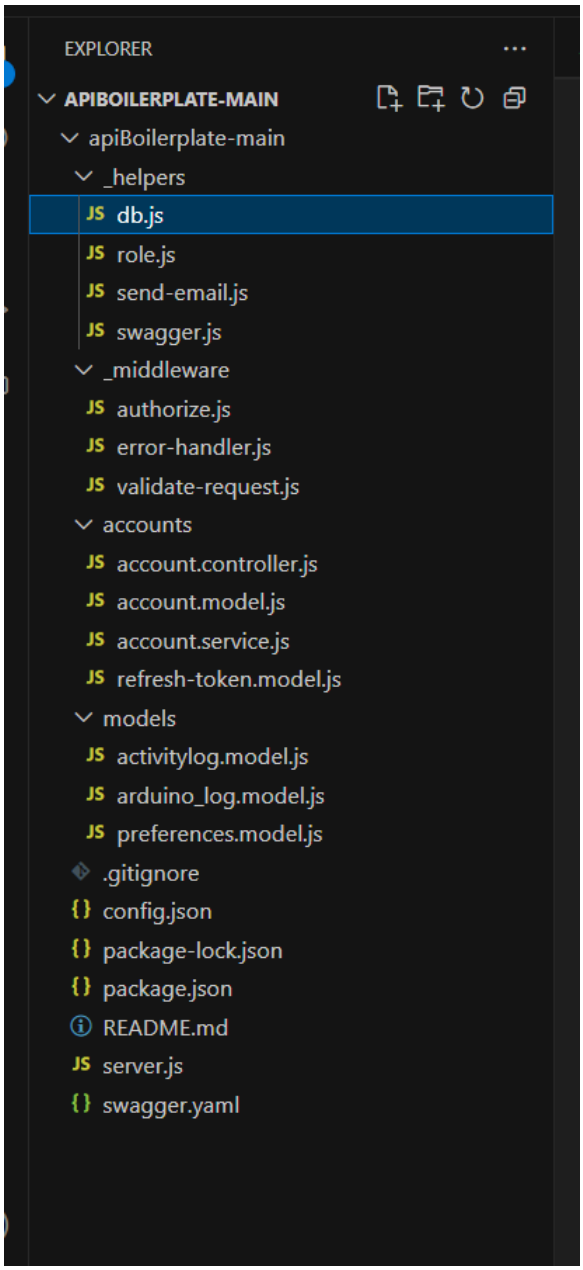
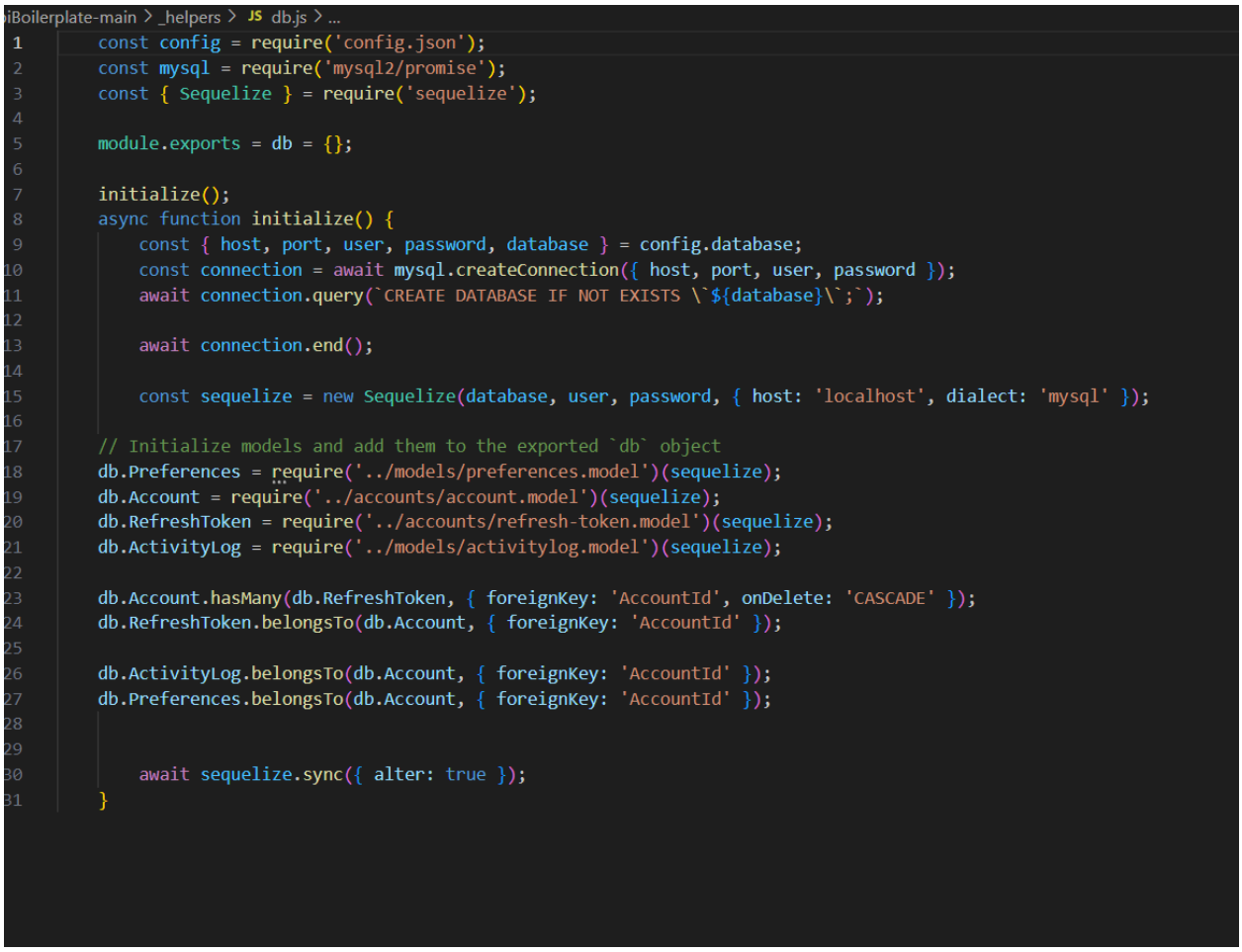
Group Project Activity: Full-Stack Application Development

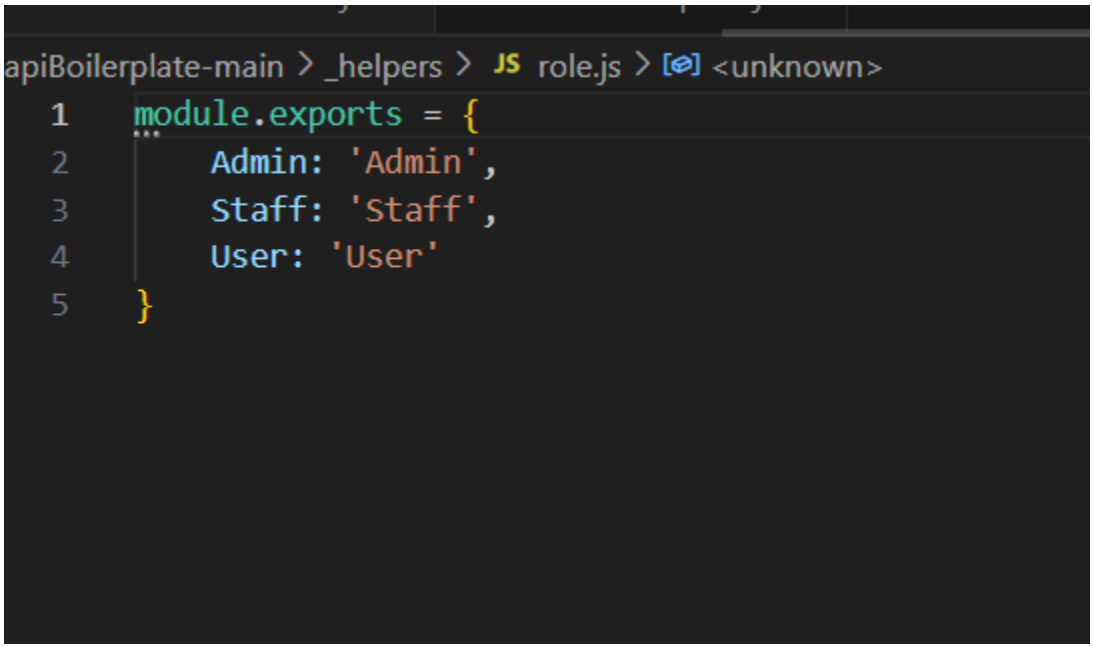
Step 1: Create a procedure



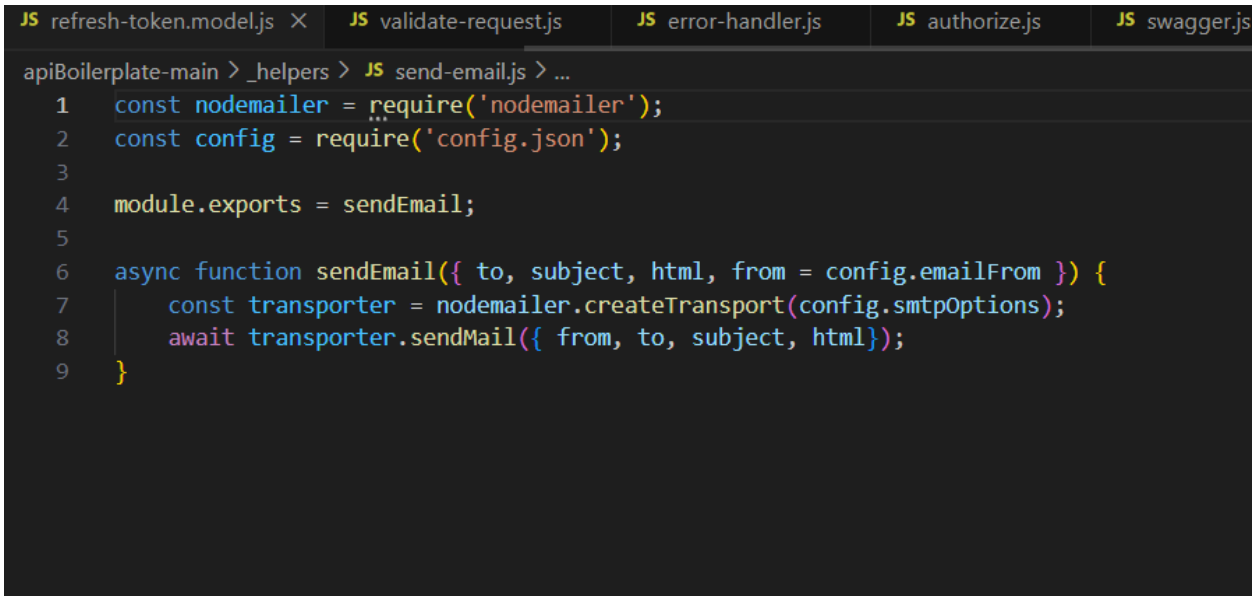
Path: /_helpers/db.js



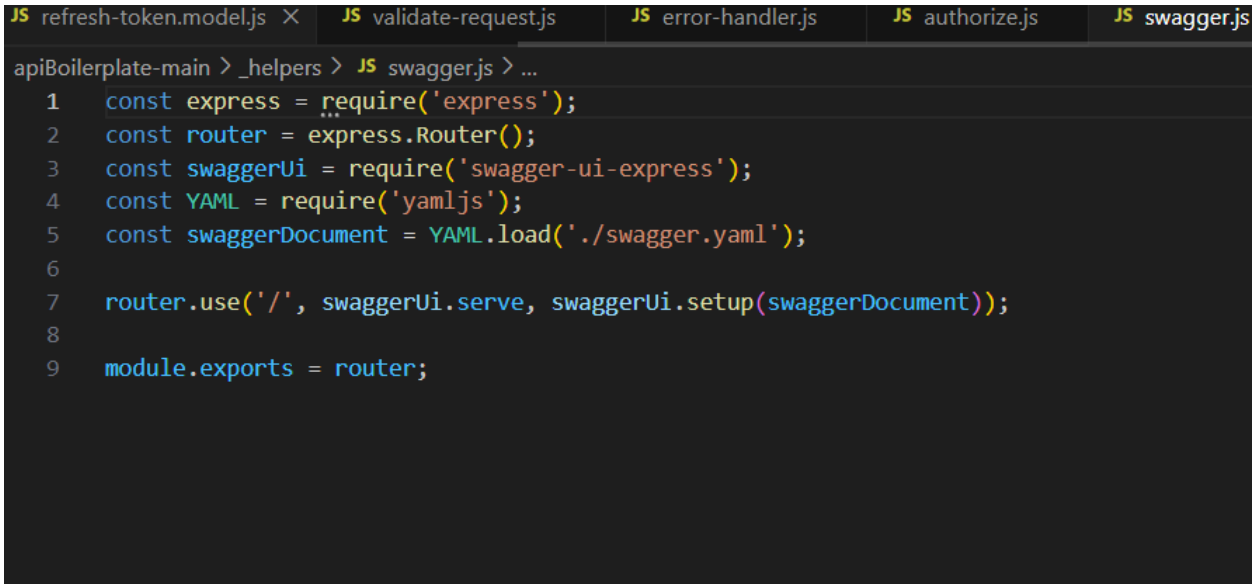
Path: /_helpers/role.js



Path: /_helpers/send-email.js

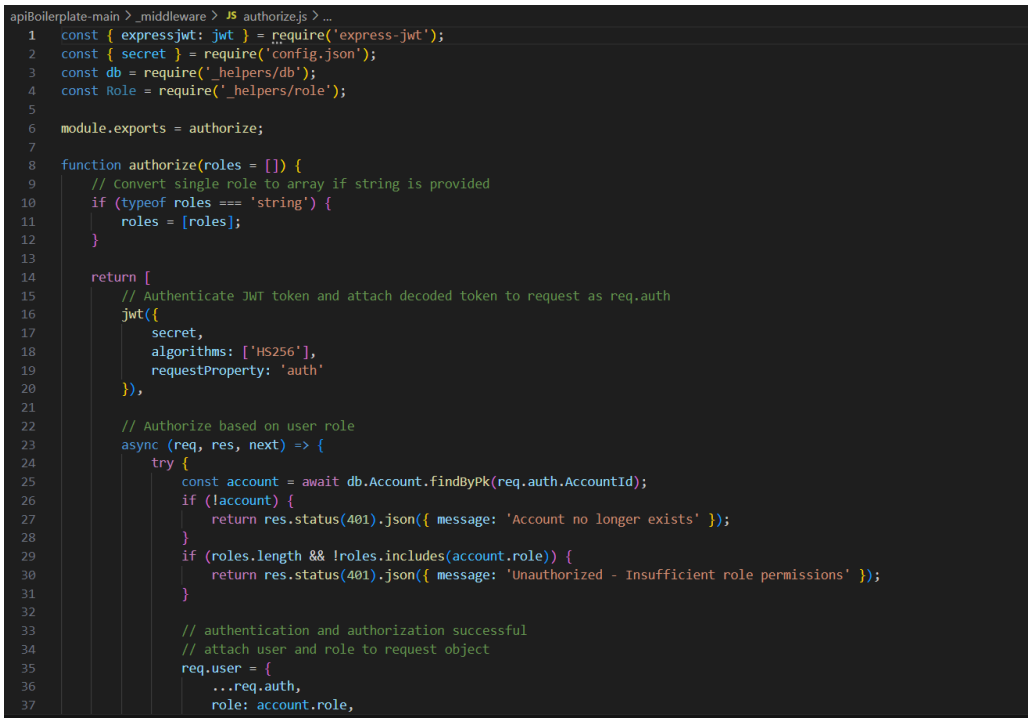


Path: /_helpers/swagger.js



Authorize Middleware

Path: /_middleware/authorize.js



```
function authorize(roles = []) {
        async (req, res, next) => {
                    role: account.role,
                    BranchId: account.BranchId  // Make sure this is being set correctly
                };

                // Add method to check if user owns a refresh token
                const refreshTokens = await account.getRefreshTokens();
                req.user.ownsToken = token => !!refreshTokens.find(x => x.token === token);

                // Log authorization attempt
                console.log(`Authorization successful for user ${account.email} with role ${account.role}`);

                next();
            } catch (error) {
                console.error('Authorization error:', error);
                return res.status(500).json({
                    success: false,
                    message: 'Internal server error during authorization'
                });
            }
        }
    ];
}
```

Path: /_middleware/error-handler.js

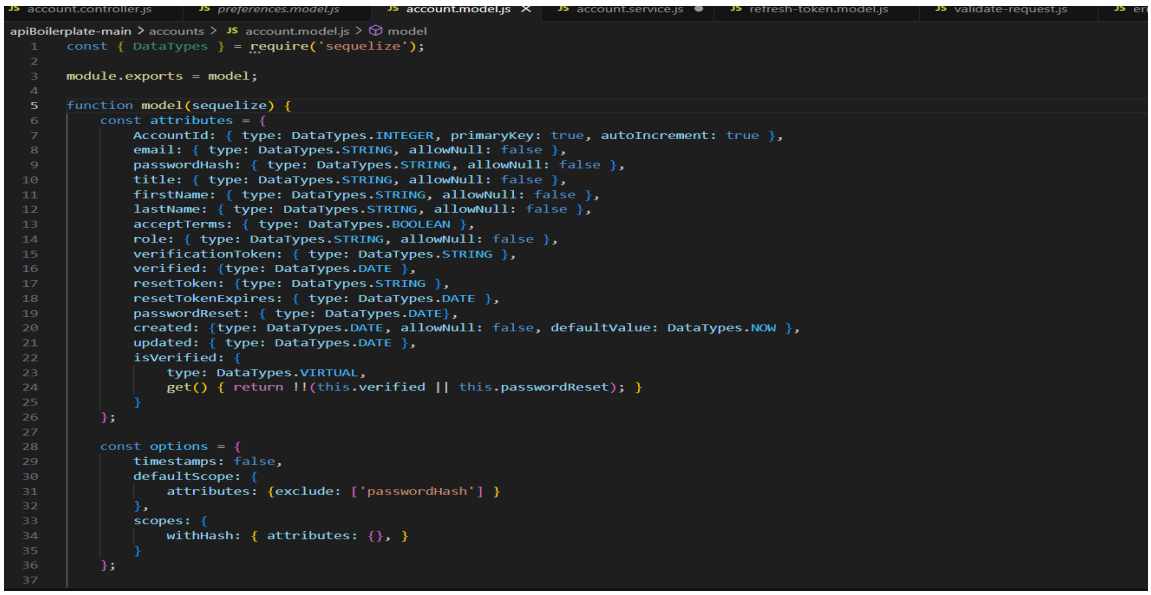
apiBoilerplate-main > _middleware > JS error-handler.js > [e] <unknown>

```
module.exports = errorHandler;

function errorHandler(err, req, res, next) {
    switch (true) {
        case typeof err === 'string':
            const is404 = err.toLowerCase().endsWith('not found');
            const statusCode = is404 ? 404 : 400;
            return res.status(statusCode).json({ message: err });
        case err.message && err.message.toLowerCase().includes('deactivated'):
            return res.status(403).json({ message: 'deactivated' });
        case err.name === 'UnauthorizedError':
            return res.status(401).json({ message: 'Unauthorized error-handler' });
        default:
            return res.status(500).json({ message: err.message });
    }
}

```

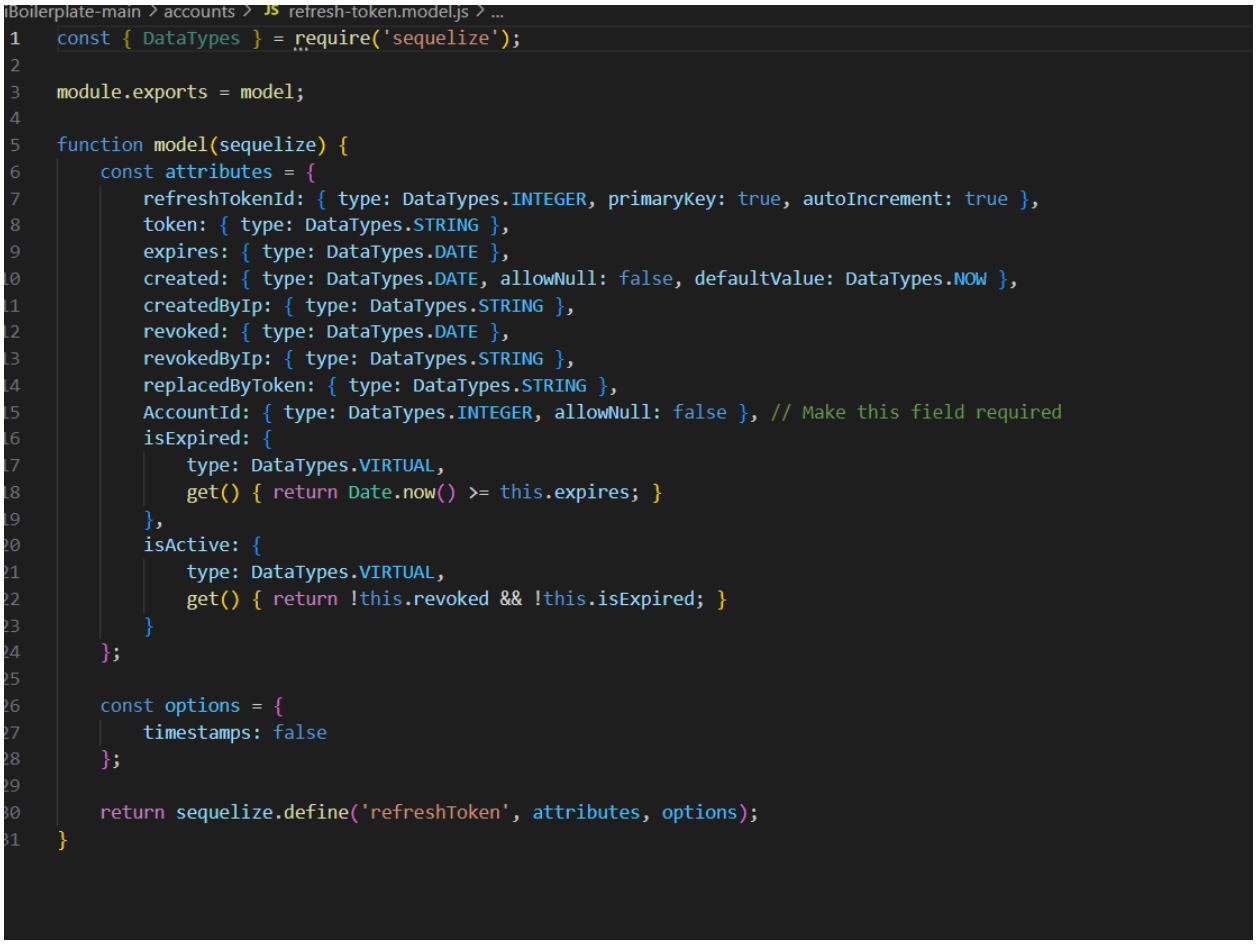
Path: /_middleware/validate-request.js

Sequelize Account Model

Path: /accounts/account.model.js



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



Path: /accounts/account.service.js

apiBoilerplate-main > accounts > account.service.js > refreshToken

```
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,
    logActivity,
    getAccountActivities,
    getAllActivityLogs,
    update,
    updatePreferences,
    getPreferences,
    delete: _delete
};

async function authenticate({ email, password, ipAddress, browserInfo }) {
    const account = await db.Account.scope('withHash').findOne({ where: { email } });

    if (!account || !account.isVerified || !(await bcrypt.compare(password, account.passwordHash))) {
      throw 'Email or password is incorrect';
    }
```

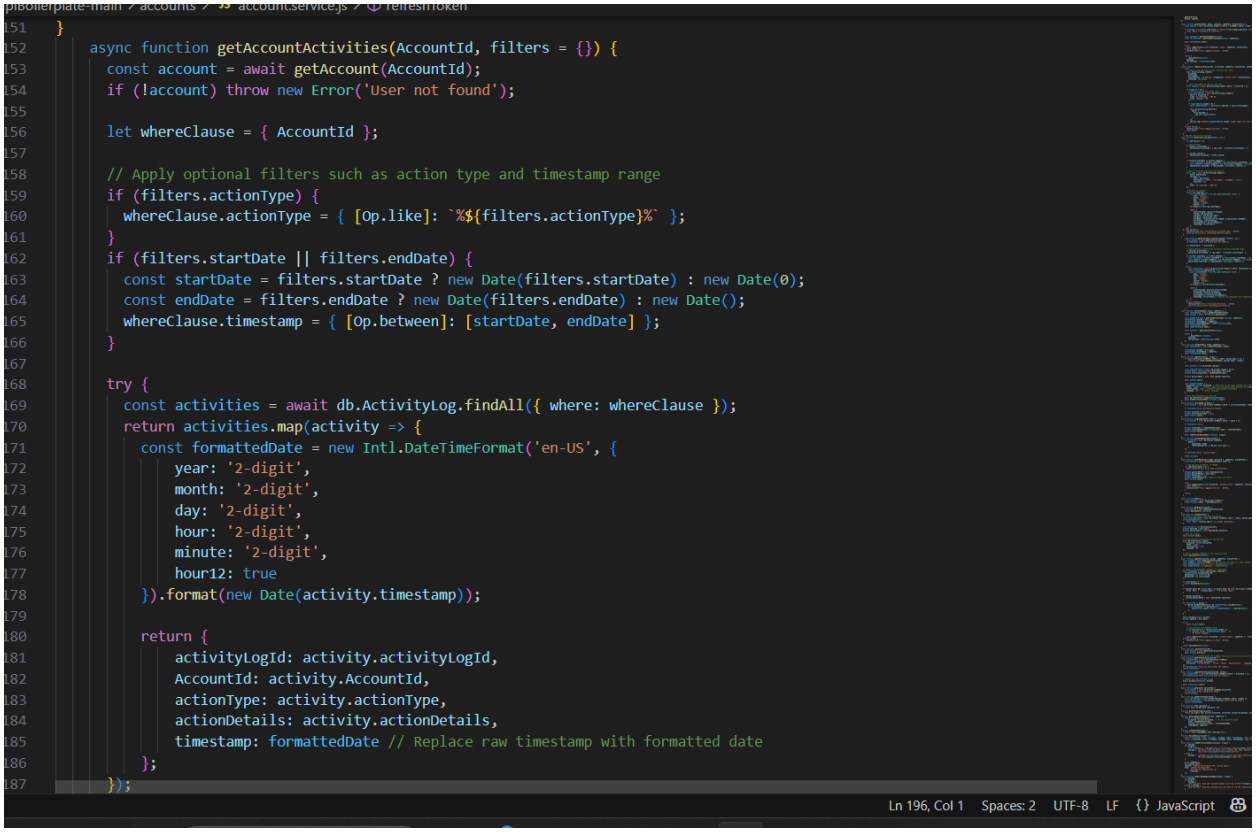
```
    const refreshToken = generateRefreshToken(account, ipAddress);

    await refreshToken.save();

    try {
      await logActivity(account.AccountId, 'login', ipAddress, browserInfo);
    } catch (error) {
      console.error('Error logging activity:', error);
    }

    return {
      ...basicDetails(account),
      jwtToken,
      refreshToken: refreshToken.token
    };
  }
async function logActivity(AccountId, actionType, ipAddress, browserInfo, updateDetails = '') {
    try {
      // Create a new log entry in the 'activity_log' table
      await db.ActivityLog.create({
        AccountId,
        actionType,
        actionDetails: `IP Address: ${ipAddress}, Browser Info: ${browserInfo}, Details: ${updateDetails}`,
        timestamp: new Date()
      });

      // Count the number of logs for the user
      const logCount = await db.ActivityLog.count({ where: { AccountId } });

      if (logCount > 10) {
        // Find and delete the oldest logs
        const logsToDelete = await db.ActivityLog.findAll({
          where: { AccountId },
          order: [['timestamp', 'ASC']],
          limit: logCount - 10
        });
```

```
        if (logsToDelete.length > 0) {
          const logIdsToDelete = logsToDelete.map(log => log.activityLogId);

          await db.ActivityLog.destroy({
            where: {
              activityLogId: {
                [Op.in]: logIdsToDelete
              }
            }
          });
          console.log(`Deleted ${logIdsToDelete.length} oldest log(s) for user ${AccountId}.`);
        }
      }
    } catch (error) {
      console.error('Error logging activity:', error);
      throw error;
    }
  }
  // Add this new service function
async function getAllActivityLogs(filters = {}) {
  try {
      let whereClause = {};

      // Apply filters
      if (filters.actionType) {
          whereClause.actionType = { [Op.like]: `%${filters.actionType}%` };
      }

      if (filters.userId) {
          whereClause.AccountId = filters.userId;
      }

      if (filters.startDate || filters.endDate) {
          const startDate = filters.startDate ? new Date(filters.startDate) : new Date(0);
          const endDate = filters.endDate ? new Date(filters.endDate) : new Date();
          whereClause.timestamp = { [Op.between]: [startDate, endDate] };
```

Ln 196, Col 1 Spaces: 2 UTF-8 LF {} JavaScrip

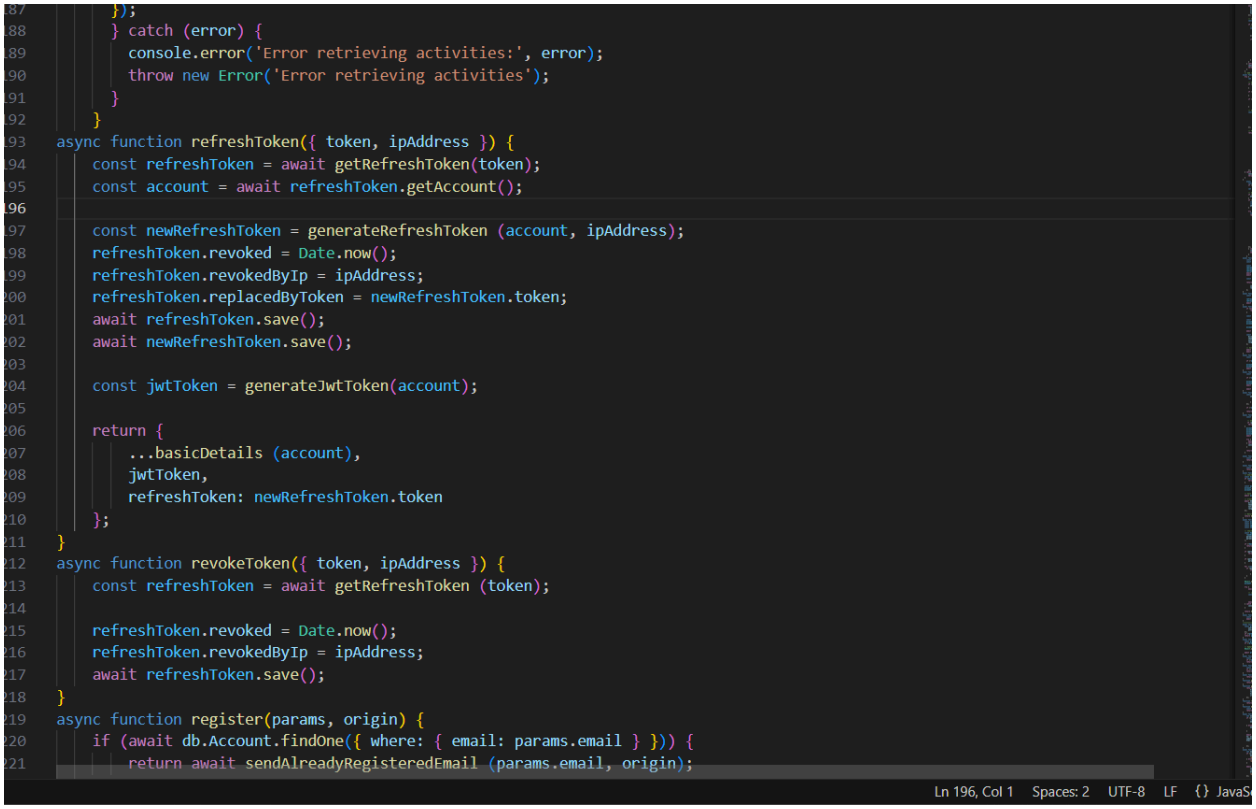
```

      // Get all activity logs with user details
      const logs = await db.ActivityLog.findAll({
          where: whereClause,
          include: [{
              model: db.Account,
              attributes: ['email', 'firstName', 'lastName', 'role'],
              required: true
          }],
          order: [['timestamp', 'DESC']]
      });

      // Format the response
      return logs.map(log => {
          const formattedDate = new Intl.DateTimeFormat('en-US', {
              year: 'numeric',
              month: '2-digit',
              day: '2-digit',
              hour: '2-digit',
              minute: '2-digit',
              hour12: true
          }).format(new Date(log.timestamp));

          return {
              activityLogId: log.activityLogId,
              userId: log.AccountId,
              userEmail: log.Account.email,
              userRole: log.Account.role,
              userName: `${log.Account.firstName} ${log.Account.lastName}`,
              actionType: log.actionType,
              actionDetails: log.actionDetails,
              timestamp: formattedDate
          };
      });
  } catch (error) {
      console.error('Error retrieving all activity logs:', error);
```

Ln 196, Col 1 Spaces: 2 UTF-8 LF {} JavaScript



```
}
    async function getAccountActivities(AccountId, filters = {}) {
      const account = await getAccount(AccountId);
      if (!account) throw new Error('User not found');

      let whereClause = { AccountId };

      // Apply optional filters such as action type and timestamp range
      if (filters.actionType) {
        whereClause.actionType = { [Op.like]: `%${filters.actionType}%` };
      }
      if (filters.startDate || filters.endDate) {
        const startDate = filters.startDate ? new Date(filters.startDate) : new Date(0);
        const endDate = filters.endDate ? new Date(filters.endDate) : new Date();
        whereClause.timestamp = { [Op.between]: [startDate, endDate] };
      }

      try {
        const activities = await db.ActivityLog.findAll({ where: whereClause });
        return activities.map(activity => {
          const formattedDate = new Intl.DateTimeFormat('en-US', {
            year: '2-digit',
            month: '2-digit',
            day: '2-digit',
            hour: '2-digit',
            minute: '2-digit',
            hour12: true
          }).format(new Date(activity.timestamp));

          return {
            activityLogId: activity.activityLogId,
            AccountId: activity.AccountId,
            actionType: activity.actionType,
            actionDetails: activity.actionDetails,
            timestamp: formattedDate // Replace raw timestamp with formatted date
          };
        });
```

Ln 196, Col 1 Spaces: 2 UTF-8 LF {} JavaScript



```
        });
        } catch (error) {
          console.error('Error retrieving activities:', error);
          throw new Error('Error retrieving activities');
        }
      }
async function refreshToken({ token, ipAddress }) {
    const refreshToken = await getRefreshToken(token);
    const account = await refreshToken.getAccount();

    const newRefreshToken = generateRefreshToken (account, ipAddress);
    refreshToken.revoked = Date.now();
    refreshToken.revokedByIp = ipAddress;
    refreshToken.replacedByToken = newRefreshToken.token;
    await refreshToken.save();
    await newRefreshToken.save();

    const jwtToken = generateJwtToken(account);

    return {
        ...basicDetails (account),
        jwtToken,
        refreshToken: newRefreshToken.token
    };
}
async function revokeToken({ token, ipAddress }) {
    const refreshToken = await getRefreshToken (token);

    refreshToken.revoked = Date.now();
    refreshToken.revokedByIp = ipAddress;
    await refreshToken.save();
}
async function register(params, origin) {
    if (await db.Account.findOne({ where: { email: params.email } })) {
        return await sendAlreadyRegisteredEmail (params.email, origin);
```

Ln 196, Col 1 Spaces: 2 UTF-8 LF {} JavaS

```
    const account = new db.Account (params);

    const isFirstAccount = (await db.Account.count()) === 0;
    account.role = isFirstAccount? Role.Admin: Role.User;
    account.verificationToken = randomTokenString();

    account.passwordHash = await hash (params.password);

    await account.save();

    const preferencesData = {
      AccountId: account.AccountId, // Reference to the newly created user's ID
      theme: 'light',  // Default theme (you can modify these defaults as needed)
      notifications: true,  // Default notifications preference
      language: 'en'   // Default language
    };

    // Save the preferences for the user
    await db.Preferences.create(preferencesData);
    await sendVerificationEmail (account, origin);
}
async function verifyEmail({token}) {
    const account = await db.Account.findOne({ where: { verificationToken: token} });

    if (!account) throw 'Verification failed';

    account.verified = Date.now();
    account.verificationToken = null;
    await account.save();
}
async function forgotPassword({ email }, origin) {
    const account = await db.Account.findOne({ where: { email } });

    if (!account) return;
```

```
    account.resetToken = randomTokenString();
    account.resetTokenExpires= new Date(Date.now() + 24*60*60*1000);
    await account.save();

    await sendPasswordResetEmail (account, origin);
}
async function validateResetToken({token}) {
    const account = await db.Account.findOne({
        where: {
            resetToken: token,
            resetTokenExpires: { [Op.gt]: Date.now() }
        }
    });

    if (!account) throw 'Invalid token';

    return account;
}
async function resetPassword({ token, password }, ipAddress, browserInfo) {
    const account = await validateResetToken({ token });

    // Add password validation if needed
    if (password.length < 6) {
      throw 'Password must be at least 6 characters';
  }
    account.passwordHash = await hash(password);
    account.passwordReset = Date.now();
    account.resetToken = null;
    account.resetTokenExpires = null; // Clear the expiry
    await account.save();

    try {
      await logActivity(account.AccountId, 'password_reset', ipAddress, browserInfo);
    } catch (error) {
```

```
    }

    return;
  }

async function getAll() {
    const accounts = await db.Account.findAll();
    return accounts.map(x => basicDetails(x));
}

async function getById(AccountId) {
    const account = await getAccount(AccountId);
    return basicDetails (account);
}
async function create(params) {
  // Check if the email is already registered
  const existingAccount = await db.Account.findOne({ where: { email: params.email } });
  if (existingAccount) {
      throw `Email "${params.email}" is already registered`;
  }

  const account = new db.Account(params);
  account.verified = Date.now();
  account.passwordHash = await hash(params.password);

  // Save the account
  await account.save();

  // Create default preferences for the new user
  await db.Preferences.create({
      AccountId: account.AccountId,
      theme: 'light',
      notifications: true,
      language: 'en'
  });
```

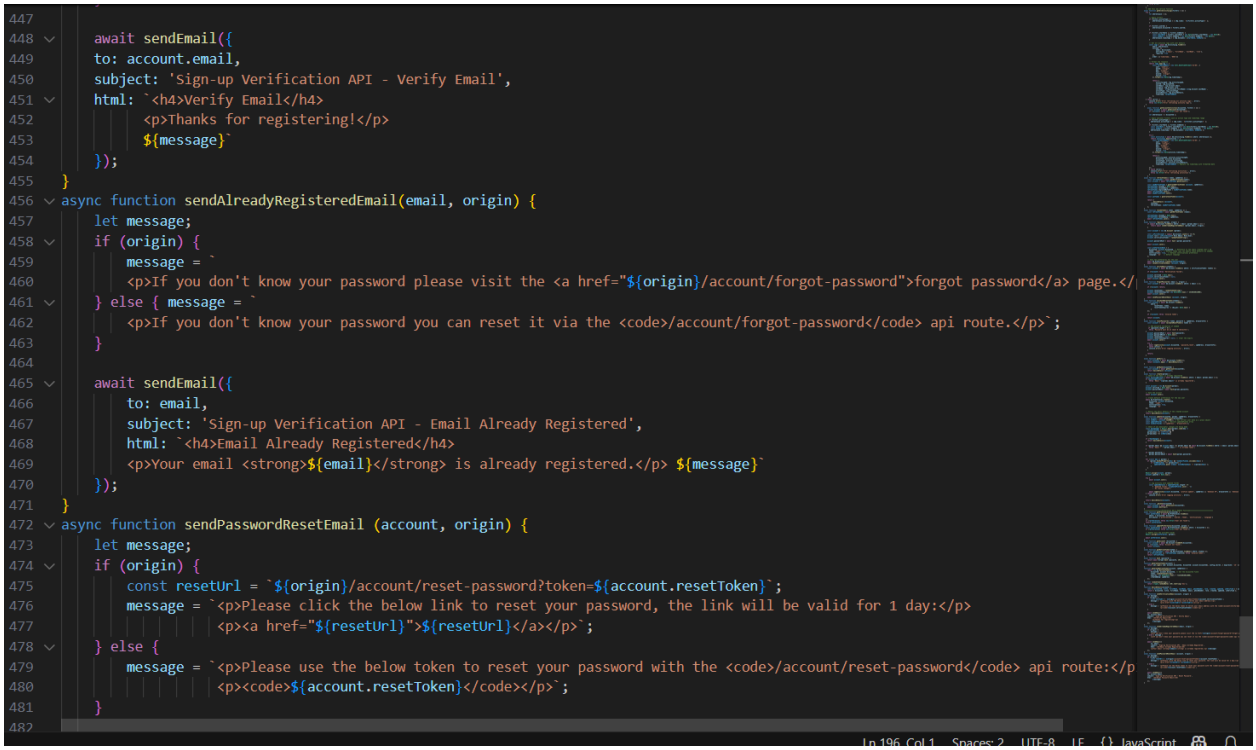
```
  return basicDetails(account);
}
async function update(AccountId, params, ipAddress, browserInfo) {
  const account = await getAccount(AccountId);
  const oldData = account.toJSON(); // Get current user data as a plain object
  const updatedFields = []; // Declare updatedFields array
  const nonUserFields = ['ipAddress', 'browserInfo'];

  // Check if any meaningful changes are being made
  const hasChanges = Object.keys(params).some(key =>
    !nonUserFields.includes(key) &&
    params[key] !== undefined &&
    params[key] !== oldData[key]
  );

  if (!hasChanges) {
    return basicDetails(account);
  }

  if (params.email && account.email !== params.email && await db.Account.findOne({ where: { email: params.email } })) {
      throw 'Email "' + params.email + '" is already taken';
  }

  if (params.password) {
      params.passwordHash = await hash(params.password);
  }

  for (const key in params) {
    if (params.hasOwnProperty(key) && !nonUserFields.includes(key)) {
        if (oldData[key] !== params[key]) {
            updatedFields.push(`${key}: ${oldData[key]} -> ${params[key]}`);
        }
    }
  }

  Object.assign(account, params);
```

```
  try {
      await account.save();

      // Log activity with updated fields
      const updateDetails = updatedFields.length > 0
          ? `Updated fields: ${updatedFields.join(', ')}`
          : 'No fields changed';

      await logActivity(account.AccountId, 'profile update', ipAddress || 'Unknown IP', browserInfo || 'Unknown Browser', updat
  } catch (error) {
      console.error('Error logging activity:', error);
  }

  return basicDetails(account);
}
async function _delete(AccountId) {
    const account = await getAccount(AccountId);
    await account.destroy();
}
//==================Preferences Get & Update Function==========================
async function getPreferences(AccountId) {
  const preferences = await db.Preferences.findOne({
      where: { AccountId: AccountId },
      attributes: ['preferenceId', 'userId','theme', 'notifications', 'language']
  });
  if (!preferences) throw new Error('User not found');
  return preferences;
}
async function updatePreferences(AccountId, params) {
  const preferences = await db.Preferences.findOne({ where: { AccountId } });
  if (!preferences) throw new Error('User not found');

  // Update only the provided fields
  Object.assign(preferences, params);
```

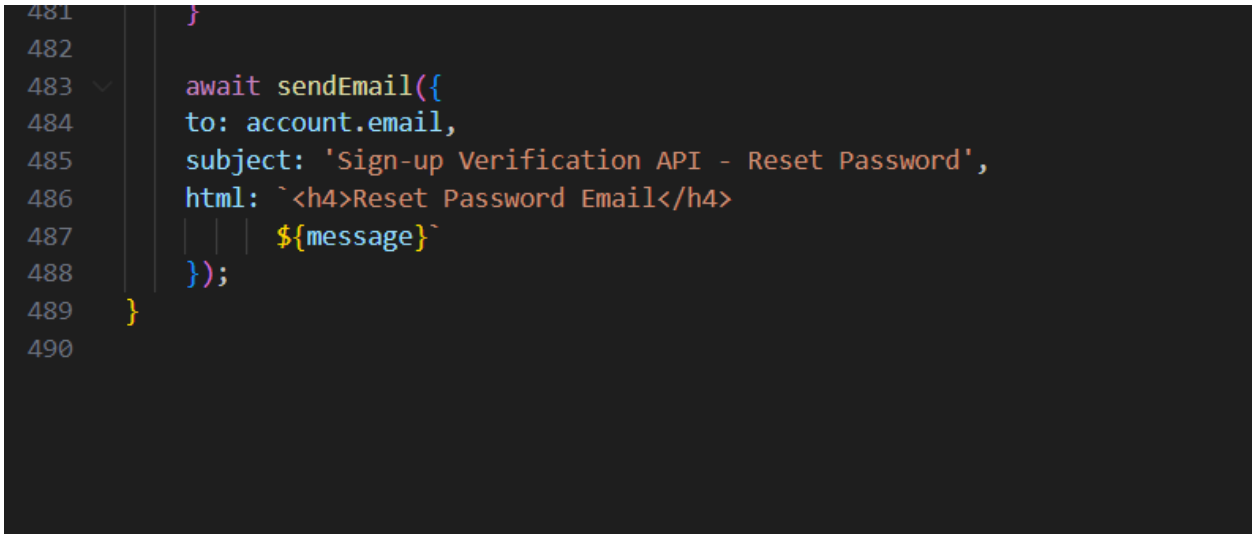
Ln 196, Col 1 Spaces: 2 UTF-8 LF

```
405   }
406   async function getAccount (AccountId) {
407       const account = await db.Account.findByPk(AccountId);
408       if (!account) throw 'Account not found';
409       return account;
410   }
411   async function getRefreshToken(token) {
412       const refreshToken = await db.RefreshToken.findOne({ where: {token} });
413       if (!refreshToken || !refreshToken.isActive) throw 'Invalid token';
414       return refreshToken;
415   }
416   async function hash (password) {
417       return await bcrypt.hash (password, 10);
418   }
419   function generateJwtToken(account) {
420       return jwt.sign({ sub: account.AccountId, AccountId: account.AccountId}, config.secret, { expiresIn: '1h' });
421   }
422   function generateRefreshToken(account, ipAddress) {
423       return new db.RefreshToken({
424           AccountId: account.AccountId, // Set the AccountId field
425           token: randomTokenString(),
426           expires: new Date(Date.now() + 7*24*60*60*1000),
427           createdByIp: ipAddress
428       });
429   }
430   function randomTokenString() {
431       return crypto.randomBytes (40).toString('hex');
432   }
433   function basicDetails(account) {
434       const { AccountId, title, firstName, lastName, email, phoneNumber, role, created, updated, isVerified } = account;
435       return { AccountId, title, firstName, lastName, email, phoneNumber, role, created, updated, isVerified };
436   }
437   async function sendVerificationEmail(account, origin) {
438       let message;
439       if (origin) {
440           const verifyUrl = `${origin}/account/verify-email?token=${account.verificationToken}`;
```

Ln 196, Col 1 Spaces: 2 UTF-8 LF {} JavaScript



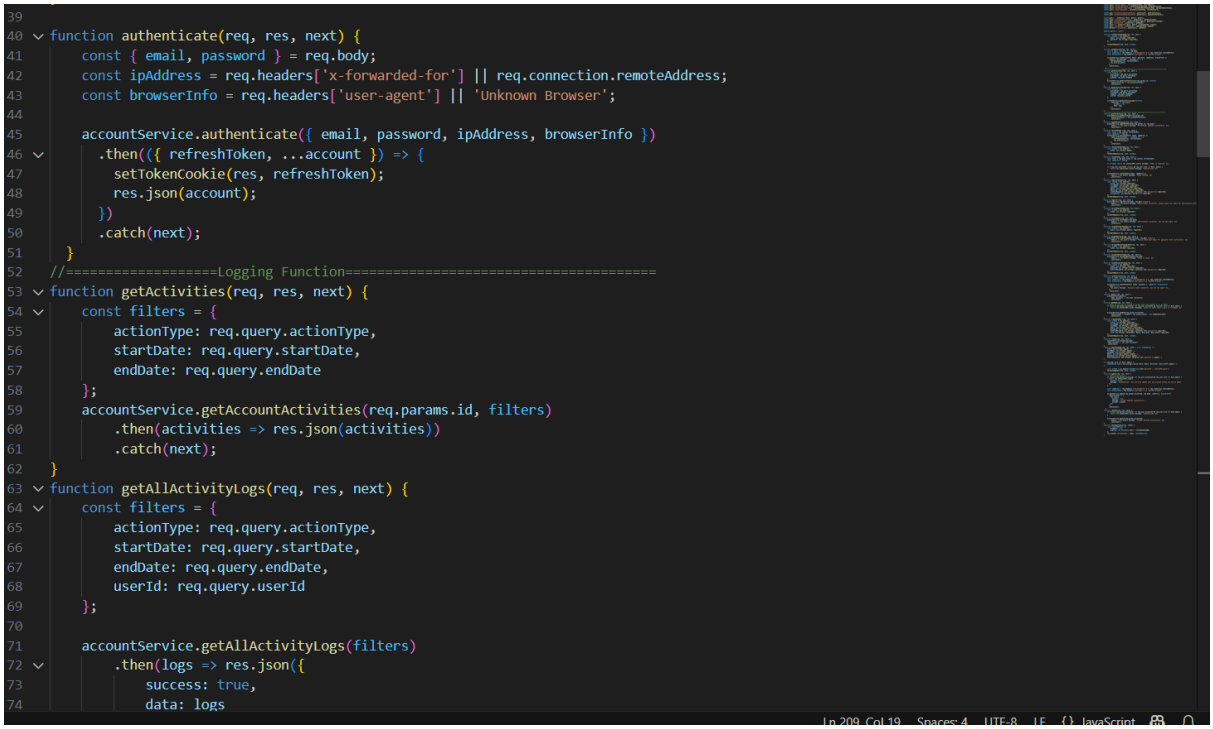
```
    await sendEmail({
    to: account.email,
    subject: 'Sign-up Verification API - Verify Email',
    html: `<h4>Verify Email</h4>
           <p>Thanks for registering!</p>
           ${message}`
    });
}
async function sendAlreadyRegisteredEmail(email, origin) {
    let message;
    if (origin) {
        message = `
        <p>If you don't know your password please visit the <a href="${origin}/account/forgot-password">forgot password</a> page.</
    } else { message = `
        <p>If you don't know your password you can reset it via the <code>/account/forgot-password</code> api route.</p>`;
    }

    await sendEmail({
        to: email,
        subject: 'Sign-up Verification API - Email Already Registered',
        html: `<h4>Email Already Registered</h4>
        <p>Your email <strong>${email}</strong> is already registered.</p> ${message}`
    });
}
async function sendPasswordResetEmail (account, origin) {
    let message;
    if (origin) {
        const resetUrl = `${origin}/account/reset-password?token=${account.resetToken}`;
        message = `<p>Please click the below link to reset your password, the link will be valid for 1 day:</p>
                   <p><a href="${resetUrl}">${resetUrl}</a></p>`;
    } else {
        message = `<p>Please use the below token to reset your password with the <code>/account/reset-password</code> api route:</p
                   <p><code>${account.resetToken}</code></p>`;
    }
```



```
    }

    await sendEmail({
    to: account.email,
    subject: 'Sign-up Verification API - Reset Password',
    html: `<h4>Reset Password Email</h4>
           ${message}`
    });
}
```

Path: /accounts/accounts.controller.js

```
piBoilerplate-main > accounts > JS account.controller.js > createSchema > schema > password
const express = require('express');
const router = express. Router();
const Joi = require('joi');
const validateRequest = require('_middleware/validate-request');
const authorize = require('_middleware/authorize');
const Role = require('_helpers/role');
const accountService = require('./account.service');

router.post('/authenticate', authenticateSchema, authenticate);
router.post('/refresh-token', refreshToken);
router.post('/revoke-token', authorize(), revokeTokenSchema, revokeToken);
router.post('/register', registerSchema, register);
router.post('/verify-email', verifyEmailSchema, verifyEmail);
router.post('/forgot-password', forgotPasswordSchema, forgotPassword);
router.post('/validate-reset-token', validateResetTokenSchema, validateResetToken);
router.post('/reset-password', resetPasswordSchema, resetPassword);

router.get('/:preferenceId/preferences',authorize(), getPreferences);
router.put('/:preferenceId/preferences',authorize(), updatePreferences);

router.get('/', authorize (Role. Admin), getAll);
router.post('/:AccountId/activity', authorize(), getActivities);
router.get('/activity-logs', authorize(Role.Admin), getAllActivityLogs);
router.get('/:AccountId', authorize(), getById);
router.post('/', authorize (Role. Admin), createSchema, create);
router.put('/:AccountId', authorize(), updateSchema, update);
router.delete('/:AccountId', authorize(), _delete);

module.exports = router;

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

    validateRequest(req, next, schema);
```



```
function authenticate(req, res, next) {
    const { email, password } = req.body;
    const ipAddress = req.headers['x-forwarded-for'] || req.connection.remoteAddress;
    const browserInfo = req.headers['user-agent'] || 'Unknown Browser';

    accountService.authenticate({ email, password, ipAddress, browserInfo })
      .then(({ refreshToken, ...account }) => {
        setTokenCookie(res, refreshToken);
        res.json(account);
      })
      .catch(next);
  }
//==================Logging Function=====================================
function getActivities(req, res, next) {
    const filters = {
        actionType: req.query.actionType,
        startDate: req.query.startDate,
        endDate: req.query.endDate
    };
    accountService.getAccountActivities(req.params.id, filters)
        .then(activities => res.json(activities))
        .catch(next);
}
function getAllActivityLogs(req, res, next) {
    const filters = {
        actionType: req.query.actionType,
        startDate: req.query.startDate,
        endDate: req.query.endDate,
        userId: req.query.userId
    };

    accountService.getAllActivityLogs(filters)
        .then(logs => res.json({
            success: true,
            data: logs
```

```
}
//==================Preferences Router Function=========================
function getPreferences(req, res, next) {
    accountService.getPreferences(req.params.id)
        .then(preferences => res.json(preferences))
        .catch(next);
}
function updatePreferences(req, res, next) {
    accountService.updatePreferences(req.params.id, req.body)
        .then(() => res.json({ message: 'Preferences updated successfully' }))
        .catch(next);
}
function refreshToken (req, res, next) {
    const token = req.cookies.refreshToken;
    const ipAddress = req.ip;
    accountService.refreshToken({ token, ipAddress })
        .then(({refreshToken, ...account }) => {
            setTokenCookie(res, refreshToken);
            res.json(account);
        })
        .catch(next);
}
function revokeTokenSchema(req, res, next) {
    const schema = Joi.object({
        token: Joi.string().empty('')
    });
    validateRequest(req, next, schema);
}
function revokeToken (req, res, next) {
    const token = req.body.token || req.cookies.refreshToken;
    const ipAddress = req.ip;

    if (!token) return res.status(400).json({ message: 'Token is required' });

    if (!req.user.ownsToken (token) && req.user.role !== Role. Admin) {
        return res.status(401).json({ message: 'Unauthorized' });
    }
```

```

    accountService.revokeToken({token, ipAddress })
        .then(() =>res.json({ message: 'Token revoked' }))
        .catch(next);
}
function registerSchema(req, res, next) {
    const schema = Joi.object({
        title: Joi.string().required(),
        firstName: Joi.string().required(),
        lastName: Joi.string().required(),
        email: Joi.string().email().required(),
        password: Joi.string().min(6).required(),
        confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
        acceptTerms: Joi.boolean().valid(true).required()
    });
    validateRequest(req, next, schema);
}
function register(req, res, next) {
    accountService.register(req.body, req.get('origin'))
        .then(() => res.json({ message: 'Registration successful, please check your email for verification instructions' }))
        .catch(next);
}
function verifyEmailSchema(req, res, next) {
    const schema = Joi.object({
        token: Joi.string().required()
    });
    validateRequest(req, next, schema);
}
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({
```

```
}
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) {
    const { token, password } = req.body;
    const ipAddress = req.headers['x-forwarded-for'] || req.connection.remoteAddress;
    const browserInfo = req.headers['user-agent'] || 'Unknown Browser';
```

```
    accountService.resetPassword({ token, password }, ipAddress, browserInfo)
      .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);
}
function getById(req: any, res: any, next: any): any
function getById(req, res, next) {
    // Check if the user is trying to access their own account or is an admin
    if (Number(req.params.AccountId) !== req.user.AccountId && req.user.role !== Role.Admin) {
        return res.status(403).json({ message: 'Access to other user\'s data is forbidden' });
    }

    accountService.getById(req.params.AccountId)
        .then(account => account ? res.json(account) : res.sendStatus(404))
        .catch(next);
}
function createSchema (req, res, next) {
    const schema = Joi.object({
        title: Joi.string().required(),
        firstName: Joi.string().required(),
        lastName: Joi.string().required(),
        email: Joi.string().email().required(),
        password: Joi.string().min(6).required(),
        confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
        role: Joi.string().valid(Role. Admin, Role.User, Role.Staff).required()
    });
    validateRequest(req, next, schema);
}
function create(req, res, next) {
    accountService.create(req.body)
```

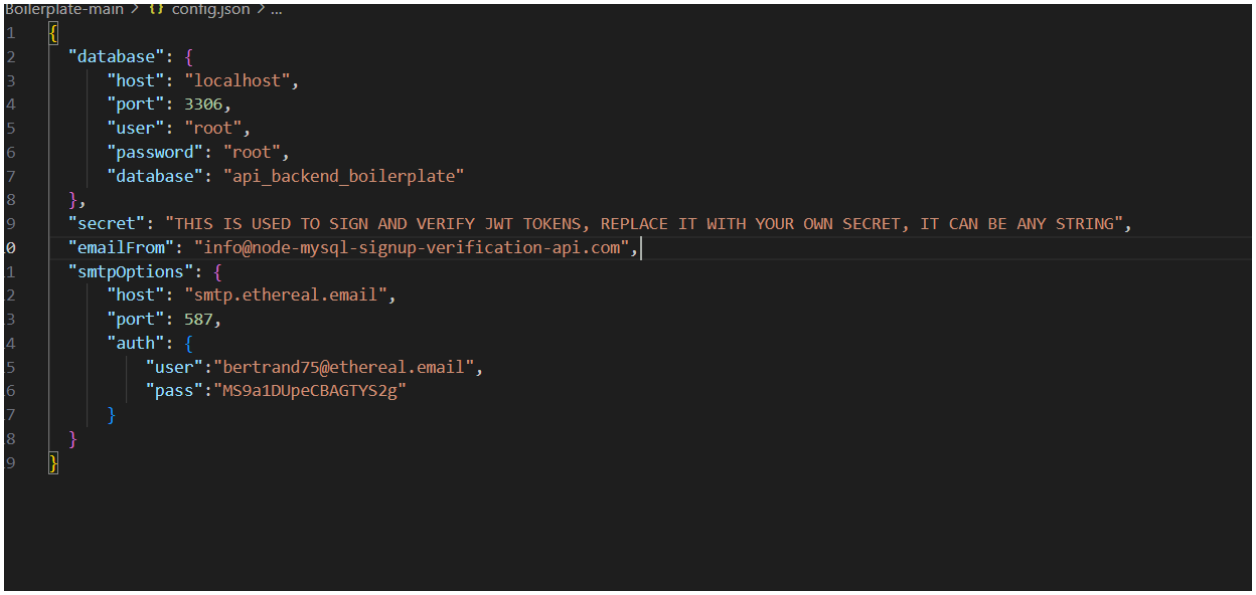
```
    .then (account => res.json (account))
    .catch(next);
}
function updateSchema(req, res, next) { const schemaRules = {
    title: Joi.string().empty(''),
    firstName: Joi.string().empty(''),
    lastName: Joi.string().empty(''),
    email: Joi.string().email().empty(''),
    password: Joi.string().min(6).empty(''),
    confirmPassword: Joi.string().valid(Joi.ref('password')).empty('')
}

if (req.user.role === Role. Admin) {
    schemaRules.role = Joi.string().valid (Role. Admin, Role.User, Role.Staff).empty('');
}

    const schema = Joi.object(schemaRules).with('password', 'confirmPassword');
    validateRequest(req, next, schema);
}
function update(req, res, next) {
    // Check authorization
    if (Number(req.params.AccountId) !== req.user.AccountId && req.user.role !== Role.Admin) {
      return res.status(401).json({
        success: false,
        message: 'Unauthorized - You can only update your own account unless you are an admin'
      });
    }

    const ipAddress = req.headers['x-forwarded-for'] || req.connection.remoteAddress;
    const browserInfo = req.headers['user-agent'] || 'Unknown Browser';

    accountService.update(req.params.AccountId, req.body, ipAddress, browserInfo)
      .then(account => {
        res.json({
```

Ln 209, Col 19 Spaces: 4 UTF-8 LF {} JavaScript

```
251             success: true,
252             message: 'Account updated successfully',
253             account: account
254           });
255         })
256         .catch(next);
257     }
258     function _delete(req, res, next) {
259         if (Number(req.params.AccountId) !== req.user.AccountId && req.user.role !== Role.Admin) {
260             return res.status(401).json({ message: 'Unauthorized' });
261         }
                                (parameter) req: any
262
263         accountService.delete(req.params.AccountId)
264             .then(() =>res.json({ message: 'Account deleted successfully' }))
265             .catch(next);
266     }
267     function setTokenCookie(res, token) {
268         const cookieOptions = {
269             httpOnly: true,
270             expires: new Date(Date.now() + 7*24*60*60*1000)
271         };
272         res.cookie('refreshToken', token, cookieOptions);
273     }
```

Api Config

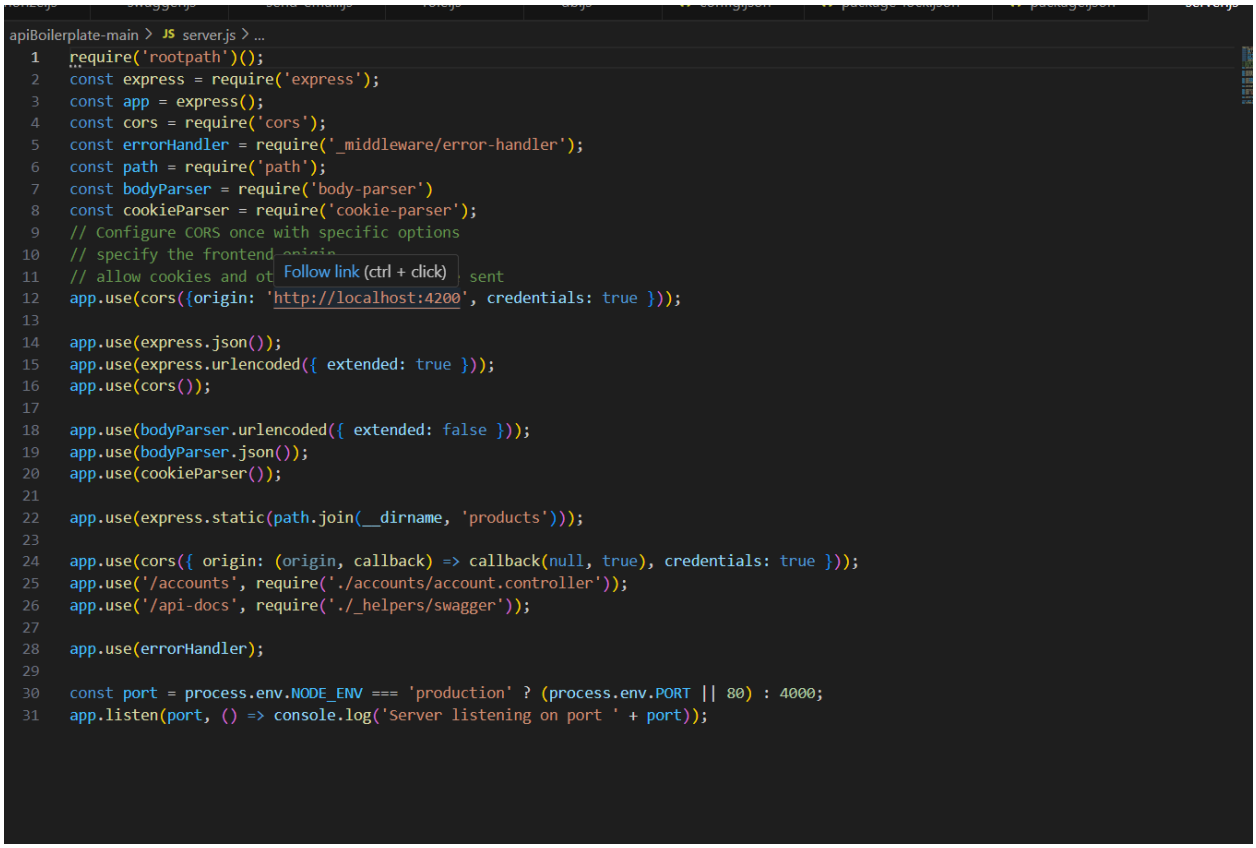
Path: /config.json



Path: /package.json

Server Startup File

Path: /server.js



Push into github

```
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main> git init
Reinitialized existing Git repository in C:/Users/Renelyn Quiamco/Downloads/apiBoilerplate-main/.git/
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main> git clone https://github.com/Cdooii/user-management-system-frontend.git
Cloning into 'user-management-system-frontend'...
remote: Enumerating objects: 98, done.
remote: Counting objects: 100% (98/98), done.
remote: Compressing objects: 100% (73/73), done.
remote: Total 98 (delta 24), reused 87 (delta 21), pack-reused 0 (from 0)
Receiving objects: 100% (98/98), 236.17 KiB | 257.00 KiB/s, done.
Resolving deltas: 100% (24/24), done.
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main>
```

```
fatal: unable to auto-detect email address (got 'Renelyn Quiamco@LAPTOP-KNQOUKVA.(none)')
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main> git commit -m "Implement email sign-up, verification, and authentication"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.
```

```
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend> git init
Reinitialized existing Git repository in C:/Users/Renelyn Quiamco/Downloads/apiBoilerplate-main/user-management-system-frontend/.git/
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend> git clone https://github.com/Cdooii/user-management-system-frontend.g
it
Cloning into 'user-management-system-frontend'...
remote: Enumerating objects: 113, done.
remote: Counting objects: 100% (113/113), done.
remote: Compressing objects: 100% (88/88), done.
remote: Total 113 (delta 34), reused 85 (delta 21), pack-reused 0 (from 0)
Receiving objects: 100% (113/113), 2.29 MiB | 61.00 KiB/s, done.
Resolving deltas: 100% (34/34), done.
```

```
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend> git checkout -b backend-signup-auth
Switched to a new branch 'backend-signup-auth'
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend> git add .
warning: adding embedded git repository: user-management-system-frontend
hint: You've added another git repository inside your current repository.
hint: Clones of the outer repository will not contain the contents of
hint: the embedded repository and will not know how to obtain it.
hint: If you meant to add a submodule, use:
hint:
hint:   git submodule add <url> user-management-system-frontend
hint:
hint: If you added this path by mistake, you can remove it from the
```

```
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend> git push origin backend-signup-auth
info: please complete authentication in your browser...
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'backend-signup-auth' on GitHub by visiting:
remote:      https://github.com/Cdooii/user-management-system-frontend/pull/new/backend-signup-auth
remote:
To https://github.com/Cdooii/user-management-system-frontend.git
 * [new branch]      backend-signup-auth -> backend-signup-auth
PS C:\Users\Renelyn Quiamco\Downloads\apiBoilerplate-main\user-management-system-frontend>
```