Workshop NodeJS for integration developers

Using NodeJS, Docker, MongoDB

SIG 28 augustus 2018

Contents

Prerequisites

For following this workshop you need the following:

- VirtualBox 5.2.12 or later (VM has been tested with 5.2.16)
- Course VM

If you do not have the course VM, you can either install (or use) the tools yourself or create a VM with Ubuntu 18.04 and install the tools in there. The tools to install are: Visual Studio Code, Docker, Node.js and Postman.

Make sure Virtualbox *5.2.12* or later is installed. Download and install it from www.virtualbox.org. If the version differs, the guest additions might not be able to communicate with the host OS. Issues you might encounter is that the bidirectional clipboard does not function.
The machine you're running the VM on requires at least 5Gb of available RAM and around 10Gb (preferably 30Gb) of free disk space.

Note that if you have Hyper-V enabled on your Windows 10 machine, you need to deactivate it before using Virtualbox.
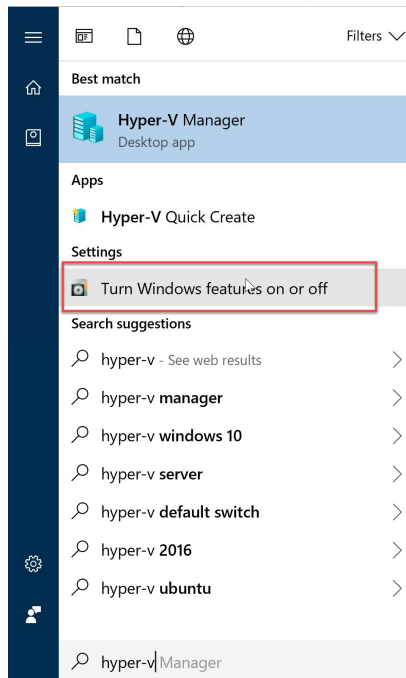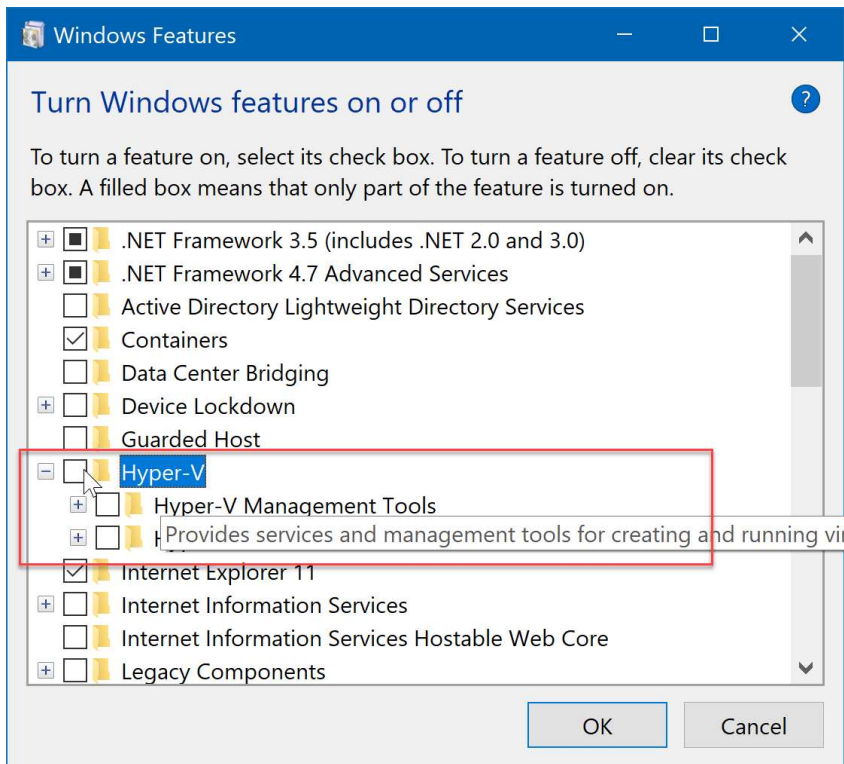
Your machine will reboot.

Importing the VM

Download the image file from: https://conclusionfutureit-my.sharepoint.com/:f:/g/personal/matthijs_breeman_amis_nl/EmkHnMTp0LVJigAoU8ZIZg0BQIljo_nDxTgaty_tPxyNNQ?e=kcJGqU *(2,7 GB)*

Import the OVA file in Virtualbox
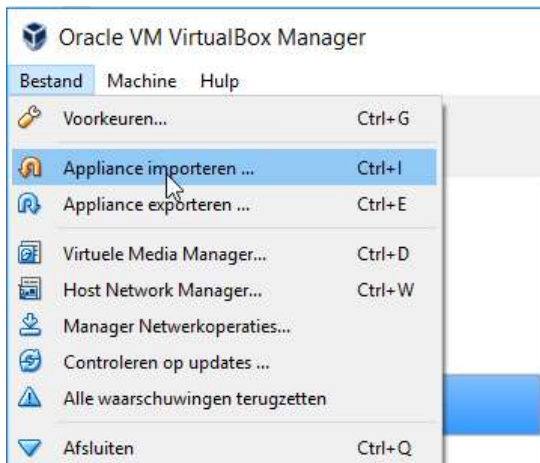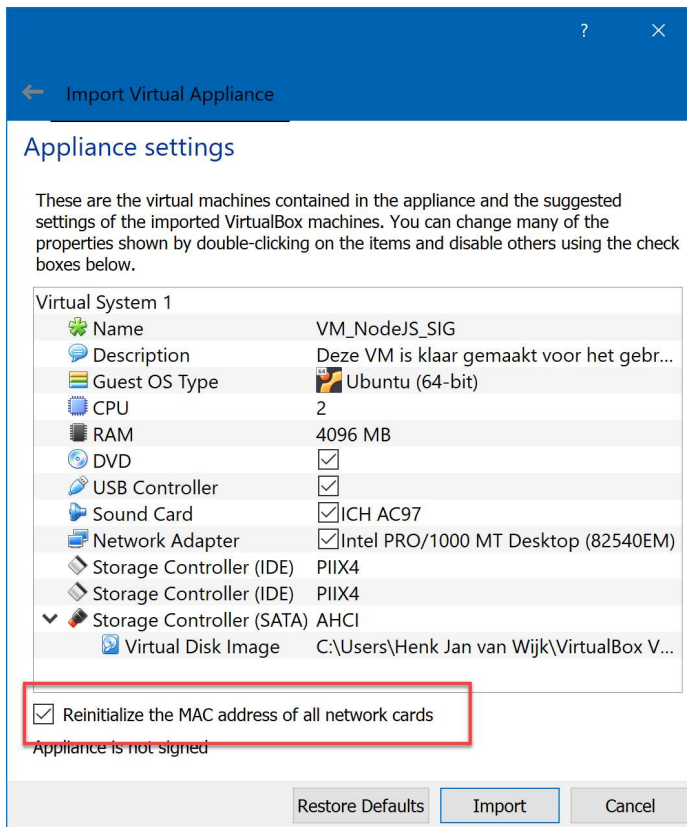
If you want, you can rename the VM by double clicking on the name.
Select '"**Reinitialize the MAC address of all network cards**" and press Import.

After import start Settings.
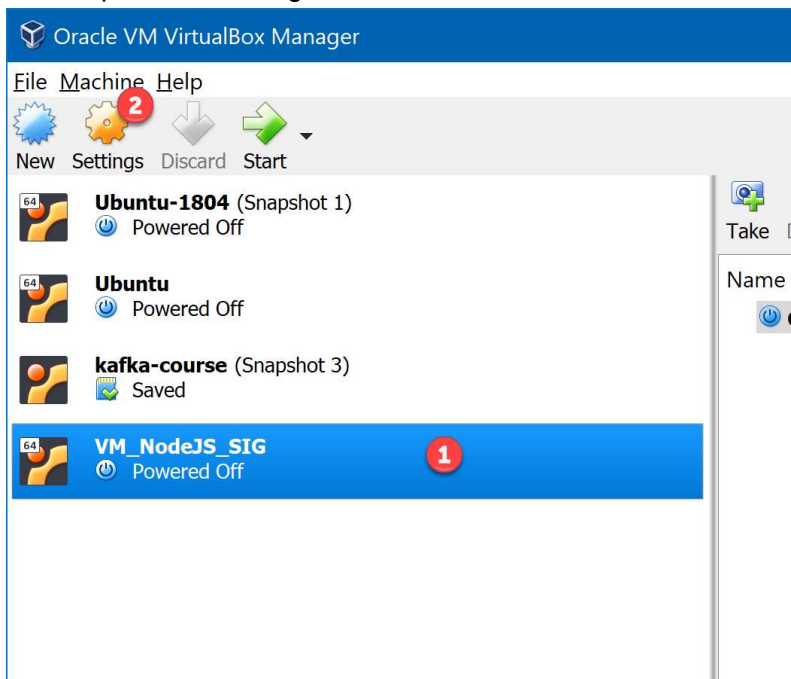
Check the network settings.

Set at least one of the network adapters to NAT so you will be able to connect to the internet from within the VM. You can either choose the Bridged Adapter or NAT.
At a certain point you will need internet access from within the VM. If the Brdiges Adapter does not work, you should switch to NAT (you can only change the settings when the VM is powered down).
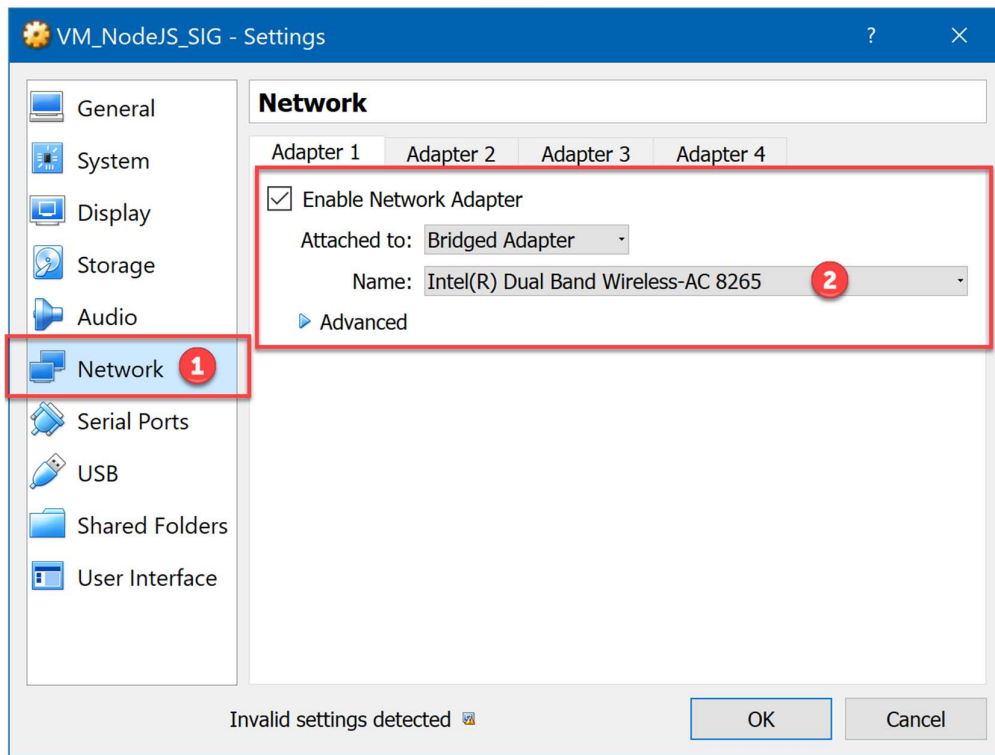
Start the VM
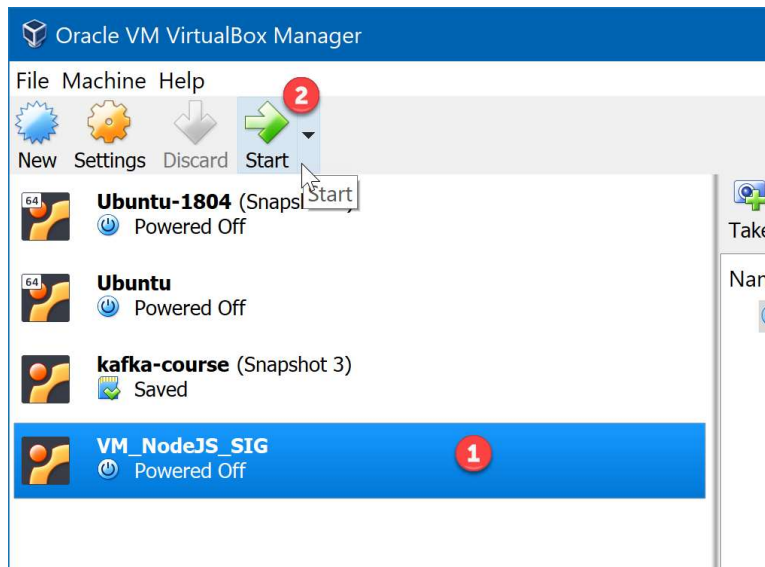
If you get the following error:

the network settings are not set correctly and check them again and make sure you select and appropriate network adapter which exists on your machine.

Introducing the VM

The VM will auto login with user developer and password **Welcome01**.
If at any time there is a request for a password, Welcome01 is the answer.

Set the shared clipboard to bidirectional to be able to copy & paste from and to the VM.
Use `ctrl+shift+v` to paste the clipboard in the VM.

Installed tools

Visual Studio Code

A shortcut to start Visual Studio Code can be found in the Application Bar on the left side of the screen.
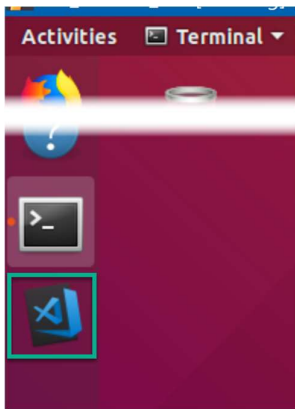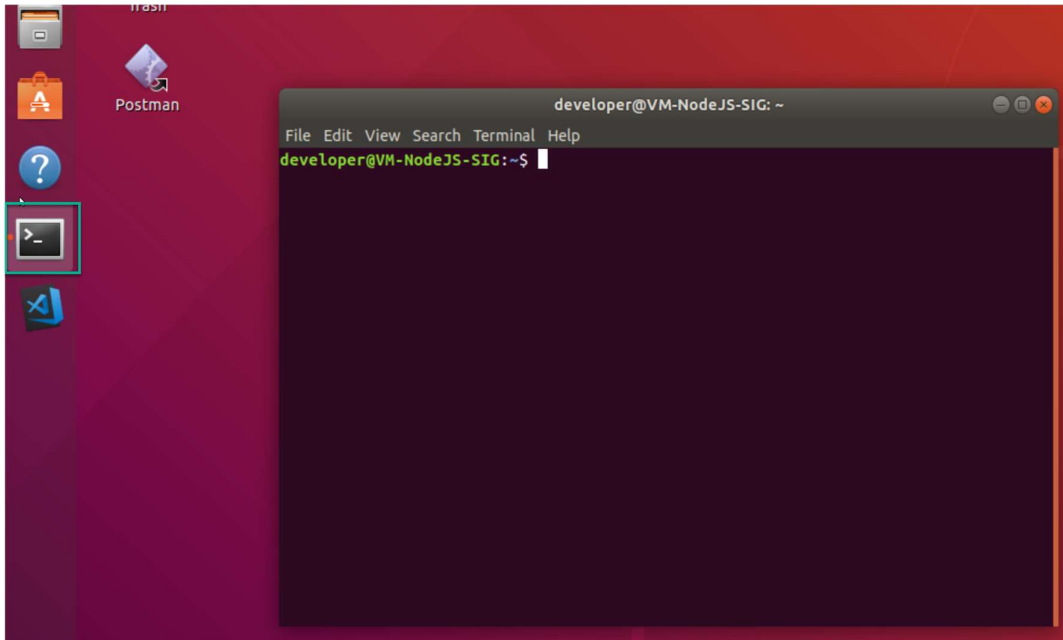

Terminal

You will need the terminal window several times, a shortcut is also available on the Application Bar.

NodeJS

NodeJS is available from the terminal window.

Type `node --version` to see the version of Node.

Docker

Docker is available from the terminal window.

Type `docker --version` to see the version of Docker.

Some useful Docker commands:

`docker ps` `docker container ls`	List the running Docker containers. Add `-a` to show also the non-running.
`docker image ls`	List the installed Docker images
`docker run --name <container name> <image name>:<image tag>`	Start and run an new Docker container. Add `--rm` to automatically remove the container upon exit / stopping.
`docker start <container name>`	Start an existing (non-running) Docker container
`docker exec -it <container name> /bin/bash`	Login into a running container using bash shell

If you see the following error when executing a docker command:

Prefix the command with sudo:

If you do not want to run every docker command using sudo, do the following:

Create the docker group.

`sudo groupadd docker`

Add your user to the docker group.

```
sudo usermod -aG docker $USER
```

Log out and log back in so that your group membership is re-evaluated.
Check it by running:

```
docker image ls
```

in the terminal window. If you get a Docker daemon error: restart the VM.

Postman

Postman will be used for testing and is available as shortcut on the desktop.

Starting out with NodeJS: Hello world!

Starting a new project

To start a new NodeJS project begin by creating a project folder:
- Open the terminal
- The terminal should be opened in your home directory: ~. You can always go back to the home directory by typing `cd` or `cd ~`
- create a new folder by typing:
  `mkdir projects`
- switch to the projects folder you just made:
  `cd projects`
- create a subdirectory called hello-world:
  `mkdir hello-world`
- switch to the hello-world folder:
  `cd hello-world`
- Open Visual Studio Code (VSC):
  `code .`
- In VSC create a new file and call it **hello.js**
  - Click on the Explorer menu
  - Hover over your folder name and click the leftmost icon
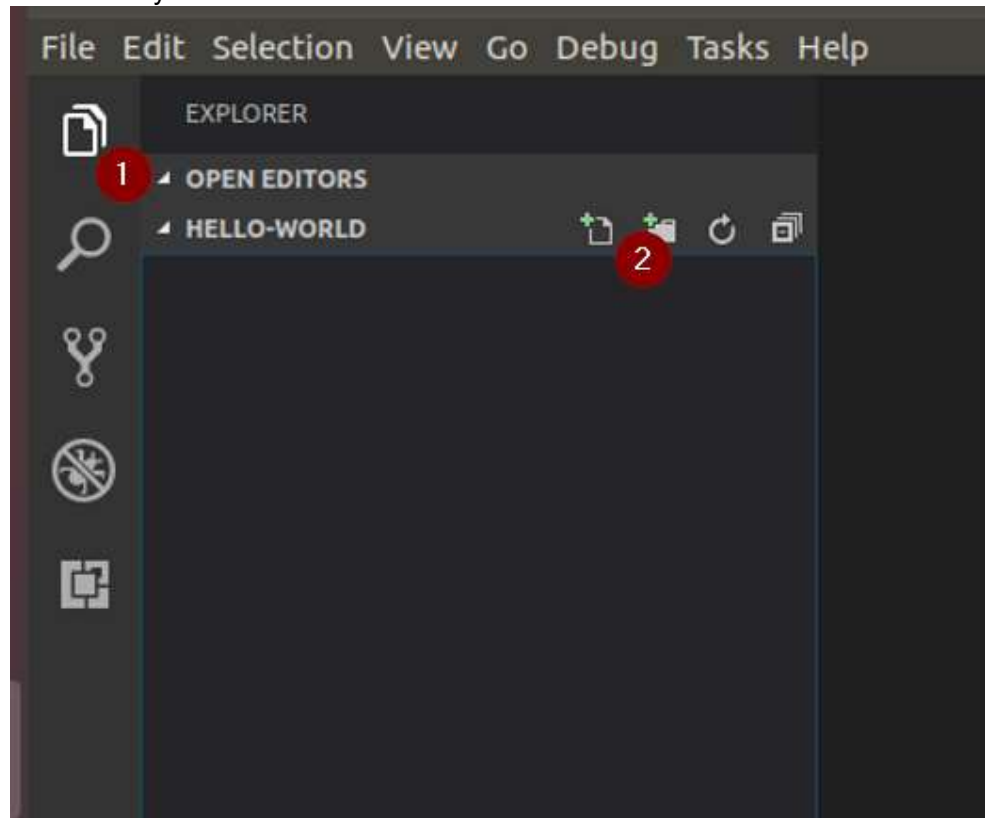
- Type out the code to print the 'Hello World!' statement to the terminal
  `console.log('Hello World!');`

- Save your file.
- Test your code by opening the integrated terminal window (View > Integrated Terminal or CTRL+`
- Type:
  `node hello.js`

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                    1: bash        ▾   ✚  ⊡  🗑  ∧  ⬚  ✕

developer@VM-NodeJS-SIG:~/projects/hello-world$ node hello.js
Hello World!
developer@VM-NodeJS-SIG:~/projects/hello-world$ █
```

- Instead of just calling the code, you can also wrap it in a function.
- Change the code to (or create a new file if you wish):

```
function say_hello() {
    console.log('Hello World!');
}

say_hello();
```

- And run the code again. You should see the same result.
- Change the code to include a function parameter and run het again:

```
function say_hello(name) {
    console.log('Hello ' + name + '!');
}

say_hello('me');
```

- Or if you wish to use a command line argument:

```
function say_hello(greet) {
    console.log('Hello ' + name + '!');
}

// assume command line: node hello.js someName
var name = process.argv[2];
say_hello(name);
```

- Run by typing `node hello.js yourname` or if you want to have spaces: `node hello.js "your name"`

Now we will add some extensions to Visual Studio Code (VSC).

- Go to the extensions menu in VSC.

  - CTRL+SHIFT+X or the icon in the left bar
- Find the Node.js Extension Pack and click install, then restart VSC
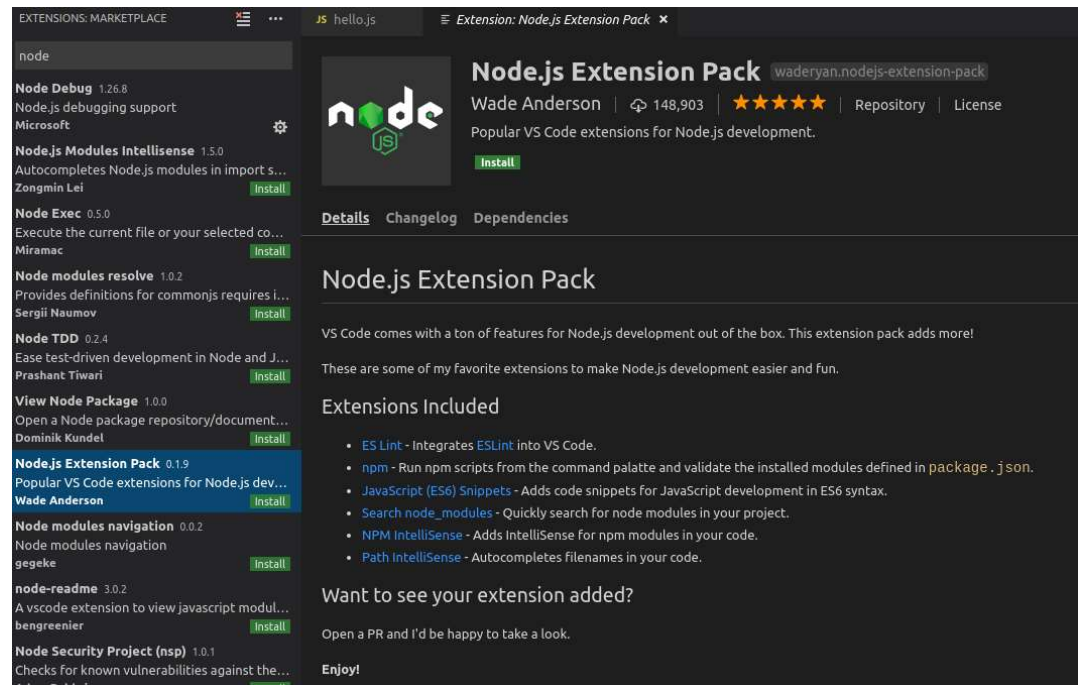
- Next we need to install ESLint by typing the command in the terminal (in VSC) and restart VSC again
  ```
  sudo npm install -g eslint
  ```
- Now create a package.json file and use the settings from the picture. The package.json file contains the settings for your new project.
  ```
  npm init
  ```

- Next execute the following command to set up ESLint. ESLint uses style guides to ensure a uniform coding style.
  `eslint --init`

```
developer@VM-NodeJS-SIG:~/Projects/hello-world$ eslint --init
? How would you like to configure ESLint? Use a popular style guide
? Which style guide do you want to follow? Airbnb (https://github.com/airbnb/javascript)
? Do you use React? No
? What format do you want your config file to be in? JSON
```

- Install the npm dependencies when asked
- Restart VSC for one last time.
- ESLint will now give out errors and a warning in the Problem tab:

```
PROBLEMS 8    OUTPUT   DEBUG CONSOLE   TERMINAL              Filter. Eg: text, **/*.ts, !...

▲ JS hello.js  8
      ⊗ [eslint] Identifier 'say_hello' is not in camel case. (camelcase) (1, 10)
      ⊗ [eslint] Expected indentation of 2 spaces but found 4. (indent) (2, 1)
      ⊗ [eslint] Unexpected string concatenation. (prefer-template) (2, 17)
      ⊗ [eslint] Trailing spaces not allowed. (no-trailing-spaces) (2, 40)
      ⊗ [eslint] All 'var' declarations must be at the top of the function scope. (vars-on-top) (6, 1)
      ⊗ [eslint] Unexpected var, use let or const instead. (no-var) (6, 1)
      ⊗ [eslint] Trailing spaces not allowed. (no-trailing-spaces) (6, 28)
      ⚠ [eslint] Unexpected console statement. (no-console) (2, 5)
```

- First have a look at the warning. ESLint throws this warning because it is unusual to have a console.log command in a front-end application. But because this is a console based application we do not want to have that check in place. The warning can be turned off by editing the .eslintrc.json file like this:

```
{
    "extends": "airbnb-base",
    "rules":{
        "no-console": "off"
    }
}
```

- When you save the file, you will notice that warning will disappear.
- Now check the errors and try to fix them for yourself.
- Hint 1: Use F2 or right click > Rename symbol to refactor
- Hint 2: press CTRL + SHIFT + P to open the VSC command palette. Here you can execute all kinds of commands (builtin or added by extensions). Search for eslint.

If you want to know more about the used style guide, go to:
https://github.com/airbnb/javascript.

Using server-side NodeJS
Using NodeJS to read a connection file

- First let's start a new project by opening or navigate to the Linux terminal again and type:

```
cd ~/projects
mkdir read-file
cd read-file
code .
```

- A new Visual Studio Code window is opened (you can close the other one if it is still open).
- Reopen the command window
- execute the `npm init` command, accept the default settings except the entry point. Name it: **read-file.js**
- make a config file called **config.json**. This contains the connection settings to mongoDB:

```json
{
    "host": "localhost",
    "port": 27017,
    "db": "catalog"
}
```

- We will use NodeJS to read the connection settings from this file. Start by creating a new file called **read-file.js**

```javascript
const fs = require('fs');

let config;

function readConfiguration() {
  fs.readFile('config.json', (err, data) => {
    if (err) {
      throw err;
    }
    config = data;
    console.log(config);
  });
}

readConfiguration();
```

- If you execute this file (in the VSC terminal: `node read-file.js`), it will show you the buffer object, into which the connection data is loaded.
- Change the line `config = data` to `config = JSON.parse(data);` to convert the buffer to JSON. If you run the file again you will see the content of the file in a readable form.

  Let's explain the code so you understand what is happening:

  ```javascript
  const fs = require('fs');
  ```

This statement declares a constant fs with the reference to the fs (file system) module. require itself is also a (core) module of node.js. You can use the fs constant in the rest of your file to call functions in that module.

```
let config;
```
Define a new variable named config in the global scope (it is not defined in a block). Note that there is no typing. config can be any type. If you want you can assign a value directly to initialize it.

```
function readConfiguration()
```
The function who read the file.

```
fs.readFile('config.json', (err, data) => {…}
```
Call the readFile function in the fs module and register a callback function with 2 arguments: err and data.
The => is called an (fat) arrow function which is a shorter syntax for: `function <function name> (arguments) {}`
The callback function is called when the file has been read. Note that the application will continue. If you add for instance the following statement after the call to readConfiguration(): `console.log('Done.');`
You will notice that in the output you will see Done first.

Arrow function

The arrow function was introduced with version ES6.

Example code in ES5 syntax:

```
function timesTwo(params) {
  return params * 2
}
console.log(timesTwo(2));
```

Some code in ES6 syntax:

```
let timesTwo = params => params * 2;
console.log(timesTwo(2));
```

Use-case

Description

We will use a very simple use case to demonstrate how to use node.js for database and API use.

Consider we have to product catalog containing computer (or parts) we sell. And we have several sales persons who are the main contact for certain product groups. We have 2 tables: one with the products and one with all the sales persons and the product groups they sell. In the end we need to support an application which can search in de product catalog and give the product details plus the salesperson which is responsible for the found products.

The products 'table' will consist of the following fields:
- productgroup (string), e.g. "Laptops"
- brand (string), e.g. "HP"
- name (string), e.g. "Elitebook 850 G5"
- modeltype (string), e.g. "i7-8550U"
- modelnumber, e.g. "3JX19EA#ABH"

The salespersons 'table' will consist of the following fields:
- name (string), e.g. "Dijkstra"
- initials (string), e.g. "J"
- prefix (string), e.g. ""
- productgroups (list), e.g. "Laptops, Desktops"

Setup

For implementing the use case we need to have access to a mongodb database. If added the developer user to the docker group you won't need to use sudo.
- Open the terminal
- Create a data directory:
  ```
  mkdir ~/data
  mkdir ~/data/mongo
  ```
- and run this command: `sudo docker run -d --name mongodb -p 27017:27017 -v ~/data/mongo:/data/db mongo:latest`
- Check if the container is running: `sudo docker exec -it mongodb mongo`
- Type `exit` to exit the mongo shell

Create a new project:
```
cd ~/projects
mkdir catalog
cd catalog
code .
```

In Visual Studio Code open the integrated terminal and enter:
```
npm init
eslint --init
```

just like we have done before. Use as entry file: start.js
Then add the required module for mongo and working with rest api's (-save adds module this to the package dependencies):

```
npm install mongodb -save
npm install mongoose -save
npm install express -save
npm install body-parser -save
```

Adding data

Now add the file **setupdb.js** to add some data to our mongodb database.

We will create a **catalog** database with a **products** collection. An add some sample data to start with.

We can do this using just the mongodb module as we see first. After that we will refactor the code to use mongoose, a ORM module for mongo.

Tip: to check if data is actually added to your database: open a Linux terminal window and type: `docker exec -it mongodb mongo`
You are now in the mongo shell.
Type: `use catalog` *to switch to the catalog database.*
Type: `db.products.find({})` *to query alle documents in the products collection.*

Step 1:
```
const mongo = require('mongodb');

const mongoClient = mongo.MongoClient;
const url = 'mongodb://localhost:27017/catalog';

mongoClient.connect(url, (err, db) => {
  if (err) throw err;
  console.log('Connected to mongodb.');
  db.close();
});
```

Try and run the code. If all goes well you should see the message.
You will get a warning. Change the code according to the given hint:
```
mongoClient.connect(url, { useNewUrlParser: true }, (err, db) => {
```
And try again. The warning should be gone.

Change the code to:
```
const mongo = require('mongodb');

const mongoClient = mongo.MongoClient;
```

```
const url = 'mongodb://localhost:27017/catalog';

mongoClient.connect(url, { useNewUrlParser: true }, (err, db) => {
  if (err) throw err;
  console.log('Connected to mongodb.');
  const dbo = db.db('catalog');
  dbo.createCollection('products', (errCollection, res) => {
    if (errCollection) throw errCollection;
    console.log('Collection products created.');
    db.close();
  });
});
```

Run the code (`node setupdb.js`) to check if you get two messages.
A new database with a collection has been created.

To insert a new record into the collection, add to following code after
console.log('Collection products created.'):

```
    products.insertOne(product, (errInsert, res) => {
    if (errInsert) throw errInsert;
    console.log('1 document inserted');
    db.close();
  });
```

Add a variable named products and one named product (after dbo):

```
  const products = dbo.collection('products');
  const product = {
    brand: 'HP',
    productgroup: 'laptops',
    name: 'EliteBook 1040 G4',
    modeltype: 'i5-7200U',
    modelnumber: '1EP75EA#ABH',
  };
```

And remove de db.close() statement in the outer block. We want to close the database
connection after inserting the record.
If you leave the db.close() statement, it will be executed before the insert!
This method of adding data could be made better of course. You could move the insert
into a separate function. But you need to keep in mind the asynchronous nature.
Let's say you add a function addProduct:

```
function addProduct(coll, product) {
  coll.insertOne(product, (errInsert, res) => {
    if (errInsert) throw errInsert;
    console.log('1 document inserted');
  });
}
```

Then you can call this function to add the product:

```
dbo.createCollection('products', (errCollection, res) => {
  if (errCollection) throw errCollection;
  console.log('Collection products created.');

  addProduct(products, product);
});
```

But now you do not close the database connection anymore. Where would you put it?

Assignment: try to figure it out for yourself (*hint: you need a callback function*).

Introducing mongoose

One way to make it easier is to use a ORM (Object Relational Mapper) module or in this case a ODM (Object Document Mapper) because mongodb is a nosql document database, no relational database. For mongo we will use the mongoose module (which we installed already). More information can be found on https://mongoosejs.com/.

For each of our 'tables' (products and salespersons) we will create a schema/model which can be used to easily work with the data.

Create a new file named products.js with the following content:

```
const mongoose = require('mongoose');

// Declare Mongoose schema for products
// Unique: productgroup + brand + modelnumber
const productSchema = new mongoose.Schema({
  productgroup: { type: String, required: true, unique: true },
  brand: { type: String, required: true, unique: true },
  name: { type: String, required: true },
  modeltype: { type: String, required: true },
  modelnumber: { type: String, required: true, unique: true },
});

// Declare Mongoose model to used as class
const Product = mongoose.model('Product', productSchema);

module.exports.Product = Product;
```

rename setupdb.js to setupdb_0.js and create a new file **setupdb.js** which will use mongoose instead of directly calling mongodb functions:

```
const mongoose = require('mongoose');
const products = require('./products');
```

```
const url = 'mongodb://localhost:27017/catalog';

mongoose.connect(url, { useNewUrlParser: true });
const db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', () => {
  // we're connected!
  const product = new products.Product({
    brand: 'HP',
    productgroup: 'laptops',
    name: 'EliteBook 850 G5',
    modeltype: 'i7-8550U',
    modelnumber: '3JX19EA#ABH',
  });
  product.save((err, record) => {
    if (err) return console.error(err);
    console.log(`Added: ${record.brand} ${record.name}`);
    db.close();
    return true;
  });
});
```

And run the code: `node setupdb.js`
And check if the record is added.
Try to run it again to see if the same record is added or not? Why?

Now create a file called salespersons.js for the Salespersons collection.
And update the setupdb.js to create add a salesperson.

REST API

In the previous paragraph we added the record directly using mongoose. Now we will
add a REST API to add new records to the products and salespersons collection and
to query the data.
For the inserts we will create two REST API methods:

1. POST method on /product
2. POST method on /salesperson

Both POST methods will have the complete document for each of the collections as
body.

For query we will create one REST API method:
1. GET method on /products

Swagger definition

Go to https://github.com/AMIS-Services/sig-nodejs-integration-dev for the complete description of the API.

You can also close the repo:

Open a Linux terminal window and type:

```
cd ~/projects/
git clone https://github.com/AMIS-Services/sig-nodejs-integration-dev.git
cd sig-nodejs-integration-dev/productapi
npm start
```

Open a webbrowser with url: http://localhost:8080/docs/
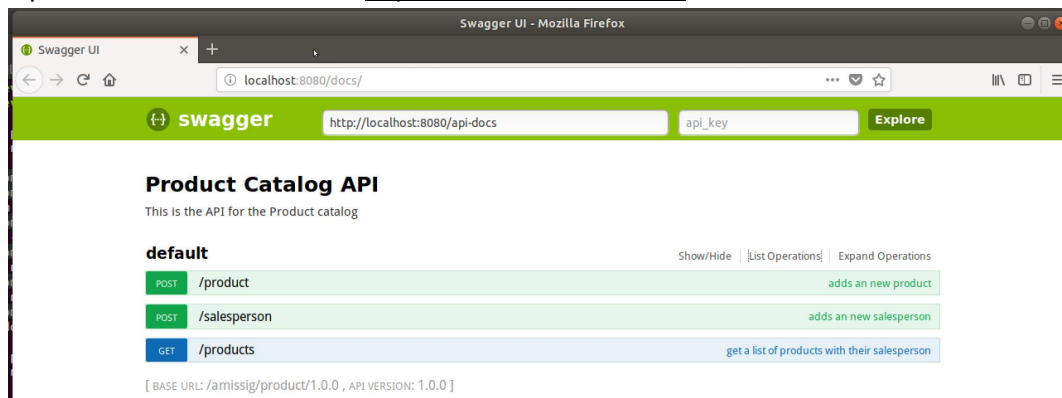
The swagger file can be found in the api subfolder.
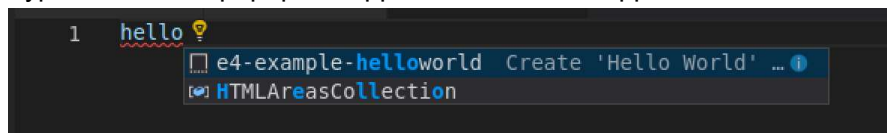
Express.js

For creating a REST API interface we will use express.js (https://expressjs.com/). We already installed it earlier.

Create a new folder **catalogapi** and a subfolder **routes**.

Create a new file: **app.js** in the **catalogapi** folder.

Since we have installed de node.js expansion pack, we can use some snippets.

Type hello and a popup will appear to select to snippet:

Select the e4-example-helloworld snippet. And some basic code is already provided for you. Just change the port to a port number, e.g. 3000 and run the application to check that it works. And fix the code according to the code style to get rid of the warnings.

```javascript
const express = require('express');

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

```
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

// Run app, then load http://localhost:3000 in a browser to see the
output.
```

Now we know that express is working. Change the contents to:

```
const express = require('express');
const bodyParser = require('body-parser');
const routes = require('./routes/routes.js');

const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

routes(app);

const server = app.listen(3000, () => {
  console.log(`app running on port ${server.address().port}`);
});
```
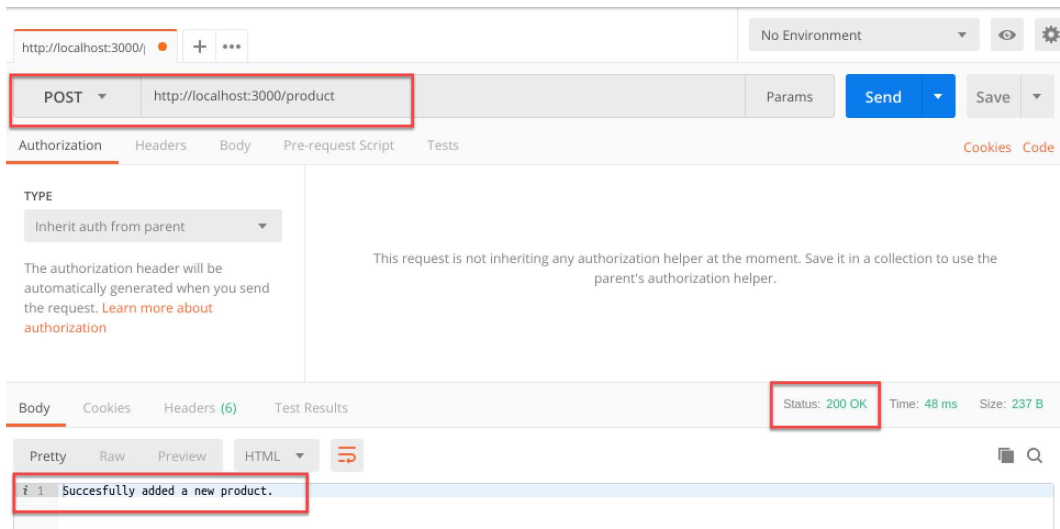
Create a new file **routes.js** in the subfolder **routes**.
Add the basic contents:

```
const appRouter = (app) => {
  app.post('/product', (req, res) => {
    res.status(200).send('Succesfully added a new product.');
  });
};

module.exports = appRouter;
```

run the app: node catalogapi/app.js

Open postman and try again to see if you get an http 200 with the message.

Now update the code so it will actually add a new record to the mongodb collection.

…

Add the post method for /salesperson.

…

Add the get method for searching within the products. First perform just a search on the product collection and later add the lookup of the salesperson(s).

…