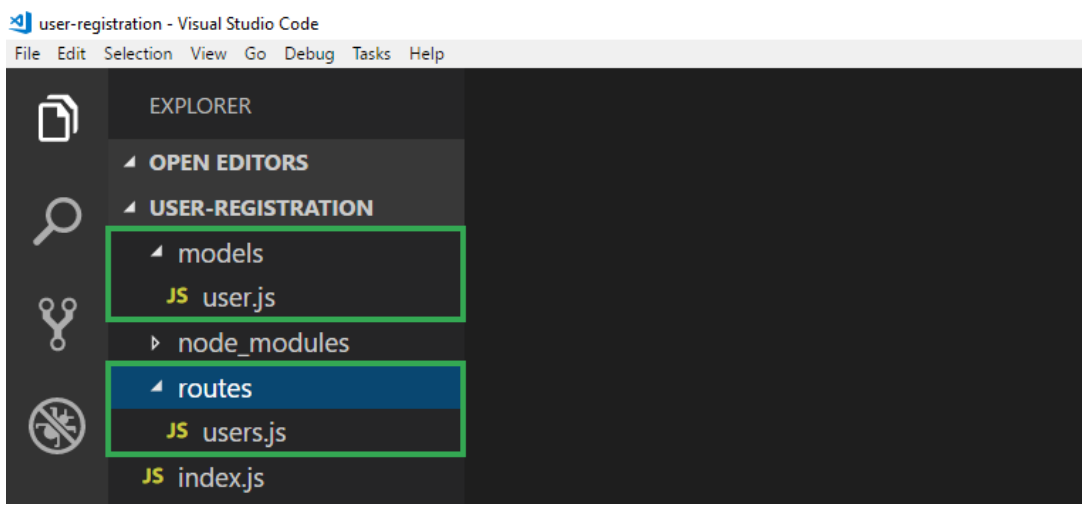
← Mongoose Relationships Tutorial | Information Expert Principle Applied To Mongoose Models →

Node.js MongoDB User Registration

So we are going to start building the most basic of User Registration systems in Node.js using MongoDB as the data store, Express as the routing system, Joi as the validator, and of course Mongoose to make interacting with Mongo from Node easy. Below we have our sample project layout. User-Registration is the top level directory which holds the index.js file, and then we have a models directory and a routes directory. We're going to see how to build the JavaScript files now to make this work.



Step 1. Create a User Model

First up we need to create a **User Model**. You can create a `user.js` file and place it in the `models`directory. At the top of the file, we require Joi and Mongoose as we will need them for validation and for creating the User **Mongodb Schema**. Then, we create the User Schema and define the requirements for name, email, and password. The User Schema is stored in `User`. Next up we create a validation function named `validateUser`. Lastly we export these modules so we can require them elsewhere.

27 Awesome VueJS Libraries
D3 DOM Selection With D3 JavaScript
How to add a WordPress Blog to your Laravel Application
Process HTML Forms With PHP
How To Use The Linux Terminal
Angular Data Binding
JavaScript Revealing Module Pattern
Vue.js Tutorial
JavaScript Callbacks vs Promises vs Async Await
The Top 15 Most Popular JavaScript String Functions
Mongoose Relationships Tutorial
Install Laravel on Windows

/models/user.js

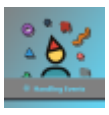
```
const Joi = require('joi');
const mongoose = require('mongoose');

const User = mongoose.model('User', new mongoose.
    name: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 50
    },
    email: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 255,
        unique: true
    },
    password: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 1024
    }
}));

function validateUser(user) {
    const schema = {
        name: Joi.string().min(5).max(50).required(),
        email: Joi.string().min(5).max(255).required().e
        password: Joi.string().min(5).max(255).requirec
    };
    return Joi.validate(user, schema);
}

exports.User = User;
exports.validate = validateUser;
```

Step 2. Set Up Users Routes

Now that we have a User Model set up which both defines the schema we need to follow and the validation rules, we can create a `users.js` routes file in our `routes`directory. In this file, the first thing we do right at the top is to require, or import, the User schema and validate schema that we just exported in user.js. Next we make sure Express is initialized. The router.post() function does all the heavy lifting here. First the http post request gets validated, then we check to see if the user

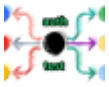

How To Create New Components In Angular Using The CLI


How To Use wp_list_pages() To Create Parent and Child Page Menus


How To Handle Events In React


Laravel Auth Tutorial


The Declarative Nature of SQL


Using Test Authentication To Allow Logged In Users To Post Replies


Creating Monitoring and Killing Processes in Linux


Mongoose Crud Tutorial


How To Protect Specific Routes With Middleware


HTML Encoding With htmlspecialchars and htmlentities


How To Add Routes and Models To Node Rest API


How To Add Post Meta In WordPress


What Does this Refer to in JavaScript?


9 New Array Functions in ES6


How To Use The Laravel Query

already exists in the database, and finally we create a new user if they do not exist in the database and also if they pass all validation requirements. Lastly we export the router, so we can use it in the index.js file.

/routes/users.js

```
1  const { User, validate } = require('../models/user');
2  const express = require('express');
3  const router = express.Router();
4
5  router.post('/', async (req, res) => {
6      // First Validate The Request
7      const { error } = validate(req.body);
8      if (error) {
9          return res.status(400).send(error.details[0].mes
10     }
11
12     // Check if this user already exisits
13     let user = await User.findOne({ email: req.body.em
14     if (user) {
15         return res.status(400).send('That user already e
16     } else {
17         // Insert the new user if they do not exist yet
18         user = new User({
19             name: req.body.name,
20             email: req.body.email,
21             password: req.body.password
22         });
23         await user.save();
24         res.send(user);
25     }
26 });
27
28 module.exports = router;
```

Step 3. Register Users Route in index.js

index.js

Most of the boilerplate here should look familiar to you if you followed the **rest api tutorial** already. The key points for this tutorial here are highlighted. Note we require the users.js file at line 4. This allows us to set up the route for `/api/users` at line 13.

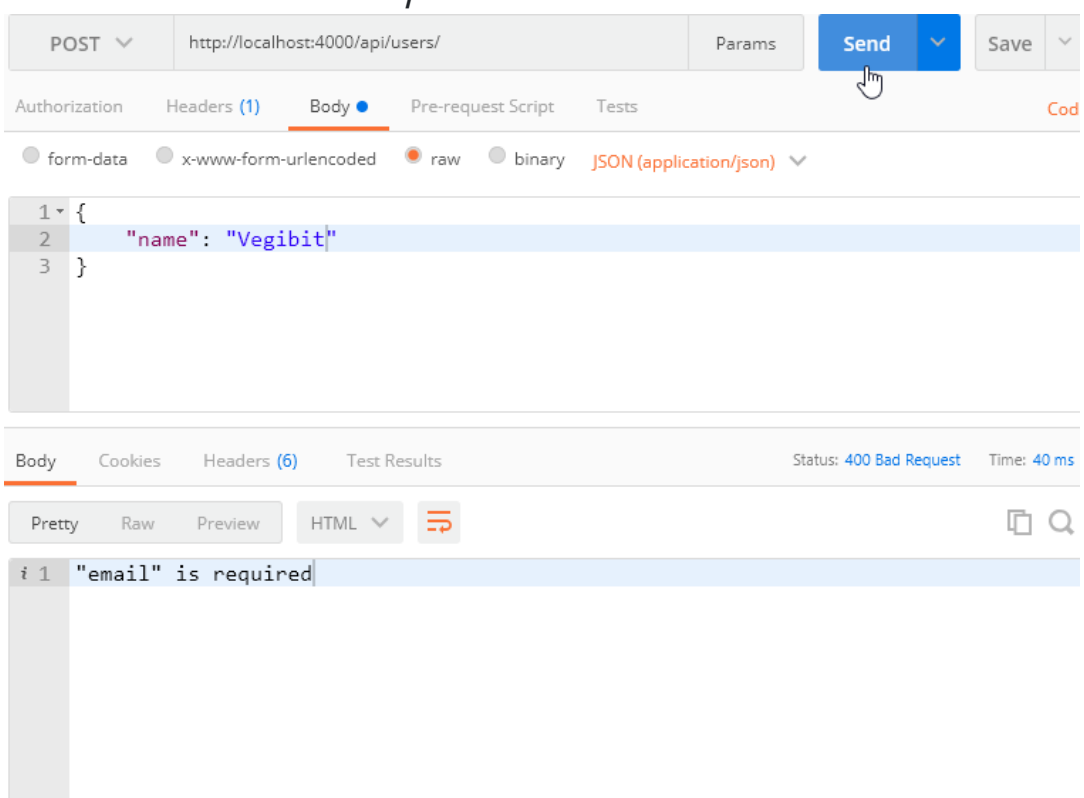
```
const Joi = require('joi');
Joi.objectId = require('joi-objectid')(Joi);
const mongoose = require('mongoose');
const users = require('./routes/users');
const express = require('express');
const app = express();

mongoose.connect('mongodb://localhost/mongo-game
    .then(() => console.log('Now connected to MongoD
    .catch(err => console.error('Something went wron

app.use(express.json());
app.use('/api/users', users);

const port = process.env.PORT || 4000;
app.listen(port, () => console.log(`Listening on port $
```

Step 4. Test Post requests with Postman

Now we can make use of Postman to send a Post request to our server to see if we can persist a new user to MongoDB. First, let's launch the application.

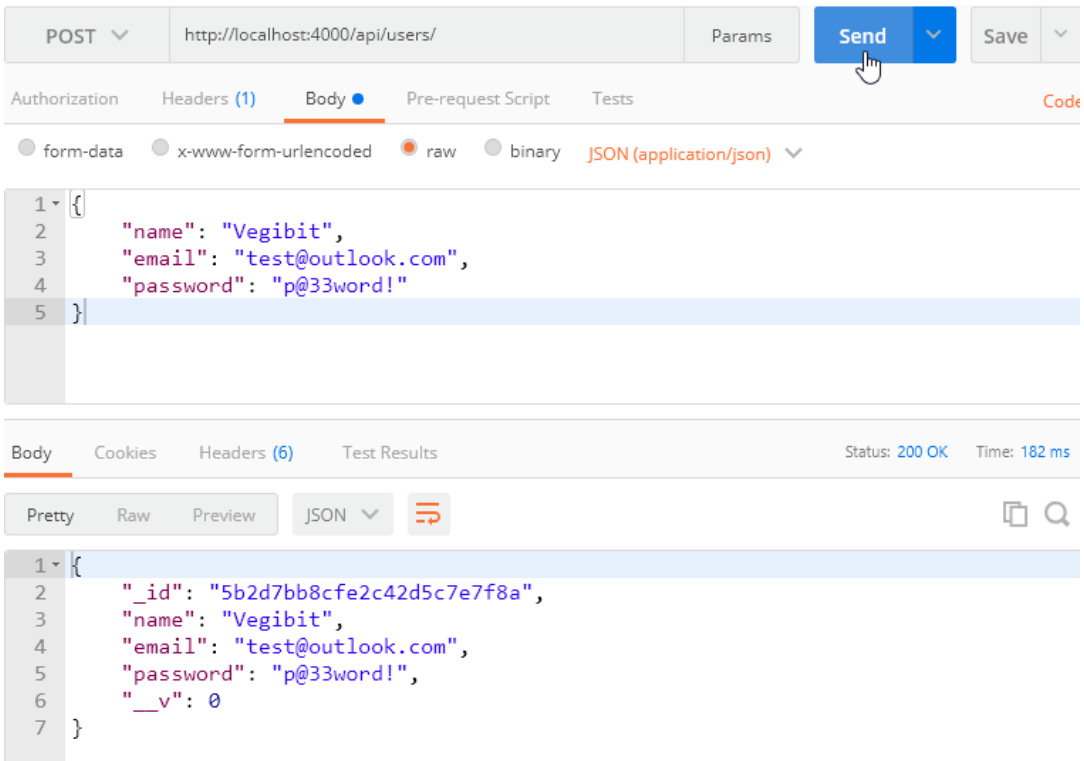
```
user-registration $node index.js
Listening on port 4000...
Now connected to MongoDB!
```

Awesome, everything is running with no crashes! Now we can test some Post requests. Here we send a post request as application/json with a json object in the body of the request. We only set the user name, but we left off both email and password. We can see that our validation is working since the response we get back from the server is *"email" is required.*

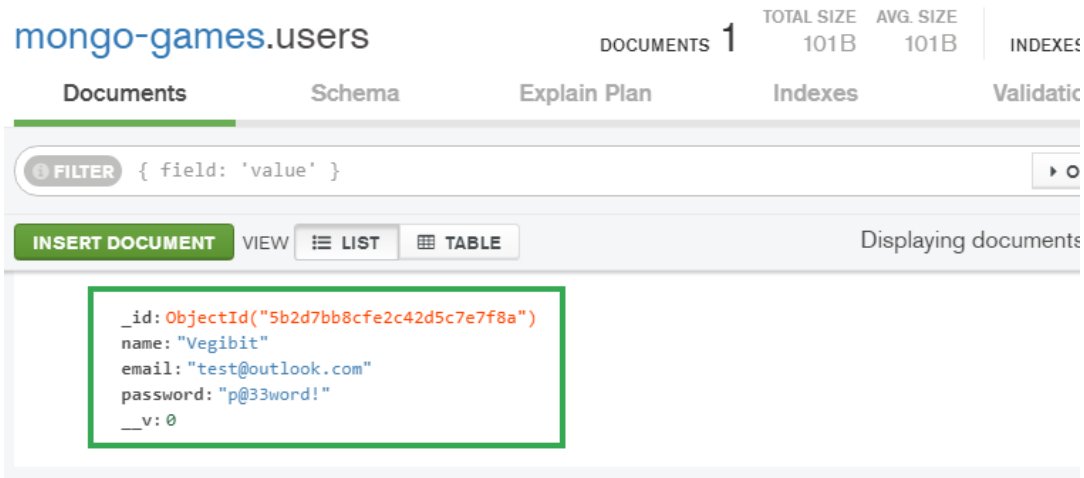


Underscore JS Map Function
What Is Twitter Bootstrap?
How To Make HTTP Requests In Angular Using Observables
Basic Laravel Routing and Views
How To Create A Child Component In VueJS
How To Use WordPress Excerpts
How To Pass Data To Views In Laravel
Angular Table Filter Component
How To Remember Form Data
Introduction To Laravel 5
Custom Helper Functions in Laravel
Laravel AJAX CRUD Tutorial
Escape Strings For MySQL To Avoid SQL Injection
PHP String Helper Functions
Laravel Migration Generator
Laravel Repository

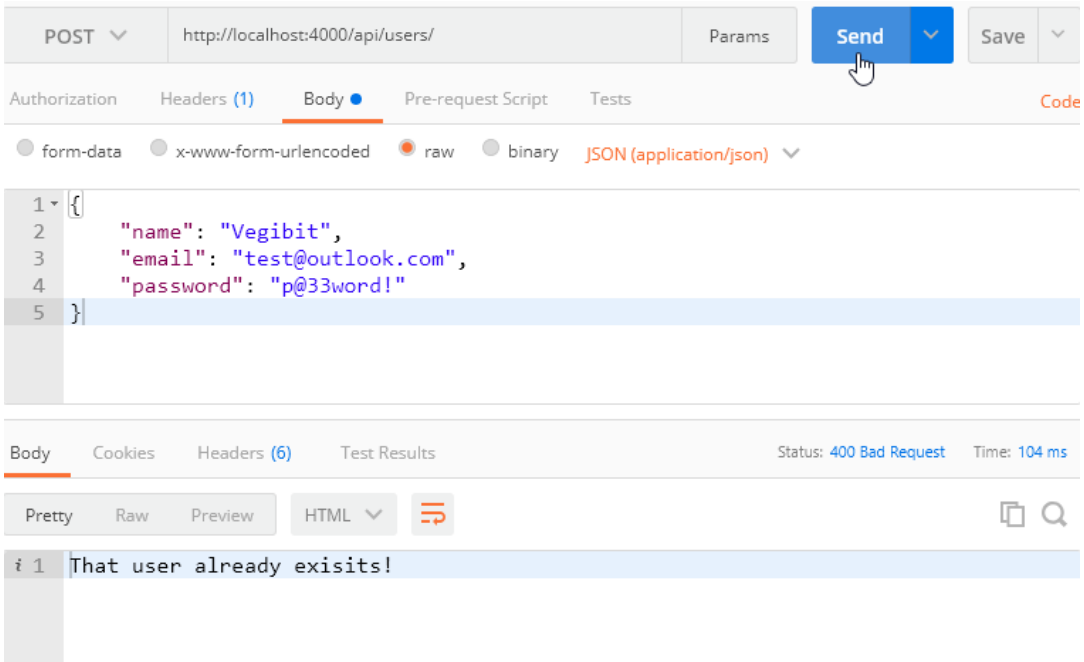
Let's now fill out a proper user object to see if we can get the User to be stored in the MongoDB database. This time around, we don't get an error back, but we see the user object. This means it was successful!



Now we can look inside **MongoDB using Compass** and see if this new user is in place. Nice!



Recall that we did put some logic in the code to make sure that if there was already a user in the database, then we should not persist that user again. To test this we send that same request again to the server, and we get back the response we expect. We are not allowed to insert the same user twice. Very nice!



Hash Passwords With Bcrypt

Pattern

Open Closed Principle

Working With HTML Images

Install Larabook On Laravel Homestead

Php Tutorials For Beginners

You Might Still Need jQuery

Hunt Down The Nouns

How Do Functions Work in Python?

Autoloading For Code Organization

CSS Selectors Tutorial

What Are Migrations In Laravel?

Build A Regular Expression Tester With Laravel

Getting Started With JavaScript Programming

Laravel hasMany and belongsTo Tutorial

The rudimentary portion of the user registration is now working however the password is in clear text. This is a big no no, so let's see how to encrypt the password before saving into the database using the bcrypt package. First up, we install it.

```
user-registration $npm i bcrypt

> bcrypt@2.0.1 install C:\node\user-
registration\node_modules\bcrypt
> node-pre-gyp install --fallback-to-build

[bcrypt] Success: "C:\node\user-
registration\node_modules\bcrypt\lib\binding\bcrypt_lib.n
ode" is installed via remote

+ bcrypt@2.0.1
added 69 packages from 47 contributors and audited 247
packages in 8.548s
found 1 low severity vulnerability
  run npm audit fix to fix them, or npm audit for details
```

Now that we have bcrypt installed, we can use it in the users.js routes file like so. At the top of the file, we now require the bcrypt package which makes it available to use further down in the file. At lines 24 and 25 we then generate a salt, and use it to hash the password before saving.

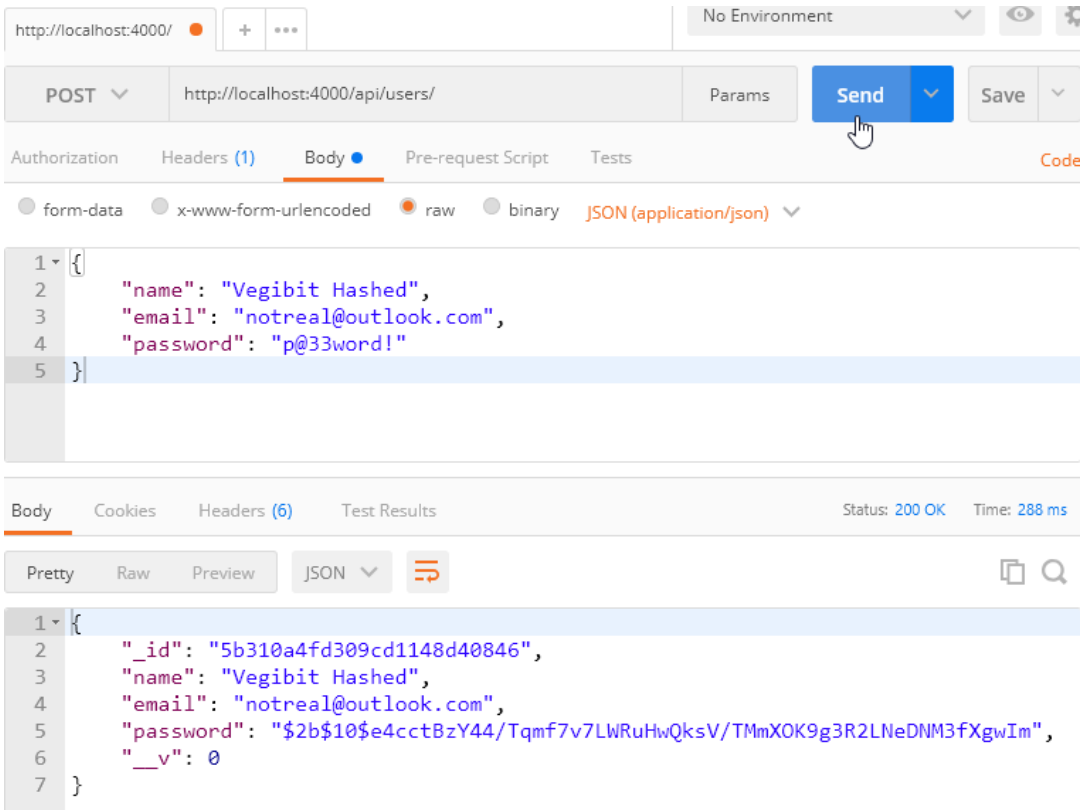
/routes/users.js

```
const bcrypt = require('bcrypt');
const { User, validate } = require('../models/user');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {
    // First Validate The Request
    const { error } = validate(req.body);
    if (error) {
        return res.status(400).send(error.details[0].mes
    }

    // Check if this user already exisits
    let user = await User.findOne({ email: req.body.em
    if (user) {
        return res.status(400).send('That user already e
    } else {
        // Insert the new user if they do not exist yet
        user = new User({
            name: req.body.name,
            email: req.body.email,
            password: req.body.password
        });
        const salt = await bcrypt.genSalt(10);
        user.password = await bcrypt.hash(user.passwor
        await user.save();
        res.send(user);
    }
});

module.exports = router;
```

Now, let's run the application and then test with Postman.

```
user-registration $node index.js
Listening on port 4000...
Now connected to MongoDB!
```

With that, we can open up Postman and send a POST request to http://localhost:4000/api/users/ with a new user specified as a JSON object in the body of the request.



Excellent! We get back a response object which means a new user was created, and notice the password field: It is fully hashed. This way, the password is safe and secure in the Mongo database. In fact, let's inspect it using Compass as well. Note the first user we had created has a password stored in plain text. The new user has a much more secure password which is properly hashed using bcrypt.

Using Lodash To Simplify Our Code

Let's go ahead an import the lodash package into our project so we can make use of it. Lodash is a powerful JavaScript utility library similar to the popular **Underscore Library**. Here we go ahead and install Lodash.

```
user-registration $npm i lodash

+ lodash@4.17.10
updated 1 package and audited 247 packages in
13.631sfound 1 low severity vulnerability
  run npm audit fix to fix them, or npm audit for details
```

Great! Now we can use lodash in our project. Specifically in this instance we are going to use the **pick function** which makes working with objects more terse. Now, once we import lodash into our file, we can make use of these handy one liners highlighted here.

```
const bcrypt = require('bcrypt');
const _ = require('lodash');
const { User, validate } = require('../models/user');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {
    // First Validate The Request
    const { error } = validate(req.body);
    if (error) {
        return res.status(400).send(error.details[0].mes
    }

    // Check if this user already exisits
    let user = await User.findOne({ email: req.body.em
    if (user) {
        return res.status(400).send('That user already e
    } else {
        // Insert the new user if they do not exist yet
        user = new User(_.pick(req.body, ['name', 'ema
        const salt = await bcrypt.genSalt(10);
        user.password = await bcrypt.hash(user.passwor
        await user.save();
        res.send(_.pick(user, ['_id', 'name', 'email']));
    }
});

module.exports = router;
```

How To Authenticate Users

Now that the user registration is in place, we can set up the process of authenticating users. First, go ahead and create an auth.js file in the routes directory. Once complete, we can start with this boilerplate.

/routes/auth.js

```
const Joi = require('joi');
const bcrypt = require('bcrypt');
const _ = require('lodash');
const { User } = require('../models/user');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {

});

module.exports = router;
```

Now we have to go back to the index.js file and set up the route for 'api/auth' like so.

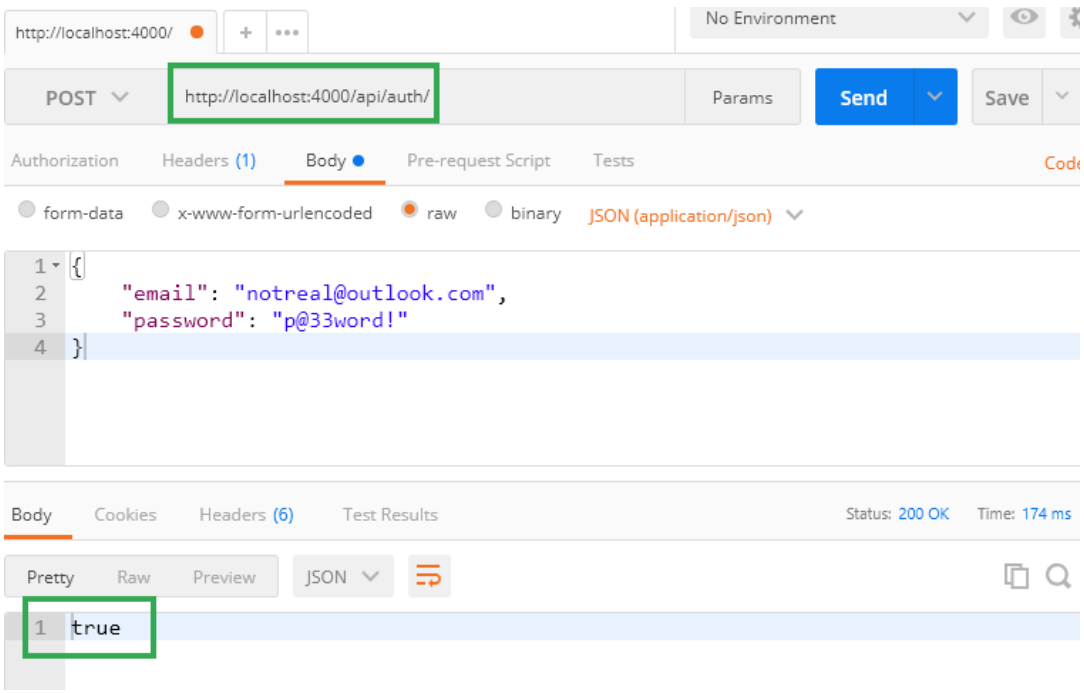
```
const Joi = require('joi');
Joi.objectId = require('joi-objectid')(Joi);
const mongoose = require('mongoose');
const users = require('./routes/users');
const auth = require('./routes/auth');
const express = require('express');
const app = express();

mongoose.connect('mongodb://localhost/mongo-game
    .then(() => console.log('Now connected to MongoD
    .catch(err => console.error('Something went wron

app.use(express.json());
app.use('/api/users', users);
app.use('/api/auth', auth);

const port = process.env.PORT || 4000;
app.listen(port, () => console.log(`Listening on port $
```

Ok back to the auth.js file. In here we need to set up the logic that will authenticate a user when the credentials are provided during a log in attempt. That means we need to validate the HTTP request being sent, find the user in the database, then use bcrypt to compare the stored password against the password provided in the request. This code will accomplish those goals.

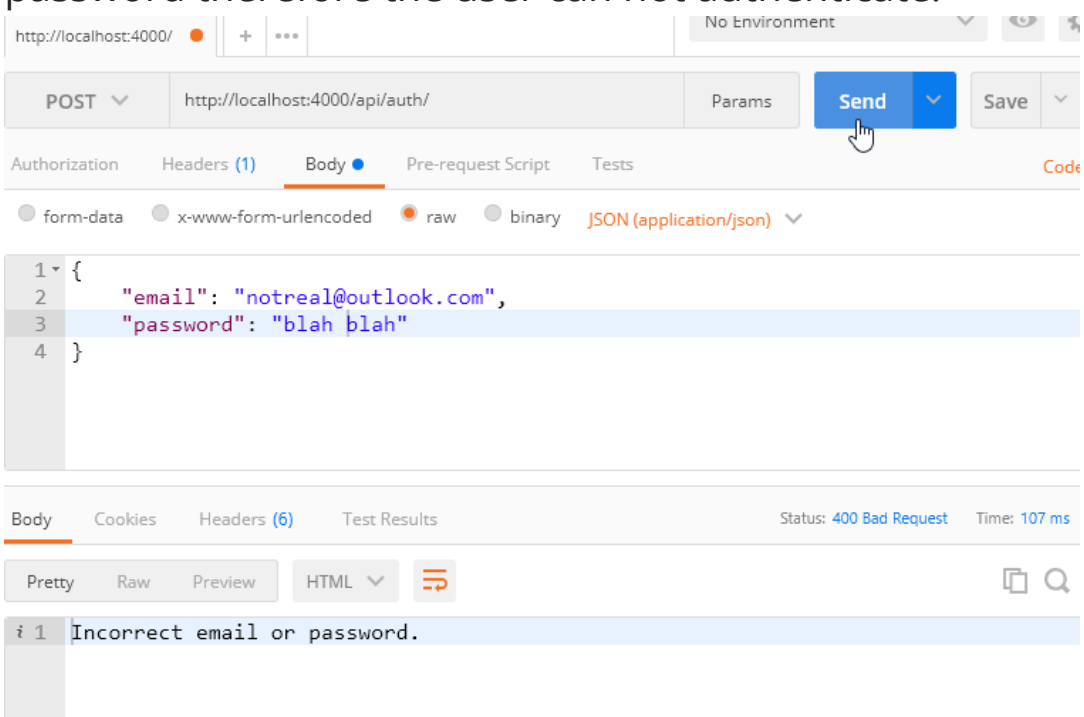
/routes/auth.js

```
1  const Joi = require('joi');
2  const bcrypt = require('bcrypt');
3  const _ = require('lodash');
4  const { User } = require('../models/user');
5  const express = require('express');
6  const router = express.Router();
7
8  router.post('/', async (req, res) => {
9      // First Validate The HTTP Request
10     const { error } = validate(req.body);
11     if (error) {
12         return res.status(400).send(error.details[0].mes
13     }
14
15     //  Now find the user by their email address
16     let user = await User.findOne({ email: req.body.em
17     if (!user) {
18         return res.status(400).send('Incorrect email or p
19     }
20
21     // Then validate the Credentials in MongoDB match
22     // those provided in the request
23     const validPassword = await bcrypt.compare(req.bo
24     if (!validPassword) {
25         return res.status(400).send('Incorrect email or p
26     }
27
28     res.send(true);
29 });
30
31 function validate(req) {
32     const schema = {
33         email: Joi.string().min(5).max(255).required().e
34         password: Joi.string().min(5).max(255).requirec
35     };
36
37     return Joi.validate(req, schema);
38 }
39
40 module.exports = router;
```

Excellent! Now let's test the auth endpoint using Postman. We can provide a valid email and password and see what happens.



Now let's send a request with the wrong password and see the result. Ah ha, looks good! It is catching the bad password therefore the user can not authenticate.



Implementing JSON Web Tokens

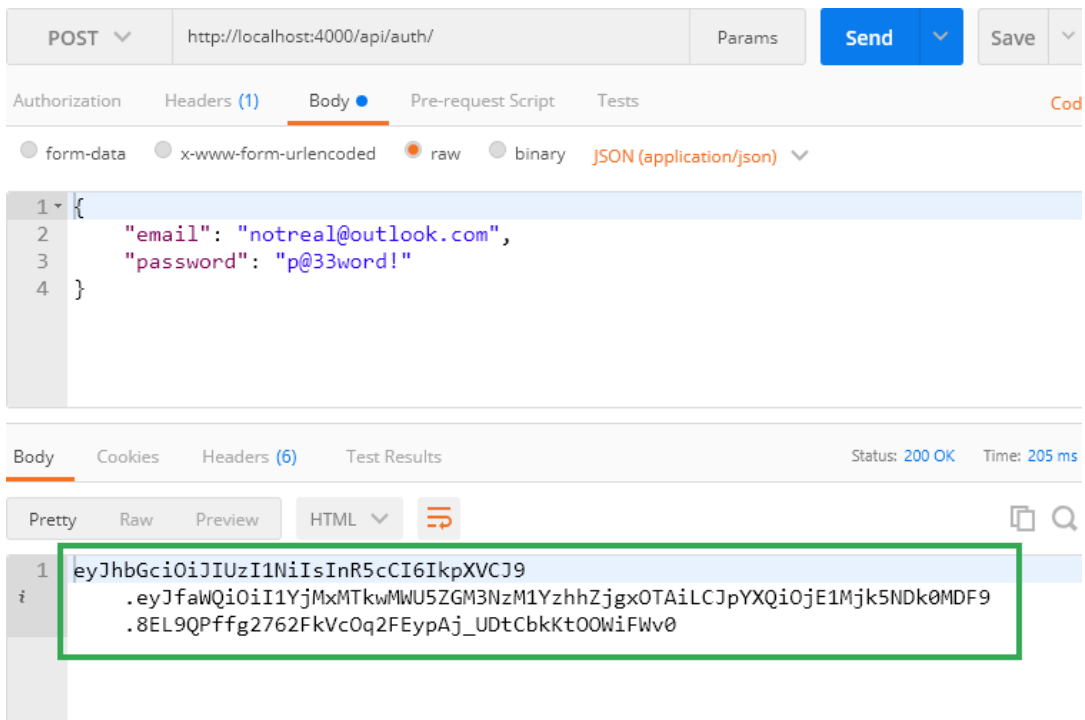
In the section above, we simply returned a `true` value when a successful login attempt was made. Now we are going to modify this response to send a **JSON web token**, which can uniquely identify any given user in the system. So in general the way it works is, the API generates a JSON Web Token upon successful login and then in the future that user must supply the JSON Web Token to identify themselves as a valid user when making various http requests to the api. On the client side, this token could be stored in local storage. That is beyond the scope of this tutorial as we will focus on the server-side here. Ok so to start generating JSON Web Tokens, we need to install an npm package to handle that for us.

```
user-registration $npm i jsonwebtoken

+ jsonwebtoken@8.3.0
added 13 packages from 9 contributors and audited 263
packages in 4.659sfound 1 low severity vulnerability
  run npm audit fix to fix them, or npm audit for details
```

Here we modify the auth.js file to make use of the jsonwebtoken package. We also use it to generate a new JSON Web Token, and send that back as a response to a proper http request.

```
1  const jwt = require('jsonwebtoken');
2  const Joi = require('joi');
3  const bcrypt = require('bcrypt');
4  const _ = require('lodash');
5  const { User } = require('../models/user');
6  const express = require('express');
7  const router = express.Router();
8
9  router.post('/', async (req, res) => {
10     // First Validate The HTTP Request
11     const { error } = validate(req.body);
12     if (error) {
13         return res.status(400).send(error.details[0].mes
14     }
15
16     //  Now find the user by their email address
17     let user = await User.findOne({ email: req.body.em
18     if (!user) {
19         return res.status(400).send('Incorrect email or p
20     }
21
22     // Then validate the Credentials in MongoDB match
23     // those provided in the request
24     const validPassword = await bcrypt.compare(req.bo
25     if (!validPassword) {
26         return res.status(400).send('Incorrect email or p
27     }
28     const token = jwt.sign({ _id: user._id }, 'PrivateKey
29     res.send(token);
30 });
31
32 function validate(req) {
33     const schema = {
34         email: Joi.string().min(5).max(255).required().e
35         password: Joi.string().min(5).max(255).required
36     };
37
38     return Joi.validate(req, schema);
39 }
40
41 module.exports = router;
```

Fantastic! Let's test out sending a valid user name and email as a POST request to our /api/auth endpoint. We see that a valid JSON Web Token is returned back to us.



We shouldn't really make the PrivateKey a part of the source code, it should be in an environment variable of some sort. Let's do this now. First we can install the config package.

```
user-registration $npm i config

+ config@1.30.0
added 3 packages from 5 contributors and audited 266
packages in 5.314sfound 1 low severity vulnerability
  run npm audit fix to fix them, or npm audit for details
```

Once installed, we can require it in the auth.js file.

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

Now let's make a config folder in our project and place a **default.json** file and a **custom-environment-variables.json** file in there.

default.json

```
{
  "PrivateKey": ""
}
```

custom-environment-variables.json

```
{
  "PrivateKey": "PrivateKey"
}
```

Now instead of referencing the private key directly, we reference it using the config.get() function like we see here.

```
const token = jwt.sign({ _id: user._id }, config.get('Priv
```

We should also include this in index.js like so.

```
const config = require('config');
const Joi = require('joi');
Joi.objectId = require('joi-objectid')(Joi);
const mongoose = require('mongoose');
const users = require('./routes/users');
const auth = require('./routes/auth');
const express = require('express');
const app = express();

if (!config.get('PrivateKey')) {
    console.error('FATAL ERROR: PrivateKey is not defin
    process.exit(1);
}

mongoose.connect('mongodb://localhost/mongo-game
    .then(() => console.log('Now connected to MongoD
    .catch(err => console.error('Something went wron

app.use(express.json());
app.use('/api/users', users);
app.use('/api/auth', auth);

const port = process.env.PORT || 4000;
app.listen(port, () => console.log(`Listening on port $
```

Lastly, we need to set the key using something like this.

```
user-registration $export PrivateKey=SecureAF
```

Setting Response Headers

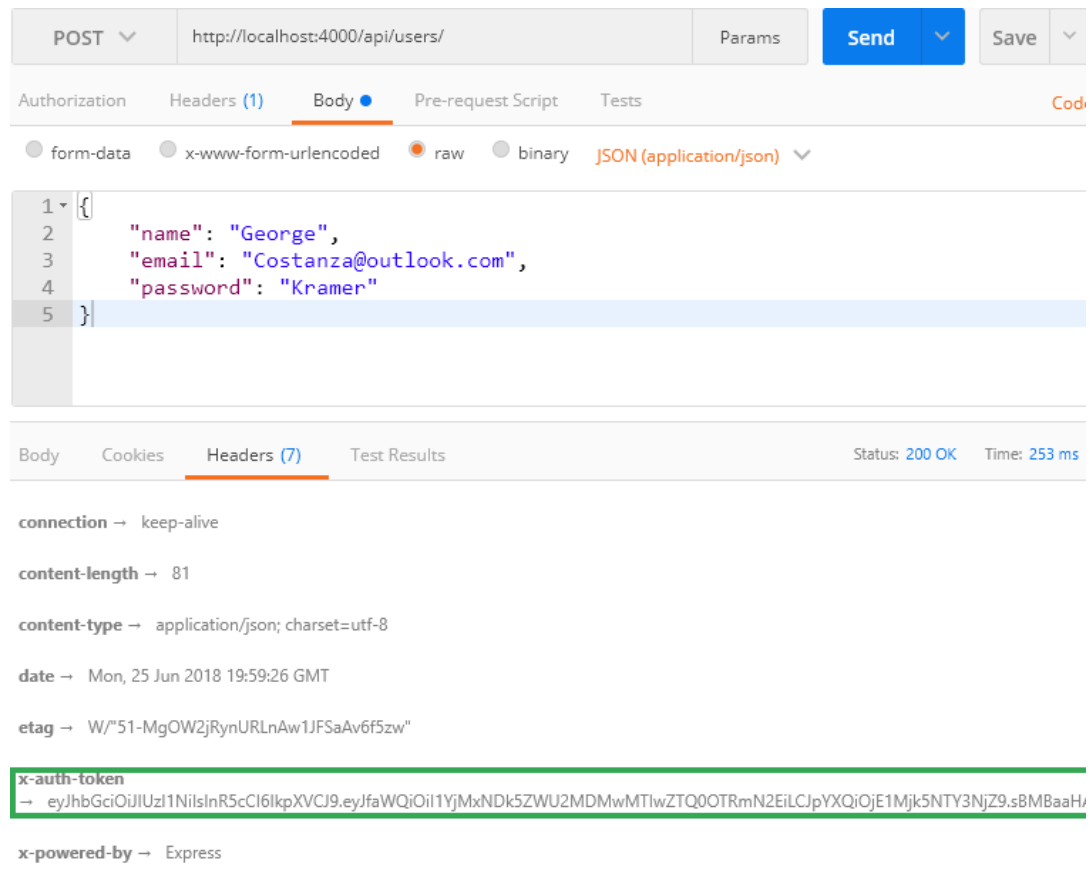
In the section above, we are successfully generating a JSON Web Token and sending it back to the client in the body of the response. Now we can make a few tweaks to send the token in the headers of the response which is a more common scenario. We can do this in the auth.js file for when a new user signs up.

```
const jwt = require('jsonwebtoken');
const config = require('config');
const bcrypt = require('bcrypt');
const _ = require('lodash');
const { User, validate } = require('../models/user');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {
    // First Validate The Request
    const { error } = validate(req.body);
    if (error) {
        return res.status(400).send(error.details[0].mes
    }

    // Check if this user already exisits
    let user = await User.findOne({ email: req.body.em
    if (user) {
        return res.status(400).send('That user already e
    } else {
        // Insert the new user if they do not exist yet
        user = new User(_.pick(req.body, ['name', 'ema
        const salt = await bcrypt.genSalt(10);
        user.password = await bcrypt.hash(user.passwor
        await user.save();
        const token = jwt.sign({ _id: user._id }, config.g
        res.header('x-auth-token', token).send(_.pick(us
    }
});

module.exports = router;
```

Now let's launch the application and create a new user from Postman.

```
user-registration $node index.js
Listening on port 4000...
Now connected to MongoDB!
```

In Postman when we send a request to create a new user and then inspect the response headers, we can see our generated JSON Web Token.



Now on the client side, this header can be read and stored for all subsequent API calls made to the server from the client.

Node.js MongoDB User Registration Summary

In this tutorial we covered the very basics of setting up user registration and authorization for a REST API powered by Node.js, Express, and MongoDB. This is for learning purposes only, and not code that should power any application in the real world! Here is what we learned.

- Authentication deals with determining if the user is who he or she claims to be by checking email and password.
- Authorization decides if the user has permission to perform certain operations.
- You should hash passwords using a package like bcrypt:

```
// To Hash a Password
const salt = await bcrypt.genSalt(10);
const hashed = await bcrypt.hash('abc123', salt);

// Validating passwords
const isValid = await bcrypt.compare(‘abc123’, hashed)
```

- A JSON Web Token is a JSON object encoded as a long string. They are used to identify users. The JWT may include a few public properties about a user in its payload. These properties cannot be tampered with because doing so requires re generating the digital signature.

- When a user logs in, you can generate a JWT on the server and return it to the client. The client can then use this token for all future API requests.
- To generate JSON Web Tokens you can use the `jsonwebtoken` package.

```
// Generating a JWT
const jwt = require('jsonwebtoken');
const token = jwt.sign({ _id: user._id}, 'privateKey');
```

- Do not store private keys in your code base. They should be stored in environment variables. The config package can then be used to read application settings stored in environment variables.
- There is no need to implement logging out on the server. It only has to be set up on the client by simply removing the JWT from the local storage.
- Do not store a JSON Web Token in plain text in the database. JSON Web Tokens should be stored on the client. If it is absolutely necessary for storing them on the server, make sure to encrypt them before storing them in a database.

#javascript #nodejs

← Mongoose Relationships Tutorial | Information Expert Principle Applied To Mongoose Models →