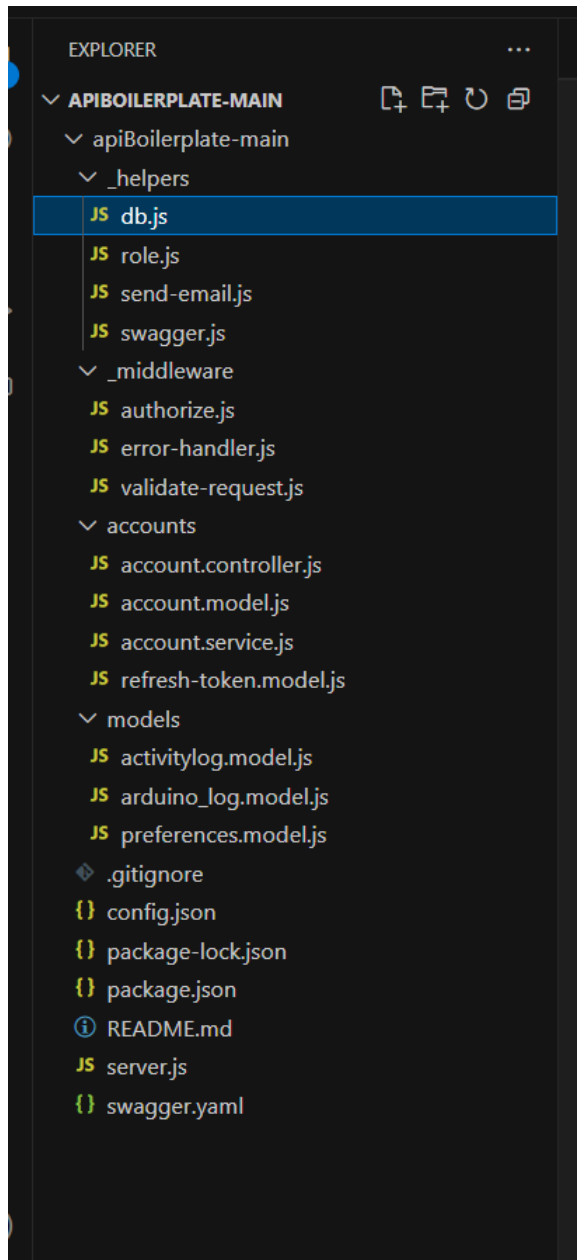


Group Project Activity: Full-Stack Application Development

Step 1: Create a procedure



Path: `/_helpers/db.js`

```
Boilerplate-main > _helpers > JS db.js > ...
1  const config = require('config.json');
2  const mysql = require('mysql2/promise');
3  const { Sequelize } = require('sequelize');
4
5  module.exports = db = {};
6
7  initialize();
8  async function initialize() {
9      const { host, port, user, password, database } = config.database;
10     const connection = await mysql.createConnection({ host, port, user, password });
11     await connection.query(`CREATE DATABASE IF NOT EXISTS \`${database}\`;`);
12
13     await connection.end();
14
15     const sequelize = new Sequelize(database, user, password, { host: 'localhost', dialect: 'mysql' });
16
17     // Initialize models and add them to the exported `db` object
18     db.Preferences = require('../models/preferences.model')(sequelize);
19     db.Account = require('../accounts/account.model')(sequelize);
20     db.RefreshToken = require('../accounts/refresh-token.model')(sequelize);
21     db.ActivityLog = require('../models/activitylog.model')(sequelize);
22
23     db.Account.hasMany(db.RefreshToken, { foreignKey: 'AccountId', onDelete: 'CASCADE' });
24     db.RefreshToken.belongsTo(db.Account, { foreignKey: 'AccountId' });
25
26     db.ActivityLog.belongsTo(db.Account, { foreignKey: 'AccountId' });
27     db.Preferences.belongsTo(db.Account, { foreignKey: 'AccountId' });
28
29     await sequelize.sync({ alter: true });
30 }
31
```

Path: `/_helpers/role.js`

```
apiBoilerplate-main > _helpers > JS role.js > [?] <unknown>
1  module.exports = {
2    ...
3    Admin: 'Admin',
4    Staff: 'Staff',
5    User: 'User'
6  }
```

Path: `/_helpers/send-email.js`

```
JS refresh-token.model.js × JS validate-request.js JS error-handler.js JS authorize.js JS swagger.js
apiBoilerplate-main > _helpers > JS send-email.js > ...
1  const nodemailer = require('nodemailer');
2  const config = require('config.json');
3
4  module.exports = sendEmail;
5
6  async function sendEmail({ to, subject, html, from = config.emailFrom }) {
7    const transporter = nodemailer.createTransport(config.smtpOptions);
8    await transporter.sendMail({ from, to, subject, html});
9  }
```

Path: `/_helpers/swagger.js`

```
JS refresh-token.model.js X JS validate-request.js JS error-handler.js JS authorize.js JS swagger.js
apiBoilerplate-main > _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;
```

Authorize Middleware

Path: `/_middleware/authorize.js`

```
apiBoilerplate-main > _middleware > JS authorize.js > ...
1  const { expressjwt: jwt } = require('express-jwt');
2  const { secret } = require('config.json');
3  const db = require('_helpers/db');
4  const Role = require('_helpers/role');
5
6  module.exports = authorize;
7
8  function authorize(roles = []) {
9    // Convert single role to array if string is provided
10   if (typeof roles === 'string') {
11     roles = [roles];
12   }
13
14   return [
15     // Authenticate JWT token and attach decoded token to request as req.auth
16     jwt({
17       secret,
18       algorithms: ['HS256'],
19       requestProperty: 'auth'
20     }),
21
22     // Authorize based on user role
23     async (req, res, next) => {
24       try {
25         const account = await db.Account.findByPk(req.auth.AccountId);
26         if (!account) {
27           return res.status(401).json({ message: 'Account no longer exists' });
28         }
29         if (roles.length && !roles.includes(account.role)) {
30           return res.status(401).json({ message: 'Unauthorized - Insufficient role permissions' });
31         }
32
33         // authentication and authorization successful
34         // attach user and role to request object
35         req.user = {
36           ...req.auth,
37           role: account.role,
```

```

8 function authorize(roles = []) {
23   async (req, res, next) => {
37     role: account.role,
38     BranchId: account.BranchId // Make sure this is being set correctly
39   };
40
41   // Add method to check if user owns a refresh token
42   const refreshTokens = await account.getRefreshTokens();
43   req.user.ownsToken = token => !!refreshTokens.find(x => x.token === token);
44
45   // Log authorization attempt
46   console.log(`Authorization successful for user ${account.email} with role ${account.role}`);
47
48   next();
49 } catch (error) {
50   console.error('Authorization error:', error);
51   return res.status(500).json({
52     success: false,
53     message: 'Internal server error during authorization'
54   });
55 }
56 }
57 ];
58 }

```

Path: `/_middleware/error-handler.js`

```

apiBoilerplate-main > _middleware > JS error-handler.js > [?] <unknown>
1 module.exports = errorHandler;
2
3 function errorHandler(err, req, res, next) {
4   switch (true) {
5     case typeof err === 'string':
6       const is404 = err.toLowerCase().endsWith('not found');
7       const statusCode = is404 ? 404 : 400;
8       return res.status(statusCode).json({ message: err });
9     case err.message && err.message.toLowerCase().includes('deactivated'):
10      return res.status(403).json({ message: 'deactivated' });
11     case err.name === 'UnauthorizedError':
12      return res.status(401).json({ message: 'Unauthorized error-handler' });
13     default:
14      return res.status(500).json({ message: err.message });
15   }
16 }
17

```

Path: `/_middleware/validate-request.js`

```
apiBoilerplate-main > _middleware > JS validate-request.js > [?] <unknown>
1  module.exports = validateRequest;
2
3  function validateRequest(req, next, schema) {
4      const options = {
5          abortEarly: false,
6          allowUnknown: true,
7          stripUnknown: true
8      };
9      const { error, value } = schema.validate(req.body, options);
10     if (error) {
11         next(`Validation error: ${error.details.map(x => x.message).join(', ')}');
12     } else {
13         req.body = value;
14         next();
15     }
16 }
17
```

Sequelize Account Model

Path: `/accounts/account.model.js`

```
JS account.controller.js JS preferences.model.js JS account.model.js X JS account.service.js JS refresh-token.model.js JS validate-request.js JS er
apiBoilerplate-main > accounts > JS account.model.js > [?] model
1  const { DataTypes } = require('sequelize');
2
3  module.exports = model;
4
5  function model(sequelize) {
6      const attributes = {
7          AccountId: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
8          email: { type: DataTypes.STRING, allowNull: false },
9          passwordHash: { type: DataTypes.STRING, allowNull: false },
10         title: { type: DataTypes.STRING, allowNull: false },
11         firstName: { type: DataTypes.STRING, allowNull: false },
12         lastName: { type: DataTypes.STRING, allowNull: false },
13         acceptTerms: { type: DataTypes.BOOLEAN },
14         role: { type: DataTypes.STRING, allowNull: false },
15         verificationToken: { type: DataTypes.STRING },
16         verified: { type: DataTypes.DATE },
17         resetToken: { type: DataTypes.STRING },
18         resetTokenExpires: { type: DataTypes.DATE },
19         passwordReset: { type: DataTypes.DATE },
20         created: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },
21         updated: { type: DataTypes.DATE },
22         isVerified: {
23             type: DataTypes.VIRTUAL,
24             get() { return !!this.verified || this.passwordReset; }
25         }
26     };
27
28     const options = {
29         timestamps: false,
30         defaultScope: {
31             attributes: { exclude: ['passwordHash'] }
32         },
33         scopes: {
34             withHash: { attributes: {}, }
35         }
36     };
37 }
```

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

```
Boilerplate-main > accounts > JS refresh-token.model.js > ...
1  const { DataTypes } = require('sequelize');
2
3  module.exports = model;
4
5  function model(sequelize) {
6    const attributes = {
7      refreshTokenId: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
8      token: { type: DataTypes.STRING },
9      expires: { type: DataTypes.DATE },
10     created: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },
11     createdByIp: { type: DataTypes.STRING },
12     revoked: { type: DataTypes.DATE },
13     revokedByIp: { type: DataTypes.STRING },
14     replacedByToken: { type: DataTypes.STRING },
15     AccountId: { type: DataTypes.INTEGER, allowNull: false }, // Make this field required
16     isExpired: {
17       type: DataTypes.VIRTUAL,
18       get() { return Date.now() >= this.expires; }
19     },
20     isActive: {
21       type: DataTypes.VIRTUAL,
22       get() { return !this.revoked && !this.isExpired; }
23     }
24   };
25
26   const options = {
27     timestamps: false
28   };
29
30   return sequelize.define('refreshToken', attributes, options);
31 }
```

Path: /accounts/account.service.js

```
apiBoilerplate-main > accounts > JS account.service.js > refreshToken
1  const config = require('config.json');
2  const jwt = require('jsonwebtoken');
3  const bcrypt = require('bcryptjs');
4  const crypto = require('crypto');
5  const { Op } = require('sequelize');
6  const sendEmail = require('_helpers/send-email');
7  const db = require('_helpers/db');
8  const Role = require('_helpers/role');
9
10 module.exports = {
11   authenticate,
12   refreshToken,
13   revokeToken,
14   register,
15   verifyEmail,
16   forgotPassword,
17   validateResetToken,
18   resetPassword,
19   getAll,
20   getById,
21   create,
22   logActivity,
23   getAccountActivities,
24   getAllActivityLogs,
25   update,
26   updatePreferences,
27   getPreferences,
28   delete: _delete
29 };
30
31 async function authenticate({ email, password, ipAddress, browserInfo }) {
32   const account = await db.Account.scope('withHash').findOne({ where: { email } });
33
34   if (!account || !account.isVerified || !(await bcrypt.compare(password, account.passwordHash))) {
35     throw 'Email or password is incorrect';
36   }
37 }
```

```
39   const refreshToken = generateRefreshToken(account, ipAddress);
40
41   await refreshToken.save();
42
43   try {
44     await logActivity(account.AccountId, 'login', ipAddress, browserInfo);
45   } catch (error) {
46     console.error('Error logging activity:', error);
47   }
48
49   return {
50     ...basicDetails(account),
51     jwtToken,
52     refreshToken: refreshToken.token
53   };
54 }
55
56 async function logActivity(AccountId, actionType, ipAddress, browserInfo, updateDetails = '') {
57   try {
58     // Create a new log entry in the 'activity_log' table
59     await db.ActivityLog.create({
60       AccountId,
61       actionType,
62       actionDetails: `IP Address: ${ipAddress}, Browser Info: ${browserInfo}, Details: ${updateDetails}`,
63       timestamp: new Date()
64     });
65
66     // Count the number of logs for the user
67     const logCount = await db.ActivityLog.count({ where: { AccountId } });
68
69     if (logCount > 10) {
70       // Find and delete the oldest logs
71       const logsToDelete = await db.ActivityLog.findAll({
72         where: { AccountId },
73         order: [['timestamp', 'ASC']],
74         limit: logCount - 10
75       });
76     }
77   } catch (error) {
78     console.error('Error logging activity:', error);
79   }
80 }
```



```

6   if (logsToDelete.length > 0) {
7       const logIdsToDelete = logsToDelete.map(log => log.activityLogId);
8
9       await db.ActivityLog.destroy({
10          where: {
11              activityLogId: {
12                  [Op.in]: logIdsToDelete
13              }
14          }
15      });
16      console.log(`Deleted ${logIdsToDelete.length} oldest log(s) for user ${AccountId}.`);
17  }
18  }
19  } catch (error) {
20      console.error('Error logging activity:', error);
21      throw error;
22  }
23  }
24  // Add this new service function
25  async function getAllActivityLogs(filters = {}) {
26      try {
27          let whereClause = {};
28
29          // Apply filters
30          if (filters.actionType) {
31              whereClause.actionType = { [Op.like]: `>${filters.actionType}%` };
32          }
33
34          if (filters.userId) {
35              whereClause.AccountId = filters.userId;
36          }
37
38          if (filters.startDate || filters.endDate) {
39              const startDate = filters.startDate ? new Date(filters.startDate) : new Date(0);
40              const endDate = filters.endDate ? new Date(filters.endDate) : new Date();
41              whereClause.timestamp = { [Op.between]: [startDate, endDate] };

```

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

```

33
34  // Get all activity logs with user details
35  const logs = await db.ActivityLog.findAll({
36      where: whereClause,
37      include: [{
38          model: db.Account,
39          attributes: ['email', 'firstName', 'lastName', 'role'],
40          required: true
41      }],
42      order: [['timestamp', 'DESC']]
43  });
44
45  // Format the response
46  return logs.map(log => {
47      const formattedDate = new Intl.DateTimeFormat('en-US', {
48          year: 'numeric',
49          month: '2-digit',
50          day: '2-digit',
51          hour: '2-digit',
52          minute: '2-digit',
53          hour12: true
54      }).format(new Date(log.timestamp));
55
56      return {
57          activityLogId: log.activityLogId,
58          userId: log.AccountId,
59          userEmail: log.Account.email,
60          userRole: log.Account.role,
61          userName: `${log.Account.firstName} ${log.Account.lastName}`,
62          actionType: log.actionType,
63          actionDetails: log.actionDetails,
64          timestamp: formattedDate
65      };
66  });
67  } catch (error) {
68      console.error('Error retrieving all activity logs:', error);

```

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

```

151 }
152 async function getAccountActivities(AccountId, filters = {}) {
153   const account = await getAccount(AccountId);
154   if (!account) throw new Error('User not found');
155
156   let whereClause = { AccountId };
157
158   // Apply optional filters such as action type and timestamp range
159   if (filters.actionType) {
160     whereClause.actionType = { [Op.like]: `%${filters.actionType}%` };
161   }
162   if (filters.startDate || filters.endDate) {
163     const startDate = filters.startDate ? new Date(filters.startDate) : new Date(0);
164     const endDate = filters.endDate ? new Date(filters.endDate) : new Date();
165     whereClause.timestamp = { [Op.between]: [startDate, endDate] };
166   }
167
168   try {
169     const activities = await db.ActivityLog.findAll({ where: whereClause });
170     return activities.map(activity => {
171       const formattedDate = new Intl.DateTimeFormat('en-US', {
172         year: '2-digit',
173         month: '2-digit',
174         day: '2-digit',
175         hour: '2-digit',
176         minute: '2-digit',
177         hour12: true
178       }).format(new Date(activity.timestamp));
179
180       return {
181         activityLogId: activity.activityLogId,
182         AccountId: activity.AccountId,
183         actionType: activity.actionType,
184         actionDetails: activity.actionDetails,
185         timestamp: formattedDate // Replace raw timestamp with formatted date
186       };
187     });
188   }

```

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

```

187   });
188   } catch (error) {
189     console.error('Error retrieving activities:', error);
190     throw new Error('Error retrieving activities');
191   }
192 }
193 async function refreshToken({ token, ipAddress }) {
194   const refreshToken = await getRefreshToken(token);
195   const account = await refreshToken.getAccount();
196
197   const newRefreshToken = generateRefreshToken(account, ipAddress);
198   refreshToken.revoked = Date.now();
199   refreshToken.revokedByIp = ipAddress;
200   refreshToken.replacedByToken = newRefreshToken.token;
201   await refreshToken.save();
202   await newRefreshToken.save();
203
204   const jwtToken = generateJwtToken(account);
205
206   return {
207     ...basicDetails(account),
208     jwtToken,
209     refreshToken: newRefreshToken.token
210   };
211 }
212 async function revokeToken({ token, ipAddress }) {
213   const refreshToken = await getRefreshToken(token);
214
215   refreshToken.revoked = Date.now();
216   refreshToken.revokedByIp = ipAddress;
217   await refreshToken.save();
218 }
219 async function register(params, origin) {
220   if (await db.Account.findOne({ where: { email: params.email } })) {
221     return await sendAlreadyRegisteredEmail(params.email, origin);
222   }

```

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

```

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;

```

```

58
59   account.resetToken = randomTokenString();
60   account.resetTokenExpires= new Date(Date.now() + 24*60*60*1000);
61   await account.save();
62
63   await sendPasswordResetEmail (account, origin);
64 }
65
66 async function validateResetToken({token}) {
67   const account = await db.Account.findOne({
68     where: {
69       resetToken: token,
70       resetTokenExpires: { [Op.gt]: Date.now() }
71     }
72   });
73
74   if (!account) throw 'Invalid token';
75
76   return account;
77 }
78
79 async function resetPassword({ token, password }, ipAddress, browserInfo) {
80   const account = await validateResetToken({ token });
81
82   // Add password validation if needed
83   if (password.length < 6) {
84     throw 'Password must be at least 6 characters';
85   }
86
87   account.passwordHash = await hash(password);
88   account.passwordReset = Date.now();
89   account.resetToken = null;
90   account.resetTokenExpires = null; // Clear the expiry
91   await account.save();
92
93   try {
94     await logActivity(account.AccountId, 'password_reset', ipAddress, browserInfo);
95   } catch (error) {

```

```

94     }
95
96     return;
97 }
98
99 async function getAll() {
100     const accounts = await db.Account.findAll();
101     return accounts.map(x => basicDetails(x));
102 }
103
104 async function getById(AccountId) {
105     const account = await getAccount(AccountId);
106     return basicDetails (account);
107 }
108
109 async function create(params) {
110     // Check if the email is already registered
111     const existingAccount = await db.Account.findOne({ where: { email: params.email } });
112     if (existingAccount) {
113         throw `Email "${params.email}" is already registered`;
114     }
115
116     const account = new db.Account(params);
117     account.verified = Date.now();
118     account.passwordHash = await hash(params.password);
119
120     // Save the account
121     await account.save();
122
123     // Create default preferences for the new user
124     await db.Preferences.create({
125         AccountId: account.AccountId,
126         theme: 'light',
127         notifications: true,
128         language: 'en'
129     });
130 }

```

```

31     return basicDetails(account);
32 }
33
34 async function update(AccountId, params, ipAddress, browserInfo) {
35     const account = await getAccount(AccountId);
36     const oldData = account.toJSON(); // Get current user data as a plain object
37     const updatedFields = []; // Declare updatedFields array
38     const nonUserFields = ['ipAddress', 'browserInfo'];
39
40     // Check if any meaningful changes are being made
41     const hasChanges = Object.keys(params).some(key =>
42         !nonUserFields.includes(key) &&
43         params[key] !== undefined &&
44         params[key] !== oldData[key]
45     );
46
47     if (!hasChanges) {
48         return basicDetails(account);
49     }
50
51     if (params.email && account.email !== params.email && await db.Account.findOne({ where: { email: params.email } })) {
52         throw `Email "${params.email}" is already taken`;
53     }
54
55     if (params.password) {
56         params.passwordHash = await hash(params.password);
57     }
58
59     for (const key in params) {
60         if (params.hasOwnProperty(key) && !nonUserFields.includes(key)) {
61             if (oldData[key] !== params[key]) {
62                 updatedFields.push(`${key}: ${oldData[key]} -> ${params[key]}`);
63             }
64         }
65     }
66
67     Object.assign(account, params);

```

```

    try {
      await account.save();

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

      await logActivity(account.AccountId, 'profile update', ipAddress || 'Unknown IP', browserInfo || 'Unknown Browser', updatedDetails);
    } 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

```

447
448 ✓    await sendEmail({
449      to: account.email,
450      subject: 'Sign-up Verification API - Verify Email',
451 ✓    html: `

#### Verify Email</h4> 452 <p>Thanks for registering!</p> 453 ${message}` 454 }); 455 } 456 ✓ async function sendAlreadyRegisteredEmail(email, origin) { 457 let message; 458 if (origin) { 459 message = ` 460 <p>If you don't know your password please visit the <a href="${origin}/account/forgot-password">forgot password</a> page.</p> 461 ✓ } else { message = ` 462 <p>If you don't know your password you can reset it via the <code>/account/forgot-password</code> api route.</p>`; 463 } 464 465 ✓ await sendEmail({ 466 to: email, 467 subject: 'Sign-up Verification API - Email Already Registered', 468 html: `Email Already Registered</h4> 469 <p>Your email <strong>${email}</strong> is already registered.</p> ${message}` 470 }); 471 } 472 ✓ async function sendPasswordResetEmail (account, origin) { 473 let message; 474 if (origin) { 475 const resetUrl = `${origin}/account/reset-password?token=${account.resetToken}`; 476 message = `


```

```

481    }
482
483    await sendEmail({
484      to: account.email,
485      subject: 'Sign-up Verification API - Reset Password',
486      html: `

#### Reset Password Email</h4> 487 ${message}` 488 }); 489 } 490


```

Path: /accounts/accounts.controller.js

```
apiBoilerplate-main > accounts > JS account.controller.js > createSchema > schema > password
1  const express = require('express');
2  const router = express.Router();
3  const Joi = require('joi');
4  const validateRequest = require('middleware/validate-request');
5  const authorize = require('middleware/authorize');
6  const Role = require('helpers/role');
7  const accountService = require('./account.service');
8
9  router.post('/authenticate', authenticateSchema, authenticate);
10 router.post('/refresh-token', refreshToken);
11 router.post('/revoke-token', authorize(), revokeTokenSchema, revokeToken);
12 router.post('/register', registerSchema, register);
13 router.post('/verify-email', verifyEmailSchema, verifyEmail);
14 router.post('/forgot-password', forgotPasswordSchema, forgotPassword);
15 router.post('/validate-reset-token', validateResetTokenSchema, validateResetToken);
16 router.post('/reset-password', resetPasswordSchema, resetPassword);
17
18 router.get('/:preferenceId/preferences', authorize(), getPreferences);
19 router.put('/:preferenceId/preferences', authorize(), updatePreferences);
20
21 router.get('/', authorize(Role.Admin), getAll);
22 router.post('/:Accountid/activity', authorize(), getActivities);
23 router.get('/activity-logs', authorize(Role.Admin), getAllActivityLogs);
24 router.get('/:Accountid', authorize(), getById);
25 router.post('/', authorize(Role.Admin), createSchema, create);
26 router.put('/:Accountid', authorize(), updateSchema, update);
27 router.delete('/:Accountid', authorize(), _delete);
28
29 module.exports = router;
30
31 function authenticateSchema(req, res, next) {
32   const schema = Joi.object({
33     email: Joi.string().required(),
34     password: Joi.string().required()
35   });
36   validateRequest(req, next, schema);
37 }
```

```
39
40 function authenticate(req, res, next) {
41   const { email, password } = req.body;
42   const ipAddress = req.headers['x-forwarded-for'] || req.connection.remoteAddress;
43   const browserInfo = req.headers['user-agent'] || 'Unknown Browser';
44
45   accountService.authenticate({ email, password, ipAddress, browserInfo })
46     .then(({ refreshToken, ...account }) => {
47       setTokenCookie(res, refreshToken);
48       res.json(account);
49     })
50     .catch(next);
51 }
52 //=====Logging Function=====
53 function getActivities(req, res, next) {
54   const filters = {
55     actionType: req.query.actionType,
56     startDate: req.query.startDate,
57     endDate: req.query.endDate
58   };
59   accountService.getAccountActivities(req.params.id, filters)
60     .then(activities => res.json(activities))
61     .catch(next);
62 }
63 function getAllActivityLogs(req, res, next) {
64   const filters = {
65     actionType: req.query.actionType,
66     startDate: req.query.startDate,
67     endDate: req.query.endDate,
68     userId: req.query.userId
69   };
70
71   accountService.getAllActivityLogs(filters)
72     .then(logs => res.json({
73       success: true,
74       data: logs
75     }));
76 }
```

```

7 }
8 //=====Preferences Router Function=====
9 function getPreferences(req, res, next) {
10   accountService.getPreferences(req.params.id)
11     .then(preferences => res.json(preferences))
12     .catch(next);
13 }
14 function updatePreferences(req, res, next) {
15   accountService.updatePreferences(req.params.id, req.body)
16     .then(() => res.json({ message: 'Preferences updated successfully' }))
17     .catch(next);
18 }
19 function refreshToken (req, res, next) {
20   const token = req.cookies.refreshToken;
21   const ipAddress = req.ip;
22   accountService.refreshToken({ token, ipAddress })
23     .then(({refreshToken, ...account }) => {
24       setTokenCookie(res, refreshToken);
25       res.json(account);
26     })
27     .catch(next);
28 }
29 function revokeTokenSchema(req, res, next) {
30   const schema = Joi.object({
31     token: Joi.string().empty('')
32   });
33   validateRequest(req, next, schema);
34 }
35 function revokeToken (req, res, next) {
36   const token = req.body.token || req.cookies.refreshToken;
37   const ipAddress = req.ip;
38
39   if (!token) return res.status(400).json({ message: 'Token is required' });
40
41   if (!req.user.ownsToken (token) && req.user.role !== Role. Admin) {
42     return res.status(401).json({ message: 'Unauthorized' });
43   }

```

```

4   accountService.revokeToken({token, ipAddress })
5     .then(() => res.json({ message: 'Token revoked' })))
6     .catch(next);
7 }
8
9 function registerSchema(req, res, next) {
10   const schema = Joi.object({
11     title: Joi.string().required(),
12     firstName: Joi.string().required(),
13     lastName: Joi.string().required(),
14     email: Joi.string().email().required(),
15     password: Joi.string().min(6).required(),
16     confirmPassword: Joi.string().valid(Joi.ref('password')).required(),
17     acceptTerms: Joi.boolean().valid(true).required()
18   });
19   validateRequest(req, next, schema);
20 }
21
22 function register(req, res, next) {
23   accountService.register(req.body, req.get('origin'))
24     .then(() => res.json({ message: 'Registration successful, please check your email for verification instructions' })))
25     .catch(next);
26 }
27
28 function verifyEmailSchema(req, res, next) {
29   const schema = Joi.object({
30     token: Joi.string().required()
31   });
32   validateRequest(req, next, schema);
33 }
34
35 function verifyEmail(req, res, next) {
36   accountService.verifyEmail(req.body)
37     .then(() => res.json({ message: 'Verification successful, you can now login' })))
38     .catch(next);
39 }
40
41 function forgotPasswordSchema(req, res, next) {
42   const schema = Joi.object({

```



```

146 }
147 ✓ function forgotPasswordSchema(req, res, next) {
148   ✓ const schema = Joi.object({
149     email: Joi.string().email().required()
150   });
151   validateRequest(req, next, schema);
152 }
153 ✓ function forgotPassword(req, res, next) {
154   accountService.forgotPassword(req.body, req.get('origin'))
155     .then(() => res.json({ message: 'Please check your email for password reset instructions' })))
156     .catch(next);
157 }
158 ✓ function validateResetTokenSchema(req, res, next) {
159   ✓ const schema = Joi.object({
160     token: Joi.string().required()
161   });
162   validateRequest(req, next, schema);
163 }
164 ✓ function validateResetToken(req, res, next) {
165   accountService.validateResetToken(req.body)
166     .then(() => res.json({ message: 'Token is valid' })))
167     .catch(next);
168 }
169 ✓ function resetPasswordSchema(req, res, next) {
170   ✓ const schema = Joi.object({
171     token: Joi.string().required(),
172     password: Joi.string().min(6).required(),
173     confirmPassword: Joi.string().valid(Joi.ref('password')).required()
174   });
175   validateRequest(req, next, schema);
176 }
177 ✓ function resetPassword(req, res, next) {
178   const { token, password } = req.body;
179   const ipAddress = req.headers['x-forwarded-for'] || req.connection.remoteAddress;
180   const browserInfo = req.headers['user-agent'] || 'Unknown Browser';
181

```

```

    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 {
    // 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

```

250         res.json({
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     }
262     (parameter) req: any
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

```
Boilerplate-main > config.json > ...
1 {
2   "database": {
3     "host": "localhost",
4     "port": 3306,
5     "user": "root",
6     "password": "root",
7     "database": "api_backend_boilerplate"
8   },
9   "secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET, IT CAN BE ANY STRING",
10  "emailFrom": "info@node-mysql-signup-verification-api.com",
11  "smtpOptions": {
12    "host": "smtp.ethereal.email",
13    "port": 587,
14    "auth": {
15      "user": "bertrand75@ethereal.email",
16      "pass": "MS9a1DUpeCBAGTYS2g"
17    }
18  }
19 }
```

Path: /package.json

```
apiBoilerplate-main > package.json > ...
1 {
2   "name": "boilerplate-api-main",
3   "version": "1.0.0",
4   "main": "server.js",
5   "scripts": {
6     "test": "echo \\\"Error: no test specified\\\" && exit 1",
7     "start": "node server.js",
8     "dev": "nodemon server.js"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "description": "",
14  "dependencies": {
15    "bcryptjs": "^2.4.3",
16    "body-parser": "^1.20.3",
17    "cookie-parser": "^1.4.6",
18    "cors": "^2.8.5",
19    "dotenv": "^16.4.5",
20    "express": "^4.21.0",
21    "express-jwt": "^8.4.1",
22    "joi": "^17.13.3",
23    "jsonwebtoken": "^9.0.2",
24    "mysql2": "^3.11.3",
25    "nodemailer": "^6.9.15",
26    "rootpath": "^0.1.2",
27    "sequelize": "^6.37.3",
28    "swagger-ui-express": "^5.0.1",
29    "yamljs": "^0.3.0"
30  },
31  "devDependencies": {
32    "nodemon": "^3.1.7"
33  }
34 }
35
```

Server Startup File

Path: /server.js

```
apiBoilerplate-main > JS server.js > ...
1  require('rootpath')();
2  const express = require('express');
3  const app = express();
4  const cors = require('cors');
5  const errorHandler = require('_middleware/error-handler');
6  const path = require('path');
7  const bodyParser = require('body-parser');
8  const cookieParser = require('cookie-parser');
9  // Configure CORS once with specific options
10 // specify the frontend origin
11 // allow cookies and other things sent
12 app.use(cors({origin: 'http://localhost:4200', credentials: true }));
13
14 app.use(express.json());
15 app.use(express.urlencoded({ extended: true }));
16 app.use(cors());
17
18 app.use(bodyParser.urlencoded({ extended: false }));
19 app.use(bodyParser.json());
20 app.use(cookieParser());
21
22 app.use(express.static(path.join(__dirname, 'products')));
23
24 app.use(cors({ origin: (origin, callback) => callback(null, true), credentials: true }));
25 app.use('/accounts', require('./accounts/account.controller'));
26 app.use('/api-docs', require('./_helpers/swagger'));
27
28 app.use(errorHandler);
29
30 const port = process.env.NODE_ENV === 'production' ? (process.env.PORT || 80) : 4000;
31 app.listen(port, () => console.log('Server listening on port ' + port));
```

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.git
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> █
```

