Extra Notes - Node.js - API with custom auth

- Dr Nick Hayward

A brief outline of a basic API for a user authenticated TODOS app with Node.js &c.

Contents

intro

We can now add users and authentication to the previous node-api-todos app.

app - Mongoose - user model

For this app, we'll need an example user model,

```
{
  email: 'test@test.com', // validated to check email structure
  password: 'jkhujshunkji897hhuh7', // password from user –
  tokens: [{
    access: 'auth',
    token: 'kjdisknksanudnjnasnlasnck'
  }]
}
```

This model includes:

- email - validated to check correct structure to match defined emails
- password - sent from the user (client-side) as plain text
    - hashed on the server-side using the *BCrypt* algorithm - a one-way hash that can't be reversed from hash
- tokens - array of objects for login
    - each token will have access type - e.g. auth
    - each token will have the token value itself - a token string that is passed back and forth to check credentials for access &c.

So, we can now update our user model for email, e.g.

```
// specify model for user
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true, // checks that email is unique in current system
    validate: {
      validator: (value) => {

      },
      message: '{VALUE} is not a valid email'
    }
  }
});
```

We add a new property for `unique` and set its value to `true` to ensure that each user's email in the system is unique.

Then, we need to validate each email address for correct, expected structure.

We can add *Mongoose Validation* for this purpose, which includes a custom option. So, we can add a validate object, and pass the value to test and validate. Then we simply return whether its valid or not.

It's a third party library, *validator*, that is performing the actual validation for the email address. It's available on NPM,

```
npm install validator --save
```

After importing the library and mongoose,

```
// require mongoose module - not custom mongoose config file
const mongoose = require('mongoose');
// require validator module
const validator = require('validator');
```

we can use the `isEmail` function to the `validate` object. e.g.

```
// specify model for user
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true, // checks that email is unique in current system
    validate: {
      validator: (value) => {
        return validator.isEmail(value);
      },
      message: '{VALUE} is not a valid email'
    }
  }
});
```

After `email`, we can then add `password`, e.g.

```
// specify model for user
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true, // checks that email is unique in current system
```

```
    validate: {
      validator: (value) => {
        return validator.isEmail(value);
      },
      message: '{VALUE} is not a valid email'
    }
  },
  password: {
    type: String,
    require: true,
    minlength: 6 // min length for user password
  }
});
```

The password is straightforward and only requires the type, a boolean value for whether it's required or not, and the minimum length for the user's password.

Then, we can add `tokens`. This is an array, and is a feature not currently available in standard SQL DBs.

```
// specify model for user
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true, // checks that email is unique in current system
    validate: {
      validator: (value) => {
        return validator.isEmail(value);
      },
      message: '{VALUE} is not a valid email'
    }
  },
  password: {
    type: String,
    require: true,
    minlength: 6 // min length for user password
  },
  tokens: [{
    access: {
      type: String,
      required: true
    },
    token: {
      type: String,
      required: true
    }
  }]
});
```

Within the `tokens` array, we can add an object that defines the properties we need for a given token. For example, we currently need to define `access` and the `token` itself.

Initially, all we need to add is the type as a string, and a boolean to ensure that both tokens are required.

app - build API - POST route for adding users

We can now update the app's `server.js` file adding a new post route to add users to the app.

We'll start with a standard POST route, e.g.

```
// POST route for adding a user - /users
app.post('/users', (req, res) => {

});
```

We're calling the `post` method on the app object, and passing the required route URL for `/users`. We then add a standard callback for the request and response.

We can then use Lodash's `pick` method to retrieve specific properties of the request, e.g.

```
var body = _.pick(req.body, ['email', 'password']);
```

The `pick()` method expects the object first, i.e. the body, and then the required properties, such as email and password.

Then, we can create a new instance of the user model, passing the recently created `body`. This contains the email and password required for the specified user model. e.g.

```
var user = new User(body);
```

We can then save this user object to the DB, and chain a `then` method to handle the response. Later, this will include authentication &c.

```
user.save().then((user) => {
  res.send(user);
}).catch((error) => {
  res.status(400).send(error);
})
```

So, our current POST route for users is as follows,

```
```

```
// POST route for adding a user - /users
app.post('/users', (req, res) => {
  var body = _.pick(req.body, ['email', 'password']);
  var user = new User(body);

  user.save().then((user) => {
    res.send(user);
  }).catch((error) => {
    res.status(400).send(400);
  })
});
```

app - testing - test POST /users route with Postman

After restarting the app's server, and wiping the current local DB for the app, we can then add a new test route to Postman.

In the app's local environment in Postman, we can add a `POST` route for the following URL,

```
{{url}}/users
```

and then add some test data to the body of the POST request, e.g.

```
{
  "email": "test@test.com",
  "password": "1234defg"
}
```

If we then send this POST request to the DB, we'll get the following saved to the DB with the expected 200 status code,

```
{
    "__v": 0,
    "email": "test@test.com",
    "password": "1234defg",
    "_id": "598da28e4f1e385524716c4e",
    "tokens": []
}
```

The password is not hashed, but we can add that later. Also, if we then try to send the same data we'll get the expected 400 status code for a duplicate entry.

app - build API - authenticating users

Our current API exposes each route to the public, creating issues if we need to publish this API for app usage. With this in mind, we need to set these routes to private, forcing a user to authenticate successfully prior to using the API.

We can use tokens to help with this authentication of API requests. Upon successful login to the app a user is provided with a token. They can use this token for subsequent authentication for each required API route. e.g. if they need to update or delete a TODO item they created.

Each token will simply be passed as part of the header request for a given route in the API.

authentication - the concept of hashing

Hashing is a one-way crypto algorithm. This means that if we provide a string &c. as the input, it will always produce the same hashed result. However, it cannot be reversed easily.

So, hashing is predictable.

For our app, we can install the NPM module `crypto-js`, which includes many cryptography algorithms, e.g.

```
npm install crypto-js --save
```

Then, we can create a new test for hashing,

```
// require npm module - crypto-js for SHA256 hashing algorithm
const {SHA256} = require('crypto-js');

// define input string
var phraseInput = 'the heart of a star';
// hash string using crypto-js
var hashOutput = SHA256(phraseInput);

// console output phrase input
console.log(`Phrase input = ${phraseInput}`);
// console output hash object as string
console.log(`Hash output = ${hashOutput.toString()}`);
```

This will output the following example result,

```
Phrase input = the heart of a star
Hash output =
5e01a837cf5a01499644eefca1a9cf97cd7600f451ad7e38ee129b47e3ace435
```

So, in a real world app we can use hashing to check a password entered by a user. A user will simply enter the string for their password, the app will hash that string, and then compare the created hash to the hash stored in the app's DB. This way, we can store the hash in the DB and not the plain text string.

authentication - hashing and salting

We can use hashing in our app as follows, e.g.

- we create an object that contains a property for the user's ID, e.g.

```
var data = {
  id: 7
}
```

This id will tell the app which user can make the request, such as deleting a TODO item for user id 7. So, a user with ID = 4 should obviously not be able to delete or modify a document belonging to a user with ID = 7.

- we need to ensure that the token we send for the requested user ID = 7 is not modified by the client to another user's ID. If this happened, of course, then the client could simply delete any user's data.
- to be able to send the user ID to the user, we can create a ***token*** object that includes the specified data object above plus a hash property. This will simply be the hashed value of the data object. e.g.

```
var token = {
  data,
  hash: SHA256(JSON.stringify(data)).toString()
}
```

This gives us a hashed token, which we can send to the user on the client-side to allow them to authenticate and modify the app for the requested user ID = 7.

However, this token is currently not foolproof.

- if the user changes the value of the data property in the token to user ID = 3, and then rehash that data, they'll be able to authenticate now as user ID = 3 instead of ID = 7. Again, this would be a serious security flaw in the app and the API.

- to prevent this modification, we can also **salt** the hash by adding a unique value to the hash that is returned in the token. This unique salting value will only be known by the app, and can then be used to check the value of the hash property in the token. This unique salt value is not known to the user.

- if the user tries to hash a user ID &c. without this salt value, the app will simply deny authentication to the app and API.

So, we may now create a token as follows,

```
// user id
var data = {
  id: 7
};
// generated token for the id data - hash and salting
var token = {
```

```
  data,
  hash: SHA256(JSON.stringify(data) + 'salted').toString()
}
```

So, the user may try to modify the value of the data property to a different user ID, but they can't replicate the expected hash without the value of the salting on the server-side of the app.

authentication - validate hashed and salted token

We can run a cursory check for the token, and the expected hashed and salted result. i.e. we can compare the passed token from the client with the expected result on the server. e.g.

```
// create expected token hash - hashed and salted
var serverHash = SHA256(JSON.stringify(token.data) + 'salted').toString();
// compare client and server token hash values
if (serverHash === token.hash) {
  console.log('token hash validated...');
} else {
  console.log('token hash did not validate...do not trust user');
}
```

We can now send a user a token, which has been hashed and salted. When the user tries to authenticate with this token, we can check it on the server side and ensure that it can't be easily changed to a different user's ID.

n.b. This standard is known as the JSON Web Token. There are many mature libraries in NPM to help us add this functionality to a live, production app.

authentication - adding authentication with JSON Web Token (JWT)

A common option for adding authentication is to use a *JSON Web Token* or JWT. Further info is available at the following URL,

- https://jwt.io

We can now install a NPM module for JWT, e.g.

```
npm install jsonwebtoken --save
```

The benefit of this library is its simplicity of usage for authentication, and its use as an industry standard for authentication.

It allows us to sign a data object, hashed and salted, and then authenticate a request as well. We can use two methods for this purpose, `sign()` and `verify()`.

`sign()` can be called on the data object that contains the user request, e.g. their user ID. This will create a signed token that is hashed and salted. e.g.

```
const jwt = require('jsonwebtoken');

var data = {
  id: 7
};
// pass data object and secret for salting...
var token = jwt.sign(data, 'salted');
// console output token
console.log(token);
```

To verify this token for subsequent requests, we can then use the JWT method, `verify()`. e.g.

```
// pass token to verify with secret for salting...
var tokenDecode = jwt.verify(token, 'salted');
// console output decoded token
console.log('token decoded = ', tokenDecode);
```

An example return is as follows,

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNTAyNDc2MDA0fQ.hu
_B9qzOtW3Mlu1YYbcWDf-3S3PxcydbdwwArjCTwJg
token decoded =  { id: 7, iat: 1502476004 }
```

The return object for the decoded token includes the original ID from the data object, i.e. id = 7, and the `iat` or *issued at timestamp*. This timestamp basically records when the token was created.

This return is known as the *payload* in JWT.

app - build API - add auth tokens

We can start by adding a return token for each new user. A user signs up for the app, and they are then logged in with a return token issued.

So, we can add this to the `POST /users` API route in `server.js`.

We'll add it as an abstracted, reusable method.

n.b. Quick segue into methods

- *model* methods - these are methods that we add to a specified model via a defined schema. These methods are then only available for objects of that model.
- *instance* methods - called against an instantiated object, which encompasses the values of the model. So, we call the instance method against an object, which may contain a document with values, e.g. `user` from `User`. So, we should expect to be able to read a user's `id` &c.

To be able to use this type of *model* methods, we need to update our `User` model to set a schema. By creating a schema for the model, we can then add our custom methods to the model. If not, node.js won't allow us to

add a method to the non-schema based original user model. (it needs a schema...)

e.g. we can update the user model file, `user-model.js` as follows:

```
var UserSchema = new mongoose.Schema({
    email: {
      type: String,
      required: true,
      trim: true,
      minlength: 1,
      unique: true, // checks that email is unique in current system
      validate: {
        validator: (value) => {
          return validator.isEmail(value);
        },
        message: '{VALUE} is not a valid email'
      }
    },
    password: {
      type: String,
      require: true,
      minlength: 6 // min length for user password
    },
    tokens: [{
      access: {
        type: String,
        required: true
      },
      token: {
        type: String,
        required: true
      }
    }]
});

// add instance method to UserSchema objects - needs standard function
syntax to provide `this` keyword
UserSchema.methods.generateAuthToken = function () {
  var user = this; // provides access to a document - the document this
method was called against...
  // get access value from tokens in schema
  var access = 'auth';
  // create token for user from schema
  var token = jwt.sign({_id: user._id.toHexString(), access},
'salted').toString(); // temporary secret value
  // update tokens array in schema - user will now be generated with the
created token values...
  user.tokens.push({access, token});
  // save user - returns promise in server.js where it will be called and
used...
  return user.save().then(() => {
    return token; // returns value as success value for next then() in
server.js call
```

```
  });
};

var User = mongoose.model('User', UserSchema);
```

n.b. we need to require the JWT library to be able to generate the token for the user, e.g.

```
// require npm module - json web token - jwt
const jwt = require('jsonwebtoken');
```

We can now use this updated model and method in the `server.js` file in the `POST /users` API route when we save a user, e.g.

```
user.save().then(() => {
  // call method from user model - use return token in promise chain...
  return user.generateAuthToken();
}).then((token) => {
  // send back user with the updated token...and send custom header for
auth token
  res.header('x-auth', token).send(user); // `x-...` in a header indicates
a custom header...
}).catch((error) => {
  res.status(400).send(error);
})
```

So, we're calling the new method `generateAuthToken()` on the user object. This method returns a promise, which we can pass along the chain to the next `then()` method. We can use this token to set the header for the response when we send the user object back. We're setting a custom header, denoted by `x-`, and setting the token as part of this custom header.

If we then test these updates with Postman, `POST /users` route, a new user will be created as follows,

```
{
    "__v": 1,
    "email": "test@test.com",
    "password": "1234defg",
    "_id": "598f09c59ce7b87e49c8af8c",
    "tokens": [
        {
            "access": "auth",
            "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1OThmMDljNTljZTdiODdlNDlj
OGFmOGMiLCJhY2Nlc3MiOiJhdXRoIiwiaWF0IjoxNTAyNTQ2MzczfQ.ptLPHSI_Ki0ysIG5iir
C_aWT10z6XzlKVaejMUUTpac",
            "_id": "598f09c59ce7b87e49c8af8d"
        }
```

```
    ]
}
```

The user object is still visible, but this is something we can fix later.

If we also check the headers for this http POST request we'll get the following response header, e.g.

```
x-auth
→eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1OThmMDljNTljZTdiODdlNDlj
OGFmOGMiLCJhY2Nlc3MiOiJhdXRoIiwiaWF0IjoxNTAyNTQ2MzczfQ.ptLPHSI_Ki0ysIG5iir
C_aWT10z6XzlKVaejMUUTpac
```

This is the custom header we send back in the user response. It's also the custom header we can send when we want to make a secure http request to todos routes in the API, e.g. POST, GET, DELETE &c. for specified todo items in the DB.

app - build API - override method in user model for security purposes

We can also override a method in the User model, which allows us to restrict the values returned in the visible object for the user. e.g. above we can see the return for a newly created user, which includes the password and tokens array. We need to restrict the properties of the user object to email and id.

So, we can update our user-model.js file to override a JSON method. e.g.

```
UserSchema.methods.toJSON = function () {
  var user = this;
  // take mongoose object and convert to regular object - only properties
from user object will exist
  var userObject = user.toObject();
  // return specific user object properties - use pick method from lodash
  return _.pick(userObject, ['_id', 'email']); // only return id and email
for a user object...
};
```

n.b. we'll need to require Lodash to use the `pick()` method in the user-model.js file.

If we now test this update with Postman, again submitting a POST http request for a new user, we'll get the following response object, e.g.

```
{
    "_id": "598f0d0b8aa4c27f7cc2a76d",
    "email": "test2@test.com"
}
```

This is now more secure, only returning the user's id and email address as part of the response object. The headers remain the same, including the token, which is what we need to authenticate other http requests in the API.

app - build API - add private route - GET /users/me

We can now add a new route to the API for `/users`, which will show details for the authenticated user. This route will need to be private, and authenticated with a `x-auth` token from the header.

```
app.get('/users/me', (req, res) => {
  // get token from header - req will sent for the returned header
  var token = req.header('x-auth'); // key from the header is `x-auth`

  ...
});
```

So, we'll start by requesting the header with the token we need to authenticate the user. This header is the custom header we set as the user is authenticated successfully.

Next, we need to verify the requested token, authenticate the user, and then return data &c. for the requested route.

As this pattern is used many times in an app, we can abstract this logic to the User model in `user-model.js` as a model method, e.g.

```
// model method for token authentication - statics defines method as a
models method
UserSchema.statics.findByToken = function (token) {
  // user model = `this` binding - because this is a model method...
  var User = this;
  // store decoded jwt values
  var decoded;

  // catch any errors for verify()
  try {
    decoded = jwt.verify(token, 'salted'); // pass token to verify plus
secret phrase for salting...

  } catch (error) {
    ...
  }

  // return promise to query (i.e. in server.js) for requested user values
  return User.findOne({
    _id: decoded._id,
    'tokens.token': token, //quotation marks required due to period in
tokens.token
    'tokens.access': 'auth'
```

```
  });
};
```

So, we can add a model method for the authentication of the user by token. To ensure this is defined as a model method, we can use `statics`. Then we can set the `this` binding for this method to the User model.

If the authentication and query is successful, we can then return a promise for the found User. This will include the `id` of the user, and the token and access values as well.

In `server.js`, we can now update the private route for `/users/me`, e.g.

```
app.get('/users/me', (req, res) => {
  // get token from header - req will sent for the returned header
  var token = req.header('x-auth'); // key from the header is `x-auth`

  // call user model method with token - check return promise for user
  User.findByToken(token).then((user) => {
    // check user return
    if (!user) {
      ...
    }

    // success - user authenticated token...
    res.send(user);
  });
});
```

So, we can now call the custom User model method to find the requested user.

This will now work assuming we send a valid `x-auth` header. We have not yet added any error handling, but the authentication is now working for this private route.

We can test this in Postman as well.

For error handling, we can update the logic as follows, e.g. in `user-model.js` we can update the `findByToken()` method

```
// model method for token authentication - statics defines method as a
models method
UserSchema.statics.findByToken = function (token) {
  // user model = `this` binding - because this is a model method...
  var User = this;
  // store decoded jwt values
  var decoded;

  // catch any errors for verify()
  try {
    decoded = jwt.verify(token, 'salted'); // pass token to verify plus
secret phrase for salting...
```

```
  } catch (error) {
    return new Promise((resolve, reject) => {
      reject();
    });
  }

  // return promise to query (i.e. in server.js) for requested user values
  return User.findOne({
    _id: decoded._id,
    'tokens.token': token, //quotation marks required due to period in
tokens.token
    'tokens.access': 'auth'
  });
};
```

In `server.js`, we can use this new error handling as follows, e.g.

```
app.get('/users/me', (req, res) => {
  // get token from header - req will sent for the returned header
  var token = req.header('x-auth'); // key from the header is `x-auth`

  // call user model method with token - check return promise for user
  User.findByToken(token).then((user) => {
    // check user return
    if (!user) {
      ...
    }

    // success - user authenticated token...
    res.send(user);
  }).catch((error) => {
    // send back 401 status - error code
    res.status(401).send();
  });
});
```

So, from the `findByToken()` method in the `user-model.js` file we can return a promise if an error is detected. This error will be a promise with a reject value.

In `server.js`, this will be detected if we chain a `catch()` method to the current promise chain. Then, we can simply return an updated status code of `401` to confirm an error in the request.

We can also update the `/users/me` route if no user is found. i.e. the user may not exist in the DB.

```
  // check user return
  if (!user) {
    return Promise.reject();
  }
```

So, the updated route is as follows,

```
app.get('/users/me', (req, res) => {
  // get token from header - req will sent for the returned header
  var token = req.header('x-auth'); // key from the header is `x-auth`

  // call user model method with token - check return promise for user
  User.findByToken(token).then((user) => {
    // check user return
    if (!user) {
      // reject promise - code execution stops and exits...
      return Promise.reject();
    }

    // success - user authenticated token...
    res.send(user);
  }).catch((error) => {
    // send back 401 status - error code
    res.status(401).send();
  });
});
```

app - build API - abstract and refactor private route authentication

We can now add some custom middleware to help us abstract the authentication process for more than a single route.

We can move most of the code we used previously in the `app.get()` route into its own `authenticate()` function, e.g.

```
// three arguments for middleware - request, response, next
var authenticate = (req, res, next) => {
  // get token from header - req will sent for the returned header
  var token = req.header('x-auth'); // key from the header is `x-auth`

  // call user model method with token - check return promise for user
  User.findByToken(token).then((user) => {
    // check user return
    if (!user) {
      // reject promise - code execution stops and exits...
      return Promise.reject();
    }

    // modify request object - user
    req.user = user; // set user in request object to user just found...
    // modify request object - token
    req.token = token; // set token to above token from `x-auth` header
    // call next to run code in function that calls this middleware
```

```
    next();
  }).catch((error) => {
    // send back 401 status - error code
    res.status(401).send();
  });
};
```

The main change to the previous authentication code in the `/users/me` route is the way we handle the user and token. We can add the found `user` and `token` to the `req` object, which can then be used in an API route that requires authentication.

To reference this middleware in the API route, we can modify our route as follows, e.g.

```
app.get('/users/me', authenticate, (req, res) => {
  // send the user from the authenticated request
  res.send(req.user);
});
```

Then, we can move this middleware to its own file, `authenticate.js` in a new directory, `middleware`.

We can also test this abstraction with Postman as before.

authentication - hashing - passwords

A user will send a plain text version of their password when submitting a request to the POST `/users/me` request. This plain text password will be validated. If validation is successful, the app will then hash the password, salt it as well, before saving it to the database.

The hashing will be completed using an algorithm called ***bcrypt***, which has salting built in as well.

We can install a NPM module for ***bcrypt***, e.g.

```
npm i bcryptjs --save
```

There are many ***bcrypt*** modules available, including the popular ***bcryptjs***. Further details on ***bcryptjs*** is available at the following URL,

- https://www.npmjs.com/package/bcryptjs

authentication - hashing - testing bcrypt

We'll start with a few basic tests for using the ***bcryptjs*** module in a new test file in the playground directory, e.g. `playground/hash-tests/hash-bcrypt.js`.

We'll start by requiring this module as usual,

```
```

```
const bcrypt = require('bcryptjs');
```

and then we can add an initial test, e.g.

```
// test pasword string
var password = "abcd1234";
```

So, we can now test salting and hashing a password. We start by generating a salt value using *bcryptjs*.

`bcrypt.genSalt()` takes two arguments, which include a callback function for the second. The first argument is the number of rounds to use to generate the salt. The more rounds, the slower the bcrypt algorithm performs. However, slower may be preferable for passwords, as it will also prevent continuous brute force attacks on the server.

So, this salting is as follows, e.g.

```
// generate salt and then hash - 15 rounds to salt password
bcrypt.genSalt(15, (error, salt) => {
  // three arguments for hash() - password to hash, salt just created, and
callback for success and failure
  bcrypt.hash(password, salt, (error, hash) => {
    console.log(hash);
  });
});
```

With this example, we create a salt using 15 rounds, and then hash the plain text password with the generated salt value. We can then console output the hash value (with salt) to check expected return.

If we run this test example, we'll get a hash value with salt output to the console, e.g.

```
$2a$15$1k.7jAGIqXSbMknskqzvCeCFjv.bENpB1kd/R7.6PJUlby/fQIpOu
```

This hash value can then be saved to a DB &c. as a way to authenticate a user.

We can test this hash against a passed plain text password, i.e. from a user submission, as follows,

```
// test hashed password
var hashedPassword =
'$2a$15$1k.7jAGIqXSbMknskqzvCeCFjv.bENpB1kd/R7.6PJUlby/fQIpOu';
// three arguments - plain text password, hashed password, and callback
function for success or fail
bcrypt.compare(password, hashedPassword, (err, res) => {
  // console output boolean result of comparison
```

```
  console.log(res);
});
```

The compare function expects the hashed password, the plain text password sent by the user, and then returns whether they match.

The third argument is a callback function, which will either return an error with the function itself or a boolean value for the comparison. If the passwords match then the boolean will be set to TRUE, and FALSE if they do not.

So, we'll use *bcrypt* as follows,

- user submits initial plain text password
- bcrypt will hash and salt plain text password
- hashed password saved to DB
- user submits plain text password to login to app
- hashed password is retrieved from DB for current user
- bcrypt compares passwords and returns boolean for match or no match
- user is either allowed to login or refused entry...

app - build API - add bcrypt hashing and salting

We can now add *bcrypt* to our app. We can also update the `user` model for the app.

To ensure that hashing is used before the app saves any document, we need to add some management for the middleware. For this, we can now add Mongoose middleware. This allows us to run some code either before the user model is executed or after execution.

In this update, we need to ensure that the hashing code is run before we try to save any document to the app's DB.

Further information for Mongoose Middleware is available at the following URL,

- https://mongoosejs.com/docs/middleware.html

So, to use this middleware we call one of the availabe methods on the defined schema. Methods include `pre()` for before the schema is executed. Then, we simply call the `next()` method in the `pre()` method, and the schema will continue to execute. e.g.

```
// run some code before the schema executes save...e.g. hash passwords
before save
UserSchema.pre('save', function(next) {
  // access document
  var user = this;
  // check if password is modified - if yes, then hash and salt is
required
  if (user.isModified('password')) {
    // hash and salt password - before running save
    bcrypt.genSalt(15, (error, salt) => { // generate salt
      // hash password with salt
```

```
      bcrypt.hash(user.password, salt, (error, hash) => {
        // update user document with the hashed and salted password
        user.password = hash;
        // call next() to continue execution of schema - doc will be saved
with hashed password...
        next();
      });
    });
  } else {
    // if not modified - continue schema execution
    next();
  }
});
```

In this example, we want the hash and salt check for the password to run before the save function is executed for the schema. We also need to check whether the password has been modified or not. If modified is true, then we can hash and salt the password. If not, we can simply continue to execute the schema code.

We need to check password modification to ensure that we don't repeatedly hash and salt a password. Otherwise, we'll end up with a hashed password that has been hashed repeatedly, and will then not match the submitted password from the user.

We also need to ensure that we `require` *bcrypt* in the `user-model.js` file.

We can then run this update and test the app with Postman.

app - build API - seed test data (dummy data) to DB for users

As we added seed data for the `todos` route, we also need some dummy data so we can effectively test the API for the `users` routes.

We'll start by adding a test file for adding some dummy data for API routes, including the current `todos` and `users`, e.g. `/server/tests/seed.js`

Then, we can refactor code from the current `server.test.js` file to seed some dummy data for each route, `todos` and `users`.

We'll start by testing the current server tests, e.g.

```
npm test
```

and then start to update the new `seed.js` file. We'll start by moving the `todos` array to the seed test file, e.g.

```
// update dummy todo items with test ID name:value pair property
const todos = [
  {
    _id: new ObjectID(),
    text: 'a todo item...'
  },
```

```
  {
    _id: new ObjectID(),
    text: 'another todo doc item...',
    completed: true,
    completedAt: 230797
  }
];
```

We'll need to require the `ObjectID` property from `mongodb` in the new test file, `seed.js`, so we can use it in the `todos` function, e.g.

```
const {ObjectID} = require('mongodb'); // ObjectID property from MongoDB
object
```

Then, we can move the function that is currently inside the `beforeEach()` function in `server.test.js` and create its own function in `seed.js`,

```
// create some dummy todos for testing
const populateTodos = (done) => {
  Todo.remove({}).then(() => {
    return Todo.insertMany(todos);
  }).then(() => done());
}
```

For this function to work, we'll also need to require the `todos` model, e.g.

```
const {Todo} = require('./../models/todo-model.js');
```

As before, we then need to export `todos` and the `populateTodos()` function,

```
module.exports = {
  todos,
  populateTodos
};
```

Having now refactored the `todos` dummy data functions, we can then require the `seed.data.js` file in the main `server.test.js` file, e.g.

```
const {todos, populateTodos} = require('./seed');
```

We can then use `populateTodos` with the `beforeEach()` function in the `server.test.js` file.

Using a similar pattern, we can also add some seed dummy data to the `users` API route.

We'll start by add the user model to the `seed.js` file, as we did for todos.

```
const {User} = require('./../models/user-model.js');
```

As with the todos, we'll need an array for the test users. One user with a valid password and token. The second user will fail for authentication. So, we can create the user array as follows, e.g.

```
// create object id from mongo for first user
const userId1 = new ObjectID();
// create onject id from mongo for second user
const userId2 = new ObjectID();
// create array for users - one with token auth, the other without auth
const users = [{
  _id: userId1,
  email: "test@test.com",
  password: "userpass1",
  tokens: [{
    access: 'auth',
    token: jwt.sign({_id: userId1, access: 'auth'}, 'salted').toString()
  }]
}, {
  _id: userId2,
  email: "tester2@test.com",
  password: "userpass2"
}];
```

So, we can create the `users` array to store two test users. The first user should authenticate successfull with the token, which has been hashed and salted.

The second user should not authenticate successfully.

Then, as we did for the `todos`, we need to create a function to populate users in the DB, e.g.

```
const populateUsers = (done) => {
  User.remove({}).then(() => {
    var user1 = new User(user[0]).save();
    var user2 = new User(user[1]).save();
    // wait for both promises to return or either to fail
    return Promise.all([user1, user2]);
  }).then(() => done());
};
```

We start by remove all existing users from the DB, and chain a `then()` method to the Promise with a standard callback. In this callback, we can create our two test users for the dummy data, and use a new ES6 promise method. `Promise.all` will wait for all of the specified promises to resolve successfully before continuing.

We can then call done to close the promise chain.

To finish this seeding example for users, we can export the `users` array and `populateUsers`. We'll also need to call `populateUsers` from `server.test.js` as we did for the `todos`, e.g.

```
beforeEach(populateUsers);
```

app - testing - API routes - POST /users and GET /users/me

We can start by updating the app's `server.test.js` file to include a new test for the API route GET `/users/me`

We'll need two test cases - the first to check return user details for a successfu authentication, and the second to check correct blocking of a user with false or missing authentication. i.e. to ensure they are not sent incorrect data and have no access to the system.

So, the first test case may be setup as follows, e.g.

```
// test GET route for /users/me
describe('GET /users/me', () => {
  // test case - check user return for successful authentication...
  it('should return user if authentication successful...', (done) => {
    request(app)
      .get('/users/me') // specify api route
      .set('x-auth', users[0].tokens[0].token) // add custom header to
request
      .expect(200) // should return a 200 status code - OK
      .expect((res) => {
        expect(res.body._id).toBe(users[0]._id.toHexString());
        expect(res.body.email).toBe(users[0].email);
      })
      .end(done);
  });
});
```

We can test this new test, and it should return successful, e.g.

```
npm test
```

Then, we can add the second test case for false or missing authentication. e.g.

```
//test case - check for missing or false authentication - no x-auth header
it('should return an error code 401 for no authentication...', (done) => {
  request(app)
    .get('/users/me')
```

```
    .expect(401) // should return a 401 status code
    .expect((res) => {
      expect(res.body).toEqual({}); // retun body should be empty - not
another user's data &c.
    })
    .end(done);
});
```

So, the completed test for this API route, GET `/users/me`, will look as follows,

```
// test GET route for /users/me
describe('GET /users/me', () => {
  // test case - check user return for successful authentication...
  it('should return user if authentication successful...', (done) => {
    request(app)
      .get('/users/me') // specify api route
      .set('x-auth', users[0].tokens[0].token) // add custom header to
request
      .expect(200) // should return a 200 status code - OK
      .expect((res) => {
        expect(res.body._id).toBe(users[0]._id.toHexString());
        expect(res.body.email).toBe(users[0].email);
      })
      .end(done);
  });

  //test case - check for missing or false authentication - no x-auth
header
  it('should return an error code 401 for no authentication...', (done) =>
{
    request(app)
      .get('/users/me')
      .expect(401) // should return a 401 status code
      .expect((res) => {
        expect(res.body).toEqual({}); // retun body should be empty - not
another user's data &c.
      })
      .end(done);
  });
});
```

We can now add some additional testing for the user signup route, POST `/users`. We'll need some test cases as follows,

- should create a new user - checks for valid email and password for a new user registration
- should return error for invalid requests - e.g. email or password structure is invalid...
- should not create a new user as email already exists - email entered by user already exists in the DB, so no new user can be created

We'll set up these test cases as follows, e.g.

```
// test POST route for /users
describe('POST /users', () => {
  //test case - check user created successfully
  it('should create a new user', (done) => {
    // set some dummy data for test case
    var email = 'tester@test.com';
    var password = 'abcdef1234';

    request(app)
      .post('/users') // POST request to /users API route
      .send({email, password}) // send dummay data for a new user
      .expect(200) // should return a 200 status code - OK
      .expect((res) => {
        expect(res.headers['x-auth']).toExist(); // check that x-auth
exists in the response header
        expect(res.body._id).toExist(); // check that body exists in
response
        expect(res.body.email).toBe(email); // check email matches
expected email format &c.
      })
    .end(done);
  });
});
```

So, this test case should now pass successfully for the dummy data sent to the POST `/users` route.

We can even take this a step further by testing the data saved for the new user. i.e. we're checking the saved data for email, password hashing &c. So, instead of simply passing `done` to the `end()` function, we'll add some extra tests, e.g.

```
...
//.end(done); // basic done to close test case
.end((error) => { // expanded end to test case - check saved user
details...
  // check for error with test case
  if (error) {
    return done(error); // simply return error to end test case
  }

  // if no error returned - check values saved
  User.findOne({email}).then((user) => { // find email saved in db
    expect(user).toExist(); // user should exist - saved in db
    expect(user.password).toNotBe(password); // password should not match
password in db - hashed in db
    done();
  });
});
```

We can then add the two remaining test cases as follows,

```
// test case - check email or password structure is invalid...
it('should return error for invalid requests', (done) => {
  request(app)
    .post('/users')
    // send some poorly formed data - wrong email structure...
    .send({
      email: 'em237',
      password: 'abc123'
    })
    .expect(400) // should return a status code for the error
    .end(done); // call done to end test case
});

// test case - check email entered by user already exists in the DB
it('should not create a new user as email already exists', (done) => {
  request(app)
    .post('/users')
    // send existing email from dummy data
    .send({
      email: users[0].email,
      password: 'abcd1234'
    })
    .expect(400) // should rerurn a status code of 400 for the error
    .end(done); // call done to end the test case
});
```

For the second test case, first in the above example code, we're simply checking the validity of the email &c. structure. If the structure is invalid, then we simply return a status code of `400`.

For the last test case, second above, we're checking the db for the passed email address. We can send one of the existing email addresses in the dummy data for the tests. If it already exists in the db, we can just return a status code of 400.

We can now run these tests, and all 16 tests should pass.

app - build API - user login route - POST /users/login

We can now add a login route for the users. This will be another POST route at the URI, `/users/login`.

We allow a user to login to the app with the email and plain text password they used for signup. We'll need to query the DB for the provided email address, and a hashed version of the plain text password. This password will be hashed and salted, and then we can use ***bcrypt's*** compare method to check the plain text password against the saved password in the db.

So, we can add a new API route for POST `/users/login`. Initial route is as follows, e.g.

```
// POST route for user login - /users/login
app.post('/users/login', (req, res) => {
  // pick data for email and password from request body
```

```
  var body = _.pick(req.body, ['email', 'password']);

  // check data is being picked correctly
  res.send(body);
});
```

This allows us to quickly check that the data is being requested correctly. We can then update this route to verify that a user exists with the passed email. We'll also need to verify the hashed version of the plain text password with the copy stored in the app's db.

We can now create a model method for the user, which will check for the requested email. e.g.

```
// static for a model method - standard function call to use `this`
UserSchema.statics.findByCredentials = function (email, password) {
  var User = this;
  // query db for user with passed email - then verify password...
  return User.findOne({email}).then((user) => {
    // check if user exists
    if (!user) {
      // reject the promise if no user exists
      return Promise.reject();
    }

    // return Promise for user found in db
    return new Promise((resolve, reject) => {
      // resolve Promise for user found...3 args expected
      bcrypt.compare(password, user.password, (err, res) => {
        if (res) {
          // if response is sent for user
          resolve(user); // resolve the promise
        } else {
          // handle the error from the request
          reject() // reject the promise - sends a 400 status code
        }
      });
    });
  });
};
```

This model method is then called in `server.js`, inside the POST route for the user login, e.g.

```
// POST route for user login - /users/login
app.post('/users/login', (req, res) => {
  // pick data for email and password from request body
  var body = _.pick(req.body, ['email', 'password']);

  // find the user by email and password - use return user object
  User.findByCredentials(body.email, body.password).then((user) => {
    // get a new token for the user
```

```
    res.send(user);
  }).catch((error) => {
    // tell user they were unable to login...
    res.status(400).send(); // send status of 400 for error...
  });
});
```

We can then test this initial route in Postman, sending a current user's email and plain text password to this POST route, `/users/login`. We should get a response with the user's `_id` and `email` in the body.

Now, we need to generate the token for the authorised users, and then send it back to the user.

So, we can now update the `server.js` file to generate the token for the success callback of the `findByCredentials()` function, e.g.

```
...
// find the user by email and password - use return user object
User.findByCredentials(body.email, body.password).then((user) => {
  // get a new token for the user - use abstracted function created for
user save...
  return user.generateAuthToken().then((token) => {
    // set the header to x-auth with token, and send response body back as
`user`
    res.header('x-auth', token).send(user);
  });
}).catch((error) => {
  // tell user they were unable to login...
  res.status(400).send(); // send status of 400 for error...
});
```

As before, we can then test this update with Postman. We can submit a POST request to the `/users/login` route, and then use the `x-auth` value from the response header to test user authentication with the GET route, `/users/me`. If successful, this GET route will then return the user's `_id` and `email` address. If not authorized with this x-auth token, this GET route will return a 402 status code for unauthorized access.

app - testing - API routes - POST /users/login

We can a couple of tests for this new route.

The first test will check that when a valid email and password is received, an `x-auth` token is then sent in the response header.

The second test will check that a 400 status code is returned for an invalid login attempt. In effect, the user credentials do not match a record in the db.

We can now update our app's `server.test.js` file as follows for the new describe block with the required two test cases. We'll also need to use the test *seed* data for users, in particular the second seed user. This second user does not have a set token, which is what are now testing, e.g.

```
describe('POST /users/login', () => {
  // test case - user login with auth token response
  it('should login the user and response with auth token', (done) => {
    request(app)
      .post('/users/login')
      // send data from 2nd seed dummy user
      .send({
        email: users[1].email,
        password: users[1].password
      })
      .expect(200) // should return a 200 status code for OK
      .expect((res) => {
        expect(res.headers['x-auth']).toExist(); // x-auth should be
available in the response headers
      })
      .end((error, res) => { // add custom end - check for error and check
user in db
        if (error) {
          return done(error); // return done for any errors
        }

        // if no errors - find user by id - user id from 2nd seed dummay
user
        User.findById(users[1]._id).then((user) => {
          // check user tokens array includes at least the following
properties
          expect(user.tokens[0]).toInclude({
            access: 'auth',
            token: res.headers['x-auth']
          });
          done();
        }).catch((error) => done(error));
      });
  });

  // test case - invalid login - no user &c. found in db...
  it('should login the user and response with auth token', (done) => {
    request(app)
      .post('/users/login')
      // send an invalid password to test validation...
      .send({
        email: users[1].email,
        password: 'password1234567'
      })
      .expect(400) // should return a 400 status code for error
      .expect((res) => {
        expect(res.headers['x-auth']).toNotExist(); // x-auth should not
be available in the response headers
      })
      .end((error, res) => { // add custom end - check for error and check
user in db
        if (error) {
          return done(error); // return done for any errors
```

```
        }

        // if no errors - find user by id - user id from 2nd seed dummy
user
        User.findById(users[1]._id).then((user) => {
          // check user tokens array includes at least the following
properties
          expect(user.tokens.length).toBe(0);
          done();
        }).catch((error) => done(error));
      });
  });
});
```

These two test cases are very similar, the main difference simply being a check for valid email address, updated status codes for each test return, and what we expect in the response for the request.

We now have 18 tests that are run for this app with Mocha, supertest and expect.js.

app - build API - user logout route - DELETE /users/me/token

We can now add a route to the API to allow a user to logout of the app. In effect, we can revoke the token for the user request.

We can add a DELETE route to allow a user to logout. This route will be private, which means a user will only be able to access the route if they are already authenticated and logged in. Therefore, there's no need to explicitly read or load the auth token from the body of a response.

If a user is not authenticated, they will simply be unable to view this route.

So, to make a private route for the user logout, we can specify it as follows,

```
// DELETE route for user logout - /users/me/token
app.delete('/users/me/token', authenticate, (req, res) => { // use
authenticate to make the route private
  // delete token set in the authentication middleware - access user via
req.user as the user is already authenticated...
  req.user.removeToken(req.token).then(() => { // call instance method to
abstract deletion of token
    // respond with 200 status code - OK
    res.status(200).send();
  }, () => { // second callback to then()
    // respond with 400 status code for error &c.
    res.status(400).send();
  });
});
```

To make the route private we can pass a call to `authenticate`. We can then use a custom *instance* method on the req object, passing the token for the current logged in user. This token is available on the `req` object.

We can then chain a `then()` method, and handle the success callback for removing the token, and a second callback for a failure.

We can add the custom instance method, `removeToken()`, in the `user-model.js` file.

```
// custom instance method - pass token to delete
UserSchema.methods.removeToken = function (token) {
  // lowercase `this` for instance method
  var user = this;

  // call update() on object to update - pass updates object
  return user.update({
    // specify what to `pull` from db - pull from tokens array
    $pull: {
      tokens: {
        token: token
      }
    }
  });
};
```

This custom instance method will be added to UserSchema and methods. To remove the actual token for the authorised user, we can use the MongoDB operator, `$pull`. This operator allows us to remove items from a db that match certain criteria.

In this example, we're checking the `tokens` array for a token property that matches the current token. If it matches the whole `tokens` object will be removed, and not just the single property. So, any other data stored in the `tokens` array will also be removed.

We can then `return` this updated user, so we can chain any call to this instance method elsewhere in the app. This is exactly the pattern we use in the `server.js` file for the DELETE route.

We can then start the app server, and test this new API route with Postman.

app - testing - API routes - DELETE route for user logout

We can now add a `describe` block with a test case for deletion of an x-auth token for user logout.

We'll add a standard `describe` block, e.g.

```
// test DELETE route for /users/me/token
describe('DELETE /users/me/token', () => {
  // test case - remove auth token on logout
  it('should delete auth token on user logout', (done) => {
    // need seed data to test user - auth token from seed, then
logout...users[0] token
    ...
  });
});
```

We can use the existing seed data for the first user, which includes an auth token we can use for the user login. Then, we can simply check the auth token has been removed from the DB, and the user updated in the DB.

So, for this test case we're making a delete request to the API route, `users/me/token`. We'll need a token from the first seed user in the dummy data, which we can set for the x-auth value. i.e. the value of the simulated token for a logged in user.

Then, we can add some standard assertions for a status code of 200, and an async call to `end()`. We can use the end call to find a user in the db, and verify that the tokens array has been pulled of data. This will simply be a check to the array length, which should equal zero.

So, the test case is as follows, e.g.

```
// test DELETE route for /users/me/token
describe('DELETE /users/me/token', () => {
  // test case - remove auth token on logout
  it('should delete auth token on user logout', (done) => {
    // need seed data to test user - auth token from seed, then
logout...users[0] token
    request(app)
      .delete('/users/me/token')
      // set token to seed dummy data - first user, first tokens array and
then token
      .set('x-auth', users[0].tokens[0].token)
      .expect(200) // should return a 200 code for OK
      // call end function with custom assertion
      .end((error, res) => {
        // add error check - call done if error
        if (error) {
          return done(error);
        }

        // add custom assertion - query db for user by id
        User.findById(users[0]._id).then((user) => {
          // assertion - check user tokens array = 0 (token has been
deleted)
          expect(user.tokens.length).toBe(0);
          // call done for the test case
          done();
        }).catch((error) => done(error));

      });
  });
});
```

We can run this test case, and check that each test now completes successfully. There are now 19 tests in the `server.test.js` file.

app - build API - update TODOS with private routes

To allow us to make these routes private, we're effectively linking specific todo items to users. So, we need to update the todo model as well, so we can now link a user's ID to a specific todo item in the DB.

Then, as a user requests a todo item, or all of their todos, we can grant private access to the applicable documents in the DB.

We can update the TODOS model in `todo-model.js` as follows,

```
...
author: {
  type: mongoose.Schema.Types.ObjectId,
  required: true
}
```

So, a user cannot now create a todo item unless they have successfully logged into the app. We also need to set the type for this model property to a user's ID, which is an ObjectId in MongoDB. We can use Mongoose's Schema constructor to access the `Types` object with a property of `ObjectId`.

We'll also need to update any `todo` objects we define for dummy seed data for the app. For example, we currently have two todo objects defined in the `seed/seed.js` file, which we need to update for the modified todo model. e.g.

```
// update dummy todo items with test ID name:value pair property
const todos = [
  {
    _id: new ObjectID(),
    text: 'a todo item...',
    author: userIdOne
  },
  {
    _id: new ObjectID(),
    text: 'another todo doc item...',
    completed: true,
    completedAt: 230797,
    author: userIdTwo
  }
];
```

So, as we seed the db with test data, we now have two example todo objects that match the required todos schema model.

We can then make todo routes in `server.js` private, e.g. the POST route `/todos` will be private as follows,

```
// POST route for todo items
app.post('/todos', authenticate, (req, res) => { // route url for all todo
items - use for post and get...
  // create todo item from model
  var todo = new Todo({
```

```
    text: req.body.text, // specify text for each todo item
    author: req.user._id // add user id for authenticated user - add
privacy to route...
  });

  todo.save().then((doc) => {
    res.send(doc); // send back to the saved document details
  }, (error) => {
    // send back errors...
    res.status(400).send(error); // send back error and status code for
request...
  });
});
```

and, we can then make the GET `/todos` route private as well, e.g.

```
// GET route for todo items
app.get('/todos', authenticate, (req, res) => { // authenticate - check x-
auth token used to fetch todos
  // first check for todo items that match the authenticated user...
  Todo.find({
    author: req.user._id
  }).then((todos) => { // promised resolved with all of the todos from the
db
    res.send({ //response - send data back from get route - all of the
todos
      todos // add todos array to object - update and modify object as
needed instead of just sending array response...
    });
  }, (error) => { // error callback if error with promise
    res.status(400).send(error); // send back error and status code for
request...
  });
});
```

So, we need to add the `authenticate` middleware to the request, and ensure that we only fetch todo items that match the id of the authenticated user. So, as expected, user id 1 can not access documents for user id 2.

We can then test these updated routes in Postman, e.g.

- signup a new user - POST /users
    - gives x-auth token to use in other API routes
    - useful to create at least two users - easier to test private docs linked to user id
- create a new todo item - POST /todos
    - use token to add author id for private doc
    - add x-auth header with user token
    - test create without x-auth token - 401 unauthorized code returned...
- fetch todos for user id specified - GET /todos
    - send x-auth header token for required user

    - retrieve todo items for logged in user
        - should only return todo items for header x-auth token...

app - testing - update test cases for private routes

To fix our test cases, we need to slightly modify the `server.test.js` file to match these newly updated private routes. e.g. for the POST `/todos` route we can update the each test case request as follows,

```
...
    // use Supertest to test POST - pass app object
    request(app)
      .post('/todos') // call post method from app object - i.e. call api
route
      .set('x-auth', users[0].tokens[0].token)
...
```

All we need to do is set an `x-auth` token for the request, which ensures that we setting a token for the new todo item.

We can then update the GET route for `/todos` to set the required x-auth token. e.g.

```
.set('x-auth', users[0].tokens[0].token)
```

The difference with this update is that we need to modify the number of expected todo items, which will then correspond to the authenticated user. e.g.

```
...
.expect((res) => { // custom assertion
  expect(res.body.todos.length).toBe(1); // todo items needs to match
total in db for authenticated user
})
...
```

app - build API - add privacy to all API routes

We can now check that we have secured the API for all available routes. We've already secured two of the TODOS routes, but we still need to add authentication checks for the remaining TODOS routes, including

- GET by ID
- DELETE by ID
- PATCH by ID

So, to update these routes we can follow a similar pattern to the first two routes. e.g.

- add `authenticate` middleware
- update the request query to include the new `author` property

Then, we can finally update the test cases to check everything is working as expected.

app - build API - add privacy to GET '/todos/:id' route

So, we can start by updating the GET `/todos/:id` route. We need to add the `authenticate` middleware, e.g.

```
// GET route with parameter - then query DB for passed ID
app.get('/todos/:id', authenticate, (req, res) => { // authenticate - check user is logged in, and give access to user via request object...
...
});
```

Then, we need to update the query for this route. We need to change the query method from `findById()` to `findOne()` otherwise a logged in user could simply find any todo item by ID, including other users' todos. So, for the updated `findOne()` method, we can now pass the ID and the author property to ensure only the original author can modify the specified todo item, e.g.

```
...
Todo.findOne({
  _id: id,
  author: req.user._id
})
...
```

app - testing - update test cases

We can then modify the required `describe` block for that route. We need to update this test as follows,

- set x-auth token for query

```
...
.set('x-auth', users[0].tokens[0].token) // set x-auth token for specified user
...
```

We can also add an extra test case to check that a logged in user can't access another user's todo item by id.

```
// test case - check logged in user can't access another user's todo items
it('should not return another user\'s doc', (done) => {
  request(app)
    // try getting second todo item from seed data - created by second user...
    .get(`/todos/${todos[1]._id.toHexString()}`)
    .set('x-auth', users[0].tokens[0].token) // user first user's token
```

```
      .expect(404)
      .end(done);
  });
```

So, we can use the first user's token from the dummy seed data to try and `get` the second todo item from the seed data. The author of this todo item has been set as user 2, so user 1 can't access this todo item.

app - build API - add privacy to DELETE '/todos/:id' route

Again, as with the above updates for the GET routes, we need to add authentication middleware, update the query, and update the test cases.

So, in `server.js` we can update our DELETE route as follows,

```
// DELETE route for single doc with ID
app.delete('/todos/:id', authenticate, (req, res) => { // add
authenication to route
  // get params ID from req
  var params_id = req.params.id;
  console.log(params_id);

  // validate passed ID - check not valid
  if (!ObjectID.isValid(params_id)) {
    // return 404 status code for invalid ID
    return res.status(404).send();
  }

  // find doc by id and author - remove from DB
  Todo.findOneAndRemove({
    _id: params_id,
    author: req.user._id
  }).then((todo) => {
    // check if return data available
    if (!todo) {
      return res.status(404).send();
    }
    // otherwise return the data for the deleted params ID
    res.send({todo});
  }).catch((error) => { // catch return errors for the query
    res.status(400).send();
  });
});
```

The two changes for this DELETE route include authentication for the route parameters, and a change in query method. Instead of using `Todo.findByIdAndRemove(params_id)`, we need to ensure that a document is found matching the passed doc id and author id. In effect, as with the GET routes, we need to ensure that a user can't delete a todo item unless they are also the author.

So, we can simply call the `findOneAndRemove()` method with an object for the doc id and author id.

app - testing - update test cases

For the test cases for DELETE `todos/:id`, we simply need to ensure that we set an x-auth token for the author of the requested doc, e.g.

```
.set('x-auth', users[1].tokens[0].token) // set x-auth token for specified
user
```

We can then test deleting a doc for the specified user's toke, and requested doc.

As with the GET route, we can also add a new test case to check that an authenticated user can't delete a todo item they do not own. e.g.

```
// test case - check user can't delete todo item they do no own...
it('should not delete a todo item they do no own', (done) => {
  // doc id should not be owned by user id in test case...
  var hexId = todos[0]._id.toHexString(); // doc's author should not match
user id -

  request(app)
    .delete(`/todos/${hexId}`) // remove the specified doc by id
    .set('x-auth', users[1].tokens[0].token) // set x-auth token for
specified user
    .expect(404) // assert a 200 status code for the successful doc
deletion
    .end((error, res) => { // finish request
      if (error) { // handle error
        return done(error); // if error exists simply return the request
as done...
      }

      // find doc id in db
      Todo.findById(hexId). then((todo) => {
        expect(todo).toExist(); // check that doc id does exist in db -
doc not deleted by non-author
        done(); // call done and finish async all
      }).catch((error) => done(error)); // catch any error for async call
- return done if error caught...
    });
});
```

In this extra test case, we're simply checking that an authenticated user cannot delete a todo item they do not own. So, we set the user id from the seed dummy data to a user that does not own the specified todo item. We update the expected status code to 404, and then test that the todo item can still be found in the DB. i.e. the todo item has not been removed...

For the remaining two test cases for the DELETE route, we simply set authentication for the route, e.g.

```
...
.set('x-auth', users[1].tokens[0].token) // set x-auth token for specified
user
...
```

app - build API - add privacy to PATCH '/todos/:id' route

As with GET and DELETE, we need to update the `describe` blocks and test cases to include authentication middleware, update the query, and update the test cases.

In the `server.js` file we can update the PATCH route by adding authentication, as above, and then modifying the find query as follows,

```
// update the requested doc in the db for id and author - using Mongoose
method, findOneAndUpdate()
findOneAndUpdate({
    _id: params_id,
    author: req.user._id
  }
  ...
)
...
```

As we saw in the previous routes, we need to ensure that the authenticated user is not able to modify a todo item by id that does not belong to them.

app - testing - update test cases

We can then modify the test cases for this route. So, for the first test case in PATCH `/todos/:id`, we need to set authentication, e.g.

```
.set('x-auth', users[0].tokens[0].token) // set x-auth token for specified
user
```

Then we add an extra test case to check that a user can't update a todo item they do not own. This is structured as follows,

```
//test case - check user can't update todo item they do not own
it('should not patch and update the todo item authored by another user',
(done) => {
  // get ID from dummy todos object - first object
  var hexId = todos[0]._id.toHexString();
  // text for testing PATCH update
  var text = 'some test new text...';
```

```
  // setup test with assertions
  request(app)
    .patch(`/todos/${hexId}`)
    .set('x-auth', users[1].tokens[0].token) // set x-auth token for
specified user
    .send({
      completed: true,
      text // ES6 shortcute for name:value pair
    })
    .expect(404) // user should not be able to update the requested todo
item
    .end(done);
});
```

Then, we can `set` and x-auth token for the final test case, and run all of the tests for this app,

```
npm test
```

app - build API - update app config

Current app config settings and variables are stored in `config.js`. However, it's not the best idea to push such config settings to a repository for a final production app. Indeed, we need to be aware of such issues for any stage of development with sensitive config.

For the current app's config, settings and variables for the URL of a local DB is not an issue. However, if we start adding API keys and secrets, DB login details &c., then we need to ensure these are not push to a remote repo or store.

So, the issue is not with publication of data and and config settings for a production app on Heroku, for example, but local and dev versions of the app. If we push such data and settings to a repo on GitHub &c. it becomes publicly accessible.

One solution is to create a local `config.json` file, which we can use to store settings, credentials &c. for local and development versions. This file, however, will not then be pushed to a remote repository.

e.g.

```
{
  "development": {
    "PORT": 3000,
    "MONGODB_URI": "mongodb://localhost:27017/NodeTodoApp"
  },
  "test": {
    "PORT": 3000,
    "MONGODB_URI": "mongodb://localhost:27017/NodeTodoAppTester"
  }
}
```

We can then load and call these objects and properties within our app's `config.js` &c. file. e.g.

```
// check environment and load appropriate settings
if (env === 'development' || env === 'test') {
    // require json - returns an object for the json
    var config = require('./config.json');
    // get config properties for specified env - e.g. development or test
    var envConf = config[env]; // store defined env


}
```

If we use Node.js `require` to load the JSON file, we get an object for the JSON in the defined variable. So, we can then load any objects and properties from the variable, which are stored in the JSON file. This is a variable useful option for `require`.

Then, we need to access the properties for the environment settings.

To use a variable to access a property, i.e. `config` variable to access a property in the JSON file, we have to use bracket notation. So, for

```
var envConf = config[env];
```

we're using the requested environment, either `development` or `test`, to access the matching property in the JSON specified by the variable, `config`.

So, with the selected env property in the JSON, we simply need to loop over the object to get the `PORT` and `MONGODB_URI` values.

Now, we could simply specify the required name:value pairs from the object in the JSON, such as `envConf.PORT` to the PORT value, and so on. However, this restricts what we can extract from the JSON file. We'd need to explicitly specify each name:value pair required. Also, it would need updating each time we added a new name:value pair to the JSON.

So, instead we can simply get the keys for a specified object and return them as an array. Then, we can loop over each key, regardless of name &c., and return the properties and values. e.g.

```
Object.keys(envConf).forEach((key) => { // iterate over keys array - pass
key to callback function
  // for each key - get value from envConf
  process.env[key] = envConf[key]; // use bracket notation to set property
of `process.env`
});
```

We now have an abstracted array and loop to handle each possible name:value pair in the config settings, which will set PORT, MONGODB_URI, and so on.

To hide the `config.json` from a git repository, we can add the name of the JSON file to our repos `.gitignore` file.

app - build API - update jwt config to hide settings

In addition to existing local config variables and settings, we also need to ensure that the `secret` for JWT's `sign()` method is neither exposed publicly, nor pushed to a public repository &c.

So, we currently use the `secret` in both the test cases and the user model. This means we can abstract this value and store it safely in an environment variable in `config.json`.

We'll start by adding a secret in `config.json`, e.g.

```
"SECRET_JWT":
"hgbyhujionkmjhnbvgvwsqawsxdcrf43234098767788976555tgyhujbnhgbvfrt76765434"
```

Add a long, random value for the secret for JWT's sign method. We'll usually add separate, different secrets and values for each environment, such as `test` and `development`.

We can then access each secret value as needed within the app's code. e.g.

```
process.env.SECRET_JWT
```

We can access each secret's value by simply using `process.env`. We can the update `seed.js` for testing, and the user model, `user-model.js`.

Then, to check everything still works as expected, we can simply run our test cases for the app, e.g.

```
npm test
```

which will return 22 passing test cases.

app - build API - update Heroku for `config.json`

We need to consider how we'll now configure a remote hosting option, such as Heroku, for use with the JWT secret variable.

So, for Heroku we can use the toolbelt CLI tool to help config the settings variables for production.

e.g.

```
heroku config
```

This will initially return the stored `Config Vars` for the Heroku hosted app. Currently, this includes the `MONGODB_URI` variable we need for remote connection to mLab.

So, we might now set a new `env` variable for the Heroku app, e.g.

```
heroku config:set NAME=test-app
```

We can then check this individual config variable has been set successfully, e.g.

```
heroku config:get NAME
```

Then, if we need to remove a config variable we can use the `unset` command, e.g.

```
heroku config:unset NAME=test-app
```

app - build API - connect RoboMongo (Robo 3T) to mLab

To monitor and view our mLab hosted DB in Robo3T, we need to create a new connection with the following settings,

- name = local name for connection, easy to remember...
- address = direct uri for db on mLab
- port = port number for db on mLab

Then, we need to set authentication for the connection,

- username = db specific username set on mLab
- password = matching password for username

Each of these settings variables can be found in the uri we set for the `MONGODB_URI` on Heroku, for example.

```
mongodb://username:password@ds054417.mlab.com:54417/db_name
```

So, we can find the username and password (separated by a colon). Then, the uri for the db on mLab, and the applicable port number, e.g. `54417`. Then, the name of the db on mLab.

We can then test everything in Postman.

testing - Postman - advanced options

Postman provides some useful advanced options for testing and working with API routes.

Instead of manually adding the value for an x-auth token, for example, we can use environment variables. The obvious benefit of this approach is that we set an env variable once, and then reuse for multiple routes.

So, we can setup an example env variable using the `Tests` tab in Postman. For a given route, there should be a tab `Tests`. In this section of Postman we can add custom code that is run after the request has been returned. e.g.

- fetch the x-auth token
- set the token to an env variable - this can then be used in other routes in Postman

We can use Postman utility methods to get this token, and then set the env variable. e.g.

```
// get the value from the response header
var xauthToken = postman.getResponseHeader('x-auth');
// set the value in an env variable in Postman
postman.setEnvironmentVariable('x-auth', token);
```

So, this `x-auth` token will be returned for a user when they create a new account, or login to the api. We can add this code, for example, to the `Tests` tab for the POST `/users` route.

Then, for a given route that requires authentication with the x-auth token, we can add our custom header for the route. The header will be set to `x-auth` with a value of `{{x-auth}}`.

We can also add an env variable for the last created todo item, which will then allow us to easily test API routes for GET, DELETE, PATCH &c. that require an id for a todo item. e.g.

```
// get JSON value and convert into js object - JSON.parse
var body = JSON.parse(responseBody); // postman access to json using
responseBody
// set env variable for todo id
postman.setEnvironmentVariable('todoId', body._id);
```

To use this `todoId` env variable, we can simply modify those API routes in Postman that require this ID appending tot he URL, e.g.

```
{{url}}/todos/{{todoId}}
```