/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/index.js

```
require('dotenv').config();
console.log(process.env);

console.log(process.env.DB_HOST); // 'localhost'
console.log(process.env.DB_USER); // 'root'
console.log(process.env.DB_PASSWORD); // 's1mpl3'
const express = require('express');
const logger = require('./src/util/utils').logger;
const userRoutes = require('./src/routes/user.routes');
const authRoutes = require('./src/routes/auth.routes');

const app = express();
const port = process.env.PORT || 3000;


// For access to application/json request body
app.use(express.json());

// Algemene route, vangt alle http-methods en alle URLs af, print
// een message, en ga naar de next URL (indien die matcht)!
app.use('*', (req, res, next) => {
  const method = req.method;
  logger.trace(`Methode ${method} is aangeroepen`);
  next();
});

// Info endpoints
app.get('/api/info', (req, res) => {
  logger.info('Get server information');
  res.status(201).json({
    status: 201,
    message: 'Server info-endpoint',
    data: {
      studentName: 'Miquel',
      studentNumber: 2159021,
      description: 'Welkom bij de server API van de share a meal.'
    }
  });
});

// Hier staan de referenties naar de routes
app.use('/api/user', userRoutes);

app.use('/api/', authRoutes);


// Wanneer geen enkele endpoint matcht kom je hier terecht. Dit is dus
// een soort 'afvoerputje' (sink) voor niet-bestaande URLs in de server.
app.use('*', (req, res) => {
  logger.warn('Invalid endpoint called: ', req.path);
  res.status(404).json({
    status: 404,
    message: 'Endpoint not found',
    data: {}
  });
});

// Express error handler
app.use((err, req, res, next) => {
  logger.error(err.code, err.message);
  res.status(err.code).json({
    statusCode: err.code,
    message: err.message,
    data: {}
  });
});

// Start de server
app.listen(port, () => {
  logger.info(`Share-a-Meal server listening on port ${port}`);
});

// Export de server zodat die in de tests beschikbaar is.
module.exports = app;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/controllers/authentication.controller.js

```
//
// Authentication controller
//
const assert = require('assert');
const jwt = require('jsonwebtoken');
const pool = require('../util/mysql-db');
const { logger, jwtSecretKey } = require('../util/utils');
const bcrypt = require('bcryptjs');

module.exports = {
  login(req, res, next) {
    const { emailAdress, password } = req.body;
    console.log("login called")

    pool.getConnection()
      .then(connection => {
```

```
      console.log("login calleddd")
      connection.query('SELECT * FROM `user` WHERE emailAdress = ?', [emailAdress])
        .then(([users]) => {
          if (users.length === 0) {
            return res.status(401).send({ error: "Not Authorized" });
          }
          const user = users[0];
          return bcrypt.compare(password, user.password)
            .then(passwordMatch => {
              if (!passwordMatch) {
                return res.status(401).send({ error: "Not Authorized" });
              }
              const payload = { userId: user.id };
              const token = jwt.sign(payload, jwtSecretKey);
              res.send({ token });
            });
        })
        .catch(error => {
          next({
            code: 500,
            message: error.message
          });
        })
        .finally(() => {
          connection.release();
        });
    })
    .catch(err => {
      logger.error('Error getting connection from pool');
      next({
        code: 500,
        message: err.code
      });
    });
  },

  validateToken(req, res, next) {
    logger.trace('validateToken called');
    const authHeader = req.headers.authorization;
    if (!authHeader) {
      next({
        code: 401,
        message: 'Authorization header missing!',
        data: undefined
      });
    } else {
      try {
        const token = authHeader.split(' ')[1];
        const decoded = jwt.verify(token, jwtSecretKey);
        req.userId = decoded.userId;
        next();
      } catch (error) {
        next({
          code: 401,
          message: 'Invalid token',
          data: undefined
        });
      }
    }
  }
};
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/controllers/user.controller.js

```
const database = require('../util/inmem-db');
const logger = require('../util/utils').logger;
const assert = require('assert');
const pool = require('../util/mysql-db');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const saltRounds = 10;
const validator = require('validator');

const userController = {
  createUser: async (req, res, next) => {
    const { emailAdress, password, firstName, lastName, street, city } = req.body;

    // Validate the incoming data
    if (!emailAdress || !password) {
      return res.status(400).send({ error: "Email and password are required" });
    }
    if (!validator.isEmail(emailAdress)) {
      return res.status(400).send({ error: "Invalid emailAdress format" });
    }
    if (password.length < 8) {
      return res.status(400).send({ error: "Password must be at least 8 characters long" });
    }

    try {
      const hashedPassword = await bcrypt.hash(password, saltRounds);

      const conn = await pool.getConnection();
      try {
        const [rows] = await conn.query('SELECT emailAdress FROM user WHERE emailAdress = ?', [emailAdress]);
        if (rows.length > 0) {
          return res.status(409).send({ error: "Email already exists" });
        } else {
          const [result] = await conn.query('INSERT INTO user (firstname, lastname, emailAdress, password, street, city) VALUES (?, ?, ?, ?, ?, ?)', [fi
          // Return the data and identification number of the added user
          return res.send({ message: `Registered emailAdress ${emailAdress}`, data: { id: result.insertId, emailAdress } });
        }
      } catch (error) {
        logger.error(error.message);
```

```
      return next({
        code: 500,
        message: 'Database error'
      });
    } finally {
      conn.release();
    }
  } catch (error) {
    logger.error(error.message);
    return next({
      code: 500,
      message: 'Internal server error'
    });
  }
},
getAllUsers: async (req, res, next) => {
  logger.info('Get all users');

  let sqlStatement = 'SELECT * FROM `user`';
  // Handle the query parameters
  const queryParams = req.query;
const validFields = ["id", "firstName", "lastName", "isActive", "emailAdress", "password", "phoneNumber", "roles", "street", "city"];

  const sqlParams = [];
  let isFirst = true;
  for (const key in queryParams) {
    // Check if the query parameter is a valid field
    if (validFields.includes(key)) {
      // Check if the query parameter value is not empty or null
      if (queryParams[key]) {
        // Add the SQL code to filter by the query parameter
        if (isFirst) {
          sqlStatement += " WHERE `" + key + "`=?";
          isFirst = false;
        } else {
          sqlStatement += " AND `" + key + "`=?";
        }
        // Add the query parameter value to the sqlParams array
        sqlParams.push(queryParams[key]);
      }
    }
  }

  try {
    const conn = await pool.getConnection();
    try {
      const [results] = await conn.query(sqlStatement, sqlParams);
      logger.info('Found', results.length, 'results');
      return res.status(200).json({
        code: 200,
        message: 'User getAll endpoint',
        data: results
      });
    } catch (error) {
      logger.error(error.message);
      return next({
        code: 409,
        message: error.message
      });
    } finally {
      conn.release();
    }
  } catch (err) {
    logger.error(err.code, err.syscall, err.address, err.port);
    return next({
      code: 500,
      message: err.code
    });
  }
},

getUserProfile: async (req, res, next) => {
  const userId = req.userId;
  logger.trace('Get user profile for user', userId);
  console.log(userId);

  let sqlStatement = 'SELECT * FROM `user` WHERE id=?';

  try {
    const conn = await pool.getConnection();
    try {
      const [results] = await conn.query(sqlStatement, [userId]);
      // Check if the user id is found
      if (results.length === 0) {
        return res.status(404).send({ error: "User not found" });
      }
      // Look up the details of the associated meals taking place today or in the future
      const [meals] = await conn.query('SELECT * FROM `meal` WHERE id=?', [userId]);
      logger.trace('Found', results.length, 'results');
      return res.status(200).json({
        code: 200,
        message: 'Get User profile',
        data: { ...results[0], meals }
      });
    } catch (error) {
      logger.error(error.message);
      return next({
        code: 409,
        message: error.message
      });
    } finally {
      conn.release();
    }
  } catch (err) {
    logger.error(err.code, err.syscall, err.address, err.port);
    return next({
      code: 500,
      message: err.code
    });
  }
},
```

```
getUserById: async (req, res, next) => {
  // Get the user id from the request parameters
  const userId = req.params.id ;

  let sqlStatement = 'SELECT * FROM `user` WHERE id=?';

  try {
    const conn = await pool.getConnection();
    try {
      const [results] = await conn.query(sqlStatement, [userId]);
      // Check if the user id is found
      if (results.length === 0) {
        return res.status(404).send({ error: "User not found" });
      }
      // Look up the details of the associated meals taking place today or in the future
      const [meals] = await conn.query('SELECT * FROM `meal` WHERE Id=1 ', [userId]);
      return res.status(200).json({
        code: 200,
        message: 'Get User by id',
        data: { ...results[0], meals }
      });
    } catch (error) {
      logger.error(error.message);
      return next({
        code: 409,
        message: error.message
      });
    } finally {
      conn.release();
    }
  } catch (err) {
    logger.error(err.code, err.syscall, err.address, err.port);
    return next({
      code: 500,
      message: err.code
    });
  }
},

updateUser: async (req, res, next) => {
  // Get the user id from the request parameters
  const userId = req.params.id;

  // Check if the user making the request is the same as the user being updated
  if (req.user !== userId) {
    return res.status(403).send({ error: "You can only update your own data" });
  }
 // Extract the necessary fields from the request body
  const { emailAdress,password, firstName, lastName, phoneNumber, isActive, roles, street, city } = req.body;

  if (!emailAdress) {
      return res.status(400).send({ error: "Email is required" });
  }
  if (!validator.isEmail(emailAdress)) {
      return res.status(400).send({ error: "Invalid emailAdress format" });
  }
  if (password && password.length < 8) {
      return res.status(400).send({ error: "Password must be at least 8 characters long" });
  }
  if (phoneNumber && !validator.isMobilePhone(phoneNumber)) {
      return res.status(400).send({ error: "Invalid phone number" });
  }

  let sqlStatement = 'UPDATE `user` SET emailAdress=?, password=?, firstName=?, lastName=?, phoneNumber=?, isActive=?, roles=?, street=?, city=? WHERE
  let sqlParams = [emailAdress, password, firstName, lastName, phoneNumber, isActive, roles, street, city, userId];

  try {
      const conn = await pool.getConnection();
      try {
          const [user] = await conn.query('SELECT * FROM `user` WHERE id=?', [userId]);
          if (user.length === 0) {
              return res.status(404).send({ error: "User not found" });
          }

          const [emailAdressCheck] = await conn.query('SELECT * FROM `user` WHERE emailAdress=? AND id<>?', [emailAdress, userId]);
          if (emailAdressCheck.length > 0) {
              return res.status(409).send({ error: "Email already exists" });
          }

          if (password) {
              const hashedPassword = await bcrypt.hash(password, saltRounds);
              sqlStatement = 'UPDATE `user` SET emailAdress=?, password=?, firstName=?, lastName=?, phoneNumber=?, isActive=?, roles=?, street=?, city
              sqlParams = [emailAdress, hashedPassword, firstName, lastName, phoneNumber, isActive, roles, street, city, userId];
          }

          const [result] = await conn.query(sqlStatement, sqlParams);

          return res.status(200).json({
              code: 200,
              message: 'Update User',
              data: { id: userId, emailAdress, phoneNumber, isActive, roles, street, city }
          });

      } catch (error) {
          logger.error(error.message);
          return next({
              code: 409,
              message: error.message
          });

      } finally {
          conn.release();
      }
  } catch (err) {
      logger.error(err.code, err.syscall, err.address, err.port);
      return next({
          code: 500,
          message: err.code
      });
  }
},
```

```
  deleteUser: async (req, res, next) => {
    // Get the user id from the request parameters
    const userId = req.params.id;

    // Check if the user making the request is the same as the user being deleted
    if (req.user !== userId) {
      return res.status(403).send({ error: "You can only delete your own data" });
    }
    let sqlStatement = 'DELETE FROM `user` WHERE id=?';

    try {
      const conn = await pool.getConnection();

      try {
        // Check if the user id is found
        const [user] = await conn.query('SELECT * FROM `user` WHERE id=?', [userId]);
        if (user.length === 0) {
          return res.status(404).send({ error: "User not found" });
        }
        // Delete the user data
        let sqlUpdateStatement = 'UPDATE `meal` SET `cookId`=NULL WHERE `cookId`=?';
        await conn.query(sqlUpdateStatement, [userId]);

        const [result] = await conn.query(sqlStatement, [userId]);
        // Return a confirmation message
        return res.status(200).json({
          code: 200,
          message: 'Delete User',
          data: { id: userId }
        });
      } catch (error) {
        logger.error(error.message);
        return next({
          code: 409,
          message: error.message
        });
      } finally {
        conn.release();
      }
    } catch (err) {
      logger.error(err.code, err.syscall, err.address, err.port);
      return next({
        code: 500,
        message: err.code
      });
    }
  },
};

module.exports = userController;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/routes/auth.routes.js

```
const express = require('express');
const router = express.Router();
const authController = require('../controllers/authentication.controller');

// Route for user login
router.post('/login', authController.login);

// Route for token validation
router.get('/validate-token', authController.validateToken, (req, res) => {
  // If the token is valid, the middleware will allow the request to reach this point
  // You can add additional logic here if needed
  res.status(200).json({ message: 'Token is valid' });
});

module.exports = router;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/routes/inf.routes.js

---

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/routes/user.routes.js

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/user.controller');
const authController = require('../controllers/authentication.controller');

// UC-201 Registreren als nieuwe user
router.post('', userController.createUser);

// UC-202 Opvragen van overzicht van users
router.get('', authController.validateToken, userController.getAllUsers);

// UC-203 Haal het userprofile op van de user die ingelogd is
router.get(
  '/profile',
```

```
  authController.validateToken,
  userController.getUserProfile
);

// UC-204 Opvragen van usergegevens bij ID
router.get('/:id',
authController.validateToken,
userController.getUserById);

// UC-205 Wijzigen van usergegevens
router.put('/:id',
 authController.validateToken,
 userController.updateUser);

// UC-206 Verwijderen van user
router.delete('/:id',
 authController.validateToken,
userController.deleteUser);

module.exports = router;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/util/inmem-db.js

```
// Onze lokale 'in memory database'. Later gaan we deze naar een
// aparte module (= apart bestand) verplaatsen.
let database = {
  users: [
    {
      id: 0,
      firstName: 'Hendrik',
      lastName: 'van Dam',
      emailAdress: 'hvd@server.nl'
      // Hier de overige velden uit het functioneel ontwerp
    },
    {
      id: 1,
      firstName: 'Marieke',
      lastName: 'Jansen',
      emailAdress: 'm@server.nl'
      // Hier de overige velden uit het functioneel ontwerp
    }
  ],

  // Ieder nieuw item in db krijgt 'autoincrement' index.
  // Je moet die wel zelf toevoegen!
  index: 2
};

module.exports = database;
// module.exports = database.index;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/util/mysql-db.js

```
const mysql = require('mysql2/promise');

const logger = require('../util/utils').logger;

// Create the connection pool. The pool-specific settings are the defaults
// const pool = mysql.createPool({
//   host: 'db-mysql-ams3-46626-do-user-8155278-0.b.db.ondigitalocean.com',
//   user: '2159021',
//   database: '2159021',
//   port: 25060,
//   password: 'secret',
//   waitForConnections: true,
//   connectionLimit: 10,
//   maxIdle: 10, // max idle connections, the default value is the same as `connectionLimit`
//   idleTimeout: 60000, // idle connections timeout, in milliseconds, the default value 60000
//   queueLimit: 0
// });

const pool = mysql.createPool({
  host: '127.0.0.1',
  user: 'root',
  database: 'shareameal',
  port: 3306,
  waitForConnections: true,
  connectionLimit: 10,
  maxIdle: 10, // max idle connections, the default value is the same as `connectionLimit`
  idleTimeout: 60000, // idle connections timeout, in milliseconds, the default value 60000
  queueLimit: 0
});

pool.on('connection', function (connection) {
  logger.info(
    `Connected to db '${connection.config.database}' on ${connection.config.host}`
  );
});

pool.on('acquire', function (connection) {
  logger.trace('Connection %d acquired', connection.threadId);
});

pool.on('release', function (connection) {
  logger.trace('Connection %d released', connection.threadId);
```

```
});

module.exports = pool;
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/src/util/utils.js

```
module.exports = {
  logger: require('tracer').console({
    level: process.env.LOGLEVEL || 'debug',
    format: '{{timestamp}} <{{title}}> {{message}} (in {{file}}:{{line}})',
    dateformat: 'HH:MM:ss.L',
    preprocess: function (data) {
      data.title = data.title.toUpperCase();
    }
  }),

  jwtSecretKey: process.env.JWT_SECRET || 'kljasdfoijqawtl,mnzfsg'
};
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/test/integration/info.test.js

```
process.env['DB_DATABASE'] = process.env.DB_DATABASE || 'shareameal-testdb';

const assert = require('assert');
const chai = require('chai');
const chaiHttp = require('chai-http');
const server = require('../../index');
require('tracer').setLevel('error');

chai.should();
chai.use(chaiHttp);

describe('UC-102 Informatie opvragen', function () {
  it('TC-102-1 - Server info should return succesful information', (done) => {
    chai
      .request(server)
      .get('/api/info')
      .end((err, res) => {
        res.body.should.be.an('object');
        res.body.should.has.property('status').to.be.equal(201);
        res.body.should.has.property('message');
        res.body.should.has.property('data');
        let { data, message } = res.body;
        data.should.be.an('object');
        data.should.has.property('studentName').to.be.equal('Miquel');
        data.should.has.property('studentNumber').to.be.equal(2159021);
        done();
      });
  });

  it('TC-102-2 - Server should return valid error when endpoint does not exist', (done) => {
    chai
      .request(server)
      .get('/api/doesnotexist')
      .end((err, res) => {
        assert(err === null);

        res.body.should.be.an('object');
        let { data, message, status } = res.body;

        status.should.equal(404);
        message.should.be.a('string').that.is.equal('Endpoint not found');
        data.should.be.an('object');

        done();
      });
  });
});
```

/Users/miquelstam/Library/CloudStorage/OneDrive-AvansHogeschool/Informatica Jaar 1/Periode 4/Programmeren 4/code/test/integration/user.db.test.js

```
process.env['DB_DATABASE'] = process.env.DB_DATABASE || 'shareameal-testdb';

const chai = require('chai');
const chaiHttp = require('chai-http');
const server = require('../../index');
const assert = require('assert');
const dbconnection = require('../../src/util/mysql-db');
const jwt = require('jsonwebtoken');
const { jwtSecretKey, logger } = require('../../src/util/utils');
require('tracer').setLevel('trace');

chai.should();
chai.use(chaiHttp);

/**
 * Db queries to clear and fill the test database before each test.
```

```
 *
 * LET OP: om via de mysql2 package meerdere queries in één keer uit te kunnen voeren,
 * moet je de optie 'multipleStatements: true' in de database config hebben staan.
 */
const CLEAR_MEAL_TABLE = 'DELETE IGNORE FROM `meal`;';
const CLEAR_PARTICIPANTS_TABLE = 'DELETE IGNORE FROM `meal_participants_user`;';
const CLEAR_USERS_TABLE = 'DELETE IGNORE FROM `user`;';
const CLEAR_DB =
  CLEAR_MEAL_TABLE + CLEAR_PARTICIPANTS_TABLE + CLEAR_USERS_TABLE;

/**
 * Voeg een user toe aan de database. Deze user heeft id 1.
 * Deze id kun je als foreign key gebruiken in de andere queries, bv insert meal.
 */
const INSERT_USER =
  'INSERT INTO `user` (`id`, `firstName`, `lastName`, `emailAdress`, `password`, `street`, `city` ) VALUES' +
  '(1, "first", "last", "name@server.nl", "secret", "street", "city");';

/**
 * Query om twee meals toe te voegen. Let op de cookId, die moet matchen
 * met een bestaande user in de database.
 */
const INSERT_MEALS =
  'INSERT INTO `meal` (`id`, `name`, `description`, `imageUrl`, `dateTime`, `maxAmountOfParticipants`, `price`, `cookId`) VALUES' +
  "(1, 'Meal A', 'description', 'image url', NOW(), 5, 6.50, 1)," +
  "(2, 'Meal B', 'description', 'image url', NOW(), 5, 6.50, 1);";

describe('Users API', () => {
  //
  // informatie over before, after, beforeEach, afterEach:
  // https://mochajs.org/#hooks
  //
  before((done) => {
    logger.trace(
      'before: hier zorg je eventueel dat de precondities correct zijn'
    );
    logger.trace('before done');
    done();
  });

  describe('UC-xyz [usecase beschrijving]', () => {
    //
    beforeEach((done) => {
      logger.trace('beforeEach called');
      // maak de testdatabase leeg zodat we onze testen kunnen uitvoeren.
      dbconnection.getConnection(function (err, connection) {
        if (err) {
          done(err);
          throw err; // no connection
        }
        // Use the connection
        connection.query(
          CLEAR_DB + INSERT_USER,
          function (error, results, fields) {
            if (error) {
              done(error);
              throw error; // not connected!
            }
            logger.trace('beforeEach done');
            // When done with the connection, release it.
            dbconnection.releaseConnection(connection);
            // Let op dat je done() pas aanroept als de query callback eindigt!
            done();
          }
        );
      });
    });

    it.skip('TC-201-1 Voorbeeld testcase, met POST, wordt nu geskipped', (done) => {
      chai
        .request(server)
        .post('/api/movie')
        .send({
          // name is missing
          year: 1234,
          studio: 'pixar'
        })
        .end((err, res) => {
          assert.ifError(err);
          res.should.have.status(401);
          res.should.be.an('object');

          res.body.should.be.an('object').that.has.all.keys('code', 'message');
          code.should.be.an('number');
          message.should.be.a('string').that.contains('error');
          done();
        });
    });

    it('TC-201-2 [naam van de test verder zelf aanvullen]', (done) => {
      // Zelf verder aanvullen
      done();
    });

    // En hier komen meer testcases
  });

  describe('UC-203 Opvragen van gebruikersprofiel', () => {
    //
    beforeEach((done) => {
      logger.trace('beforeEach called');
      // maak de testdatabase leeg zodat we onze testen kunnen uitvoeren.
      dbconnection.getConnection(function (err, connection) {
        if (err) {
          done(err);
          throw err; // no connection
        }
        // Use the connection
        connection.query(
          CLEAR_DB + INSERT_USER,
          function (error, results, fields) {
            if (error) {
```

```
            done(error);
            throw error; // not connected!
          }
          logger.trace('beforeEach done');
          // When done with the connection, release it.
          dbconnection.releaseConnection(connection);
          // Let op dat je done() pas aanroept als de query callback eindigt!
          done();
        }
      );
    });
  });

  it.skip('TC-203-1 Ongeldig token', (done) => {
    chai
      .request(server)
      .get('/api/user/profile')
      .set('authorization', 'Bearer hier-staat-een-ongeldig-token')
      .end((err, res) => {
        assert.ifError(err);
        res.should.have.status(401);
        res.should.be.an('object');

        res.body.should.be
          .an('object')
          .that.has.all.keys('code', 'message', 'data');
        let { code, message, data } = res.body;
        code.should.be.an('number');
        message.should.be.a('string').equal('Not authorized');
        done();
      });
  });

  it.skip('TC-203-2 Gebruiker ingelogd met geldig token', (done) => {
    // Gebruiker met id = 1 is toegevoegd in de testdatabase. We zouden nu
    // in deze testcase succesvol het profiel van die gebruiker moeten vinden
    // als we een valide token meesturen.
    chai
      .request(server)
      .get('/api/user/profile')
      .set('authorization', 'Bearer ' + jwt.sign({ userId: 1 }, jwtSecretKey))
      .end((err, res) => {
        assert.ifError(err);
        res.should.have.status(200);
        res.should.be.an('object');

        res.body.should.be
          .an('object')
          .that.has.all.keys('code', 'message', 'data');
        let { code, message, data } = res.body;
        code.should.be.an('number');
        message.should.be.a('string').that.contains('Get User profile');
        data.should.be.an('object');
        data.id.should.equal(1);
        data.firstName.should.equal('first');
        // Zelf de overige validaties aanvullen!
        done();
      });
  });
});

describe('UC-303 Lijst van maaltijden opvragen', () => {
  //
  beforeEach((done) => {
    logger.debug('beforeEach called');
    // maak de testdatabase opnieuw aan zodat we onze testen kunnen uitvoeren.
    dbconnection.getConnection(function (err, connection) {
      if (err) {
        done(err);
        throw err; // not connected!
      }
      connection.query(
        CLEAR_DB + INSERT_USER + INSERT_MEALS,
        function (error, results, fields) {
          // When done with the connection, release it.
          dbconnection.releaseConnection(connection);
          // Handle error after the release.
          if (err) {
            done(err);
            throw err;
          }
          // Let op dat je done() pas aanroept als de query callback eindigt!
          logger.debug('beforeEach done');
          done();
        }
      );
    });
  });

  it.skip('TC-303-1 Lijst van maaltijden wordt succesvol geretourneerd', (done) => {
    chai
      .request(server)
      .get('/api/meal')
      // wanneer je authenticatie gebruikt kun je hier een token meesturen
      // .set('authorization', 'Bearer ' + jwt.sign({ id: 1 }, jwtSecretKey))
      .end((err, res) => {
        assert.ifError(err);

        res.should.have.status(200);
        res.should.be.an('object');

        res.body.should.be
          .an('object')
          .that.has.all.keys('message', 'data', 'code');

        const { code, data } = res.body;
        code.should.be.an('number');
        data.should.be.an('array').that.has.length(2);
        data[0].name.should.equal('Meal A');
        data[0].id.should.equal(1);
        done();
      });
```

```
    });
    // En hier komen meer testcases
  });
});
```
