

Node Modules

import, export, require

Lesson Plan

- What are modules and why use them
- Creating our first module
- importing our module
- importing built-in module
- npm - using community module

Summary of this lesson is in this url:

<https://www.nerdeez.com/articles/node/modules>

What are modules

- The code we write today, we might want to use tomorrow as well
- It's good practice to split our code to reusable blocks and use those blocks today and in the future.
- a module is a library consisting of one or more files.
- The module consists of private and public sections
- The public sections expose a certain API which we can use when importing the module
- Node programs are built to be modular and we must keep that in mind when writing our own code and keep our code modular as well.

How node runs a module

- node treats every file as a module
- when executing a module node will wrap the module in a function:

```
function(exports, require, module, __filename, __dirname) {  
  
    ...  
  
}
```

- We can use the arguments of that function in the module we will write

Wrapping arguments

- node wraps every module with a function with the following arguments
 - **__dirname** - absolute path of directory of current module
 - **__filename** - absolute path of the file of the current module
 - **exports** - you can attach properties here and they will be exported
 - **module** - module object of the current module
 - **require** - used to require modules

Our first module

- let's create our first module
- In the module we will create a class called **Person** with a public property called **name**
- we will create a method in that class called **sayHello** which prints that name
- We will create another class in that file called **Student** which inherits from **Person** but also has a property called **grade**
- the **Student** class extends the **sayHello** and also prints the student grade
- From this file we would like to export only the **Student** class

Importing our module

- Now lets create another file and in that file use the module we just created.
- Notice that **require** can get a relative path to a file
- the relative path is relative to the current module that is doing the import

Javascript classes, and modules in ES6

- lets go over how we create classes with js
- how we create instances
- and es6 module import and export.

Built-in modules

- modules that are built-in in node's binary
- when doing require with no relative path node will first look for the module in the built in library.
- lets try to use the built in module for dealing with the file system, this module is called: **fs**

Community packages

- one of the advantages of using node is the large community, and how easy it is to share you packages with the outside world.
- There are thousands of open source packages you can install and use in your code.
- Some of those packages are in really high quality, tested, with a lot of contributors.
- The usage of community packages saves you a lot of time and make you focus on your app and not the tools you will need.
- Lets learn how we deal with external packages in node.

What is NPM

- Node Package Manager
- with npm you can
 - Use community packages
 - Publish your package
 - Maintain package version and easily update the package you are using
- npm started as NodeJs backend package manager but today it's also used to install frontend package

Install NPM

- Install NodeJs - npm arrives with node
- <https://nodejs.org/en/>
- you can verify npm is installed by typing: **npm -v**

package.json

- each project or module that has use of npm will have a **package.json** file
- The file will contain information on the current project you are creating
- The file will be in **JSON** format
- Information includes:
 - package name
 - package version
 - dependencies
 - devDependencies
 - peerDependencies
 - author
 - git repo
- to create **package.json** file: **npm init** or **npm init --yes**

package.json - important fields

- **name** and **version** together will form an identifier for the package and has to be unique
- **bugs** will hold the url and email to send bug reports
- **main** - entry point for your package
- **dependencies** - list of packages that your package depends on and will be installed when you install your package
- **devDependencies** - are not needed to run the package only for preparing it
- **peerDependencies** - packages that your package assumes that are already installed, npm will issue a warning if those packages are not installed
- **engines** - it's common to specify the node engine that is required for the package

npm registry

- npm registry is where the packages are saved
- when using npm to install a package or to publish a package, npm will refer to the registry, to see from which registry your npm is set you can type:
 - `npm get registry`
- by default when you install npm it is set to: <https://registry.npmjs.org>
- to set a new url for your registry:
 - `npm set registry <registry-url>`
- It's important for companies to have there private npm registry

Install NPM package

- a package can be installed local or global
- if installed locally the package will be added to the **node_modules** folder where you **package.json** file is located
- local: **npm install <package-name> --save/--save-dev**
- global: **npm install -g <package-name>** (might require admin privileges)
- most of the packages we will install locally, global packages are usually used to add programs to the command line
- the **--save/--save-dev** will determine where to save the package version in the **package.json**
- it's recommended not to push **node_modules** to the repository
- by default

Uninstall Package

- `npm uninstall <package-name> --save`
- `npm uninstall <package-name> --save-dev`
- `npm uninstall -g <package-name>`

Scoped Packages

- namespaces for npm modules
- used for grouping related packages together
- scope begins with @
- recommend to prefix company private packages with prefix @hcl
- you can create a private npm repo and make all the scoped packages be pushed to private repo
- <https://github.com/verdaccio/verdaccio>

publishing your package

- After you finished creating your package you can try and publish it to the private npm registry
- to publish your package:
 - **npm publish**
- you might need to create a user to publish a package
 - **npm adduser**
 - or login if you have a user: **npm login**

Summary - best practices

- in your angular projects, consider which which items will be required across multiple projects, those should be separated to different packages
- when creating a package specify in the **package.json** the **engines** for the node versions required
- make sure to use in your team **nvm** and **.nvmrc** to force the team to work with the same node and npm versions
- Usually a company decides of a scoped package of the company **@pa/hello-world**
- you can also set up that packages scoped with: **@pa** will be published to the private registry and all other packages will be published to the public registry

Using community package

- lets try to install community package called **lodash** and try and use that package

Student EX.

- The goal is to publish a package to the company private npm registry
- create a module which exports a function which prints hello world to the console
- Create a **package.json** des
- the package name should be **@pa/hello-world-<your-name>**
- create a **main** key in the package json pointing to your file.
- publish your package to the private registry and open another project and try to use the package you published.

Summary

- knowing how modules work, how to import and export them, and more importantly how to publish your modules, is the first step of a modular application, which is the most important aspect of microservice programming.