Emerging Technologies

COMP-308
Winter 2018

Lesson 5 Review

- **NoSQL**
  - **key-value** storage solutions were designed for **better availability**, **simple querying**, and **horizontal scaling**.
  - More **robust** than SQL databases
  - Document storage
    - **store hierarchical documents in standard formats**, such as **JSON** and **XML**
    - **faster read operations**

- **MongoDB**
  - Schema-less
  - JSON-like storage format named BSON (**BinaryJSON) Request object**
  - **Supports indexing replica, and and sharding**
  - use of the _**id field as primary key**
  - If the document does not specify an _**id** field, then MongoDB will add the _id field and assign a unique **ObjectId** for the document before inserting

- **Mongo shell commands**
  - **use**, **show**, **find**, **drop**, etc.

Lesson 5 Review

- **MongoDB ad hoc queries**
  - **find method**
    - **Find all documents**
    - Find Documents that Match Query Criteria
  - **Query for Equality using _id**
  - **Query Using Operators:**
    - **$in**
  - **Query for ranges:**
    - **$gt**
    - **$lt**
  - Querying **Arrays**
  - Query **Embedded Documents**
  - The **find()** method returns a cursor to the results.
    - **next()** method
    - **hasNext()** method
- **CRUD Operations**
  - find
  - Insert
  - save
  - update
  - remove
- Mongoose
  - Node.js **Object Document Model**(ODM) module that adds MongoDB support to your Express application
  - Enforces a schema from the Express application

Lesson 5 Review

- **Schema object**
  - Property of **mongoose** instance
  - **to define the document list of properties**, each with its own **type** and **constraints**
  - **to enforce the document structure**
  - define a **schema and model** for your feature
  - **use a model instance** to create, retrieve, and update user documents
    - **save** method
    - **find** method

```
Schema = mongoose.Schema;
var UserSchema = new Schema({
    firstName: String,
    lastName: String,
    email: String,
    username: String,
    password: String
});
// use schema to define the User model
mongoose.model('User', UserSchema);
```

Using Mongoose with MongoDB

Objectives:

❑ Use the model's methods and perform **CRUD operations**

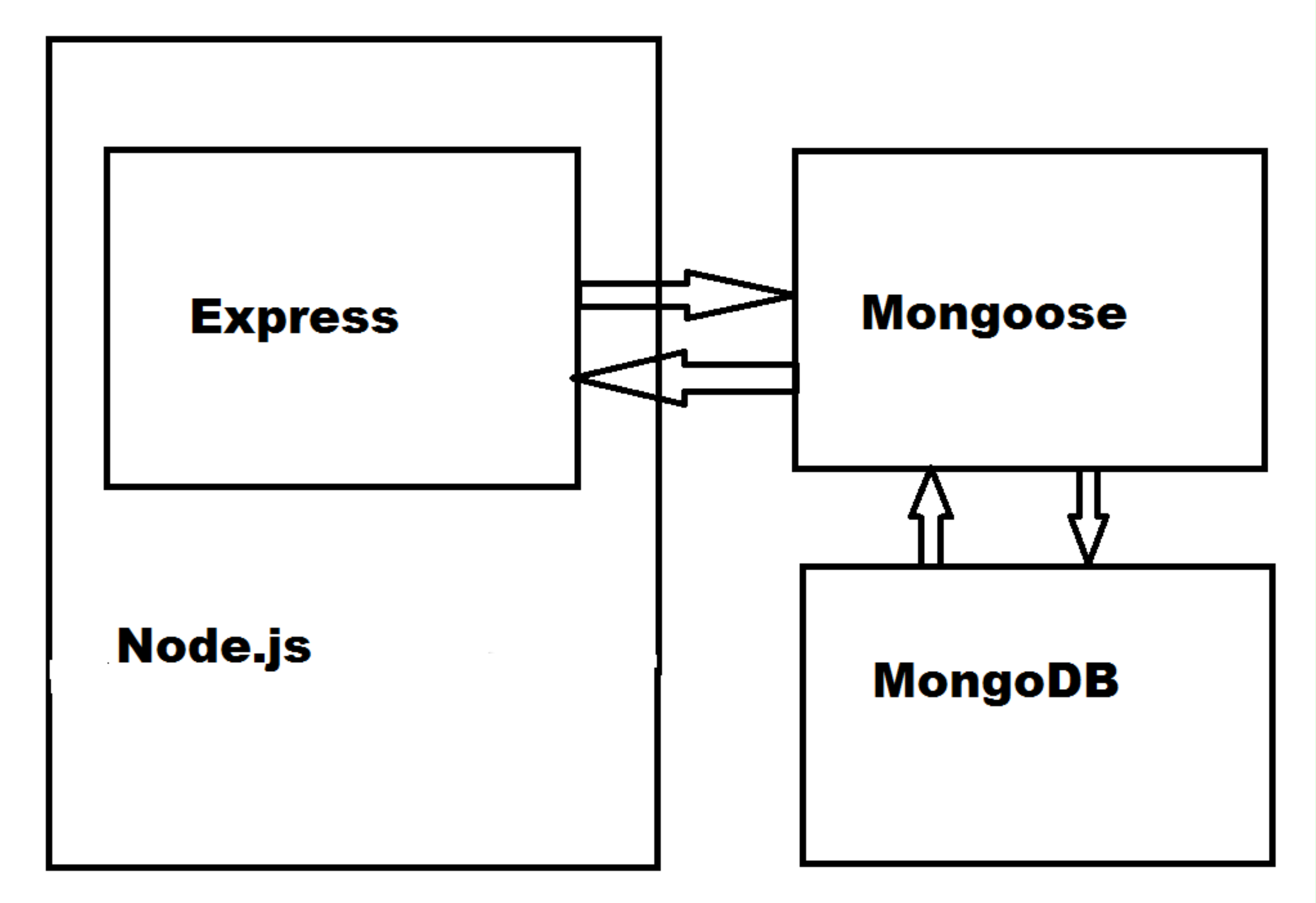
❑ Verify your data using **predefined** and **custom validators**

❑ Use **middleware to intercept** the model's methods

❑ **Referencing** of a document from another document using a **DBRef** convention

Adding MongoDB to Express app

- Express application architecture including a MongoDB database:



Building a REST API with Mongoose

- Building a REST API allows remote clients to perform CRUD operations on MongoDB collections.
- A Representational State Transfer (REST) interface provides a **set of operations that can be invoked by a remote client** (which could be another service) over a network, **using the HTTP protocol**.
- REST convention indicates which of HTTP methods should be used for which types of operation:
  - POST: to add new data
  - GET: to retrieve data
  - PUT: to update data
  - DELETE: to remove data

CRUD Operations

- **C**reating documents using save() method
- **R**etrieving documents using find() and findOne() methods
- **U**pdating documents using update(), findOneAndUpdate(), and findByIdAndUpdate() methods
- **D**eleting documents using remove(), findOneAndRemove(), and findByIdAndRemove() methods

Finding multiple user documents using find()

- The **find() method is a model method** that **retrieves multiple documents stored in the same collection using a query** and is a Mongoose implementation of the MongoDB **find()** collection method.
- Add the following **list()** method in your *app/controllers/users.server.controller.js* file:

```
exports.list = function(req, res, next) {
        User.find({}, function(err, users) {
                if (err) {
                        return next(err);
                } else {
                        res.json(users);
                }
        });
};
```

Finding multiple user documents using find()

- ❑ To use the new method **list()**, register a route for in *app/routes/users.server.routes.js* file and change it to look like the following code snippet:

```
const users =
require('../../app/controllers/users.server.controller');

module.exports = function(app) {
app.route('/users')
.post(users.create)
.get(users.list);
};
```

- ❑ **Run the application** and retrieve a list of your users by visiting http://localhost:3000/users in your browser

Reading a single user document using findOne()

- The **findOne()** method, is very similar to the find() method, but **retrieves only the first document of the subset**.
- Add the following lines of code at the end of your *app/controllers/ users.server.controller.js* file:

```
exports.read = function(req, res) {
        res.json(req.user); //JSON representation of the req.user object
};
exports.userByID = function(req, res, next, id) { //populating the req.user object
        User.findOne({
                _id: id
                }, (err, user) => {
                        if (err) {
                                return next(err);
                        } else {
                          req.user = user;
                          next();
                }
        });
};
```

Reading a single user document using findOne()

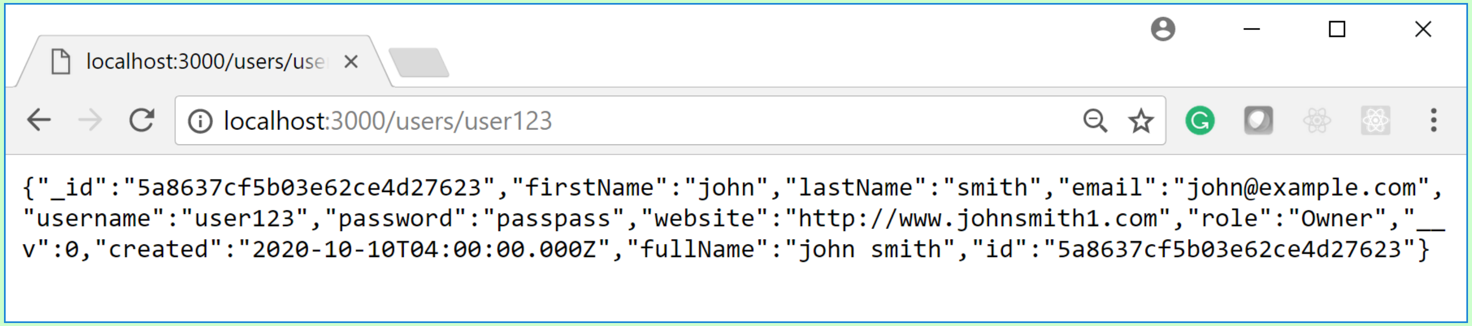
- You can use the userById() method as a **middleware to deal with the manipulation of single documents** when performing read, delete, and update operations.
- To do so, modify your *app/routes/users.server.routes.js* file to look like the following lines of code:

```
const users = require('../../app/controllers/users.server.controller');
module.exports = function(app) {
        app.route('/users')
        .post(users.create)
        .get(users.list);
        app.route('/users/:userId').get(users.read);
        app.param('userId', users.userByID);
};
```

- Run **the application,** navigate to *http://localhost:3000/users* in your browser, copy one of your users' _id values, and navigate to http://localhost:3000/users/[id], replacing the [id] part with the user's _id value.

Reading a single user document using findOne()

```
exports.userByID = function (req, res, next, username) {
        // Use the 'User' static 'findOne' method to retrieve a specific user
        User.findOne({
        username: username //using the username instead of id
        }, (err, user) => {
        if (err) {
                // Call the next middleware with an error message
                return next(err);
        } else {
                // Set the 'req.user' property
                req.user = user;
                // Call the next middleware
                next();
        }
        });
};
```



```
{"_id":"5a8637cf5b03e62ce4d27623","firstName":"john","lastName":"smith","email":"john@example.com",
"username":"user123","password":"passpass","website":"http://www.johnsmith1.com","role":"Owner","__
v":0,"created":"2020-10-10T04:00:00.000Z","fullName":"john smith","id":"5a8637cf5b03e62ce4d27623"}
```

Updating an existing user document

- The Mongoose model has several available methods to update an existing document - **update()**, **findOneAndUpdate()**, and **findByIdAndUpdate()**.
- Since we already use the userById() middleware, the easiest way to update an existing document would be to use the **findByIdAndUpdate()** method.
- Go back to your *app/controllers/users.server.controller.js* file, and add a new update() method:

```
exports.update = function(req, res, next) {
User.findByIdAndUpdate(req.user.id, req.body, function(err, user) {
        if (err) {
                return next(err);
        } else {
                res.json(user);
        }
});
};
```

Updating an existing user document

- ❑ Wire your new update() method in your users' routing module.
- ❑ Go to *app/routes/users.server.routes.js* file and change it to look like the following code snippet:

```
const users = require('../../app/controllers/users.server.controller');
module.exports = function(app) {
app.route('/users')
.post(users.create)
.get(users.list);
app.route('/users/:userId')
.get(users.read)
.put(users.update);
app.param('userId', users.userByID);
};
```

- ❑ Run **the application** and test using: $ curl -X PUT -H "Content-Type: application/json" -d '{"lastName":"Updated"}' localhost:3000/users/[id]

Deleting an existing user document

- The Mongoose model has several available methods to remove an existing document - **remove()**, **findOneAndRemove()**, and **findByIdAndRemove()**.
- Because we use the userById() middleware, the easiest way to remove an existing document would be to simply **use the remove() method**. Go back to your *app/controllers/ users.server.controller.js* file, and add the following delete() method:

```
exports.delete = function(req, res, next) {
// use the user object to remove the correct document
req.user.remove(function(err) {
        if (err) {
                return next(err);
        } else {
                res.json(req.user);
        }
})
};
```

Deleting an existing user document

- Use your new delete() method in your users' routing file.
- Go to your *app/routes/users.server.routes.js* file and change it to look like the following code snippet:

```
const users = require('../../app/controllers/users.server.controller');
module.exports = function(app) {
app.route('/users')
.post(users.create)
.get(users.list);
app.route('/users/:userId')
.get(users.read)
.put(users.update)
.delete(users.delete);
app.param('userId', users.userByID);
};
```

- Run **the application** and test using:

```
$ curl -X DELETE localhost:3000/users/[id]
```

Implementing PUT and DELETE

- HTML5 does not support PUT and DELETE methods
- If the purpose is to create a REST API, use method-override module to provide support for them:

```
app.use(methodOverride());
//override with POST having ?_method=DELETE
app.use(methodOverride('_method'));
```

- In ejs page:

```
<form method="post" action="/delete?_method=DELETE">
     <input type="text" name="username" />
     <button type="submit" >Delete User</button>
 </form>
```

- The route handling code:

```
app.route('/delete').delete(users.deleteByUserName);
```

- Do the same for **update** operation

Using schema modifiers

- Sometimes, you may want to **perform a manipulation over schema fields before saving** them **or presenting** them to the client.
  - To do this, Mongoose uses a feature called **modifiers**.
  - A **modifier can either change the field's value before saving the document** or represent it differently at query time.
- The simplest modifiers are the **predefined** ones included with Mongoose.
  - For instance, **string-type fields can have a trim modifier to remove whitespaces**, an **uppercase modifier** to uppercase the field value, and so on.

Using schema modifiers

- ❑ Ex: Let's make sure the username of your users is **clear from a leading and trailing whitespace**.
- ❑ Change your *app/models/user.server.model.js* file to look like the following code snippet:

```
const mongoose = require('mongoose'),
Schema = mongoose.Schema;
const UserSchema = new Schema({
firstName: String, lastName: String, email: String,
username: {
        type: String,
        trim: true
        },
password: String,
created: {
        type: Date,
        default: Date.now
        }
});
mongoose.model('User', UserSchema);
```

Custom setter modifiers

- Custom **setter modifiers handle data manipulation before saving the document**.
- Let's add a new *website* field to User model.
- The website field should begin with 'http://' or 'https://', but instead of forcing your customer to add this in the UI, you can simply **write a custom modifier that validates the existence** of these prefixes and adds them when necessary.
- To add your custom modifier, you will need to create the new *website* field with a *set* property as follows:

Custom setter modifiers

```
const UserSchema = new Schema({
...
website: {
    type: String,
    set: function(url) {
        if (!url) {
            return url;
        } else {
            if (url.indexOf('http://') !== 0 && url.indexOf('https://')
            !== 0) {
            url = 'http://' + url;
        }
            return url;
        }
    }
},
...
});
```

Custom getter modifiers

- Getter modifiers are **used to modify existing data before outputting the documents to next layer**.
- For instance, in our previous example, a getter modifier would sometimes be better to change already existing user documents by **modifying their website field at query time** instead of going over your MongoDB collection and updating each document.
- To do so, all you have to do is change your **UserSchema** like the following code snippet:

Custom getter modifiers

```
const UserSchema = new Schema({
...
website: {
    type: String,
    get: function(url) {
    if (!url) {
        return url;
    } else {
        if (url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0) {
            url = 'http://' + url;
        }
            return url;
        }
    }
},
...
});
UserSchema.set('toJSON', { getters: true });
```

Custom getter modifiers

- You simply changed the setter modifier to a getter modifier by **changing the *set* property to *get***.
- You configured your schema using **UserSchema.set()**.
- This will force Mongoose to include **getters** when converting the MongoDB document to a JSON representation and will allow the output of documents using **res.json()** to include the getter's behavior.
- If you didn't include this, you would have your document's JSON representation ignoring the getter modifiers.

Adding virtual attributes

- Sometimes you may want to have **dynamically calculated document properties**, which are not really presented in the document.
- These properties are called **virtual attributes** and can be used to address several common requirements.
- For instance, let's say you want to add a new *fullName* field, which will represent the concatenation of the user's first and last names.
- To do so, you will have to use the **virtual()** schema method, so a modified **UserSchema** would include the following code snippet:

```
UserSchema.virtual('fullName').get(function() {
return this.firstName + ' ' + this.lastName;
});
UserSchema.set('toJSON', { getters: true, virtuals: true });
```

Adding virtual attributes

- **Virtual attributes can also have setters** to help you save your documents as you prefer instead of just adding more field attributes.
- Let's say you wanted to break an input's *fullName* field into your **first and last name fields**.
- To do so, a modified virtual declaration would look like the following code snippet:

```
UserSchema.virtual('fullName').get(function() {
return this.firstName + ' ' + this.lastName;
}).set(function(fullName) {
var splitName = fullName.split(' ');
this.firstName = splitName[0] || '';
this.lastName = splitName[1] || '';
});
```

Optimizing queries using indexes

- ❑ Mongoose also supports the **indexing functionality** and even allows you to define **secondary indexes**.
- ❑ The basic example of indexing is the **unique index**, which validates the uniqueness of a document field across a collection.
- ❑ In our example, it is common to **keep usernames unique**, so in order to tell that to MongoDB, you will need to **modify your UserSchema definition** to include the following code snippet:

```
const UserSchema = new Schema({
...
username: {
        type: String,
        trim: true,
        unique: true // unique index keeps usernames unique
},
...
});
```

Optimizing queries using indexes

- Mongoose also supports the **creation of secondary indexes** using the **index property**.
  - For example, if your application will use a lot of queries involving the email field, you could **optimize these queries** by creating an e-mail secondary index as follows:

```
const UserSchema = new Schema({
...
email: {
type: String,
index: true // secondary index
},
...
});
```

Defining custom static methods

- Model **static methods** allow to **perform model-level operations**, such as **adding extra find methods**.
- For instance, let's say you want to search users by their username.
  - To add a static method, you will need to declare it as a member of your schema's **statics** property.
  - In our case, adding a *findOneByUsername()* method would look like the following code snippet:

```
UserSchema.statics.findOneByUsername = function (username, callback) {
this.findOne({ username: new RegExp(username, 'i') }, callback);
};
```

Defining custom static methods

- Using the new findOneByUsername() method would be similar to using a standard static method by calling it directly from the User model as follows:

```
User.findOneByUsername('username', function(err,
user){
...
});
```

Defining custom instance methods

- Mongoose offers support for **instance methods**, helping you slim down your code base and properly **reuse your application code**.
- To add an instance method, you will need to **declare it as a member of your schema's *methods* property**.
- Let's say you want to **validate your user's password** with an authenticate() method.
  - Here is how you declare it:

```
UserSchema.methods.authenticate = function(password) {
return this.password === password;
};
```

- You can call the **authenticate()** method from any User model instance as follows:

```
user.authenticate('password');
```

Model validation

- When users input information to your application, you'll often have to **validate that information before passing it on to MongoDB**.
- It is more useful to **do validation at the model level**.
- Mongoose supports both simple **predefined validators** and more **complex custom validators**.
- Validators are **defined at the field level** of a document and are **executed when the document is being saved**.
- If a validation error occurs, the ***save* operation is aborted and the *error* is passed to the callback**.

Predefined validators

- ❑ Mongoose supports different types of predefined validators, most of which are type-specific.
- ❑ **Required validator** example: let's say you want to verify the existence of a *username* field before you save the user document.
- ❑ Change your UserSchema:

```
const UserSchema = new Schema({
...
username: {
        type: String,
        trim: true,
        unique: true,
        required: true // username is a required field
},
...
});
```

Predefined validators

- ❑ Mongoose also includes **type-based predefined validators**, such as the **enum** and **match** validators for strings.
- ❑ For instance, to **validate your email field**, change your UserSchema as follows:

```
const UserSchema = new Schema({
...
email: {
        type: String,
        index: true,
        match: /.+\@.+\..+/
},
...
});
```

- ❑ **match** validator here will make sure the **email field value matches the given regex expression**, thus preventing the saving of any document where the e-mail doesn't conform to the right pattern.

enum validator

- Can help you define a **set of strings that are available for that field value**.
- Let's say you **add a role field**. A possible validation would look like this:

```
const UserSchema = new Schema({
...
role: {
      type: String,
      enum: ['Admin', 'Owner', 'User']
},
...
});
```

- The preceding condition will allow the **insertion of only these three possible strings**, and thus prevent you from saving the document.

Custom validators

- Mongoose also enables you to define your own **custom validators** using the ***validate*** **property**.
- The validate property value should be an **array** consisting of a **validation function** and an **error message**.
- Ex: to validate the length of your user's password - make these changes in your UserSchema:

```
const UserSchema = new Schema({
...
password: {
        type: String,
        validate: [
                 function(password) {
                 return password.length >= 6;
                 },
          'Password should be longer'
        ]
        },
...
});
```

Using Mongoose middleware

- Mongoose middleware are functions that can intercept the process of the *init*, *validate*, *save*, and *remove* instance methods.
- Middleware are **executed at the instance level** and have two types: **pre middleware** and **post middleware**.
- Pre middleware gets **executed before the operation happens**.
  - For instance, a **pre-save middleware** will get executed before the saving of the document.
- This functionality makes pre middleware **perfect for more complex validations and default values assignment**.

Using pre middleware

- A pre middleware is **defined using the pre() method of the schema object**, so validating your model using a pre middleware will look like the following code snippet:

```
UserSchema.pre('save', function(next) {
if (...) {
    next()
} else {
    next(new Error('An Error Occured'));
}
});
```

Using post middleware

- A **post middleware** gets **executed after the operation happens**.
- For instance, a post-save middleware will get executed after saving the document - perfect to log your application logic.
- A post middleware is **defined using the post() method of the schema object**, so logging your model's save() method using a post middleware will look something like the following code snippet:

```
UserSchema.post('save', function(next) {
if(this.isNew) {
    console.log('A new user was created.');
} else {
    console.log('A user updated is details.');
}
});
```

Using Mongoose DBRef

- Although **MongoDB doesn't support joins**, it does **support the reference of a document to another document using a convention named *DBRef***.
- Mongoose includes support for DBRefs using the ObjectID schema type and the use of the ***ref*** property.
- Mongoose also **supports the population of the parent document with the child document** when querying the database.
- Ex: Let's create another schema for blog posts called **PostSchema**.
- Because a user authors a blog post, PostSchema will contain an *author* field that will be populated by a User model instance.
- So, a PostSchema will have to look like the following code snippet:

Using Mongoose DBRef

```
const PostSchema = new Schema({
title: {
      type: String, required: true
},
content: {
      type: String, required: true
},
author: {
      type: Schema.ObjectId, ref: 'User'
}
});
mongoose.model('Post', PostSchema);
```

- ❑ Notice the ***ref*** property telling Mongoose that **the *author* field will use the User model to populate the value**.

Using Mongoose DBRef

- Using this new schema is a simple task.
- To create a new blog post, you will need to retrieve or create an instance of the User model, create an instance of the Post model, and then assign the post *author* property with the user instance.
- Ex:

```
const user = new User();
user.save();
var post = new Post();
post.author = user;
post.save();
```

- Mongoose will create a DBRef in the MongoDB post document and will later use it to retrieve the referenced document.

Using Mongoose DBRef

- Since the DBRef is only an ObjectID reference to a real document, Mongoose **will have to populate the post instance with the user instance**.
- To do so, you'll have to tell Mongoose to populate the *post* object using the **populate()** method **when retrieving the document**.
- For instance, a **find()** method that populates the author property will look like the following code snippet:

```
Post.find().populate('author').exec(function(err, posts) {
...
});
```

- Mongoose will then **retrieve all the documents in the *posts* collection and populate their *author* attribute**.

References

- Textbook
- https://docs.mongodb.com/manual/reference/method/
- http://mongoosejs.com/docs/
- http://expressjs.com/
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
- https://github.com/expressjs/method-override
- https://github.com/philipmat/method-override-examples/blob/master/index.html
- http://mongoosejs.com/docs/populate.html
- https://github.com/Microsoft/nodejstools/wiki/Debugging