

Node.JS + MySQL - Boilerplate API with Email Sign Up & Verification, Authentication & Forgot Password

Objectives

The project aims to build a boilerplate sign up and authentication API with Node.js and MySQL that includes:

- Email sign up and verification
- JWT authentication with refresh tokens
- Role based authorization with support for two roles (User & Admin)
- Forgot password and reset password functionality
- Account management (CRUD) routes with role based access control
- Swagger api documentation route

Required Tools

- NodeJS - an open source server environment. It allows us to run JavaScript on the server.
- Visual Studio Code -A code editor to view and edit the API code
- MySQL - an instance for the API to connect to and store data.
- Ethereal - a fake SMTP service, mostly aimed at Nodemailer and EmailEngine users. We're using it to randomly generate emails and test our API's authentication capability.
- Git and GitHub - to initialize and save our API project remotely.

Initializing the Project

- Create a new folder from your local machine and name it however you want.
- Within the folder directory run the terminal and initialize the node package manager by typing in `npm init`
- Then after that follow the rest of the process below to install the required dependencies.

```
C:\Windows\System32\cmd.exe X + v
C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>npm i express --save
added 99 packages, and audited 100 packages in 5s
13 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>npm i bcryptjs body-parser cookie-parser cors
added 5 packages, and audited 105 packages in 3s
13 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>npm i express-jwt joi jsonwebtoken mysql2 nodemailer
added 22 packages, and audited 127 packages in 4s
13 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>npm i rootpath sequelize swagger-ui-express yamls
added 16 packages, and audited 143 packages in 6s
14 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

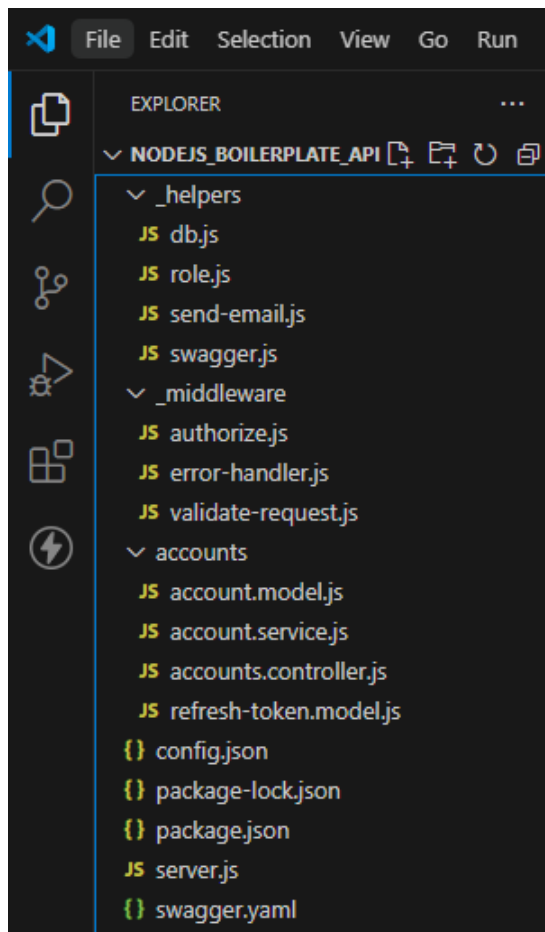
C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>npm i -D nodemon
added 26 packages, and audited 169 packages in 5s
18 packages are looking for funding
  run 'npm fund' for details
found 0 vulnerabilities

C:\Users\mathe\Downloads\New folder\nodejs_boilerplate_api>
```

Project Structure

The Project is composed of multiple files and directories, each serving their own purpose.

- Feature folders (accounts)
- Non-feature/Shared Component folders (_helpers, _middleware)
- Config.json
- Server.js
- Swagger.yaml



-Path: /_helpers

Contents :

- db.js
- role.js
- send-email.js
- swagger.js

-Path: /_middleware

Contents :

- authorize.js
- error-handler.js
- validate-request.js

-Path: /_accounts

Contents :

- account.model.js
- account.service.js
- accounts.controller.js
- refresh-token.model.js

-Config.json

-Package.json

-server.js

-swagger.yaml

MySQL Database Wrapper (Path: /_helpers/db.js)

```
JS db.js  M X
_helpers > JS db.js > initialize
1  const config = require('config.json');
2  const mysql = require('mysql2/promise');
3  const { Sequelize } = require('sequelize');
4
5  module.exports = db = {};
6
7  initialize();
8
9  async function initialize() {
10     const { host, port, user, password, database } = config.database;
11     const connection = await mysql.createConnection({ host, port, user, password });
12     await connection.query('CREATE DATABASE IF NOT EXISTS `${database}`');
13
14     const sequelize = new Sequelize(database, user, password, { dialect: 'mysql' });
15
16     db.Account = require('../accounts/account.model')(sequelize);
17     db.RefreshToken = require('../accounts/refresh-token.model')(sequelize);
18
19     db.Account.hasMany(db.RefreshToken, { onDelete: 'CASCADE' });
20     db.RefreshToken.belongsTo(db.Account);
21
22     await sequelize.sync();
23 }
```

Connects to MySQL using Sequelize to handle functions like handling database records by representing the data as objects.

Role Object / Enum (Path: `/_helpers/role.js`)

```
JS db.js M X JS role.js X
_helpers > JS role.js > [?] <unknown>
1  module.exports = {
2    Admin: 'Admin',
3    User: 'User'
4  }
```

Defines all the roles in the project application. Using it as an enum to avoid passing roles as strings. Therefore we use it as `Role.Admin` or `Role.User`

Send Email Helper (Path: `/_helpers/send-email.js`)

```
JS db.js M JS send-email.js X
_helpers > JS send-email.js > sendEmail
1  const nodemailer = require('nodemailer');
2  const config = require('config.json');
3
4  module.exports = sendEmail;
5
6  async function sendEmail({ to, subject, html, from = config.emailFrom }) {
7    const transporter = nodemailer.createTransport(config.smtpOptions);
8    await transporter.sendMail({ from, to, subject, html });
9  }
```

Simplifies sending emails in the application. Used by the account service to send account verification and password reset emails.

Swagger API Docs Route Handler (Path: `/_helpers/swagger.js`)

```
JS swagger.js M X
_helpers > JS swagger.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const YAML = require('yamljs');
4  const swaggerUi = require('swagger-ui-express');
5
6  const swaggerDocument = YAML.load('./swagger.yaml');
7
8  module.exports = router; {
9    router.use('/', swaggerUi.serve, swaggerUi.setup(swaggerDocument))
10 }
```

Auto-generates Swagger UI documentation based on the `swagger.yaml` file from the `/api-docs` path of the api.

Authorize Middleware (Path: /_middleware/authorize.js)

```
JS db.js M JS authorize.js M X
_middleware > JS authorize.js > authorize
1  const jwt = require('express-jwt');
2  const { secret } = require('config.json');
3  const db = require('_helpers/db');
4
5  module.exports = authorize;
6
7  function authorize(roles = []) {
8    if(typeof roles === 'string') {
9      roles = [roles];
10   }
11   return [
12     jwt({ secret, algorithms: ['HS256'] }),
13
14     async (req, res, next) => {
15       const account = await db.Account.findByPk(req.user.id);
16
17       if(!account || (roles.length && !roles.includes(account.role))) {
18         return res.status(401).json({ message: 'Unauthorized' });
19       }
20
21       req.user.role = account.role;
22       const refreshTokens = await account.getRefreshTokens();
23       req.user.ownsToken = token => !!refreshTokens.find(x => x.token === token);
24       next();
25     }
26   ];
27 }
```

Added to restrict access to any route which only authenticated users with specified roles can access. It is used by the accounts controller to handle authorization to CRUD routes as well as revoke token routes.

Global Error Handler Middleware (Path: /_middleware/error-handler.js)

```
JS db.js M JS authorize.js M JS error-handler.js X
_middleware > JS error-handler.js > errorHandler
1  module.exports = errorHandler;
2
3  function errorHandler(err, req, res, next) {
4    switch (true) {
5      case typeof err === 'string':
6        const is404 = err.toLowerCase().endsWith('not found');
7        const statusCode = is404 ? 404 : 400;
8        return res.status(statusCode).json({ message: err });
9      case err.name === 'UnauthorizedError':
10       return res.status(401).json({ message: 'Unauthorized' });
11      default:
12       return res.status(500).json({ message: err.message });
13    }
14  }
```

Catches all errors and removes the need for duplicated error handling code throughout the boilerplate application.

Validate Request Middleware (Path: `/_middleware/validate-request.js`)

```
JS dbjs M JS authorizejs M JS validate-requestjs M X
_middleware > JS validate-requestjs > validateRequest
1 module.exports = validateRequest;
2
3 function validateRequest(req, next, schema) {
4   const options = {
5     abortEarly: false,
6     allowUnknown: true,
7     stripUnknown: true
8   };
9   const { error, value } = schema.validate(req.body, options);
10  if(error) {
11    next(`Validation error: ${error.details.map(x => x.message).join(', ')}');
12  } else {
13    req.body = value;
14    next();
15  }
16 }
```

Validates the body of a request against a Joi schema object. Used by the accounts controller

Sequelize Account Model (Path: `/accounts/account.model.js`)

```
JS dbjs M JS authorizejs M JS validate-requestjs M JS account.modeljs X
accounts > JS account.modeljs > ...
1 const { DataTypes } = require('sequelize');
2
3 module.exports = model;
4
5 function model(sequelize) {
6   const attributes = {
7     email: { type: DataTypes.STRING, allowNull: false },
8     passwordHash: { type: DataTypes.STRING, allowNull: false },
9     title: { type: DataTypes.STRING, allowNull: false },
10    firstname: { type: DataTypes.STRING, allowNull: false },
11    lastname: { type: DataTypes.STRING, allowNull: false },
12    acceptTerms: { type: DataTypes.BOOLEAN },
13    role: { type: DataTypes.STRING, allowNull: false },
14    verificationToken: { type: DataTypes.STRING },
15    verified: { type: DataTypes.DATE },
16    resetToken: { type: DataTypes.STRING },
17    resetTokenExpires: { type: DataTypes.DATE },
18    passwordReset: { type: DataTypes.DATE },
19    created: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },
20    updated: { type: DataTypes.DATE },
21    isVerified: {
22      type: DataTypes.VIRTUAL,
23      get() { return !(this.verified || this.passwordReset); }
24    }
25  };
26
27  const options = {
28    timestamps: false,
29    defaultScope: {
30      attributes: { exclude: ['passwordHash'] }
31    },
32    scopes: {
```

Uses Sequelize to define the schema for the accounts table in the MySQL database. The exported Sequelize model object gives full access to perform CRUD operations on accounts in MySQL.

```
JS db.js M X JS authorize.js M JS validate-request.js M JS account.model.js X
accounts > JS account.model.js > ...
  5  function model(sequelize) {
27    const options = {
32      scopes: {
33        withHash: { attributes: {}, }
34      }
35    };
36
37    return sequelize.define('account', attributes, options);
38  }
```

Sequelize Refresh Token Model

(Path:/accounts/refresh-token.model.js)

```
JS db.js M JS authorize.js M JS validate-request.js M JS refresh-token.model.js M X
accounts > JS refresh-token.model.js > model
1  const { DataTypes } = require('sequelize');
2
3  module.exports = model;
4
5  function model(sequelize) {
6    const attributes = {
7      token: { type: DataTypes.STRING },
8      expires: { type: DataTypes.DATE },
9      created: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },
10     createdByIp: { type: DataTypes.STRING },
11     revoked: { type: DataTypes.DATE },
12     revokedByIp: { type: DataTypes.STRING },
13     replacedByToken: { type: DataTypes.STRING },
14     isExpired: {
15       type: DataTypes.VIRTUAL,
16       get() { return DATE.now() >= this.expires; }
17     },
18     isActive: {
19       type: DataTypes.VIRTUAL,
20       get() { return !this.revoked && !this.isExpired; }
21     }
22   };
23
24   const options = {
25     timestamps: false
26   };
27
28   return sequelize.define('refreshToken', attributes, options);
29 }
```

Uses Sequelize to define the schema for the refreshTokens table in the MySQL database. The exported Sequelize model object gives full access to perform CRUD operations on refresh tokens in MySQL

Account Service (Path:/accounts/account.service.js)

The service encapsulates all interaction with the Sequelize account models and exposes a simple set of methods which are used by the accounts controller.

```
{ package.json M    JS servers.js M    {} swagger.yaml M    JS swagger.js M    JS account.service.js M X    JS send-email.js    JS authorize.js M  ...
```

```
accounts > JS account.service.js > sendPasswordResetEmail
1  const config = require('config.json');
2  const jwt = require('jsonwebtoken');
3  const bcrypt = require('bcryptjs');
4  const crypto = require('crypto');
5  const { Op } = require('sequelize');
6  const sendEmail = require('_helpers/send-email');
7  const db = require('_helpers/db');
8  const Role = require('_helpers/role');
9
10 module.exports = {
11   authenticate,
12   refreshToken,
13   revokeToken,
14   register,
15   verifyEmail,
16   forgotPassword,
17   validateResetToken,
18   resetPassword,
19   getAll,
20   getIdById,
21   create,
22   update,
23   delete: _delete
24 }
25
26 async function authenticate({ email, password, ipAddress }) {
27   const account = await db.Account.scope('withHash').findOne({ where: { email } });
28
29   if (!account || !account.isVerified || !(await bcrypt.compare(password, account.passwordHash))) {
30     throw 'Email or password is incorrect';
31   }
32 }
```

```
package.json M JS server.js M {} swagger.yaml M JS swagger.js M JS account.service.js X JS send-email.js JS authorize.js M ...
accounts > JS account.service.js > sendPasswordResetEmail

26 async function authenticate({ email, password, ipAddress }) {
34     const jwtToken = generateJwtToken(account);
35     const refreshToken = generateRefreshToken(account, ipAddress);
36
37
38     await refreshToken.save();
39
40
41     return {
42         ...basicDetails(account),
43         jwtToken,
44         refreshToken: refreshToken.token
45     };
46 }
47
48 async function refreshToken ({ token, ipAddress }) {
49     const refreshToken = await getRefreshToken(token);
50     const account = await refreshToken.getAccount();
51
52
53     const newRefreshToken = generateRefreshToken(account, ipAddress);
54     refreshToken.revoked = Date.now();
55     refreshToken.revokedByIp = ipAddress;
56     refreshToken.replacedByToken = newRefreshToken.token;
57     await refreshToken.save();
58     await newRefreshToken.save();
59
60
61     const jwtToken = generateJwtToken(account);
62
63     return {
64         ...basicDetails(account),
65         jwtToken,
66         refreshToken: newRefreshToken.token
67     };
68 }
```



```
accounts > JS account.service.js > sendPasswordResetEmail
48 async function refreshToken ({ token, ipAddress }) {
64     return {
65         ...basicDetails(account),
66         jwtToken,
67         refreshToken: newRefreshToken.token
68     };
69 }
70
71 async function revokeToken({ token, ipAddress }) {
72     const refreshToken = await getRefreshToken(token);
73
74     refreshToken.revoked = date.now();
75     refreshToken.revokedByIp = ipAddress;
76     await refreshToken.save();
77 }
78
79 async function register(params, origin) {
80
81     if (await db.Account.findOne({ where: { email: params.email } })) {
82         return await sendAlreadyRegisteredEmail(params.email, origin);
83     }
84
85
86     const account = new db.Account(params);
87
88
89     const isFirstAccount = (await db.Account.count()) === 0;
90     account.role = isFirstAccount ? Role.Admin : Role.User;
91     account.verificationToken = randomTokenString();
92
93 }
```

```
accounts > JS account.service.js > sendPasswordResetEmail
79 async function register(params, origin) {
95     account.passwordHash = await hash(params.password);
96
97
98     await account.save();
99
100     await sendVerificationEmail(account, origin);
101
102 }
103
104 async function verifyEmail({token}) {
105     const account = await db.Account.findOne({ where: { verificationToken: token } });
106
107     if (!account) throw 'Verification failed';
108
109     account.verified = Date.now();
110     account.verificationToken = null;
111     await account.save();
112 }
113
114 async function forgotPassword({ email }, origin) {
115     const account = await db.Account.findOne({ where: { email } });
116
117     if (!account) return;
118
119
120
121     account.resetToken = randomTokenString();
122     account.resetTokenExpires = new Date(Date.now() + 24*60*60*1000);
123     await account.save();
124
125 }
```

```
accounts > JS account.service.js > sendPasswordResetEmail
114 async function forgotPassword({ email }, origin) {
126   await sendPasswordResetEmail(account, origin);
127 }
128
129 async function validateResetToken({ token }) {
130   const account = await db.Account.findOne({
131     where: {
132       resetToken: token,
133       resetTokenExpires: { [Op.gt]: Date.now() }
134     }
135   });
136
137   if (!account) throw 'Invalid token';
138
139   return account;
140 }
141
142 async function resetPassword({ token, password }) {
143   const account = await validateResetToken({ token });
144
145   account.passwordHash = await hash(password);
146   account.passwordReset = Date.now();
147   account.resetToken = null;
148   await account.save();
149 }
150
151
152 async function getAll() {
153   const accounts = await db.Account.findAll();
154   return accounts.map(x => basicDetails(x));
155 }
156
```

```
accounts > JS account.service.js > sendPasswordResetEmail
157 async function getById(id) {
158   const account = await getAccount(id);
159   return basicDetails(account);
160 }
161
162 async function create(params) {
163
164   if (await db.Account.findOne({ where: { email: params.email } })) {
165     throw 'Email ' + params.email + ' is already registered';
166   }
167
168   const account = new db.Account(params);
169   account.verified = Date.now();
170
171   account.passwordHash = await hash(params.password);
172
173   await account.save();
174
175   return basicDetails(account);
176 }
177
178
179
180 async function update(id, params) {
181   const account = await getAccount(id);
182
183   if (params.email && account.email !== params.email && await db.Account.findOne({ where: { email: params.email } })) {
184     throw 'Email ' + params.email + ' is already taken';
185   }
186
187   ...

```

```
accounts > JS account.service.js > sendPasswordResetEmail
188 async function update(id, params) {
189   if (params.password) {
190     params.passwordHash = await hash(params.password);
191   }
192
193   Object.assign(account, params);
194   account.updated = Date.now();
195   await account.save();
196
197   return basicDetails(account);
198 }
199
200
201 async function _delete(id) {
202   const account = await getAccount(id);
203   await account.destroy();
204 }
205
206 async function getAccount(id) {
207   const account = db.Account.findOne({ id });
208   if (!account) throw 'Account not found';
209   return account;
210 }
211
212 async function getRefreshToken(token) {
213   const refreshToken = await db.RefreshToken.findOne({ where: { token } });
214   if (!refreshToken || !refreshToken.isActive) throw 'Invalid token';
215   return refreshToken;
216 }
217
218 async function hash(password) {
219   return await bcrypt.hash(password, 10);
220 }
```

```
accounts > JS account.service.js > sendPasswordResetEmail
218 async function hash(password) {
219   return await bcrypt.hash(password, 10);
220 }
221
222 function generateJwtToken(account) {
223   return jwt.sign({ sub: account.id, id: account.id }, config.secret, { expiresIn: '15m' });
224 }
225
226 function generateRefreshToken(account, ipAddress) {
227   return new db.RefreshToken({
228     accountId: account.id,
229     token: randomTokenString(),
230     expires: new Date(Date.now() + 7*24*60*60*1000),
231     createdByIp: ipAddress
232   });
233 }
234
235 function randomTokenString() {
236   return crypto.randomBytes(40).toString('hex');
237 }
238
239 function basicDetails(account) {
240   const { id, title, firstName, lastName, email, role, created, updated, isVerified } = account;
241   return { id, title, firstName, lastName, email, role, created, updated, isVerified };
242 }
243
244 async function sendVerificationEmail(account, origin) {
245   let message;
246   if (origin) {
247     const verifyUrl = `${origin}/account/verify-email?token=${account.verificationToken}`;
248     message = `<p>Please click the below link to verify your email address:</p>`;
249   }
250 }
```

```
accounts > JS account.service.js > sendPasswordResetEmail
246 async function sendVerificationEmail(account, origin) {
251     <p><a href="${verifyUrl}">${verifyUrl}</a></p>;
252 } else {
253     message = `<p>Please use the below token to verify your email address with the <code>/account/verify-email/</code> api route</p>`;
254     <p><code>${account.verificationToken}</code></p>;
255 }
256
257 await sendEmail({
258     to: account.email,
259     subject: 'Sign-up Verification API - Verify Email',
260     html: `<h4>Verify Email</h4>
261     <p>Thanks for registering!</p>
262     ${message}`
263 });
264 }
265
266 async function sendAlreadyRegisteredEmail(email, origin) {
267     let message;
268     if (origin) {
269         message = `
270         <p>If you don't know your password please visit the <a href="${origin}/account/forgot-password">forgot password</a> api route</p>`;
271     } else {
272         message = `
273         <p>If you don't know your password you can reset it via the <code>/account/forgot-password/</code> api route</p>`;
274     }
275
276     await sendEmail({
277         to: email,
278         subject: 'Sign-up Verification API - Email Already Registered',
279         html: `<h4>Email Already Registered</h4>
280         <p>Your email <strong>${email}</strong> is already registered.</p>
281         ${message}`
282     });
283 }
284 }
```

```
accounts > JS account.service.js > sendPasswordResetEmail
286 async function sendAlreadyRegisteredEmail(email, origin) {
287     let message;
288     if (origin) {
289         const resetUrl = `${origin}/account/reset-password?token=${account.resetToken}`;
290         message = `<p>Please click the below link to reset your password, the link will be valid for 1 day:</p>
291         <p><a href="${resetUrl}">${resetUrl}</a></p>`;
292     } else {
293         message = `<p>Please use the below token to reset your password with the <code>/account/reset-password/</code> api route</p>
294         <p><code>${account.resetToken}</code></p>`;
295     }
296
297     await sendEmail({
298         to: account.email,
299         subject: 'Sign-up Verification API - Reset Password',
300         html: `<h4>Reset Password Email</h4>
301         ${message}`
302     });
303 }
```

Accounts Controller (*Path:/accounts/accounts.controller.js*)

Defines all /accounts routes for the Node.js + MySQL boilerplate api, the route definitions are grouped together at the top of the file and the implementation functions are below, followed by local helper functions. The controller is bound to the /accounts path in the main server.js file.

```
accounts > JS accounts.controller.js > setTokenCookie
1  const express = require('express');
2  const router = express.Router();
3  const Joi = require('joi');
4  const validateRequest = require('middleware/validate-request');
5  const authorize = require('middleware/authorize');
6  const Role = require('helpers/role');
7  const accountService = require('../account.service');
8
9  router.post('/authenticate', authenticateSchema, authenticate);
10 router.post('/refresh-token', refreshToken);
11 router.post('/revoke-token', authorize(), revokeTokenSchema, revokeToken);
12 router.post('/register', registerSchema, register);
13 router.post('/verify-email', verifyEmailSchema, verifyEmail);
14 router.post('/forgot-password', forgotPasswordSchema, forgotPassword);
15 router.post('/validate-reset-token', validateResetTokenSchema, validateResetToken);
16 router.post('/reset-password', resetPasswordSchema, resetPassword);
17 router.get('/', authorize(Role.Admin), getAll);
18 router.get('/:id', authorize(), getById);
19 router.post('/', authorize(Role.Admin), createSchema, create);
20 router.put('/:id', authorize(), updateSchema, update);
21 router.delete('/:id', authorize(), _delete);
22
23 module.exports = router;
24
25 function authenticateSchema(req, res, next) {
26   const schema = Joi.object({
27     email: Joi.string().required(),
28     password: Joi.string().required()
29   });
30   validateRequest(req, res, schema);
31 }
32
```

```
accounts > JS accounts.controller.js > setTokenCookie
33 function authenticate(req, res, next) {
34   const { email, password } = req.body;
35   const ipAddress = req.ip;
36   accountService.authenticate({ email, password, ipAddress })
37     .then(({ refreshToken, ...account }) => {
38       setTokenCookie(res, refreshToken);
39       res.json(account);
40     })
41     .catch(next);
42 }
43
44 function refreshToken(req, res, next) {
45   const token = req.cookies.refreshToken;
46   const ipAddress = req.ip;
47   accountService.refreshToken({ token, ipAddress })
48     .then(({ refreshToken, ...account }) => {
49       setTokenCookie(res, refreshToken);
50       res.json(account);
51     })
52     .catch(next);
53 }
54
55 function revokeTokenSchema(req, res, next) {
56   const schema = Joi.object({
57     token: Joi.string().empty('')
58   });
59   validateRequest(req, next, schema);
60 }
61
62 function revokeToken(req, res, next) {
63
```

```
accounts > JS accounts.controller.js > setTokenCookie
62 function revokeToken(req, res, next) {
63
64     const token = req.body.token || req.cookies.refreshToken;
65     const ipAddress = req.ip;
66
67     if(!token) return res.status(400).json({ message: 'Token is required' });
68
69
70     if (!req.user.ownsToken(token) && req.user.role !== Role.Admin) {
71         return res.status(401).json({ message: 'Unauthorized' });
72     }
73     accountService.revokeToken({ token, ipAddress })
74         .then(() => res.json({ message: 'Token revoked' }))
75         .catch(next);
76 }
77
78 function registerSchema(req, res, next) {
79     const schema = Joi.object({
80         title: Joi.string().required(),
81         firstname: Joi.string().required(),
82         lastname: Joi.string().required(),
83         email: Joi.string().email().required(),
84         password: Joi.string().min(6).required(),
85         confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
86         acceptTerms: Joi.boolean().valid(true).required()
87     });
88     validateRequest(req, next, schema);
89 }
90
91 function register(req, res, next) {
92     accountService.register(req.body, req.get('origin'))
93         .then(() => res.json({ message: 'Registration successful, please check your email for verification instructions' }))
94         .catch(next);
95 }
```

```
accounts > JS accounts.controller.js > setTokenCookie
91 function register(req, res, next) {
92     accountService.register(req.body, req.get('origin'))
93         .then(() => res.json({ message: 'Registration successful, please check your email for verification instructions' }))
94         .catch(next);
95 }
96
97 function verifyEmailSchema(req, res, next) {
98     const schema = Joi.object({
99         token: Joi.string().required()
100     });
101     validateRequest(req, next, schema);
102 }
103
104 function verifyEmail(req, res, next) {
105     accountService.verifyEmail(req.body)
106         .then(() => res.json({ message: 'Verification successful, you can now login' }))
107         .catch(next);
108 }
109
110 function forgotPasswordSchema(req, res, next) {
111     const schema = Joi.object({
112         email: Joi.string().email().required()
113     });
114     validateRequest(req, next, schema);
115 }
116
117 function forgotPassword(req, res, next) {
118     accountService.forgotPassword(req.body, req.get('origin'))
119         .then(() => res.json({ message: 'Please check your email for password reset instructions' }))
120         .catch(next);
121 }
122
123 function validateResetTokenSchema(req, res, next) {
124     const schema = Joi.object({
125         token: Joi.string().required()
126     });
127     validateRequest(req, next, schema);
128 }
```

```
accounts > JS accounts.controllers.js > setTokenCookie
123 function validateResetTokenSchema(req, res, next) {
124   const schema = Joi.object({
125     token: Joi.string().required()
126   });
127   validateRequest(req, next, schema);
128 }
129
130 function validateResetToken(req, res, next) {
131   accountService.validateResetToken(req.body)
132     .then(() => res.json({ message: 'Token is valid' }))
133     .catch(next);
134 }
135
136 function resetPasswordSchema(req, res, next) {
137   const schema = Joi.object({
138     token: Joi.string().required(),
139     password: Joi.string().min(6).required(),
140     confirmPassword: Joi.string().valid(Joi.ref('password')).required()
141   });
142   validateRequest(req, next, schema);
143 }
144
145 function resetPassword(req, res, next) {
146   accountService.resetPassword(req.body)
147     .then(() => res.json({ message: 'Password reset successful, you can now log in' }))
148     .catch(next);
149 }
150
151 function getAll(req, res, next){
152   accountService.getAll()
153     .then(accounts => res.json(accounts))
154     .catch(next);
155 }
```

Ln 235, Col 2 Spaces: 4 UTF-8 CRLF {} JavaScript

```
accounts > JS accounts.controllers.js > setTokenCookie
151 function getAll(req, res, next){
152 }
153
154 function getById(req, res, next) {
155   if(Number(req.params.id) !== req.user.id && req.user.role !== Role.Admin) {
156     return res.status(401).json({ message: 'Unauthorized' });
157   }
158
159   accountService.getById(req.params.id)
160     .then(account => account ? res.json(account) : res.sendStatus(404))
161     .catch(next);
162 }
163
164 function createSchema(req, res, next) {
165   const schema = Joi.object({
166     title: Joi.string().required(),
167     firstname: Joi.string().required(),
168     lastname: Joi.string().required(),
169     email: Joi.string().email().required(),
170     password: Joi.string().min(6).required(),
171     confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
172     role: Joi.string().valid(Role.Admin, Role.User).required()
173   });
174   validateRequest(req, next, schema);
175 }
176
177 function create(req, res, next) {
178   accountService.create(req.body)
179     .then(account => res.json(account))
180     .catch(next);
181 }
182 }
```

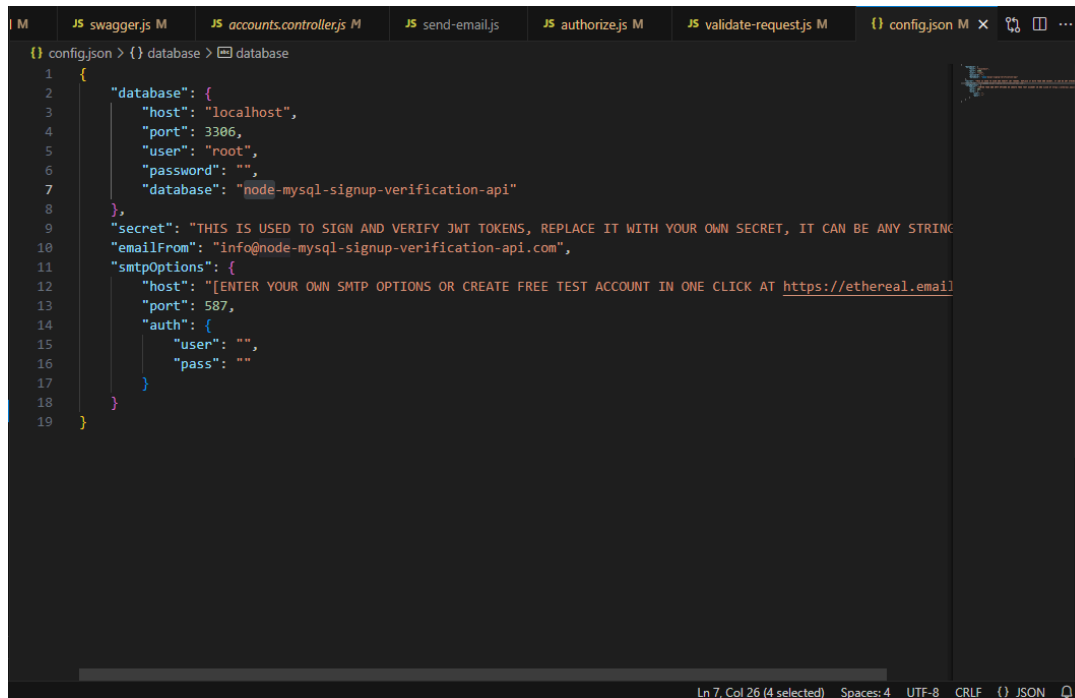
Ln 235, Col 2 Spaces: 4 UTF-8 CRLF {} JavaScript

```
accounts > JS accounts.controllerjs > setTokenCookie
187 function updateSchema(req, res, next) {
188   const schemaRules = {
189     title: Joi.string().empty(''),
190     firstname: Joi.string().empty(''),
191     lastname: Joi.string().empty(''),
192     email: Joi.string().email().empty(''),
193     password: Joi.string().min(6).empty(''),
194     confirmPassword: Joi.string().valid(Joi.ref('password')).empty('')
195   };
196
197   if (req.user.role === Role.Admin) {
198     schemaRules.role = Joi.string().valid(Role.Admin, Role.User).empty('');
199   }
200
201   const schema = Joi.object(schemaRules).with('password', 'confirmPassword');
202   validateRequest(req, next, schema);
203 }
204
205 function update(req, res, next) {
206
207   if (Number(req.params.id) !== req.user.id && req.user.role !== Role.Admin) {
208     return res.status(401).json({ message: 'Unauthorized' });
209   }
210
211   accountService.update(req.params.id, req.body)
212     .then(account => res.json(account))
213     .catch(next);
214 }
215
216 function _delete(req, res, next) {
217
218
```

```
accounts > JS accounts.controllerjs > setTokenCookie
217 function _delete(req, res, next) {
218
219   if (Number(req.params.id) !== req.user.id && req.user.role !== Role.Admin) {
220     return res.status(401).json({ message: 'Unauthorized' });
221   }
222
223   accountService.delete(req.params.id)
224     .then(() => res.json({ message: 'Account deleted successfully' }))
225     .catch(next);
226 }
227
228 function setTokenCookie(res, token) {
229
230   const cookieOptions = {
231     httpOnly: true,
232     expires: new Date(Date.now() + 7*24*60*60*1000)
233   };
234   res.cookie('refreshToken', token, cookieOptions);
235 }
```

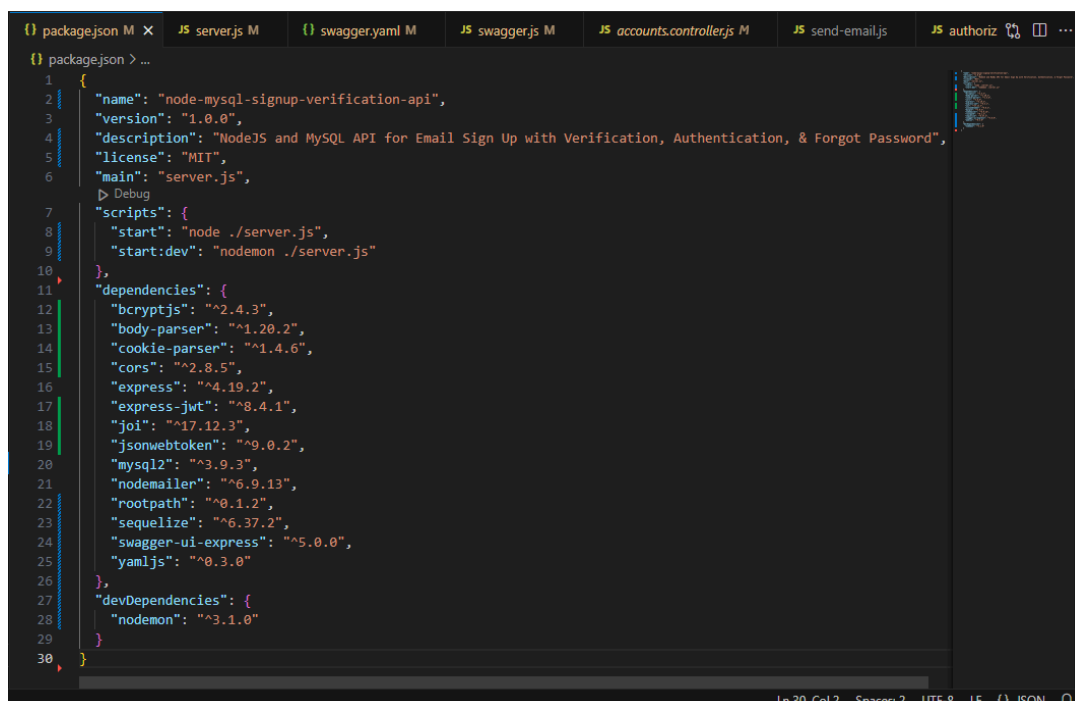

API Config (Path:/config.json)

Contains configuration data for the boilerplate api, it includes the database connection options for the MySQL database, the secret used for signing and verifying JWT tokens, the emailFrom address used to send emails, and the smtpOptions used to connect and authenticate with an email server.

A screenshot of a VS Code editor window with the 'config.json' file open. The file contains a JSON object with the following structure: { "database": { "host": "localhost", "port": 3306, "user": "root", "password": "", "database": "node-mysql-signup-verification-api" }, "secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET, IT CAN BE ANY STRING", "emailFrom": "info@node-mysql-signup-verification-api.com", "smtpOptions": { "host": "[ENTER YOUR OWN SMTP OPTIONS OR CREATE FREE TEST ACCOUNT IN ONE CLICK AT https://ethereal.email]", "port": 587, "auth": { "user": "", "pass": "" } } }. The editor has a dark theme and shows line numbers from 1 to 19. The status bar at the bottom indicates 'Ln 7, Col 26 (4 selected)', 'Spaces: 4', 'UTF-8', 'CRLF', and 'JSON'.

Package.json (Path:/package.json)

The package.json file contains project configuration information including package dependencies which get installed when you run npm install.

A screenshot of a VS Code editor window with the 'package.json' file open. The file contains a JSON object with the following structure: { "name": "node-mysql-signup-verification-api", "version": "1.0.0", "description": "NodeJS and MySQL API for Email Sign Up with Verification, Authentication, & Forgot Password", "license": "MIT", "main": "server.js", "scripts": { "start": "node ./server.js", "start:dev": "nodemon ./server.js" }, "dependencies": { "bcryptjs": "^2.4.3", "body-parser": "^1.20.2", "cookie-parser": "^1.4.6", "cors": "^2.8.5", "express": "^4.19.2", "express-jwt": "^8.4.1", "joi": "^17.12.3", "jsonwebtoken": "^9.0.2", "mysql2": "^3.9.3", "nodemailer": "^6.9.13", "rootpath": "^0.1.2", "sequelize": "^6.37.2", "swagger-ui-express": "^5.0.0", "yamljs": "^0.3.0" }, "devDependencies": { "nodemon": "^3.1.0" } }. The editor has a dark theme and shows line numbers from 1 to 30. The status bar at the bottom indicates 'Ln 30, Col 2', 'Spaces: 2', 'UTF-8', 'LF', and 'JSON'.

Server Startup File (*Path:/server.js*)

The server.js file is the entry point into the boilerplate Node.js api, it configures application middleware, binds controllers to routes and starts the Express web server for the api.

```
1 require('rootpath')();
2
3 const express = require('express');
4 const app = express();
5 const bodyParser = require('body-parser');
6 const cookieParser = require('cookie-parser');
7 const cors = require('cors');
8 const errorHandler = require('_middleware/error-handler');
9
10 app.use(bodyParser.urlencoded({ extended: false }));
11 app.use(bodyParser.json());
12 app.use(cookieParser());
13
14 app.use(cors({ origin: (origin, callback) => callback(null, true), credentials:true }));
15
16 app.use('/accounts', require('./accounts/accounts.controller'));
17
18 app.use('/api-docs', require('_helpers/swagger'));
19
20 app.use(errorHandler);
21
22 const port = process.env.NODE_ENV === 'production' ? (process.env.PORT || 80) : 4000;
23 app.listen(port, () => console.log('Server listening on port ' + port));
```

Swagger API Documentation (*Path:/swagger.yaml*)

The YAML documentation is used by the swagger.js helper to automatically generate and serve interactive Swagger UI documentation on the /api-docs route of the boilerplate api. To preview the Swagger UI documentation without running the api simply copy and paste the below YAML into the swagger editor at [Swagger Editor](#).

```
openapi: 3.0.0
info:
  title: Node.js Sign-up and Verification API
  description: Node.js and MySQL - API with email sign-up, verification, authentication and forgot password
  version: 1.0.0

servers:
  - url: http://localhost:4000
    description: Local development server

paths:
```

```
/accounts/authenticate:
  post:
    summary: Authenticate account credentials and return a JWT token and
a cookie with a refresh token
    description: Accounts must be verified before authenticating.
    operationId: authenticate
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              email:
                type: string
                example: "jason@example.com"
              password:
                type: string
                example: "pass123"
            required:
              - email
              - password
    responses:
      "200":
        description: Account details, a JWT access token and a refresh
token cookie
        headers:
          Set-Cookie:
            description: "`refreshToken`"
            schema:
              type: string
              example:
refreshToken=51872eca5efedcf424db4cf5afd16a9d00ad25b743a034c9c221afc85d18d
cd5e4ad6e3f08607550; Path=/; Expires=Tue, 16 Jun 2020 09:14:17 GMT;
HttpOnly
        content:
          application/json:
            schema:
              type: object
              properties:
                id:
                  type: string
                  example: "5eb12e197e06a76ccdefc121"
                title:
                  type: string
                  example: "Mr"
                firstName:
```

```

        type: string
        example: "Jason"
    lastName:
        type: string
        example: "Watmore"
    email:
        type: string
        example: "jason@example.com"
    role:
        type: string
        example: "Admin"
    created:
        type: string
        example: "2020-05-05T09:12:57.848Z"
    isVerified:
        type: boolean
        example: true
    jwtToken:
        type: string
        example:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1ZWlzMmUxOTdlMDZhNzZjY2RlZmMxMjEiLCJpZCI6IjVlYjEyZTE5N2UwNmE3NmNjZGVmYzEyMSIsImhhdCI6MTU4ODc1ODE1N3O.xR9H0STbFOPskUGA9jHNZOJ6eS7umHHqKRhI807YT1Y"
    "400":
        description: The email or password is incorrect
        content:
            application/json:
                schema:
                    type: object
                    properties:
                        message:
                            type: string
                            example: "Email or password is incorrect"
/accounts/refresh-token:
    post:
        summary: Use a refresh token to generate a new JWT token and a new
refresh token
        description: The refresh token is sent and returned via cookies.
        operationId: refreshToken
        parameters:
            - in: cookie
              name: refreshToken
              description: The `refreshToken` cookie
              schema:
                  type: string

```

```
    example:
51872eca5efedcf424db4cf5afd16a9d00ad25b743a034c9c221afc85d18dcd5e4ad6e3f08
607550
  responses:
    "200":
      description: Account details, a JWT access token and a new
refresh token cookie
      headers:
        Set-Cookie:
          description: "`refreshToken`"
          schema:
            type: string
            example:
refreshToken=51872eca5efedcf424db4cf5afd16a9d00ad25b743a034c9c221afc85d18d
cd5e4ad6e3f08607550; Path=/; Expires=Tue, 16 Jun 2020 09:14:17 GMT;
HttpOnly
      content:
        application/json:
          schema:
            type: object
            properties:
              id:
                type: string
                example: "5eb12e197e06a76ccdefc121"
              title:
                type: string
                example: "Mr"
              firstName:
                type: string
                example: "Jason"
              lastName:
                type: string
                example: "Watmore"
              email:
                type: string
                example: "jason@example.com"
              role:
                type: string
                example: "Admin"
              created:
                type: string
                example: "2020-05-05T09:12:57.848Z"
              isVerified:
                type: boolean
                example: true
              jwtToken:
                type: string
```

```

        example:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1ZWlzMmUxOTdlMDZhNzZjY2RlZmMxMjEiLCJpZCI6IjVlYjEyZTE5N2UwNmE3NmNjZGVmYzEyMSIsImhhdCI6MTU0ODc1ODE1N3O.xR9H0STbFOpSkuGA9jHNZOJ6eS7umHHqKRhI807YT1Y"
    "400":
        description: The refresh token is invalid, revoked or expired
        content:
            application/json:
                schema:
                    type: object
                    properties:
                        message:
                            type: string
                            example: "Invalid token"
/accounts/revoke-token:
    post:
        summary: Revoke a refresh token
        description: Admin users can revoke the tokens of any account,
        regular users can only revoke their own tokens.
        operationId: revokeToken
        security:
            - bearerAuth: []
        parameters:
            - in: cookie
              name: refreshToken
              description: The refresh token can be sent in a cookie or the
        post body, if both are sent the token in the body is used.
        schema:
            type: string
            example:
51872eca5efedcf424db4cf5afd16a9d00ad25b743a034c9c221afc85d18dcd5e4ad6e3f08
607550
        requestBody:
            content:
                application/json:
                    schema:
                        type: object
                        properties:
                            token:
                                type: string
                                example:
"51872eca5efedcf424db4cf5afd16a9d00ad25b743a034c9c221afc85d18dcd5e4ad6e3f0
8607550"
        responses:
            "200":
                description: The refresh token was successfully revoked
                content:

```

```
    application/json:
      schema:
        type: object
        properties:
          message:
            type: string
            example: "Token revoked"
  "400":
    description: The refresh token is invalid
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Invalid token"
  "401":
    $ref: "#/components/responses/UnauthorizedError"
/accounts/register:
  post:
    summary: Register a new user account and send a verification email
    description: The first account registered in the system is assigned
the `Admin` role, other accounts are assigned the `User` role.
    operationId: register
    requestBody:
      required: true
    content:
      application/json:
        schema:
          type: object
          properties:
            title:
              type: string
              example: "Mr"
            firstName:
              type: string
              example: "Jason"
            lastName:
              type: string
              example: "Watmore"
            email:
              type: string
              example: "jason@example.com"
            password:
              type: string
              example: "pass123"
```

```

        confirmPassword:
          type: string
          example: "pass123"
        acceptTerms:
          type: boolean
      required:
        - title
        - firstName
        - lastName
        - email
        - password
        - confirmPassword
        - acceptTerms
    responses:
      "200":
        description: The registration request was successful and a
        verification email has been sent to the specified email address
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: "Registration successful, please check your
email for verification instructions"
            /accounts/verify-email:
              post:
                summary: Verify a new account with a verification token received by
                email after registration
                operationId: verifyEmail
                requestBody:
                  required: true
                  content:
                    application/json:
                      schema:
                        type: object
                        properties:
                          token:
                            type: string
                            example:
"3c7f8d9c4cb348ff95a0b74a1452aa24fc9611bb76768bb9eafeeb826ddae2935f1880bc7
713318f"
                      required:
                        - token
                responses:
                  "200":

```


description: Verification was successful so you can now login to the account

content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 example: "Verification successful, you can now login"

"400":

description: Verification failed due to an invalid token
content:
 application/json:
 schema:
 type: object
 properties:
 message:
 type: string
 example: "Verification failed"

/accounts/forgot-password:

post:

summary: Submit email address to reset the password on an account

operationId: forgotPassword

requestBody:

required: true

content:

application/json:
 schema:
 type: object
 properties:
 email:
 type: string
 example: "jason@example.com"
 required:
 - email

responses:

"200":

description: The request was received and an email has been sent to the specified address with password reset instructions (if the email address is associated with an account)

content:
 application/json:
 schema:
 type: object
 properties:
 message:

```

        type: string
        example: "Please check your email for password reset
instructions"
/accounts/validate-reset-token:
  post:
    summary: Validate the reset password token received by email after
submitting to the /accounts/forgot-password route
    operationId: validateResetToken
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              token:
                type: string
                example:
"3c7f8d9c4cb348ff95a0b74a1452aa24fc9611bb76768bb9eafeeb826ddae2935f1880bc7
713318f"
            required:
              - token
    responses:
      "200":
        description: Token is valid
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: "Token is valid"
      "400":
        description: Token is invalid
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: "Invalid token"
/accounts/reset-password:
  post:
    summary: Reset the password for an account
    operationId: resetPassword

```

```

requestBody:
  required: true
  content:
    application/json:
      schema:
        type: object
        properties:
          token:
            type: string
            example:
"3c7f8d9c4cb348ff95a0b74a1452aa24fc9611bb76768bb9eafeeb826ddae2935f1880bc7
713318f"
          password:
            type: string
            example: "newPass123"
          confirmPassword:
            type: string
            example: "newPass123"
        required:
          - token
          - password
          - confirmPassword
responses:
  "200":
    description: Password reset was successful so you can now login
to the account with the new password
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Password reset successful, you can now
login"
  "400":
    description: Password reset failed due to an invalid token
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Invalid token"
/accounts:
  get:

```

```

summary: Get a list of all accounts
description: Restricted to admin users.
operationId: getAllAccounts
security:
  - bearerAuth: []
responses:
  "200":
    description: An array of all accounts
    content:
      application/json:
        schema:
          type: array
          items:
            type: object
            properties:
              id:
                type: string
                example: "5eb12e197e06a76ccdefc121"
              title:
                type: string
                example: "Mr"
              firstName:
                type: string
                example: "Jason"
              lastName:
                type: string
                example: "Watmore"
              email:
                type: string
                example: "jason@example.com"
              role:
                type: string
                example: "Admin"
              created:
                type: string
                example: "2020-05-05T09:12:57.848Z"
              updated:
                type: string
                example: "2020-05-08T03:11:21.553Z"
  "401":
    $ref: "#/components/responses/UnauthorizedError"
post:
  summary: Create a new account
  description: Restricted to admin users.
  operationId: createAccount
  security:
    - bearerAuth: []

```

```
requestBody:
  required: true
  content:
    application/json:
      schema:
        type: object
        properties:
          title:
            type: string
            example: "Mr"
          firstName:
            type: string
            example: "Jason"
          lastName:
            type: string
            example: "Watmore"
          email:
            type: string
            example: "jason@example.com"
          password:
            type: string
            example: "pass123"
          confirmPassword:
            type: string
            example: "pass123"
          role:
            type: string
            enum: [Admin, User]
        required:
          - title
          - firstName
          - lastName
          - email
          - password
          - confirmPassword
          - role
responses:
  "200":
    description: Account created successfully, verification is not
    required for accounts created with this endpoint. The details of the new
    account are returned.
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
```

```

        type: string
        example: "5eb12e197e06a76ccdefc121"
    title:
        type: string
        example: "Mr"
    firstName:
        type: string
        example: "Jason"
    lastName:
        type: string
        example: "Watmore"
    email:
        type: string
        example: "jason@example.com"
    role:
        type: string
        example: "Admin"
    created:
        type: string
        example: "2020-05-05T09:12:57.848Z"
"400":
    description: Email is already registered
    content:
        application/json:
            schema:
                type: object
                properties:
                    message:
                        type: string
                        example: "Email 'jason@example.com' is already
registered"
"401":
    $ref: "#/components/responses/UnauthorizedError"
/accounts/{id}:
    parameters:
        - in: path
          name: id
          description: Account id
          required: true
          example: "5eb12e197e06a76ccdefc121"
          schema:
              type: string
    get:
        summary: Get a single account by id
        description: Admin users can access any account, regular users are
restricted to their own account.
        operationId: getAccountById

```

```

security:
  - bearerAuth: []
responses:
  "200":
    description: Details of the specified account
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: string
              example: "5eb12e197e06a76ccdefc121"
            title:
              type: string
              example: "Mr"
            firstName:
              type: string
              example: "Jason"
            lastName:
              type: string
              example: "Watmore"
            email:
              type: string
              example: "jason@example.com"
            role:
              type: string
              example: "Admin"
            created:
              type: string
              example: "2020-05-05T09:12:57.848Z"
            updated:
              type: string
              example: "2020-05-08T03:11:21.553Z"
  "404":
    $ref: "#/components/responses/NotFoundError"
  "401":
    $ref: "#/components/responses/UnauthorizedError"
put:
  summary: Update an account
  description: Admin users can update any account including role,
regular users are restricted to their own account and cannot update role.
  operationId: updateAccount
  security:
    - bearerAuth: []
  requestBody:
    required: true

```

```
content:
  application/json:
    schema:
      type: object
      properties:
        title:
          type: string
          example: "Mr"
        firstName:
          type: string
          example: "Jason"
        lastName:
          type: string
          example: "Watmore"
        email:
          type: string
          example: "jason@example.com"
        password:
          type: string
          example: "pass123"
        confirmPassword:
          type: string
          example: "pass123"
        role:
          type: string
          enum: [Admin, User]
responses:
  "200":
    description: Account updated successfully. The details of the
updated account are returned.
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: string
              example: "5eb12e197e06a76ccdefc121"
            title:
              type: string
              example: "Mr"
            firstName:
              type: string
              example: "Jason"
            lastName:
              type: string
              example: "Watmore"
```



```

        email:
          type: string
          example: "jason@example.com"
        role:
          type: string
          example: "Admin"
        created:
          type: string
          example: "2020-05-05T09:12:57.848Z"
        updated:
          type: string
          example: "2020-05-08T03:11:21.553Z"
      "404":
        $ref: "#/components/responses/NotFoundError"
      "401":
        $ref: "#/components/responses/UnauthorizedError"
    delete:
      summary: Delete an account
      description: Admin users can delete any account, regular users are
restricted to their own account.
      operationId: deleteAccount
      security:
        - bearerAuth: []
      responses:
        "200":
          description: Account deleted successfully
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: "Account deleted successfully"
        "404":
          $ref: "#/components/responses/NotFoundError"
        "401":
          $ref: "#/components/responses/UnauthorizedError"

components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
  responses:
    UnauthorizedError:

```

```

    description: Access token is missing or invalid, or the user does
not have access to perform the action
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Unauthorized"
NotFoundError:
  description: Not Found
  content:
    application/json:
      schema:
        type: object
        properties:
          message:
            type: string
            example: "Not Found"

```

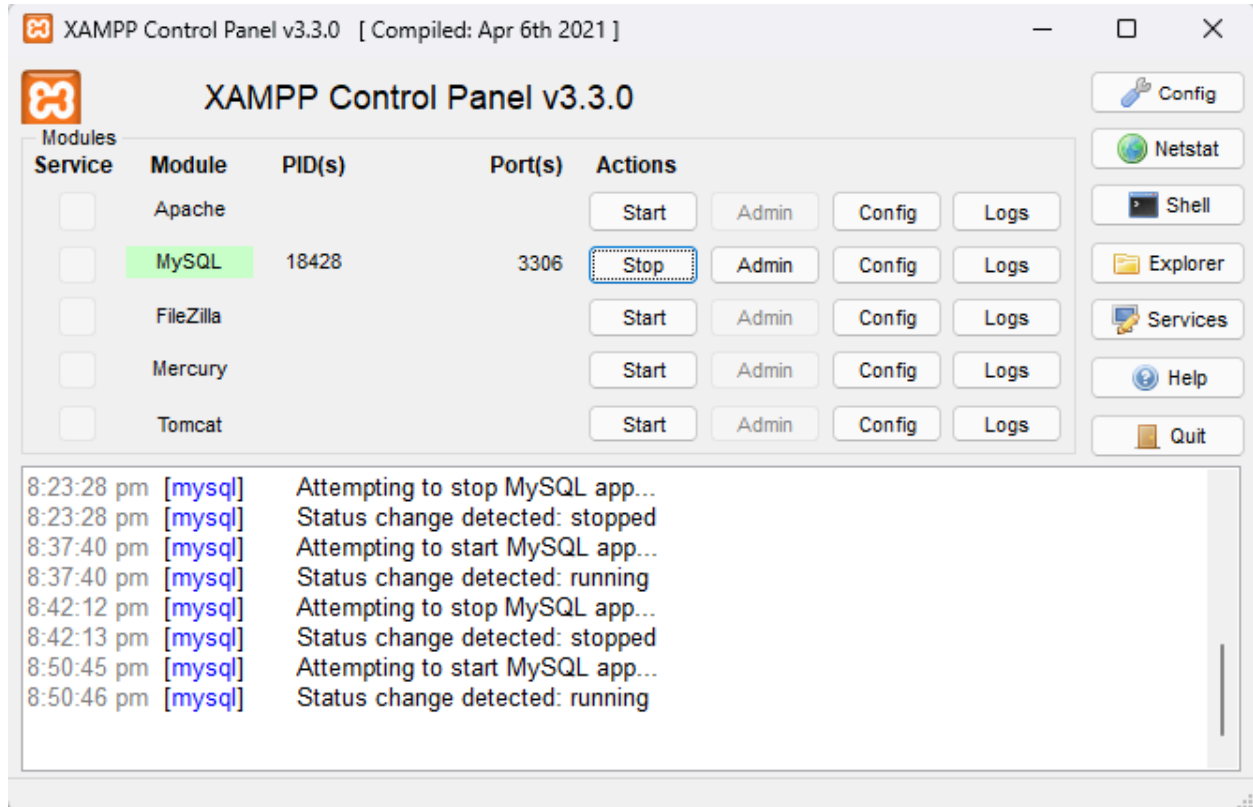
Preparations

Before the test, Ensure that you have followed the following guidelines:

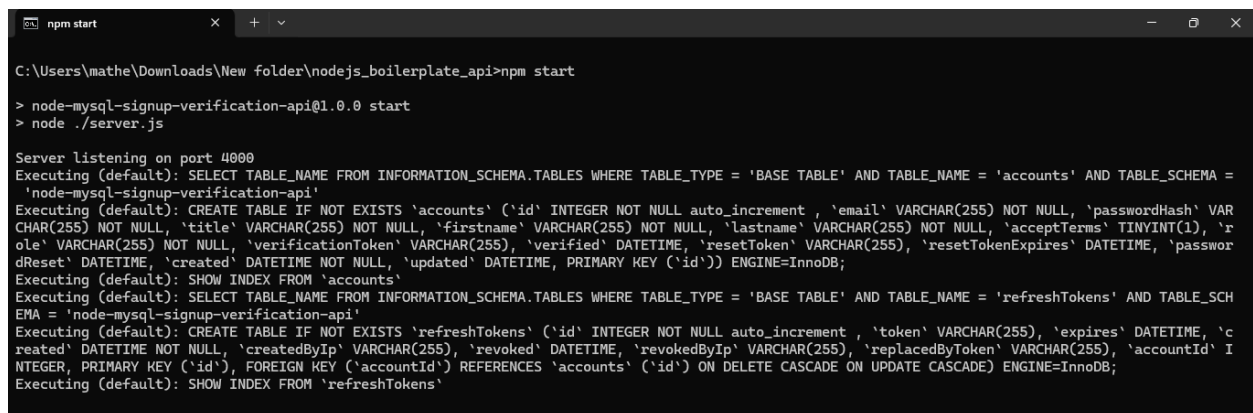
- Installed NodeJS from their official website [Download Node.js](#)
- Installed MySQL from their website [Download MySQL Community Server](#). Or use the XAMPP installer [Apache Friends](#) to run MySQL(MariaDB) on the XAMPP Control Panel.
- Prepared the API project source code.
- Installed the required npm packages by running the command `npm i` or `npm install` in the command-line within the root folder of the project.
- Configured the SMTP setting for email within the `smtpOptions` property in the `/src/config.json` file. We use [Ethereal Email](#) for testing.
- Updated the secret property in the `config.json` file as it is used for signing and verifying JWT tokens for authentication. We use [GUID Generator](#) to join a couple of GUIDs together and make a long random string.
- And finally start the API by running `npm start` (or `npm run start:dev` to start with nodemon) from the command line in the project root folder, you should see the message Server listening on port 4000, and you can view the Swagger API documentation at <http://localhost:4000/api-docs>.

Testing the API Locally using NodeJS

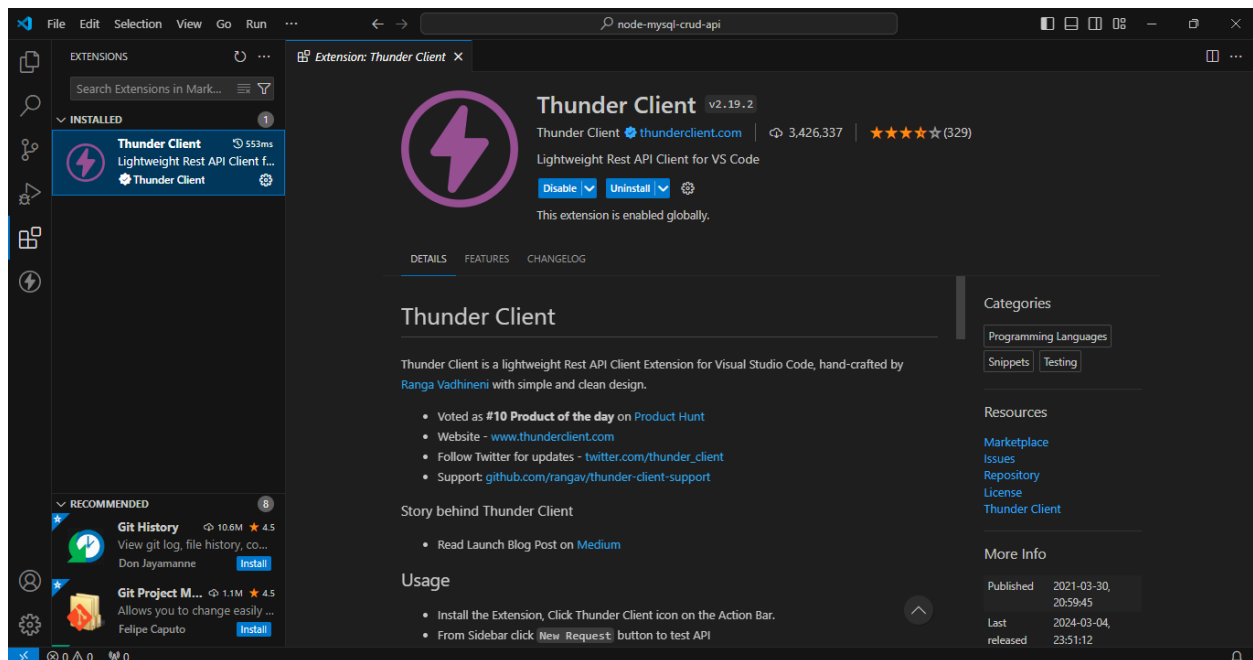
First thing to do is run an instance of your MySQL Server



Then type 'npm start' or 'npm run start:dev' in the terminal within your project folder's root directory to start the server. It should show the Sequelize taking action.

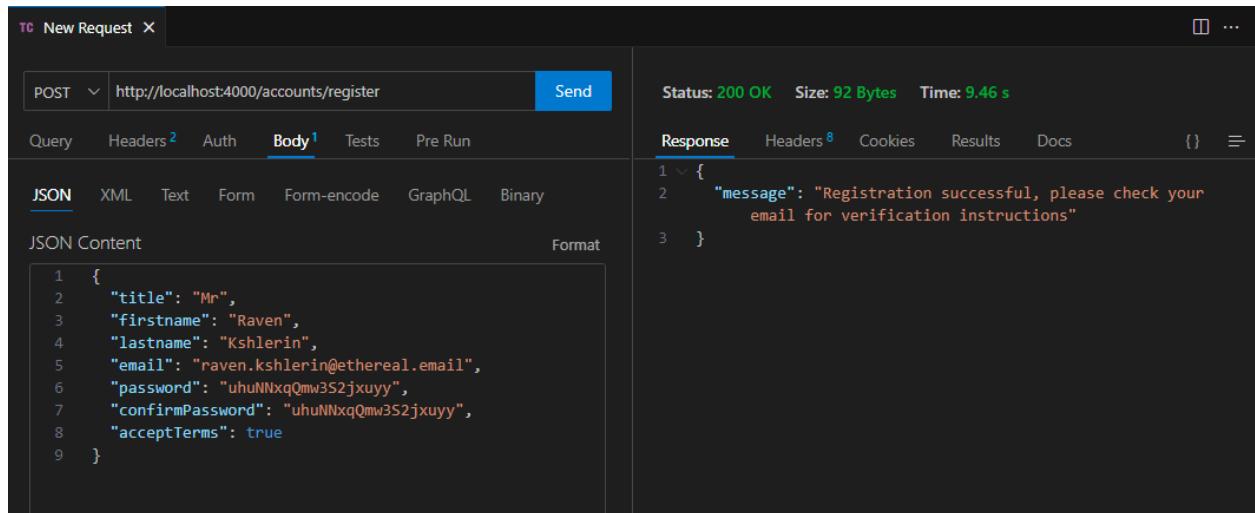


You can test the API directly with a tool such as Postman or VSCode extension ThunderClient. This time around we use Thunder Client.

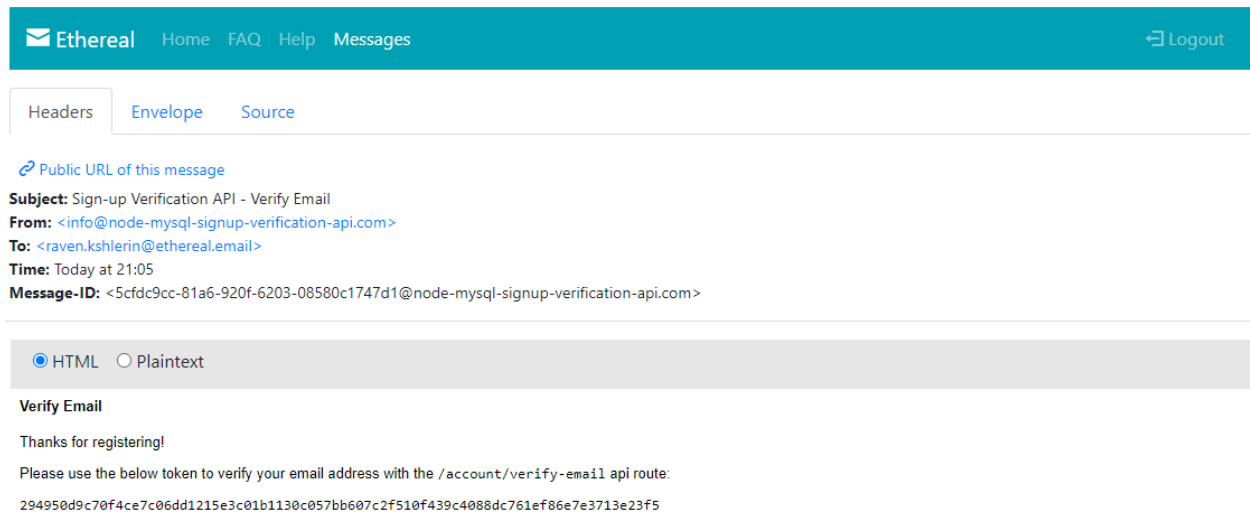


To register a new account with the Node.js boilerplate api follow these steps:

- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the register route of your local API - `http://localhost:4000/accounts/register`
- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object containing the required user properties in the "Body" textarea, e.g:
- Click the "Send" button, you should receive a "200 OK" response with a "registration successful" message in the response body.



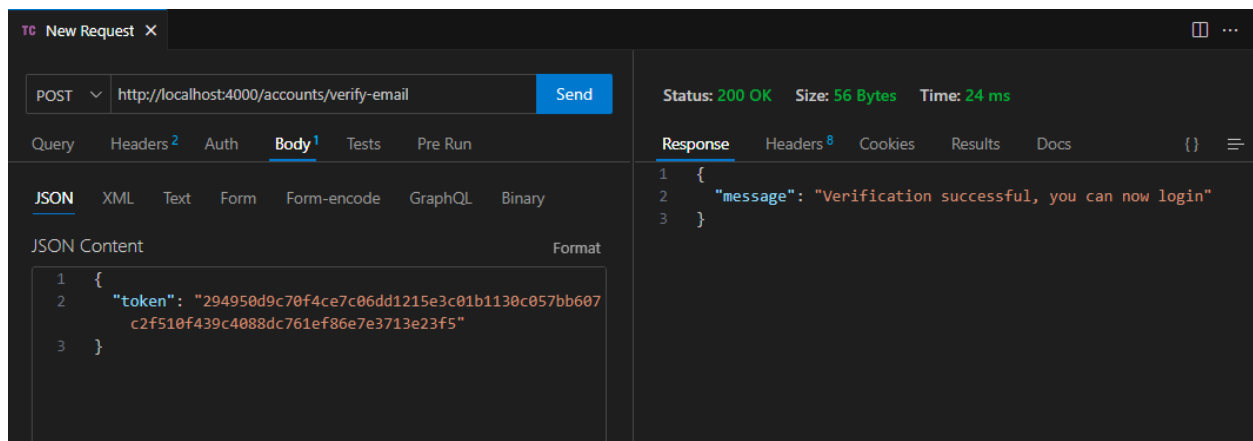
Received a verification email with the token to verify the newly created account.



To verify an account with the Node api follow these steps:

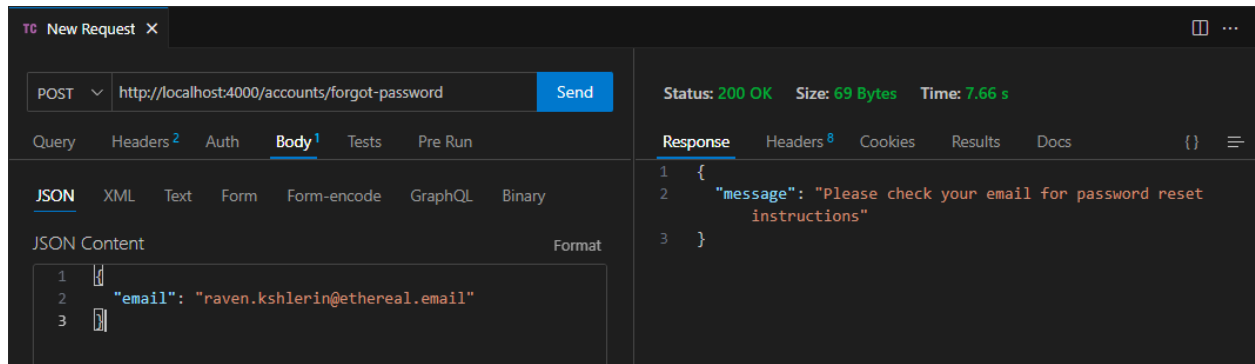
- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/verify-email`

- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object containing the token received in the verification email (in the previous step) in the "Body" textarea, e.g:
- Click the "Send" button, you should receive a "200 OK" response with a "verification successful" message in the response body.

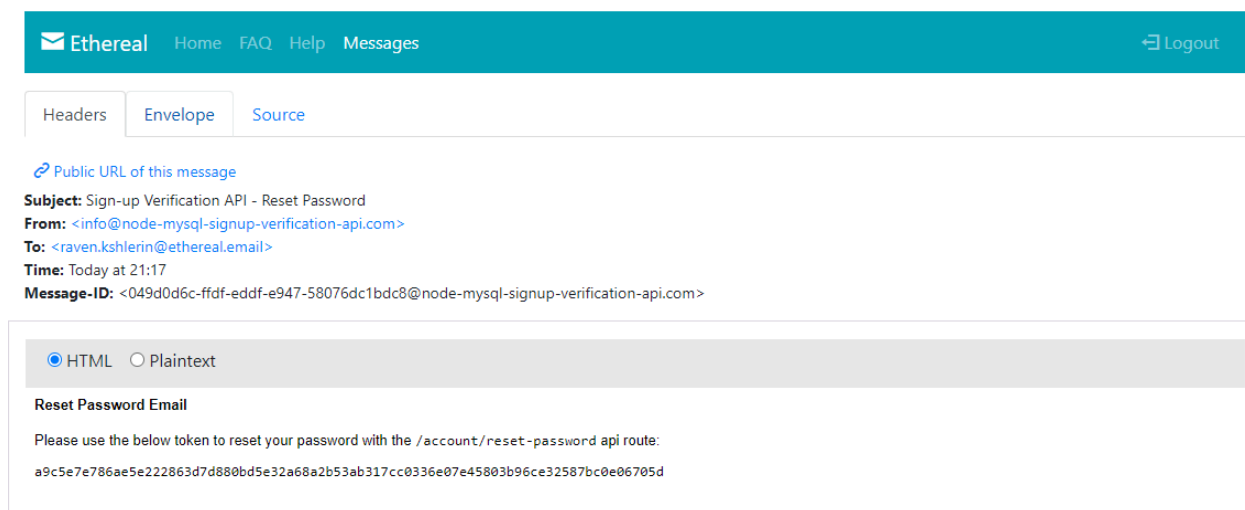


Follow these steps in ThunderClient if you forgot the password for an account:

- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/forgot-password`
- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object containing the email of the account with the forgotten password in the "Body" textarea, e.g:
- Click the "Send" button, you should receive a "200 OK" response with the message "Please check your email for password reset instructions" in the response body.



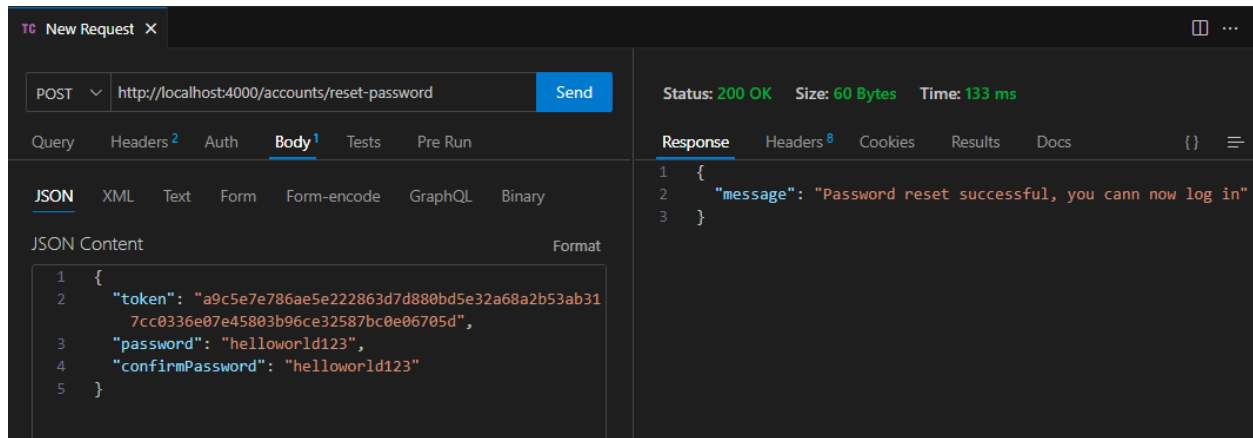
Received a verification email with the token to reset the password of the account.



To reset the password of an account with the api follow these steps:

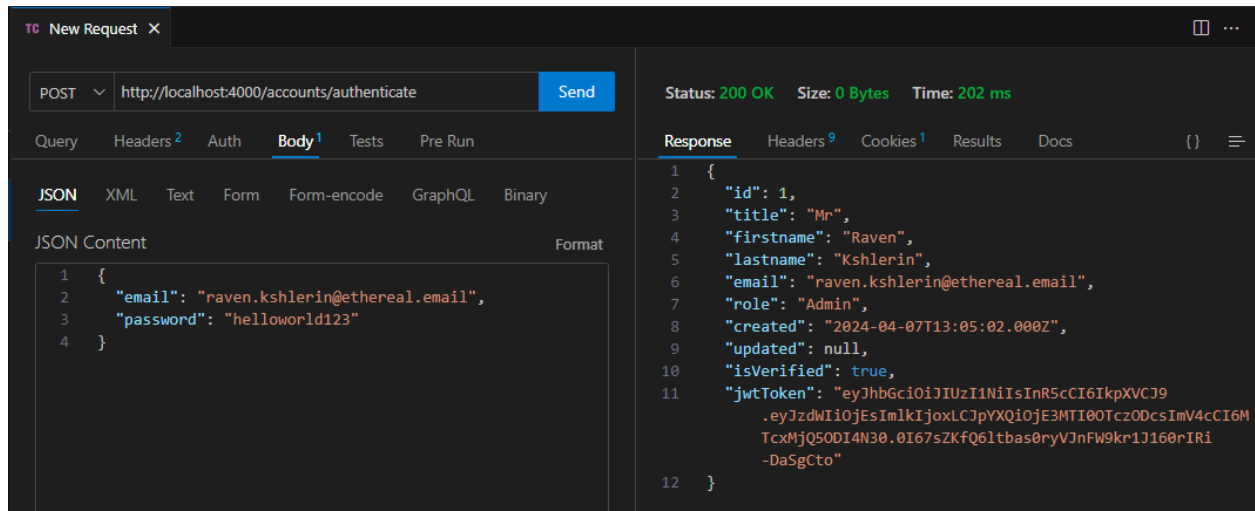
- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/reset-password`
- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object containing the password reset token received in the email from the forgot password step, along with a new password and matching confirmPassword, into the "Body" textarea, e.g:

- Click the "Send" button, you should receive a "200 OK" response with the message "Please check your email for password reset instructions" in the response body.

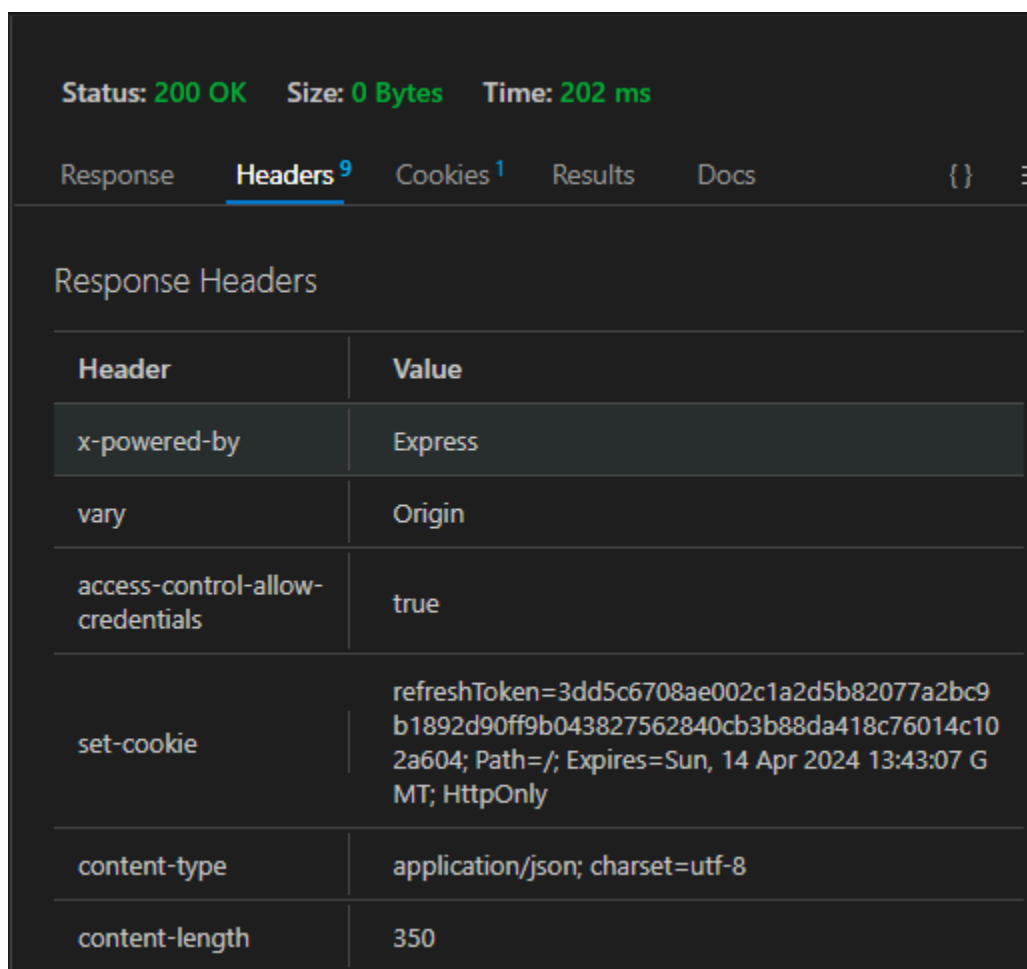


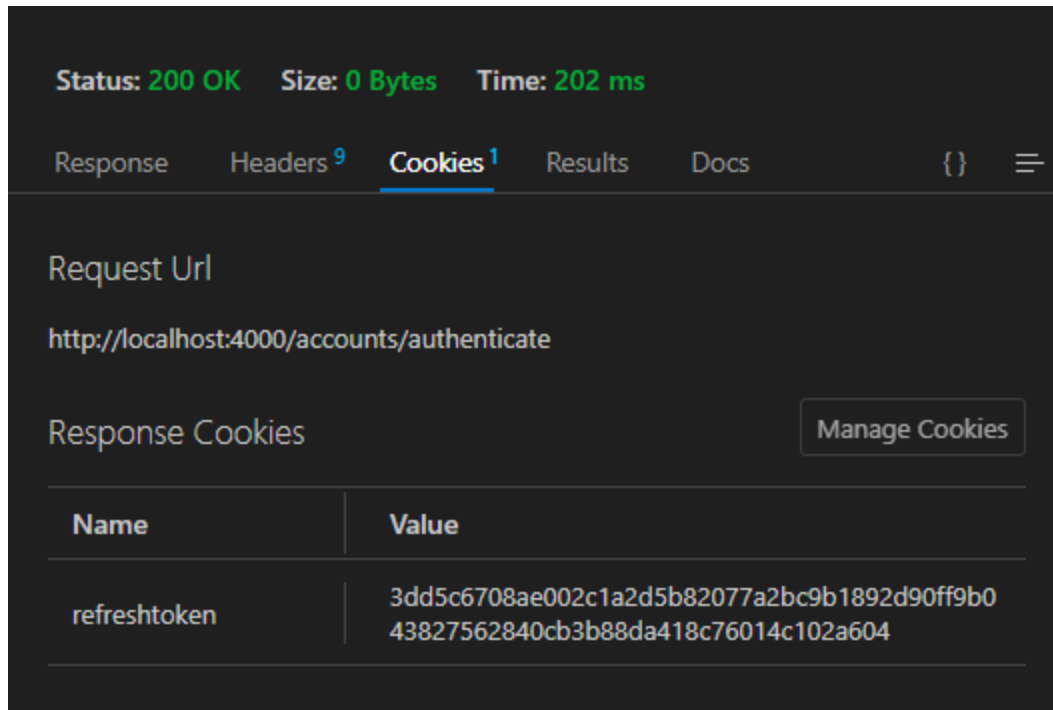
To authenticate an account with the api and get a JWT token follow these steps:

- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/authenticate`
- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object containing the account email and password in the "Body" textarea:
- Click the "Send" button, you should receive a "200 OK" response with a "password reset successful" message in the response body.
- Copy the JWT token value because we'll be using it in the next steps to make authenticated requests.



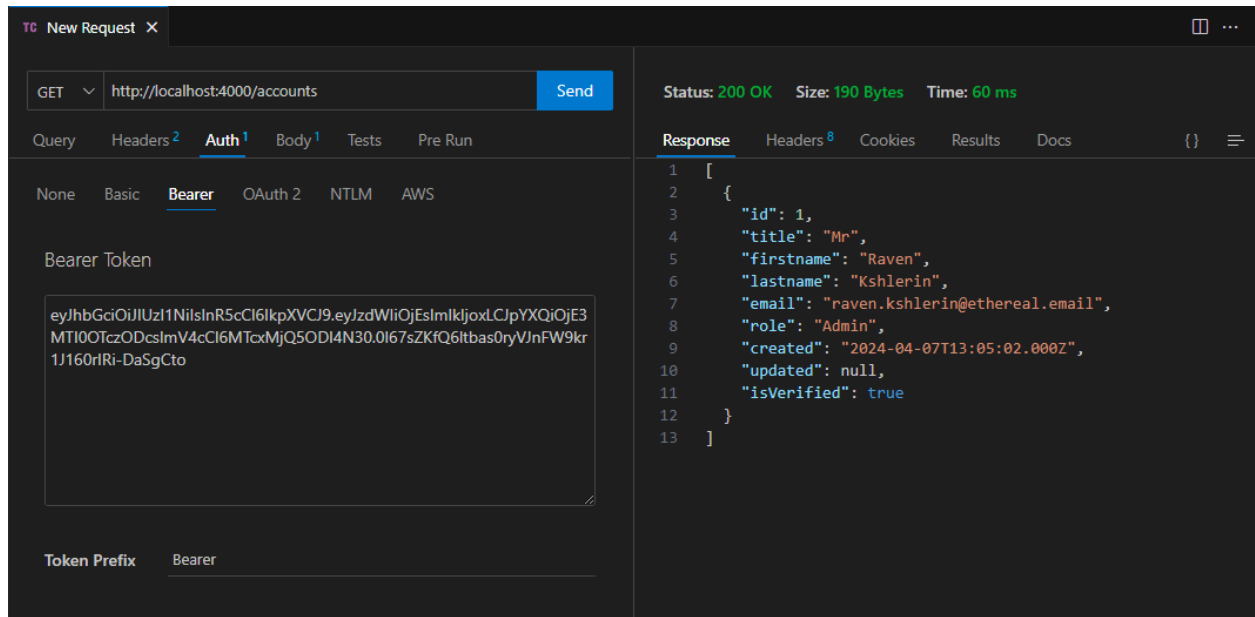
Response from the Headers and Cookies tab with the refresh token





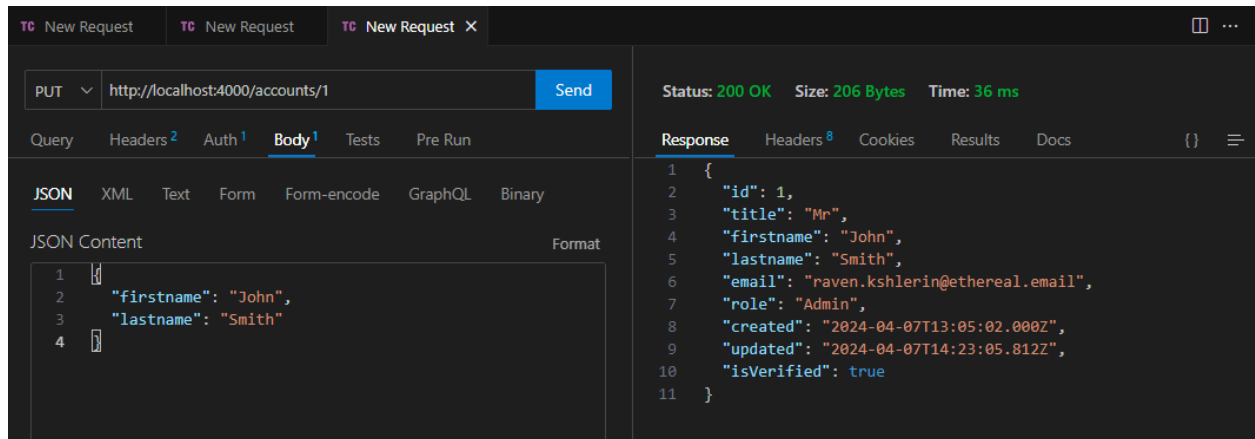
To get a list of all accounts from the Node boilerplate api follow these steps:

- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "GET" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the users route of your local API - `http://localhost:4000/accounts`
- Select the "Authorization" tab below the URL field, change the type to "Bearer Token" in the type dropdown selector, and paste the JWT token from the previous authenticate step into the "Token" field.
- Click the "Send" button, you should receive a "200 OK" response containing a JSON array with all of the account records in the system.



To update an account with the api follow these steps:

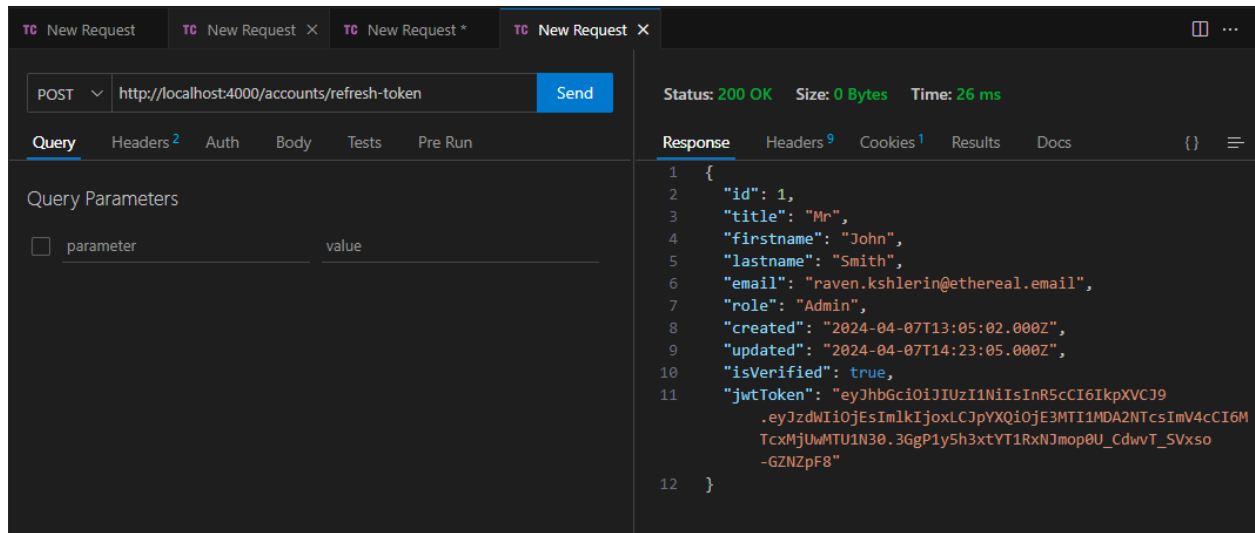
- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "PUT" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the /accounts/{id} route with the id of the account you want to update, e.g -
http://localhost:4000/accounts/1
- Select the "Authorization" tab below the URL field, change the type to "Bearer Token" in the type dropdown selector, and paste the JWT token from the previous authenticate step into the "Token" field.
- Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
- Enter a JSON object in the "Body" textarea containing the properties you want to update, for example to update the first and last names:
- Click the "Send" button, you should receive a "200 OK" response with the updated account details in the response body.



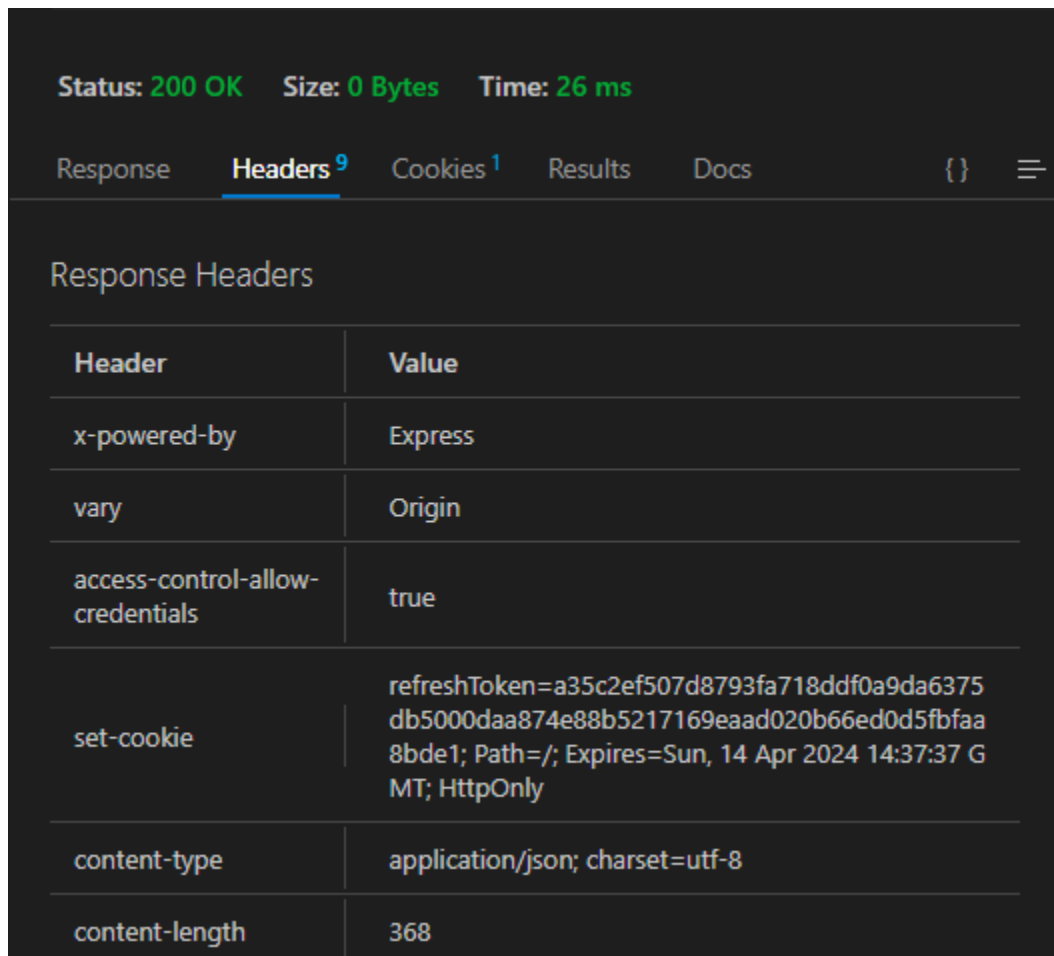
To use a refresh token cookie to get a new JWT token and a new refresh token follow these steps:

- Open a new request tab by clicking the plus (+) button at the end of the tabs.
- Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
- In the URL field enter the address to the refresh token route of your local API - `http://localhost:4000/accounts/refresh-token`
- Click the "Send" button, you should receive a "200 OK" response with the account details including a new JWT token in the response body and a new refresh token in the response cookies.
- Copy the JWT token value because we'll be using it in the next steps to make authenticated requests.

Response after the request is sent and the token has been refreshed



Response from the Headers and Cookies tab with the newly refreshed token



Status: 200 OK Size: 0 Bytes Time: 26 ms

Response

Headers⁹

Cookies¹

Results

Docs

{ } ≡

Request Url

http://localhost:4000/accounts/refresh-token

Response Cookies

Manage Cookies

Name	Value
refreshtoken	a35c2ef507d8793fa718ddf0a9da6375db5000daa874e88b5217169eaad020b66ed0d5fbfaa8bde1