Helpers

So far, we have looked at some of the basic functionalities known as CRUD (Create, Read, Update, Delete) operations, but Mongoose also provides the ability to configure several types of helper methods and properties. These can be used to further simplify working with data.

Let’s create a user schema in `./src/models/user.js` with the fields `firstName` and `lastName`:

```
let mongoose = require('mongoose')
let userSchema = new mongoose.Schema({
  firstName: String,
  lastName: String
})
module.exports = mongoose.model('User', userSchema)
```

Helpers

Virtual Property

A virtual property is not restricted the database. We can add it to our schema as a helper to get and set values.

Let’s create a virtual property called `fullName` which can be used to set values on `firstName` and `lastName` and retrieve them as a combined value when read:

```
userSchema.virtual('fullName').get(function() {
  return this.firstName + ' ' + this.lastName
})
userSchema.virtual('fullName').set(function(name) {
  let str = name.split(' ')

  this.firstName = str[0]
  this.lastName = str[1]
})
```

Callbacks for get and set must use the function keyword as we need to access the model via the **this** keyword. Using fat arrow functions will change what **this** refers to.

Now, we can set `firstName` and `lastName` by assigning a value to `fullName`:

```
let model = new UserModel()

model.fullName = 'Thomas Anderson'

console.log(model.toJSON())  // Output model fields as JSON

console.log()

console.log(model.fullName)  // Output the full name
```

The code above will output the following:

```
{ _id: 5a7a4248550ebb9fafd898cf,

  firstName: 'Thomas',

  lastName: 'Anderson' }

Thomas Anderson
```

Helpers

Instance Methods

We can create custom helper methods on the schema and access them via the model instance. These methods will have access to the model object and they can be used quite creatively. For instance, we could create a method to find all the people who have the same first name as the current instance.

In this example, let's create a function to return the initials for the current user. Let's add a custom helper method called `getInitials` to the schema:

```
userSchema.methods.getInitials = function() {

  return this.firstName[0] + this.lastName[0]

}
```

This method will be accessible via a model instance:

```
let model = new UserModel({

  firstName: 'Thomas',

  lastName: 'Anderson'
})
let initials = model.getInitials()
console.log(initials) // This will output: TA
```

Helpers

Static Methods

Similar to instance methods, we can create static methods on the schema. Let’s create a method to retrieve all users in the database:

```
userSchema.statics.getUsers = function() {
  return new Promise((resolve, reject) => {
    this.find((err, docs) => {
      if(err) {
        console.error(err)
        return reject(err)
      }
      resolve(docs)
    })
  })
}
```

Calling `getUsers` on the Model class will return all the users in the database:

```
UserModel.getUsers()
  .then(docs => {
    console.log(docs)
  })
  .catch(err => {
    console.error(err)
  })
```

Adding instance and static methods is a nice approach to implement an interface to database interactions on collections and records.

Helpers

Middleware

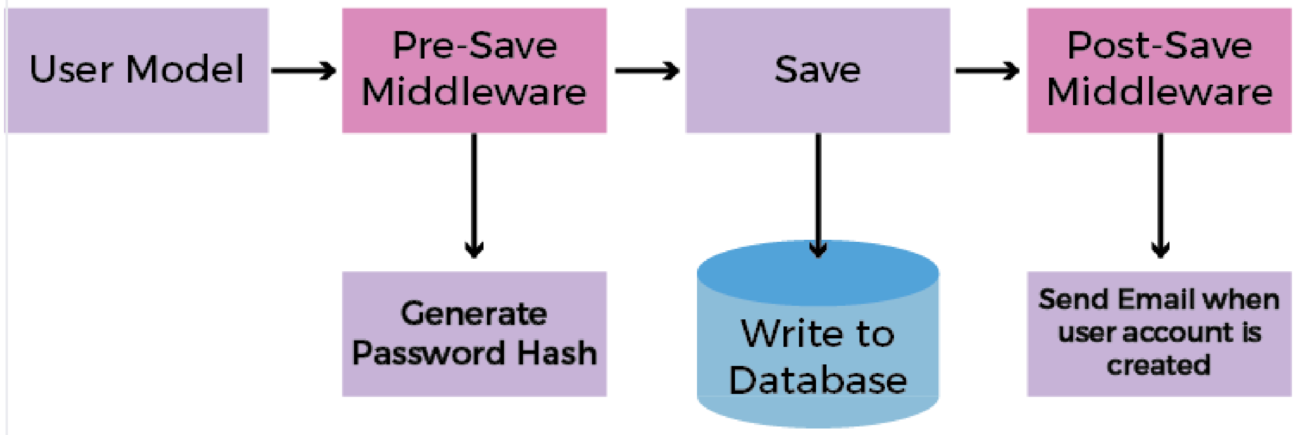
Middleware is a set of functions that run at specific stages of a pipeline. Mongoose supports middleware for the following operations:

- Aggregate
- Document
- Model
- Query

For instance, models have `pre` and `post` functions that take two parameters:

1. Type of event ('init', 'validate', 'save', 'remove')
2. A callback that is executed with **this** referencing the model instance

Example of Middleware (a.k.a. pre and post hooks):



Helpers

Middleware

Let's demonstrate this, as an example, by adding two fields called `createdAt` and `updatedAt` to our schema:

```
let mongoose = require('mongoose')

let userSchema = new mongoose.Schema({

  firstName: String,

  lastName: String,

  createdAt: Date,

  updatedAt: Date
})
module.exports = mongoose.model('User', userSchema)
```

When `model.save()` is called, there is a `pre('save', …)` and `post('save', …)` event that is triggered. For the second parameter, you can pass a function that is called when the event is triggered. These functions take a parameter to the next function in the middleware chain.

Let's add a pre-save hook and set values for `createdAt` and `updatedAt`:

```
userSchema.pre('save', function (next) {
  let now = Date.now()

  this.updatedAt = now
  // Set a value for createdAt only if it is null
  if (!this.createdAt) {
    this.createdAt = now
  }
  // Call the next function in the pre-save chain
  next()
})
```

Let's create and save our model:

```
let UserModel = require('./user')
let model = new UserModel({
  fullName: 'Thomas Anderson'
}
msg.save()
   .then(doc => {
     console.log(doc)
   })
   .catch(err => {
     console.error(err)
   })
```

You should see values for `createdAt` and `updatedAt` when the created record is printed:

```
{ _id: 5a7bbbeebc3b49cb919da675,
  firstName: 'Thomas',
  lastName: 'Anderson',
  updatedAt: 2018-02-08T02:54:38.888Z,
  createdAt: 2018-02-08T02:54:38.888Z,
  __v: 0 }
```

Helpers

Plugins

Suppose that we want to track when a record was created and last updated on every collection in our database. Instead of repeating the above process, we can create a plugin and apply it to every schema.

Let’s create a file `./src/model/plugins/timestamp.js` and replicate the above functionality as a reusable module:

```
module.exports = function timestamp(schema) {
  // Add the two fields to the schema
  schema.add({
    createdAt: Date,
    updatedAt: Date
  })

  // Create a pre-save hook
  schema.pre('save', function (next) {
    let now = Date.now()

    this.updatedAt = now
    // Set a value for createdAt only if it is null
    if (!this.createdAt) {
      this.createdAt = now
    }
   // Call the next function in the pre-save chain
   next()
  })
}
```

To use this plugin, we simply pass it to the schemas that should be given this functionality:

```
let timestampPlugin = require('./plugins/timestamp')

emailSchema.plugin(timestampPlugin)
userSchema.plugin(timestampPlugin)
```