Node.js Modules

This lesson gives a brief overview of benefits of modularity and how to use modules in Node.js

We'll cover the following

The Benefits of Modularity

The general idea behind modules is pretty straightforward and similar to the one behind functions. Instead of writing all the code in one place, thus creating a monolithic application, it's often better to split the functionalities into smaller, loosely coupled parts. Each part should focus on a specific task, making it far easier to understand and reuse. The general application's behavior results from the interactions between these building blocks.

These smaller parts are sometimes referred to as components in other environments. In Node, they are called *modules* and can come under different forms. The general definition of a module is: anything that can be loaded using Node's `require()` function. The Node.js platform adheres to the CommonJS module format.

Creating a Module

The simplest form of module is a single JavaScript file, containing special commands to *export* specific pieces of code. The rest of the code is private to the module and won't be visible outside of it.

For example, a `greetings.js` module could contain the following code.

```
// Create three functions
const sayHello = name => `Hello, ${name}`;
const flatter = () => `Look how gorgeous you are today!`;
const sayGoodbye = name => `Goodbye, ${name}`;

// Export two of them
module.exports.sayHello = sayHello;
module.exports.flatter = flatter;
```

In Node, functions can be exported (made accessible outside) by specifying additional properties on the special `module.exports` object. Here, two functions are exported under the names `sayHello` and `flatter`. The third one is not exported. This module could have been written in a slightly more concise way by directly defining the functions as properties of the `module.exports` object.

```
// Create and export two functions
module.exports.sayHello = name => `Hello, ${name}`;
module.exports.flatter = () => `Look how gorgeous you are today!`;

// Create a non-exported function
const sayGoodbye = name => `Goodbye, ${name}`;
```

Loading a Module

Assuming both files are located in the same directory, another JavaScript file could load the previously created module by using the `require()` function provided by Node.js.

```
// Load the module "greetings.js"
const greetings = require("./greetings.js");

// Use exported functions
console.log(greetings.sayHello("Baptiste")); // "Hello, Baptiste"
console.log(greetings.flatter()); // "Look how gorgeous you are today!"
console.log(greetings.sayGoodbye("Baptiste")); // Error: sayGoodbye doesn't exist
```

The parameter passed to `require()` identifies the module to load. Here, the `"./"` substring at the beginning indicates a relative path: the module should be searched for in the same directory as the file that loads it. The result of the

call to `require()` is an object, named `greetings` here. This object references

the value of the `module.exports` object defined inside the module. Thus, the greetings object has two functions `sayHello` and `flatter` as properties. Trying to access its non-existent `sayGoodbye` property triggers an error during execution.

> Giving the object resulting from a call to `require()` the same name as the loaded module's name, through not mandatory, is a common practice.