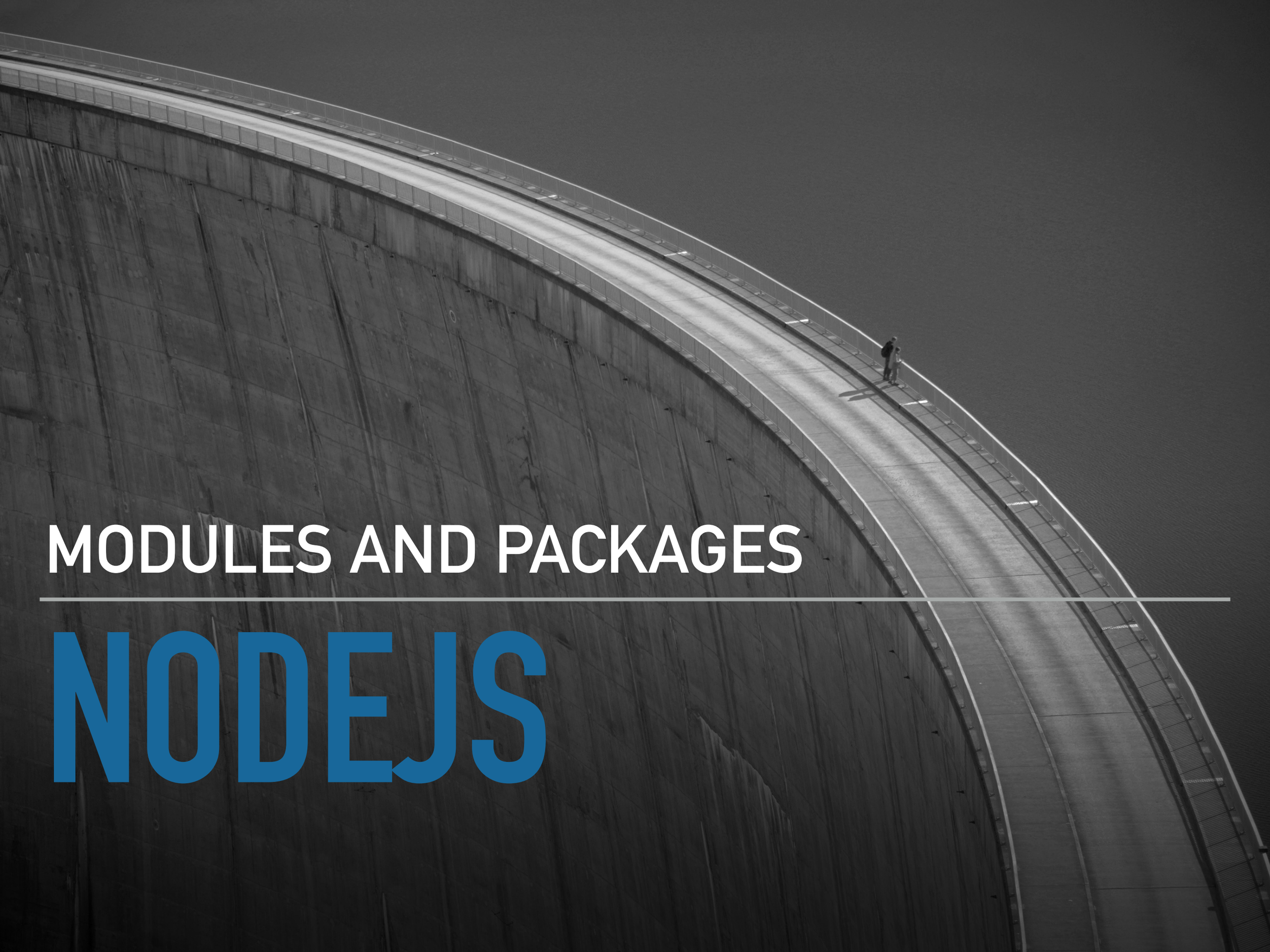

MODULES AND PACKAGES

NODEJS

AGENDA

- Introduction
- Using modules
- Creating modules
- Using packages
- Creating packages
- Labs modules and packages

INTRODUCTION

- Most functionality is encapsulated in modules
  - Binary (.node), JSON (.json), Script (.js)
- Built in modules: Http, fs, EventEmitter, etc….
- Community modules: NPM (more on this later)
- Custom modules: Build your own (more on this later)

INTRODUCTION

- Modules can be packaged as a package
- Modules and files have a 1-to-1 relationship
- Modules are for reusability
- Modules can have dependencies
- Modules are cached when loaded

USING MODULES

- require('<module name>')

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
            + circle.area(4));
```

USING MODULES

- inside require

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"
```

USING MODULES

- inside require

```
LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.json is a file, parse X.json to a JavaScript Object.  STOP
4. If X.node is a file, load X.node as binary addon.  STOP
```

```
LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
3. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP
4. If X/index.node is a file, load X/index.node as binary addon.  STOP
```

- X can be a file or a directory

USING MODULES

- inside require

```
LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)
```

```
NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let I = count of PARTS - 1
3. let DIRS = []
4. while I >= 0,
   a. if PARTS[I] = "node_modules" CONTINUE
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
5. return DIRS
```

- NODE_MODULES_PATHS returns array of node_modules paths from deepest nested to root

USING MODULES

- Example: if you make a request to load the module, "utils":

```
var utils = require( "utils" );
```

- Node.js will perform a hierarchical directory search for "node_modules" and "utils" in the following ways:

```
./node_modules/utils.js

./node_modules/utils/index.js

./node_modules/utils/package.json
```

- If it still can't find the file, Node.js will look at the "require.paths" array
- If the path does not exist, require() throws error 'MODULE_NOT_FOUND'.

CREATING MODULES

- File modules
  - .js files contain JavaScript
  - .json files contain JSON text
  - .node contains binary data
- Folder modules
  - package.json defines main entry script file
  - index.js/index.node alternatively

CREATING MODULES

- Remember?

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
             + circle.area(4));
```

- Here is the module implementation

The contents of `circle.js` :

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

CREATING MODULES

- NodeJS wraps module loading in IIFE

```
var Module = require('module');
console.log(Module.wrapper);
[ '(function (exports, require, module, __filename, __dirname) { ',
  '\n});' ]
```

- Inside your module file, you have access to module and exports. You do not pollute the global namespace.
- HINT: you also always have access to __filename and __dirname. These are the file and directory values of your script module

CREATING MODULES

- Modules are compiled. Which means evaluated and returned.

```
Module.prototype._compile = function(content, filename) {
  var self = this;
  function require(path) { // 1
    return self.require(path);
  }
  ...
  var wrapper = Module.wrap(content);
  var compiledWrapper = runInThisContext(wrapper, { filename: filename });
  var args = [self.exports, require, self, filename, dirname];
  return compiledWrapper.apply(self.exports, args);
};
```

CREATING MODULES

- Modules are cached

```
Module._load = function(request, parent, isMain) {
  if (parent) {
    debug('Module._load REQUEST  ' + (request) + ' parent: ' + parent.id);
  }

  var filename = Module._resolveFilename(request, parent);

  var cachedModule = Module._cache[filename];
  if (cachedModule) {
    return cachedModule.exports;
  }

  var module = new Module(filename, parent);

  if (isMain) {
    process.mainModule = module;
    module.id = '.';
  }

  Module._cache[filename] = module;
```

Demonstreer ook de global.require functie en laat met deze link de code van module module zien

https://github.com/nodejs/node-v0.x-archive/blob/master/lib/module.js#L345

DEMO MODULES

LABS MODULES AND PACKAGES

EXERCISE 1 AND 2

CHECKOUT E657126

USING PACKAGES

- Node Package consists of
  - 1 or more module files
  - package.son
- Node packages can be published to global registry
  - https://www.npmjs.com
  - They cannot be deleted!

USING PACKAGES

```
$ npm -v
1.4.28
```

- Local packages
  - Use require('dirname') in your code
- NPM packages
  - use 'npm install <packagename> (-g)'
  - install can be global and local

USING PACKAGES

- In local mode it installs in `node_modules` folder in your parent working directory.
  - This location is owned by the current user.
- Global packages are installed in `{prefix}/lib/node_modules/`
  - owned by root
  - This means you would have to use `sudo` to install packages globally, which could cause permission errors when resolving third-party dependencies, as well as being a security concern.

USING PACKAGES

- You can change the location of the global packages

```
$ npm config get prefix
/usr/local
```

CREATES AND FILLS CONFIG FILE FOR NPM

```
$ cd && mkdir .node_modules_global
$ npm config set prefix=$HOME/.node_modules_global
```

```
$ npm config get prefix
/home/sitepoint/.node_modules_global
$ cat .npmrc
prefix=/home/sitepoint/.node_modules_global
```

USING PACKAGES

- You can change the location of the global packages

```
$ npm install npm --global
npm@2.6.0 /home/sitepoint/.node_modules_global/lib/node_modules/npm
```

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

```
$ which npm
/home/sitepoint/.node_modules_global/bin/npm
$ npm -v
2.6.0
```

CHECK FOR SUCCESS

USING PACKAGES

- NPM commands:

  access, adduser, bin, bugs, build, bundle, cache, completion, config, dedupe, deprecate, dist-tag, docs, edit, explore, help, help-search, init, install, install-test, link, logout, ls, npm, outdated, owner, pack, ping, prefix, prune, publish, rebuild, repo, restart, root, run-script, search, shrinkwrap, star, stars, start, stop, tag, team, test, uninstall, unpublish, update, version, view, whoami

- Most used commands:
  - init -> creates a new package.json file in the current folder
  - install -> installs module(s)

demonstreer ook de version, ls en config commandos

DEMO USING PACKAGES

CREATING PACKAGES

- Node Package consists of
  - 1 or more module files
  - package.json

    http://browsenpm.org/package.json

CREATING PACKAGES

- NPM can trigger scripts
  - npm test a.k.a npm run test runs test script
  - npm start a.k.a npm run start runs start script
  - npm run <command> runs <command> script

CREATING PACKAGES

- Commmand line interfaces
  - Use bin in package.json
  - Create script and use shebang for node. Place script in ./bin folder
  - Execute $ npm link
  - Test CLI
- At install on windows, NPM will create a .cmd file

```
},
"bin": {
  "hello" : "./bin/hello"
},
"author": "",
```

```
1  #!/usr/bin/env node
2  console.log("Starting the script here!!!");
```

```
1. bash
Johns-MacBook-Pro:testpackage johngorter$ sudo npm link_
```

```
1. bash
Johns-MacBook-Pro:testpackage johngorter$ hello
Starting the script here!!!
Johns-MacBook-Pro:testpackage johngorter$ _
```

CREATING PACKAGES

```
{
  "name": "testpackage",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "echo \"Start!!\"",
    "john": "echo \"john\""
  },
  "author": "",
  "license": "ISC"
}
```

1. bash

```
Johns-MacBook-Pro:testpackage johngorter$ npm run john

> testpackage@1.0.0 john /Users/johngorter/Desktop/testpackage
> echo "john"

john
Johns-MacBook-Pro:testpackage johngorter$ _
```

CREATING PACKAGES

- Command Line Interface tips
  - remove the main entry from package.json
    - this is only used for modules that will be used through the module system (eg var _ = require('underscore');).
  - add preferGlobal and set it to true in package.json
    - if someone installs without the -g option, they will be warned.

CREATING PACKAGES

- Packages can be published to online global repository
  - Create a valid package
  - Use command npm adduser to add or create a user

    Note: If you created one on the site, use npm login to store the credentials on the client.
  - Use npm publish to publish the package.
  - Go to http://npmjs.com/package/<package>. You should see the information for your new package.

CREATING PACKAGES

- Packages can be updated
  - Use npm version patch|minor|major to increment version

    Note: This command will change the version number in package.json
  - Use npm publish to publish the package.
  - Go to http://npmjs.com/package/<package>. You should see the information for your new package.

CREATING PACKAGES

- Packages can be removed
  - Use npm unpublish <pkg> to remove the package

    If no version is specified, or if all versions are removed then the root package entry is removed from the registry entirely.
  - Even if a package version is unpublished, that specific name and version combination can never be reused.

Warning: It is generally considered bad behavior to remove versions of a library that others are depending on! Consider using the deprecate command instead.

demonstreer nam init . script entries en de bin mogelijkheden

DEMO CREATING PACKAGES

LABS MODULES AND PACKAGES

EXERCISE 3

CHECKOUT F751341