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

How 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

Node + MySQL Boilerplate Overview

There are no users registered in the node.js boilerplate api by default, in order to authenticate you must first register and verify an account. The api sends a verification email after registration with a token to verify the account. Email SMTP settings must be set in the config.json file for email to work correctly, you can create a free test account in one click at https://ethereal.email/ and copy the options below the title *Nodemailer configuration*.

The first account registered is assigned to the `Admin` role and subsequent accounts are assigned to the regular `User` role. Admins have full access to CRUD routes for managing all accounts, while regular users can only modify their own account.

JWT authentication with refresh tokens

Authentication is implemented with JWT access tokens and refresh tokens. On successful authentication the boilerplate api returns a short lived JWT access token that expires after 15 minutes, and a refresh token that expires after 7 days in a HTTP Only cookie. The JWT is used for accessing secure routes on the api and the refresh token is used for generating new JWT access tokens when (or just before) they expire. HTTP Only cookies are used for increased security because they are not accessible to client-side javascript which prevents XSS (cross site scripting), and the refresh token can only be used to fetch a new JWT token from the `/accounts/refresh-token` route which prevents CSRF (cross site request forgery).

Refresh token rotation

As an added security measure in the `refreshToken()` method of the account service, each time a refresh token is used to generate a new JWT token, the refresh token is revoked and replaced by a new refresh token. This technique is known as *Refresh Token Rotation* and increases security by reducing the lifetime of refresh tokens, which makes it less likely that a compromised token will be valid (or valid for long). When a refresh token is rotated the new token is saved in the `replacedByToken` property of the revoked token to create an audit trail in the MySQL database.

Node.js + MySQL Boilerplate API Project Structure

This project is structured into feature folders (accounts) and non-feature / shared component folders (_helpers, _middleware). Shared component folders contain code that can be used by multiple features and other parts of the application, and are prefixed with an underscore `_` to group them together and make it easy to differentiate between feature folders and non-feature folders.

The boilerplate example only contains a single (accounts) feature at the moment, but could be easily extended with other features by copying the accounts folder and following the same pattern.

Project structure:

- _helpers
  - db.js
  - role.js
  - send-email.js
  - swagger.js
- _middleware
  - authorize.js
  - error-handler.js
  - validate-request.js
- accounts
  - account.model.js
  - refresh-token.model.js
  - account.service.js
  - accounts.controller.js
- config.json
- package.json
- server.js
- swagger.yaml

Helpers Folder

Path: /_helpers

The helpers folder contains all the bits and pieces that don't fit into other folders but don't justify having a folder of their own.

MySQL Database Wrapper

Path: /_helpers/db.js

The MySQL database wrapper connects to MySQL using Sequelize and the MySQL2 client, and exports an object containing all of the database model

objects in the application (currently only `Account` and `RefreshToken`). It provides an easy way to access any part of the database from a single point.

The `initialize()` function is executed once on api startup and performs the following actions:

- Connects to MySQL server using the `mysql2` db client and executes a query to create the database if it doesn't already exist.
- Connects to the database with the Sequelize ORM.
- Initializes the `Account` and `RefreshToken` models and attaches them to the exported `db` object.
- Defines the one-to-many relationship between accounts and refresh tokens and configures refresh tokens to be deleted when the account they belong to is deleted.
- Automatically creates tables in MySQL database if they don't exist by calling `await sequelize.sync()`. For more info on Sequelize model synchronization options see https://sequelize.org/master/manual/model-basics.html#model-synchronization.

```
const config = require('config.json');
const mysql = require('mysql2/promise');
const { Sequelize } = require('sequelize');

module.exports = db = {};

initialize();

```

```
async function initialize() {
    // create db if it doesn't already exist
    const { host, port, user, password, database } = config.database;
    const connection = await mysql.createConnection({ host, port, user, password });
    await connection.query(`CREATE DATABASE IF NOT EXISTS \`${database}\`;`);

    // connect to db
    const sequelize = new Sequelize(database, user, password, { dialect: 'mysql' });

    // init models and add them to the exported db object
    db.Account = require('../accounts/account.model')(sequelize);
    db.RefreshToken = require('../accounts/refresh-token.model')(sequelize);

    // define relationships
    db.Account.hasMany(db.RefreshToken, { onDelete: 'CASCADE' });
    db.RefreshToken.belongsTo(db.Account);

    // sync all models with database
    await sequelize.sync();
}
```

Role Object / Enum

Path: /_helpers/role.js

The role object defines all the roles in the example application. I created it to use like an enum to avoid passing roles around as strings, so instead of `'Admin'` and `'User'` we can use `Role.Admin` and `Role.User`.

```
module.exports = {
    Admin: 'Admin',
    User: 'User'
}
```

Send Email Helper

Path: /_helpers/send-email.js

The send email helper is a lightweight wrapper around the `nodemailer` module to simplify sending emails from anywhere in the application. It is used by the account service to send account verification and password reset emails.

```
const nodemailer = require('nodemailer');
const config = require('config.json');

module.exports = sendEmail;

async function sendEmail({ to, subject, html, from = config.emailFrom }) {
    const transporter = nodemailer.createTransport(config.smtpOptions);
    await transporter.sendMail({ from, to, subject, html });
}
```

Swagger API Docs Route Handler (`/api-docs`)

Path: /_helpers/swagger.js

The Swagger docs route handler uses the Swagger UI Express module to serve auto-generated Swagger UI documentation based on the swagger.yaml

file from the `/api-docs` path of the api. The route handler is bound to the `/api-docs` path in the main server.js file.

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

For more info on swagger-ui-express see

https://www.npmjs.com/package/swagger-ui-express.

Swagger UI | Node.js + MySQL - Boilerplate

localhost:4000/api-docs/#/

GSC Faculty Portal | GSC OLMS: Log in... | Java Basics - Java... | Institution Manage... | Wilson Portfolio | 2024

Swagger

Node.js Sign-up and Verification API 1.0.0 OAS3

Node.js + MySQL - API with email sign-up, verification, authentication and forgot password

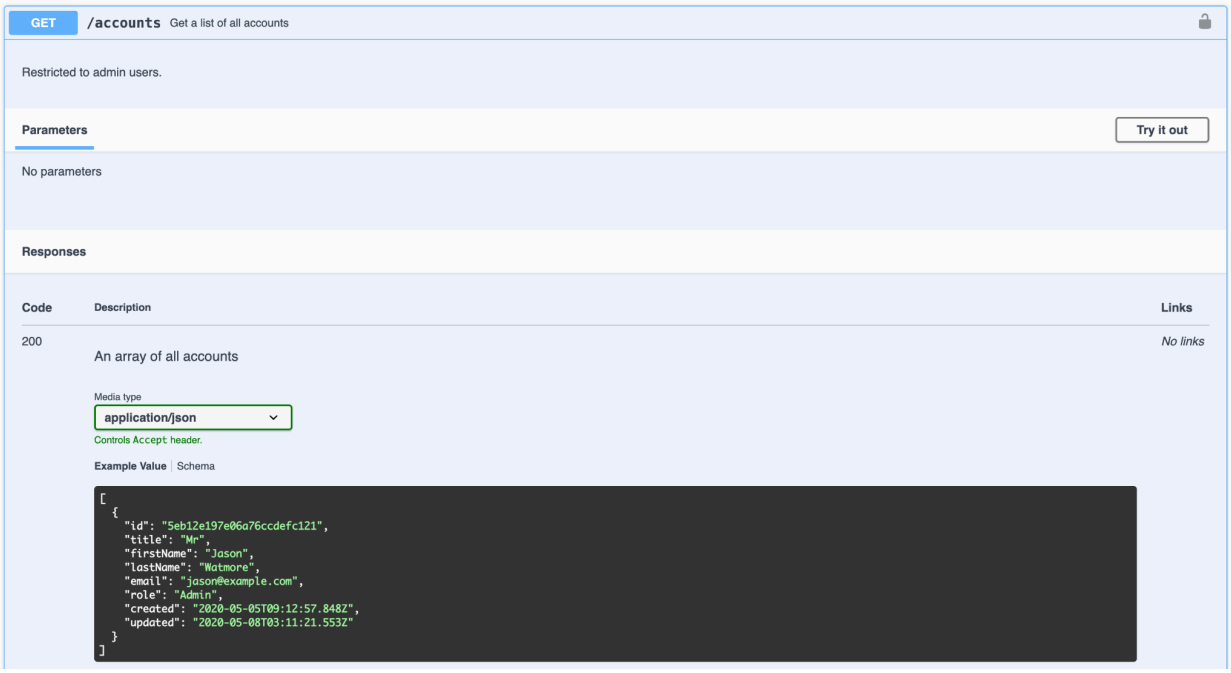
Servers

http://localhost:4000 - Local development server

Authorize

default

POST	/accounts/authenticate	Authenticate account credentials and return a JWT token and a cookie with a refresh token
POST	/accounts/refresh-token	Use a refresh token to generate a new JWT token and a new refresh token
POST	/accounts/revoke-token	Revoke a refresh token
POST	/accounts/register	Register a new user account and send a verification email
POST	/accounts/verify-email	Verify a new account with a verification token received by email after registration
POST	/accounts/forgot-password	Submit email address to reset the password on an account
POST	/accounts/validate-reset-token	Validate the reset password token received by email after submitting to the /accounts/forgot-password route
POST	/accounts/reset-password	Reset the password for an account
GET	/accounts	Get a list of all accounts
POST	/accounts	Create a new account
GET	/accounts/{id}	Get a single account by id
PUT	/accounts/{id}	Update an account
DELETE	/accounts/{id}	Delete an account



Express.js Middleware Folder

Path: /_middleware

The middleware folder contains Express.js middleware functions that can be used by different routes / features within the Node.js boilerplate api.

Authorize Middleware

Path: /_middleware/authorize.js

The authorized middleware can be added to any route to restrict access to the route to authenticated users with specified roles. If the `roles` parameter is omitted (i.e. `authorize()`) then the route will be accessible to all

authenticated users regardless of role. It is used by the accounts controller to restrict access to account CRUD routes and revoke token routes.

The authorize function returns an array containing two middleware functions:

- The first (`jwt({ ... })`) authenticates the request by validating the JWT access token in the "Authorization" header of the http request. On successful authentication a `user` object is attached to the `req` object that contains the data from the JWT token, which in this example includes the user id (`req.user.id`).
- The second authorizes the request by checking that the authenticated account still exists and is authorized to access the requested route based on its `role`. The second middleware function also attaches the `role` property and the `ownsToken` method to the `req.user` object so they can be accessed by controller functions.

If either authentication or authorization fails then a `401 Unauthorized` response is returned.

```
const jwt = require('express-jwt');
const { secret } = require('config.json');
const db = require('_helpers/db');

module.exports = authorize;

function authorize(roles = []) {
    // roles param can be a single role string (e.g. Role.User or 'User')
    // or an array of roles (e.g. [Role.Admin, Role.User] or ['Admin', 'User'])
    if (typeof roles === 'string') {
        roles = [roles];
    }

```

```
    return [
        // authenticate JWT token and attach user to request object (req.user)
        jwt({ secret, algorithms: ['HS256'] }),

        // authorize based on user role
        async (req, res, next) => {
            const account = await db.Account.findByPk(req.user.id);

            if (!account || (roles.length && !roles.includes(account.role))) {
                // account no longer exists or role not authorized
                return res.status(401).json({ message: 'Unauthorized' });
            }

            // authentication and authorization successful
            req.user.role = account.role;
            const refreshTokens = await account.getRefreshTokens();
            req.user.ownsToken = token => !!refreshTokens.find(x => x.token === token);
            next();
        }
    ];
}
```

Global Error Handler Middleware

Path: /_middleware/error-handler.js

The global error handler is used to catch all errors and remove the need for duplicated error handling code throughout the boilerplate application. It's configured as middleware in the main server.js file.

By convention errors of type `'string'` are treated as custom (app specific) errors, this simplifies the code for throwing custom errors since only a string needs to be thrown (e.g. `throw 'Invalid token'`). Further to this if a custom error ends with the words `'not found'` a 404 response code is returned, otherwise a standard 400 response is returned. See the account service for some examples of custom errors thrown by the api, errors are caught in the accounts controller for each route and passed to `next(err)` which passes them to this global error handler.

```
module.exports = errorHandler;

function errorHandler(err, req, res, next) {
    switch (true) {
        case typeof err === 'string':
            // custom application error
            const is404 = err.toLowerCase().endsWith('not found');
            const statusCode = is404 ? 404 : 400;
            return res.status(statusCode).json({ message: err });
        case err.name === 'UnauthorizedError':
            // jwt authentication error
            return res.status(401).json({ message: 'Unauthorized' });
        default:
            return res.status(500).json({ message: err.message });
    }
}
```

Validate Request Middleware

Path: /_middleware/validate-request.js

The validate request middleware function validates the body of a request against a Joi schema object.

It is used by schema middleware functions in controllers to validate the request against the schema for a specific route (e.g. `authenticateSchema` in the accounts controller).

```
module.exports = validateRequest;

function validateRequest(req, next, schema) {
    const options = {
        abortEarly: false, // include all errors
        allowUnknown: true, // ignore unknown props
        stripUnknown: true // remove unknown props
    };
    const { error, value } = schema.validate(req.body, options);
    if (error) {
        next(`Validation error: ${error.details.map(x => x.message).join(', ')}`);
    } else {
        req.body = value;
        next();
    }
}
```

Accounts Feature Folder

Path: /accounts

The accounts folder contains all code that is specific to the accounts feature of the node.js + mysql boilerplate api.

Sequelize Account Model

Path: /accounts/account.model.js

The account model 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 (create, read, update, delete) operations on accounts in MySQL, see the account service below for examples of it being used (via the `db` helper).

Fields with the type `DataTypes.VIRTUAL` are sequelize virtual fields that are not persisted in the database, they are convenience properties on the model that can include multiple field values (e.g. `isVerified`).

The `defaultScope` configures the model to exclude the password hash from query results by default. The `withHash` scope can be used to query accounts and include the password hash field in results.

The one-to-many relationship between accounts and refresh tokens is defined in the database wrapper.

```
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          // disable default timestamp fields (createdAt and updatedAt)
29          timestamps: false,
30          defaultScope: {
31              // exclude password hash by default
32              attributes: { exclude: ['passwordHash'] }
33          },
34          scopes: {
35              // include hash with this scope
36              withHash: { attributes: {}, }
37          }
38      };
39
40      return sequelize.define('account', attributes, options);
41  }
```

Sequelize Refresh Token Model

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

The refresh token model 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 (create, read, update, delete) operations on refresh tokens in MySQL, see the account service below for examples of it being used (via the `db` helper).

The `DataTypes.VIRTUAL` properties are convenience properties available on the sequelize model that don't get persisted to the MySQL database.

The one-to-many relationship between accounts and refresh tokens is defined in the database wrapper.

```
const { DataTypes } = require('sequelize');

module.exports = model;

```

```
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          // disable default timestamp fields (createdAt and updatedAt)
26          timestamps: false
27      };
28
29      return sequelize.define('refreshToken', attributes, options);
30  }
```

Account Service

Path: /accounts/account.service.js

The account service contains the core business logic for account sign up & verification, authentication with JWT & refresh tokens, forgot password & reset password functionality, as well as CRUD methods for managing account data. The service encapsulates all interaction with the sequelize account models and exposes a simple set of methods which are used by the accounts controller.

The top of the file contains the exported service object with just the method names to make it easy to see all the methods at a glance, the rest of the file contains the implementation functions for each service method, followed by local helper functions.

```
const config = require('config.json');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const crypto = require("crypto");
const { Op } = require('sequelize');
const sendEmail = require('_helpers/send-email');
const db = require('_helpers/db');
const Role = require('_helpers/role');

module.exports = {
    authenticate,
    refreshToken,
    revokeToken,
    register,
    verifyEmail,
    forgotPassword,
    validateResetToken,
    resetPassword,
    getAll,
    getById,
    create,
    update,
    delete: _delete
};
```

```
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
33      // authentication successful so generate jwt and refresh tokens
34      const jwtToken = generateJwtToken(account);
35      const refreshToken = generateRefreshToken(account, ipAddress);
36
37      // save refresh token
38      await refreshToken.save();
39
40      // return basic details and tokens
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      // replace old refresh token with a new one and save
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      // generate new jwt
61      const jwtToken = generateJwtToken(account);
62
63      // return basic details and tokens
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      // revoke token and save
75      refreshToken.revoked = Date.now();
76      refreshToken.revokedByIp = ipAddress;
77      await refreshToken.save();
78  }
```

```
79
80  async function register(params, origin) {
81      // validate
82      if (await db.Account.findOne({ where: { email: params.email } })) {
83          // send already registered error in email to prevent account enumeration
84          return await sendAlreadyRegisteredEmail(params.email, origin);
85      }
86
87      // create account object
88      const account = new db.Account(params);
89
90      // first registered account is an admin
91      const isFirstAccount = (await db.Account.count()) === 0;
92      account.role = isFirstAccount ? Role.Admin : Role.User;
93      account.verificationToken = randomTokenString();
94
95      // hash password
96      account.passwordHash = await hash(params.password);
97
98      // save account
99      await account.save();
100
101     // send email
102     await sendVerificationEmail(account, origin);
103 }
```

```
104
105 async function verifyEmail({ token }) {
106     const account = await db.Account.findOne({ where: { verificationToken: token } });
107
108     if (!account) throw 'Verification failed';
109
110     account.verified = Date.now();
111     account.verificationToken = null;
112     await account.save();
113 }
114
115 async function forgotPassword({ email }, origin) {
116     const account = await db.Account.findOne({ where: { email } });
117
118     // always return ok response to prevent email enumeration
119     if (!account) return;
120
121     // create reset token that expires after 24 hours
122     account.resetToken = randomTokenString();
123     account.resetTokenExpires = new Date(Date.now() + 24*60*60*1000);
124     await account.save();
125
126     // send email
127     await sendPasswordResetEmail(account, origin);
128 }
```

```
129
130 async function validateResetToken({ token }) {
131     const account = await db.Account.findOne({
132         where: {
133             resetToken: token,
134             resetTokenExpires: { [Op.gt]: Date.now() }
135         }
136     });
137
138     if (!account) throw 'Invalid token';
139
140     return account;
141 }
142
143 async function resetPassword({ token, password }) {
144     const account = await validateResetToken({ token });
145
146     // update password and remove reset token
147     account.passwordHash = await hash(password);
148     account.passwordReset = Date.now();
149     account.resetToken = null;
150     await account.save();
151 }
```

```
152
153   async function getAll() {
154       const accounts = await db.Account.findAll();
155       return accounts.map(x => basicDetails(x));
156   }
157
158   async function getById(id) {
159       const account = await getAccount(id);
160       return basicDetails(account);
161   }
162
163   async function create(params) {
164       // validate
165       if (await db.Account.findOne({ where: { email: params.email } })) {
166           throw 'Email "' + params.email + '" is already registered';
167       }
168
169       const account = new db.Account(params);
170       account.verified = Date.now();
171
172       // hash password
173       account.passwordHash = await hash(params.password);
174
175       // save account
176       await account.save();
177
178       return basicDetails(account);
179   }
```

```
180
181   async function update(id, params) {
182       const account = await getAccount(id);
183
184       // validate (if email was changed)
185       if (params.email && account.email !== params.email && await db.Account.findOne({ where: { email: params.email } })) {
186           throw 'Email "' + params.email + '" is already taken';
187       }
188
189       // hash password if it was entered
190       if (params.password) {
191           params.passwordHash = await hash(params.password);
192       }
193
194       // copy params to account and save
195       Object.assign(account, params);
196       account.updated = Date.now();
197       await account.save();
198
199       return basicDetails(account);
200   }
201
202   async function _delete(id) {
203       const account = await getAccount(id);
204       await account.destroy();
205   }
```

```
206
207  // helper functions
208
209  async function getAccount(id) {
210      const account = await db.Account.findByPk(id);
211      if (!account) throw 'Account not found';
212      return account;
213  }
214
215  async function getRefreshToken(token) {
216      const refreshToken = await db.RefreshToken.findOne({ where: { token } });
217      if (!refreshToken || !refreshToken.isActive) throw 'Invalid token';
218      return refreshToken;
219  }
220
221  async function hash(password) {
222      return await bcrypt.hash(password, 10);
223  }
224
225  function generateJwtToken(account) {
226      // create a jwt token containing the account id that expires in 15 minutes
227      return jwt.sign({ sub: account.id, id: account.id }, config.secret, { expiresIn: '15m' });
228  }
```

```
229
230  function generateRefreshToken(account, ipAddress) {
231      // create a refresh token that expires in 7 days
232      return new db.RefreshToken({
233          accountId: account.id,
234          token: randomTokenString(),
235          expires: new Date(Date.now() + 7*24*60*60*1000),
236          createdByIp: ipAddress
237      });
238  }
239
240  function randomTokenString() {
241      return crypto.randomBytes(40).toString('hex');
242  }
243
244  function basicDetails(account) {
245      const { id, title, firstName, lastName, email, role, created, updated, isVerified } = account;
246      return { id, title, firstName, lastName, email, role, created, updated, isVerified };
247  }
```

```
248
249 async function sendVerificationEmail(account, origin) {
250     let message;
251     if (origin) {
252         const verifyUrl = `${origin}/account/verify-email?token=${account.verificationToken}`;
253         message = `<p>Please click the below link to verify your email address:</p>
254                    <p><a href="${verifyUrl}">${verifyUrl}</a></p>`;
255     } else {
256         message = `<p>Please use the below token to verify your email address with the <code>/account/verify-email</code> api route:</p>
257                    <p><code>${account.verificationToken}</code></p>`;
258     }
259
260     await sendEmail({
261         to: account.email,
262         subject: 'Sign-up Verification API - Verify Email',
263         html: `<h4>Verify Email</h4>
264                <p>Thanks for registering!</p>
265                ${message}`
266     });
267 }
```

```
268
269 async function sendAlreadyRegisteredEmail(email, origin) {
270     let message;
271     if (origin) {
272         message = `
273         <p>If you don't know your password please visit the <a href="${origin}/account/forgot-password">forgot password</a> page.</p>`;
274     } else {
275         message = `
276         <p>If you don't know your password you can reset it via the <code>/account/forgot-password</code> api route.</p>`;
277     }
278
279     await sendEmail({
280         to: email,
281         subject: 'Sign-up Verification API - Email Already Registered',
282         html: `<h4>Email Already Registered</h4>
283                <p>Your email <strong>${email}</strong> is already registered.</p>
284                ${message}`
285     });
286 }
```

```
287
288 async function sendPasswordResetEmail(account, origin) {
289     let message;
290     if (origin) {
291         const resetUrl = `${origin}/account/reset-password?token=${account.resetToken}`;
292         message = `<p>Please click the below link to reset your password, the link will be valid for 1 day:</p>
293                    <p><a href="${resetUrl}">${resetUrl}</a></p>`;
294     } else {
295         message = `<p>Please use the below token to reset your password with the <code>/account/reset-password</code> api route:</p>
296                    <p><code>${account.resetToken}</code></p>`;
297     }
298
299     await sendEmail({
300         to: account.email,
301         subject: 'Sign-up Verification API - Reset Password',
302         html: `<h4>Reset Password Email</h4>
303                ${message}`
304     });
305 }
```

Express.js Accounts Controller

Path: /accounts/accounts.controller.js

The accounts controller 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.

Routes that require authorization include the middleware function `authorize()` and optionally specify a role (e.g. `authorize(Role.Admin)`, if a role is specified then the route is restricted to users in that role, otherwise the route is restricted to all authenticated users regardless of role. The auth logic is located in the authorize middleware.

The route functions `revokeToken`, `getById`, `update` and `_delete` include an extra custom authorization check to prevent non-admin users from accessing accounts other than their own. So regular user accounts (`Role.User`) have CRUD access to their own account but not to others, and admin accounts (`Role.Admin`) have full CRUD access to all accounts.

Routes that require schema validation include a middleware function with the naming convention `<route>Schema` (e.g. `authenticateSchema`). Each schema validation function defines a schema for the request body using the Joi library and calls `validateRequest(req, next, schema)` to ensure the request body is valid. If validation succeeds the request continues to the next middleware function (the route function), otherwise an error is returned with

details of why validation failed. For more info about Joi schema validation see https://www.npmjs.com/package/joi.

Express is the web server used by the boilerplate api, it's one of the most popular web application frameworks for Node.js. For more info see https://expressjs.com/.

```
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   // routes
10  router.post('/authenticate', authenticateSchema, authenticate);
11  router.post('/refresh-token', refreshToken);
12  router.post('/revoke-token', authorize(), revokeTokenSchema, revokeToken);
13  router.post('/register', registerSchema, register);
14  router.post('/verify-email', verifyEmailSchema, verifyEmail);
15  router.post('/forgot-password', forgotPasswordSchema, forgotPassword);
16  router.post('/validate-reset-token', validateResetTokenSchema, validateResetToken);
17  router.post('/reset-password', resetPasswordSchema, resetPassword);
18  router.get('/', authorize(Role.Admin), getAll);
19  router.get('/:id', authorize(), getById);
20  router.post('/', authorize(Role.Admin), createSchema, create);
21  router.put('/:id', authorize(), updateSchema, update);
22  router.delete('/:id', authorize(), _delete);
23
24  module.exports = router;
25
```

```
26  function authenticateSchema(req, res, next) {
27      const schema = Joi.object({
28          email: Joi.string().required(),
29          password: Joi.string().required()
30      });
31      validateRequest(req, next, schema);
32  }
33
34  function authenticate(req, res, next) {
35      const { email, password } = req.body;
36      const ipAddress = req.ip;
37      accountService.authenticate({ email, password, ipAddress })
38          .then(({ refreshToken, ...account }) => {
39              setTokenCookie(res, refreshToken);
40              res.json(account);
41          })
42          .catch(next);
43  }
44
45  function refreshToken(req, res, next) {
46      const token = req.cookies.refreshToken;
47      const ipAddress = req.ip;
48      accountService.refreshToken({ token, ipAddress })
49          .then(({ refreshToken, ...account }) => {
50              setTokenCookie(res, refreshToken);
51              res.json(account);
52          })
53          .catch(next);
54  }
```

```
56  function revokeTokenSchema(req, res, next) {
57      const schema = Joi.object({
58          token: Joi.string().empty('')
59      });
60      validateRequest(req, next, schema);
61  }
62
63  function revokeToken(req, res, next) {
64      // accept token from request body or cookie
65      const token = req.body.token || req.cookies.refreshToken;
66      const ipAddress = req.ip;
67
68      if (!token) return res.status(400).json({ message: 'Token is required' });
69
70      // users can revoke their own tokens and admins can revoke any tokens
71      if (!req.user.ownsToken(token) && req.user.role !== Role.Admin) {
72          return res.status(401).json({ message: 'Unauthorized' });
73      }
74
75      accountService.revokeToken({ token, ipAddress })
76          .then(() => res.json({ message: 'Token revoked' }))
77          .catch(next);
78  }
79
```

```
80  function registerSchema(req, res, next) {
81      const schema = Joi.object({
82          title: Joi.string().required(),
83          firstName: Joi.string().required(),
84          lastName: Joi.string().required(),
85          email: Joi.string().email().required(),
86          password: Joi.string().min(6).required(),
87          confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
88          acceptTerms: Joi.boolean().valid(true).required()
89      });
90      validateRequest(req, next, schema);
91  }
92
93  function register(req, res, next) {
94      accountService.register(req.body, req.get('origin'))
95          .then(() => res.json({ message: 'Registration successful, please check your email for verification instructions' }))
96          .catch(next);
97  }
98
99  function verifyEmailSchema(req, res, next) {
100     const schema = Joi.object({
101         token: Joi.string().required()
102     });
103     validateRequest(req, next, schema);
104 }
```

```
function verifyEmail(req, res, next) {
    accountService.verifyEmail(req.body)
        .then(() => res.json({ message: 'Verification successful, you can now login' }))
        .catch(next);
}

function forgotPasswordSchema(req, res, next) {
    const schema = Joi.object({
        email: Joi.string().email().required()
    });
    validateRequest(req, next, schema);
}

function forgotPassword(req, res, next) {
    accountService.forgotPassword(req.body, req.get('origin'))
        .then(() => res.json({ message: 'Please check your email for password reset instructions' }))
        .catch(next);
}

function validateResetTokenSchema(req, res, next) {
    const schema = Joi.object({
        token: Joi.string().required()
    });
    validateRequest(req, next, schema);
}
```

```
function validateResetToken(req, res, next) {
    accountService.validateResetToken(req.body)
        .then(() => res.json({ message: 'Token is valid' }))
        .catch(next);
}

function resetPasswordSchema(req, res, next) {
    const schema = Joi.object({
        token: Joi.string().required(),
        password: Joi.string().min(6).required(),
        confirmPassword: Joi.string().valid(Joi.ref('password')).required()
    });
    validateRequest(req, next, schema);
}

function resetPassword(req, res, next) {
    accountService.resetPassword(req.body)
        .then(() => res.json({ message: 'Password reset successful, you can now login' }))
        .catch(next);
}

function getAll(req, res, next) {
    accountService.getAll()
        .then(accounts => res.json(accounts))
        .catch(next);
}
```

```
159 function getById(req, res, next) {
160     // users can get their own account and admins can get any account
161     if (Number(req.params.id) !== req.user.id && req.user.role !== Role.Admin) {
162         return res.status(401).json({ message: 'Unauthorized' });
163     }
164
165     accountService.getById(req.params.id)
166         .then(account => account ? res.json(account) : res.sendStatus(404))
167         .catch(next);
168 }
169
170 function createSchema(req, res, next) {
171     const schema = Joi.object({
172         title: Joi.string().required(),
173         firstName: Joi.string().required(),
174         lastName: Joi.string().required(),
175         email: Joi.string().email().required(),
176         password: Joi.string().min(6).required(),
177         confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
178         role: Joi.string().valid(Role.Admin, Role.User).required()
179     });
180     validateRequest(req, next, schema);
181 }
182
183 function create(req, res, next) {
184     accountService.create(req.body)
185         .then(account => res.json(account))
186         .catch(next);
187 }
```

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

```
219 function _delete(req, res, next) {
220     // users can delete their own account and admins can delete any account
221     if (Number(req.params.id) !== req.user.id && req.user.role !== Role.Admin) {
222         return res.status(401).json({ message: 'Unauthorized' });
223     }
224
225     accountService.delete(req.params.id)
226         .then(() => res.json({ message: 'Account deleted successfully' }))
227         .catch(next);
228 }
229
230 // helper functions
231
232 function setTokenCookie(res, token) {
233     // create cookie with refresh token that expires in 7 days
234     const cookieOptions = {
235         httpOnly: true,
236         expires: new Date(Date.now() + 7*24*60*60*1000)
237     };
238     res.cookie('refreshToken', token, cookieOptions);
239 }
```

Api Config

Path: /config.json

The api config file 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.

Configure SMTP settings for email within the `smtpOptions` property. For testing you can create a free account in one click at https://ethereal.email/ and copy the options below the title *Nodemailer configuration*.

IMPORTANT: The `secret` property is used to sign and verify JWT tokens for authentication, change it with your own random string to ensure nobody else can generate a JWT with the same secret to gain unauthorized access to your api. A quick and easy way is join a couple of GUIDs together to make a long random string (e.g. from https://www.guidgenerator.com/).

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

Package.json

Path: /package.json

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

The `scripts` section contains scripts that are executed by running the command `npm run <script name>`, the start script can also be run with the shortcut command `npm start`.

The `start` script starts the boilerplate api normally using `node`, and the `start:dev` script starts the api in development mode using `nodemon` which automatically restarts the server when a file is changed.

For more info see https://docs.npmjs.com/files/package.json.

```
{
    "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",
    "scripts": {
        "start": "node ./server.js",
        "start:dev": "nodemon ./server.js"
    },
    "dependencies": {
        "bcryptjs": "^2.4.3",
        "body-parser": "^1.19.0",
        "cookie-parser": "^1.4.5",
        "cors": "^2.8.5",
        "express-jwt": "^6.0.0",
        "express": "^4.17.1",
        "joi": "^17.2.1",
        "jsonwebtoken": "^8.5.1",
        "mysql2": "^2.1.0",
        "nodemailer": "^6.4.11",
        "rootpath": "^0.1.2",
        "sequelize": "^6.3.4",
        "swagger-ui-express": "^4.1.4",
        "yamljs": "^0.3.0"
    },
    "devDependencies": {
        "nodemon": "^2.0.3"
    }
}
```

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   const express = require('express');
3   const app = express();
4   const bodyParser = require('body-parser');
5   const cookieParser = require('cookie-parser');
6   const cors = require('cors');
7   const errorHandler = require('_middleware/error-handler');
8
9   app.use(bodyParser.urlencoded({ extended: false }));
10  app.use(bodyParser.json());
11  app.use(cookieParser());
12
13  // allow cors requests from any origin and with credentials
14  app.use(cors({ origin: (origin, callback) => callback(null, true), credentials: true }));
15
16  // api routes
17  app.use('/accounts', require('./accounts/accounts.controller'));
18
19  // swagger docs route
20  app.use('/api-docs', require('_helpers/swagger'));
21
22  // global error handler
23  app.use(errorHandler);
24
25  // start server
26  const port = process.env.NODE_ENV === 'production' ? (process.env.PORT || 80) : 4000;
27  app.listen(port, () => console.log('Server listening on port ' + port));
```

Swagger API Documentation

Path: /swagger.yaml

The Swagger YAML file describes the entire Node.js Boilerplate API using the *OpenAPI Specification* format, it includes descriptions of all routes and HTTP methods on each route, request and response schemas, path parameters, and authentication methods.

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 https://editor.swagger.io/.

File: swagger.yaml

Run the Node + MySQL Boilerplate API Locally

1. Install NodeJS and NPM from https://nodejs.org/en/download/.
2. Install MySQL Community Server from https://dev.mysql.com/downloads/mysql/ and ensure it is started. Installation instructions are available at https://dev.mysql.com/doc/refman/8.0/en/installing.html.
3. Project source code
4. Install all required npm packages by running `npm install` or `npm i` from the command line in the project root folder (where the package.json is located).
5. Configure SMTP settings for email within the `smtpOptions` property in the `/src/config.json` file. For testing you can create a free account in one click at https://ethereal.email/ and copy the options below the title *Nodemailer configuration*.
6. 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`.

Before running in production

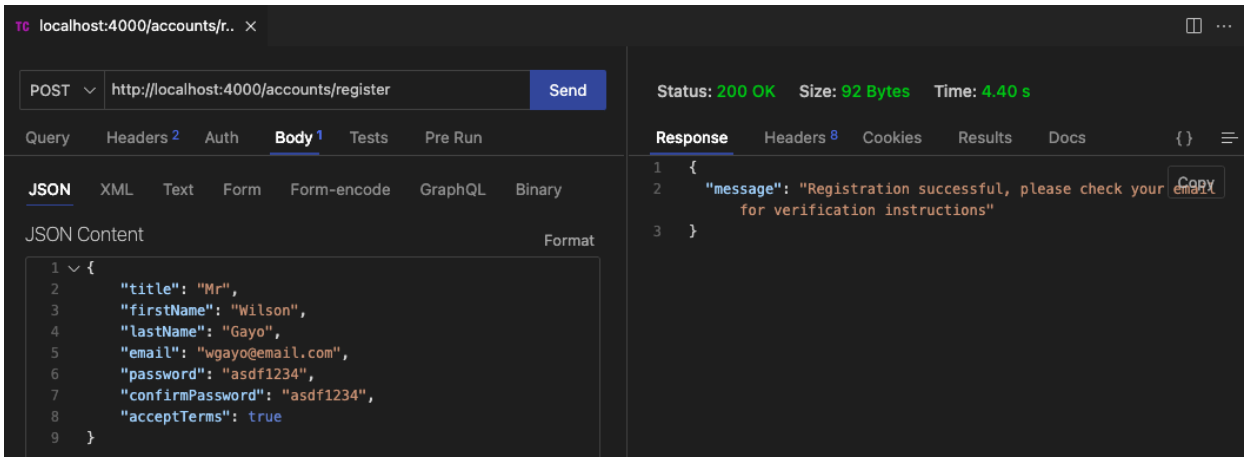
Before running in production also make sure that you update the `secret` property in the config.json file, it is used to sign and verify JWT tokens for authentication, change it to a random string to ensure nobody else can generate a JWT with the same secret and gain unauthorized access to your api. A quick and easy way is join a couple of GUIDs together to make a long random string (e.g. from https://www.guidgenerator.com/).

Test the Node.js Boilerplate API

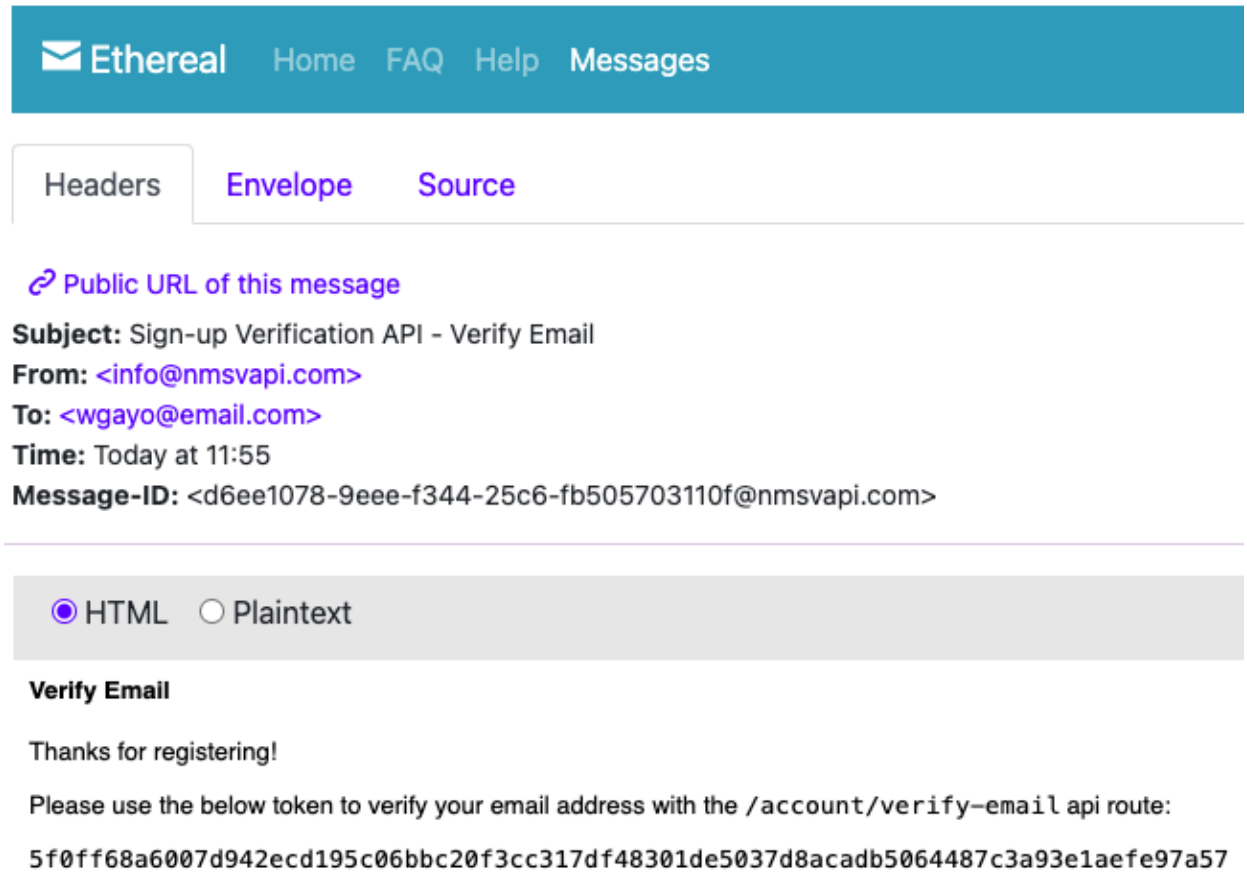
To register a new account with the Node.js boilerplate api follow these steps:

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

6. Click the "Send" button, you should receive a "200 OK" response with a "registration successful" message in the response body.



And this is a screenshot of the verification email received with the token to verify the account:

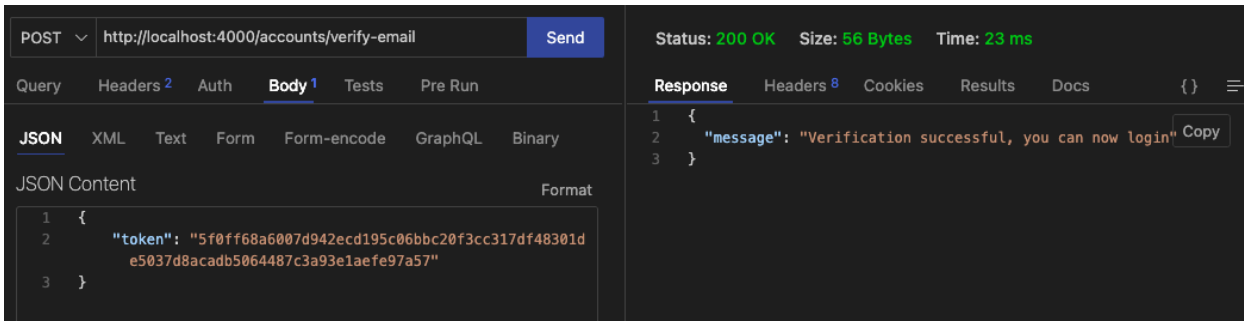


How to verify an account with Postman / ThunderCLient

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

1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "POST" with the dropdown selector on the left of the URL input field.

3. In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/verify-email`
4. Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
5. Enter a JSON object containing the token received in the verification email (in the previous step) in the "Body" textarea, e.g:
6. Click the "Send" button, you should receive a "200 OK" response with a "verification successful" message in the response body.



How to access an account if you forgot the password with Postman / ThunderCLient

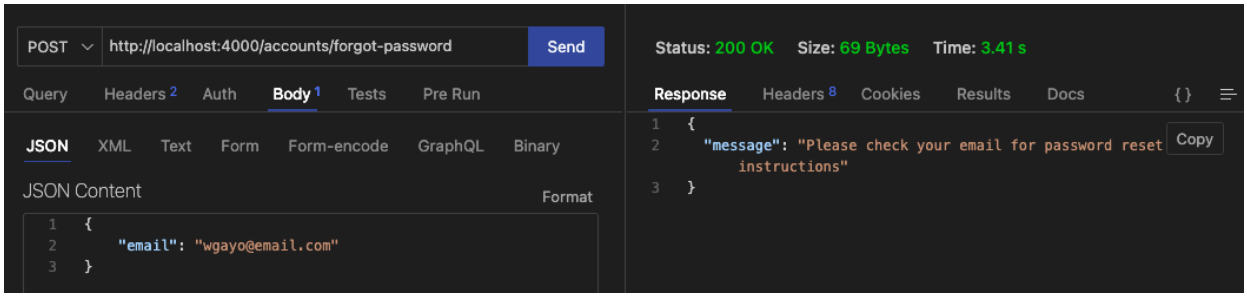
To re-enable access to an account with a forgotten password you need to submit the email address of the account to the `/account/forgot-password` route, the route will then send a token to the email which will allow you to reset the password of the account in the next step.

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

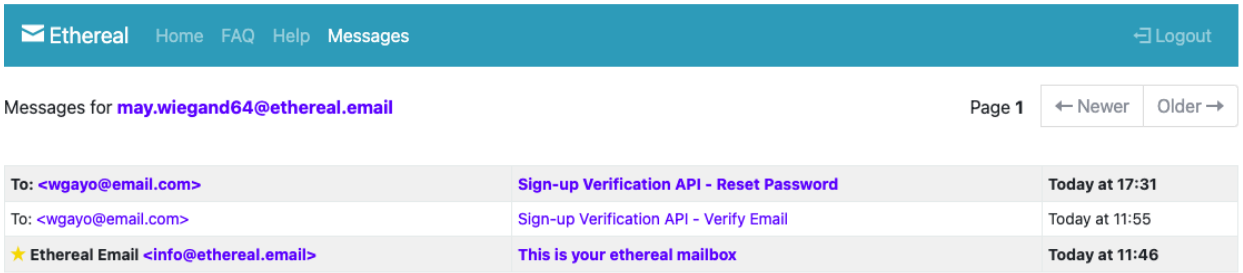
1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
3. In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/forgot-password`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

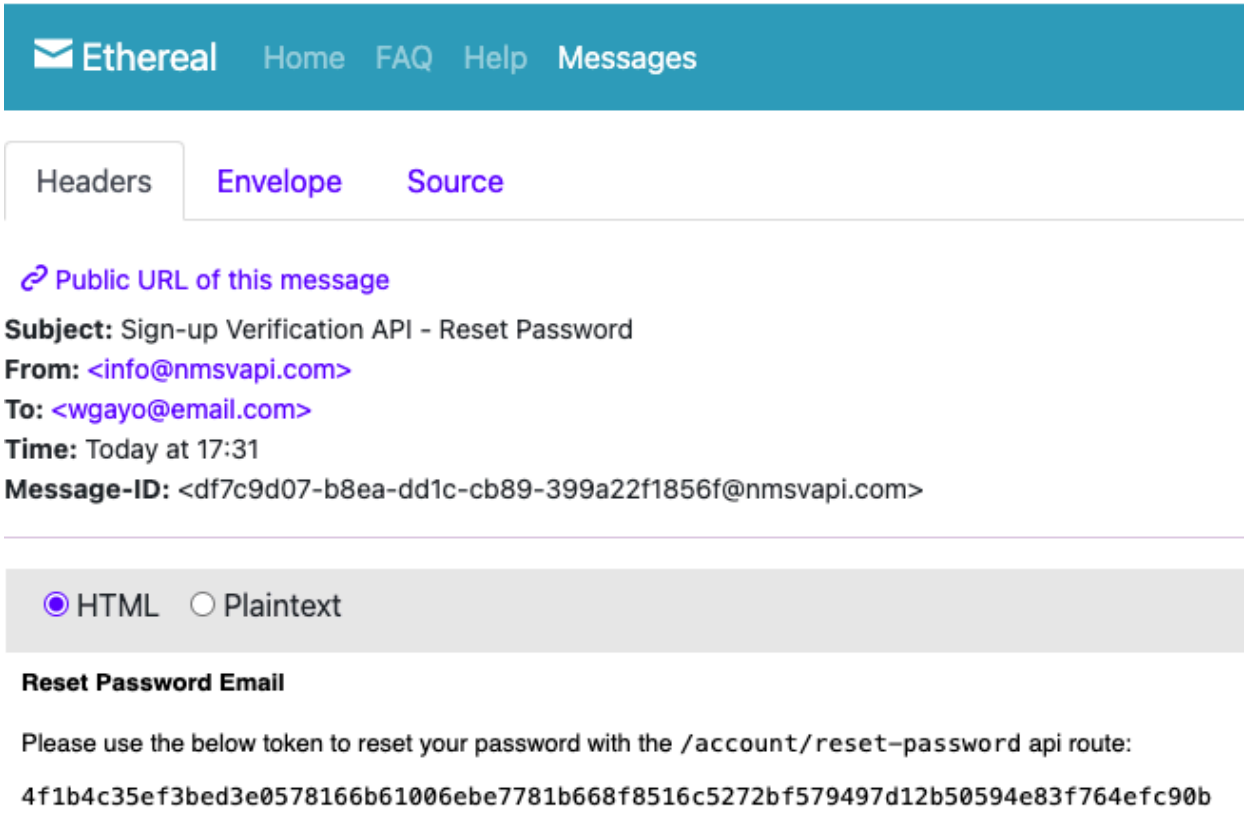
```
1  {
2      "email": "wgayo@email.com"
3  }
```

6. 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.



And this is a screenshot of the email received with the token to reset the password of the account:





How to reset the password of an account with Postman / ThunderCLient

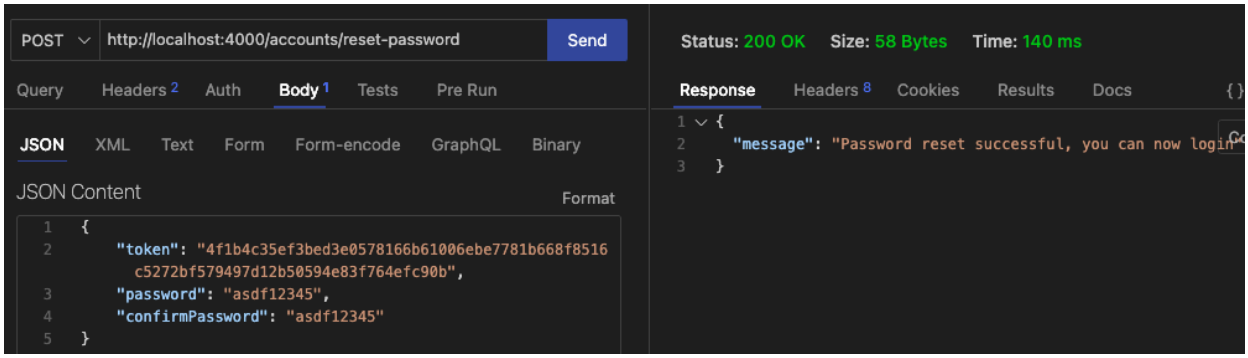
To reset the password of an account with the api follow these steps:

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

4. Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
5. 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:

```
{
    "token": "REPLACE THIS WITH YOUR TOKEN",
    "password": "new-super-secret-password",
    "confirmPassword": "new-super-secret-password"
}
```

6. Click the "Send" button, you should receive a "200 OK" response with a "password reset successful" message in the response body.



How to authenticate with Postman / ThunderCLient

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

1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "POST" with the dropdown selector on the left of the URL input field.
3. In the URL field enter the address to the authenticate route of your local API - `http://localhost:4000/accounts/authenticate`
4. Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
5. Enter a JSON object containing the account email and password in the "Body" textarea:

```
{
  "email": "wgayo@email.com",
  "password": "asdf12345"
}
```

6. Click the "Send" button, you should receive a "200 OK" response with the user details including a JWT token in the response body and a refresh token in the response cookies.
7. Copy the JWT token value because we'll be using it in the next steps to make authenticated requests.

POST http://localhost:4000/accounts/authenticate Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
{
  "email": "wgayo@email.com",
  "password": "asdf12345"
}
```

Status: 200 OK Size: 0 Bytes Time: 207 ms

Response Headers 9 Cookies 1 Results Docs { } Copy


```
{
  "id": 1,
  "title": "Mr",
  "firstName": "Wilson",
  "lastName": "Gayo",
  "email": "wgayo@email.com",
  "role": "Admin",
  "created": "2024-03-21T03:55:40.000Z",
  "updated": null,
  "isVerified": true,
  "jwtToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
      .eyJzdWIiOjEsImlkIjoxLCJpYXQiOjE3MTEwNjg0ODEsImV4cCI6MTcxMTA2OTM4MX0
      .bKaboDo1W3gA5CKgMdyQvMJaviotu5Xifc_4nM587U8"
}
```


Response Chart

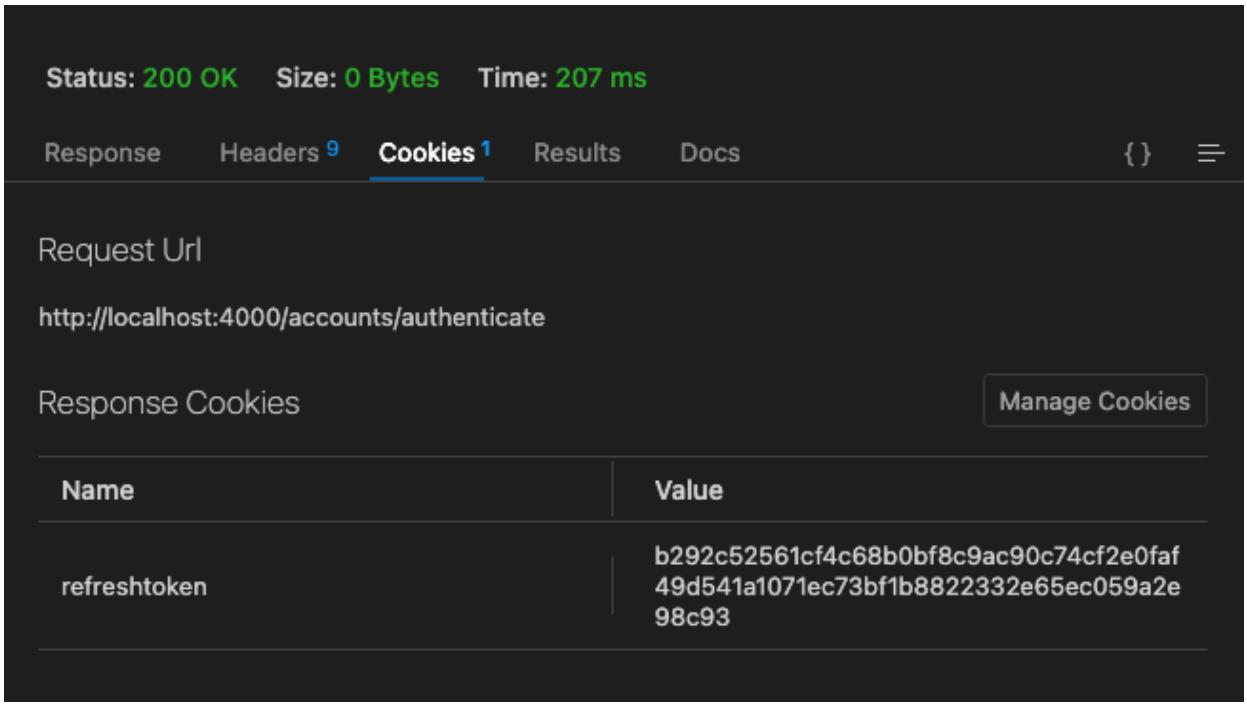
And this is the response cookies tab with the refresh token:

Status: 200 OK Size: 0 Bytes Time: 207 ms

Response Headers 9 Cookies 1 Results Docs { }

Response Headers

Header	Value
x-powered-by	Express
vary	Origin
access-control-allow-credentials	true
set-cookie	refreshToken=b292c52561cf4c68b0bf8c9ac90c74cf2e0faf49d541a1071ec73bf1b8822332e65ec059a2e98c93; Path=/; Expires=Fri, 29 Mar 2024 00:48:01 GMT; HttpOnly
content-type	application/json; charset=utf-8
content-length	333



How to get a list of all accounts with Postman / ThunderCLient

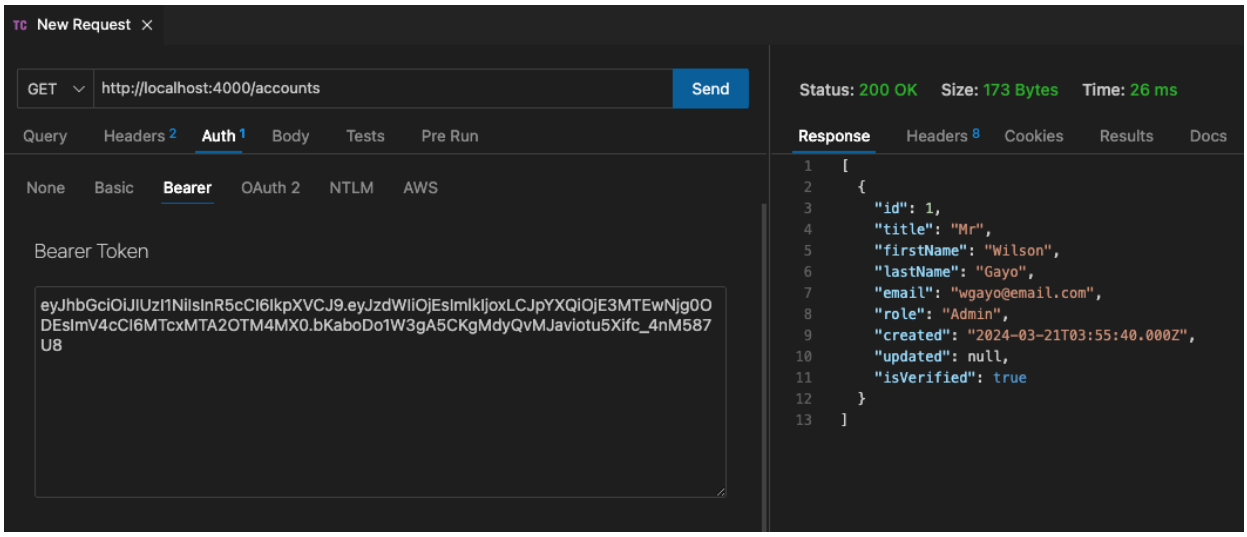
This is a secure request that requires a JWT authentication token from the authenticate step. The api route is restricted to admin users.

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

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

4. 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.
5. Click the "Send" button, you should receive a "200 OK" response containing a JSON array with all of the account records in the system.

Here's a screenshot of Postman after making an authenticated request to get all accounts:



How to update an account with Postman / ThunderCLient

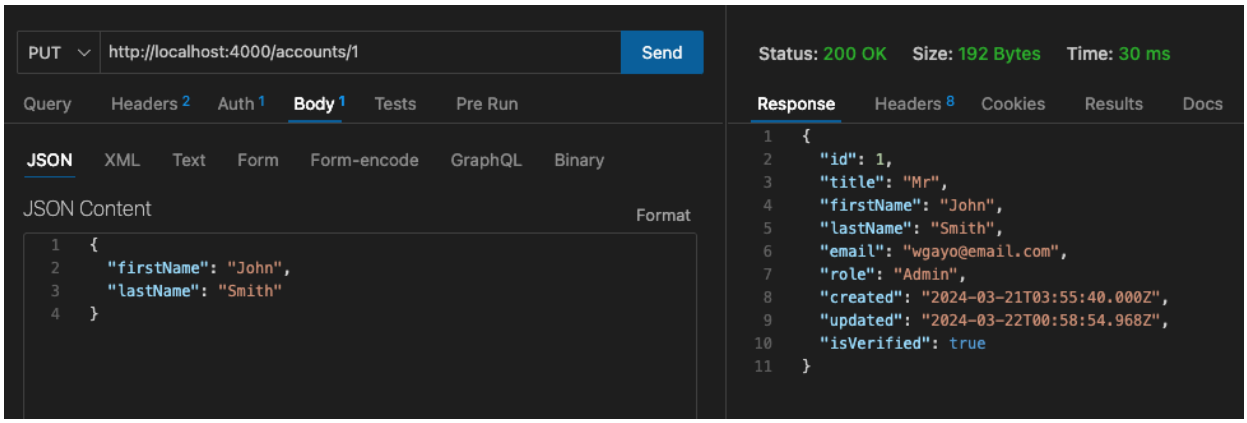
This is a secure request that requires a JWT authentication token from the authenticate step. Admin users can update any account including its role, while regular users are restricted to their own account and cannot update roles. Omitted or empty properties are not updated.

To update an account with the api follow these steps:

1. Open a new request tab by clicking the plus (+) button at the end of the tabs.
2. Change the http request method to "PUT" with the dropdown selector on the left of the URL input field.
3. 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`
4. 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.
5. Select the "Body" tab below the URL field, change the body type radio button to "raw", and change the format dropdown selector to "JSON".
6. Enter a JSON object in the "Body" textarea containing the properties you want to update, for example to update the first and last names:

```
{
  "firstName": "John",
  "lastName": "Smith"
}
```

7. Click the "Send" button, you should receive a "200 OK" response with the updated account details in the response body.



How to use a refresh token to get a new JWT token

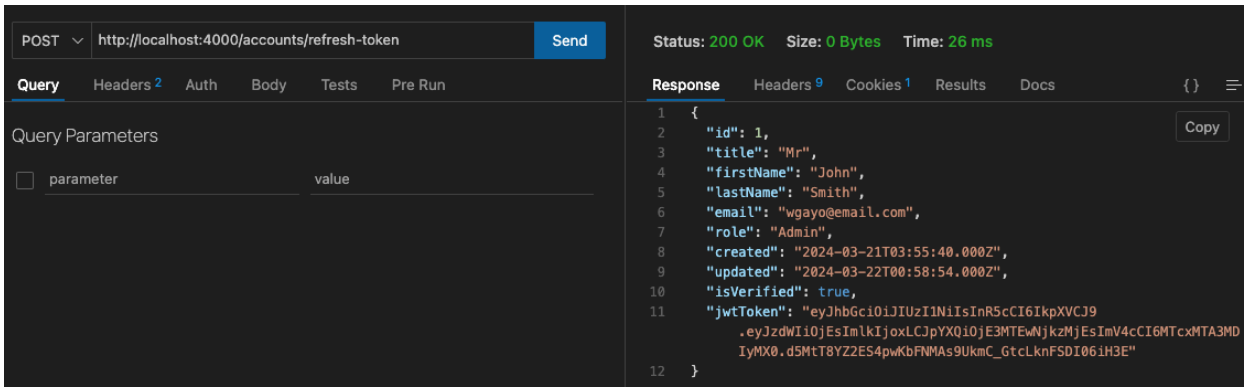
This step can only be done after the authenticate step because a valid refresh token cookie is required.

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

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

4. 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.
5. Copy the JWT token value because we'll be using it in the next steps to make authenticated requests.

Here's a screenshot of Postman after the request is sent and the token has been refreshed:



And this is the response cookies tab with the new refresh token:

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

Response Headers 9 Cookies 1 Results Docs { }

Response Headers

Header	Value
x-powered-by	Express
vary	Origin
access-control-allow-credentials	true
set-cookie	refreshToken=8fd16a422d0346e16e4c72fc2e3cbeaccc2001108630742371570ba8474ce24ad6141ebfa2dfa33; Path=/; Expires=Fri, 29 Mar 2024 01:02:01 GMT; HttpOnly
content-type	application/json; charset=utf-8