5b2bdc233e939402b41d90bf, companyName: 'Nintendo' } ]
```

To clean things up just a bit more, lets remove the _id from the output in the query by adding `-_id`.

```
async function listGames() {
    const games = await Game
        .find()
        .populate('publisher', 'companyName -_id')
        .select('title publisher');
    console.log(games);
}

listGames();
```

Nice! It looks like it is working great.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b2bdca63c61aa362c25dc7b, title: 'Super Smash Bros' },
  { _id: 5b2bdd5fd056be34c08986c2,    title: 'Super Smash Bros',
    publisher: { companyName: 'Nintendo' } } ]
```

Embedded Documents In MongoDB

Now we want to see how to embed a document within another document instead of using the reference approach. In the section above, we had a game document that made a reference to a separate publisher document. Now, we are going to change the code so that when a game document is saved, we will also embed a publisher document at the same time. What you can see below is that we are now embedding the publisherSchema right inside of the gameSchema.

```
// Publisher Schema
const publisherSchema = new mongoose.Schema({
    companyName: String,
    firstParty: Boolean,
    website: String
})

// Publisher Model
const Publisher = mongoose.model('Publisher', publisherSchem

// Game Schema
const gameSchema = new mongoose.Schema({
    title: String,
    publisher: publisherSchema
})

// Game Model
const Game = mongoose.model('Game', gameSchema);
```

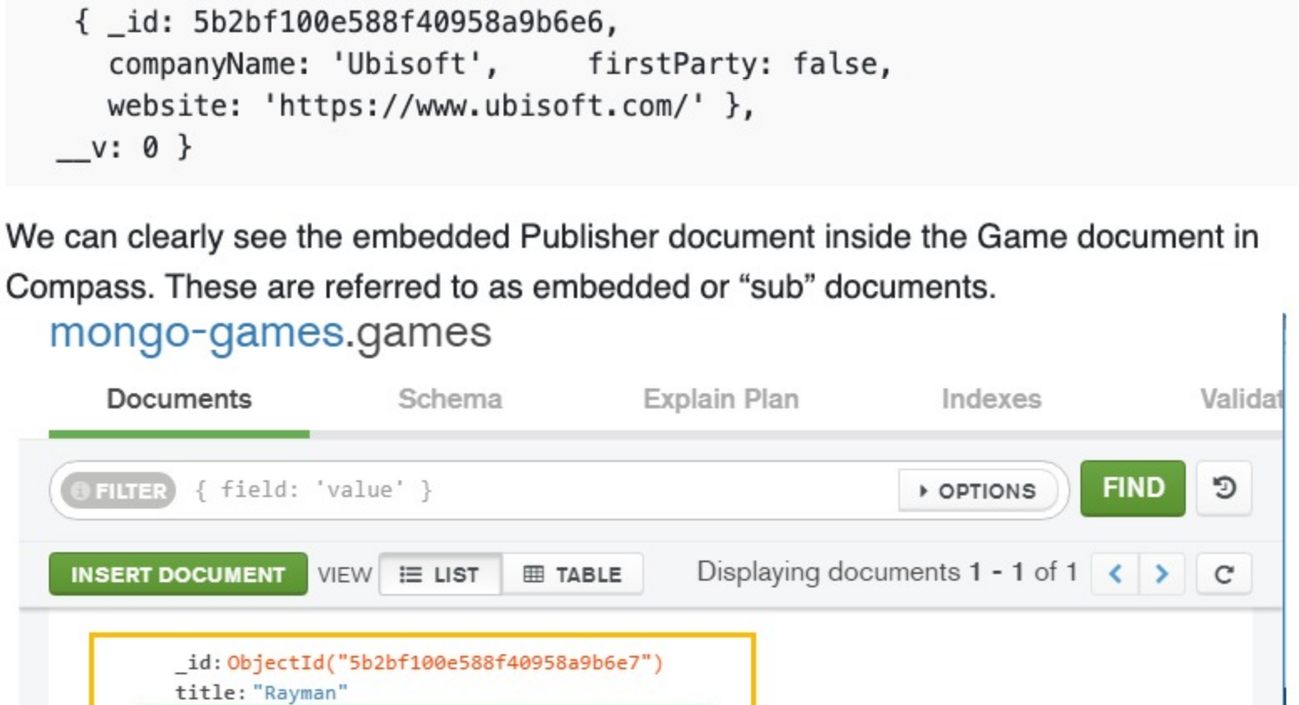
Great! Let's create a new game and embed a publisher in one shot.

```
async function createGame(title, publisher) {
    const game = new Game({
        title,
        publisher
    });

    const result = await game.save();
    console.log(result);
}

createGame('Rayman', new Publisher({ companyName: 'Ubisoft', fi
```

Ah ha! Note that publisher is an object, and it contains all the properties of a Publisher.

```
mongo-crud $node index.jsNow connected to MongoDB!
{ _id: 5b2bf100e588f40958a9b6e7,
  title: 'Rayman',  publisher:
   { _id: 5b2bf100e588f40958a9b6e6,
     companyName: 'Ubisoft',     firstParty: false,
     website: 'https://www.ubisoft.com/' },
  __v: 0 }
```

We can clearly see the embedded Publisher document inside the Game document in Compass. These are referred to as embedded or "sub" documents.



So let's say you want to update the publisher, but it is embedded in a game. How can we do that? You would have to find the game first, then update the publisher within the game. Last, you would save the game.

```
async function updatePublisher(gameId) {
    const game = await Game.findById(gameId);
    game.publisher.companyName = 'Epic Games';
    game.publisher.website = 'https://epicgames.com/';
    game.save();
}

updatePublisher('5b2bf100e588f40958a9b6e7');
```

Run the function in the terminal.

```
mongo-crud $node index.js
```

If we check out the document in Compass, we can see it is now updated successfully.

It is also possible to update a sub document directly. Here is how to do that.

```
async function updatePublisher(gameId) {
    const game = await Game.update({ _id: gameId }, {
        $set: {
            'publisher.companyName': 'Bethesda Softworks',
            'publisher.website': 'https://bethesda.net/'
        }
    });
}

updatePublisher('5b2bf100e588f40958a9b6e7');
```

Run the function in the terminal.

```
mongo-crud $node index.js
```

Once again, Compass shows us we successfully updated the document right in the database.

Removing a sub document with unset

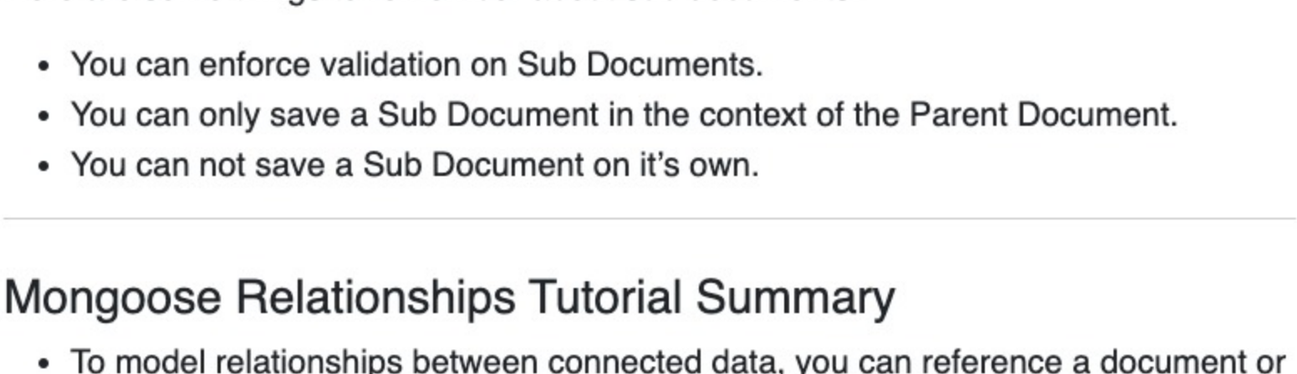
To remove a sub document you can use unset like so.

```
async function updatePublisher(gameId) {
    const game = await Game.update({ _id: gameId }, {
        $unset: {
            'publisher': ''
        }
    });
}

updatePublisher('5b2bf100e588f40958a9b6e7');
```

Run the function in the terminal.

```
mongo-crud $node index.js
```

Sure enough, the sub document is now gone in Compass.



Here are some things to remember about sub documents.

- You can enforce validation on Sub Documents.
- You can only save a Sub Document in the context of the Parent Document.
- You can not save a Sub Document on it's own.

Mongoose Relationships Tutorial Summary

- To model relationships between connected data, you can reference a document or embed it in another document as a sub document.
- Referencing a document does not create a "real" relationship between these two documents as does with a relational database.
- Referencing documents is also known as **normalization**. It is good for data consistency but creates more queries in your system.
- Embedding documents is also known as **denormalization**. The benefit of this approach is getting all the data you need about a document and it's sub document(s) with a single query. Therefore, this approach is very fast. The drawback is that data may not stay as consistent in the database.
- ObjectIDs are generated by MongoDB driver to uniquely identify each document. ObjectIDs consist of 12 bytes
  - 4 bytes: timestamp
  - 3 bytes: machine identifier
  - 2 bytes: process identifier
  - 3 byes: counter

To reference a document in Mongoose, you can use mongoose.Schema.Types.ObjectId like this.

```
const gameSchema = new mongoose.Schema({
    publisher: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Publisher'
    }
})
```

The more common approach with NoSQL databases is to embed a sub document like so.

```
const gameSchema = new mongoose.Schema({
    publisher: {
        type: new mongoose.Schema({
            companyName: String,
        })
    }
})
```

Embedded documents don't have a save method. They can only be saved via their parent.

```
const game = await Game.findById(gameId);
game.publisher.companyName = 'New Company Name';
game.save();
```

#javascript #nodejs