Relationships and Joins with Population

Relationships and Joins with Population

Although, there are no relationships stored in a NoSQL database such as MongoDB, we can do so in the application layer. Mongoose provides a feature called *population*. It allows us to fill certain parts of the document from a different collection. Let's say we have `posts` and `users` collections. We can reference posts in the user schema:

```
const mongoose = require('mongoose'),
  Schema = mongoose.Schema

const userSchema = Schema({
  _id     : Number,
  name: String,
  posts: [{ type: Schema.Types.ObjectId, ref: 'Post' }]
};

const postSchema = Schema({
  _creator: { type: Number, ref: 'User' },
  title: String,
  text: String
})

let Post  = mongoose.model('Post', postSchema)
let User = mongoose.model('User', userSchema)

User.findOne({ name: /azat/i })
  .populate('posts')
  .exec(function (err, user) {
    if (err) return handleError(err)
    console.log('The user has % post(s)', user.posts.length)
  })
```

Note `ObjectId`, `Number`, `String`, and `Buffer` are valid data types to use as references.

In the previous query, we used a regular expression (RegExp), this feature is not exclusive to Mongoose. In fact, the native driver and its other wrappers, along with the `mongo` console all support RegExps. The syntax is the same as in normal JavaScript/Node.js RegExp patterns. Therefore, in a way, we perform a join query on our `Post` and `User` models.

It’s possible to return only a portion of populated results. For example, we can limit the number of posts to the first 10 only:

```
.populate({
  path: 'posts',
  options: { limit: 10, sort: 'title' }
})
```

Sometimes it's more practical to return only certain fields instead of the full document. This can be done with `select`:

```
.populate({
    path: 'posts',
    select: 'title',
    options: { limit: 10, sort: 'title' }
  })
```

In addition, Mongoose can filter the populated results by a query! For example, we can apply RegExp for "node.js" to the text (a `match` query property):

```
.populate({
    path: 'posts',
    select: '_id title text',
    match: {text: /node\.js/i},
    options: { limit: 10, sort: '_id' }
  })
```

Here, it takes selected properties (`select` and then the field names of `_id`, `title`, `text`) and can be as customized as you want it to be. The best practice is to populate only the required fields because this avoids potential leakage of sensitive information and reduces overhead on the system.

The `populate` method also works on multiple document queries. For example, we can use `find` instead of `findOne`:

```
User.find({}, {limit: 10, sort:{ _id: -1}})
  .populate('posts')
  .exec(function (err, user) {
    if (err) return handleError(err);
    console.log('The user has % post(s)', user.posts.length);
  })
```

Tip For custom sorting, we can add properties using `name: -1` or `name: 1` patterns and can pass the resulting object to the `sort` option. Again, this is a standard MongoDB interface and is not exclusive to Mongoose.