NodeJS + Express

Roi Yehoshua
2018

Agenda

- NodeJS
- NPM
- Express
- Templates (pug)
- Cookies
- Session
- Uploading files
- Cache management

NodeJS

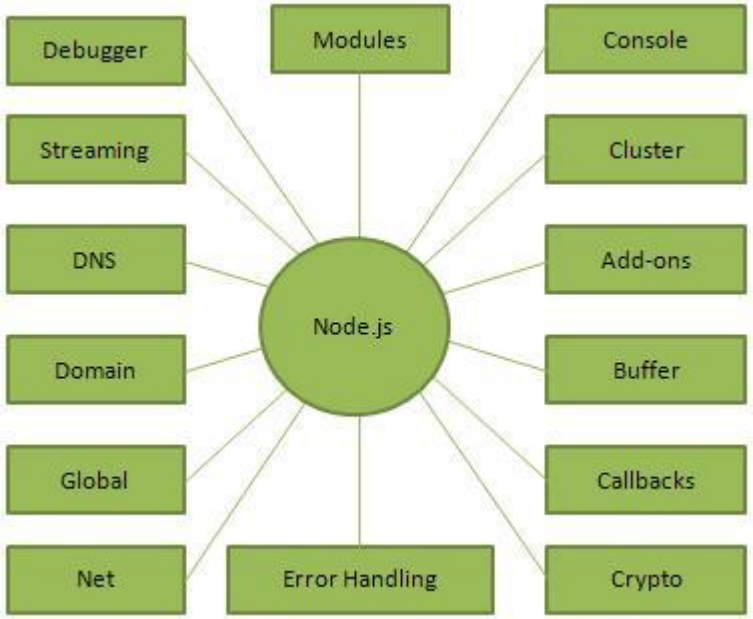
- An open-source, cross-platform JavaScript run-time environment that executes JavaScript code server-side
- Based on Google's V8 engine
- Provides core server functionality
- Asynchronous programming model using non-blocking I/O and asynchronous events
- Aims to optimize throughput and scalability in web applications with many input/output operations
- Node.js was originally written by Ryan Dahl in 2009
- http://nodejs.org

NodeJS Releases

Release	Code name	Release date	LTS status	Active LTS start	Maintenance start	Maintenance end
v0.10.x		2013-03-11	End-of-life	-	2015-10-01	2016-10-31
v0.12.x		2015-02-06	End-of-life	-	2016-04-01	2016-12-31
4.x	Argon	2015-09-08	Maintenance	2015-10-01	2017-04-01	2018-04-30
5.x		2015-10-29	No LTS	N/A		
6.x	Boron	2016-04-26	**Active**	2016-10-18	2018-04-30	April 2019
7.x		2016-10-25	No LTS	N/A		
8.x	Carbon[65]	2017-05-30	**Active**	2017-10-31	April 2019	December 2019
9.x		2017-10-01	No LTS	N/A		
10.x		Apr 2018	Pending	October 2018	April 2020	April 2021

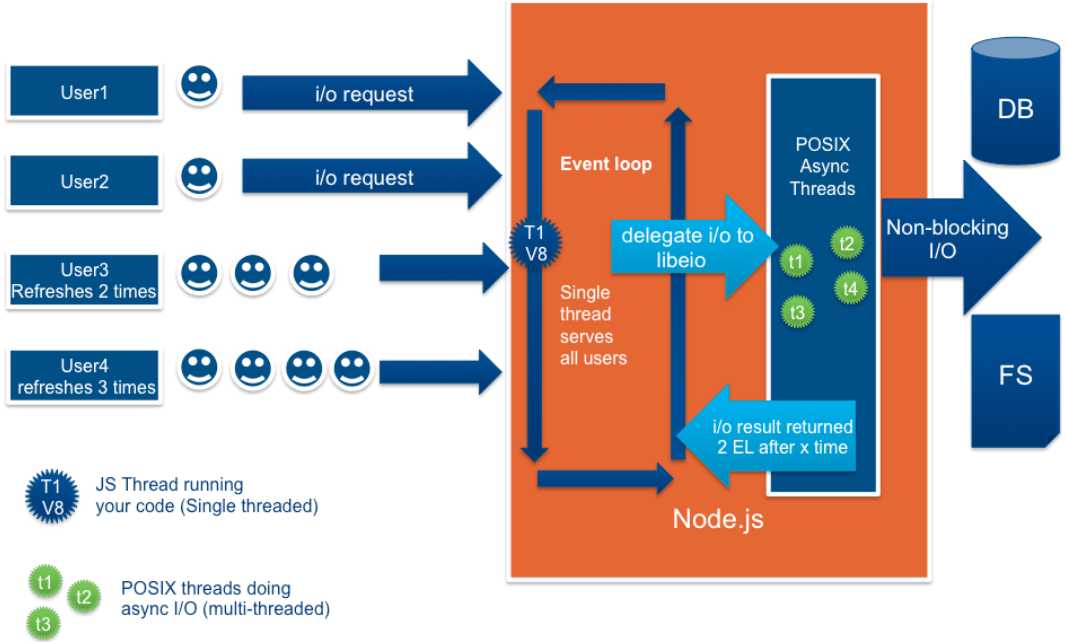
NodeJS Core Functionality

- File system I/O
- Networking (DNS, HTTP, HTTPS, TCP, TLS/SSL,UDP)
- Binary data (buffers)
- Cryptography functions
- Data streams



NodeJS Architecture

- NodeJS is based on a **single-threaded event loop** model to handle multiple concurrent client requests
  - In contrast to other web servers, like PHP/Java/ASP.NET, in which every client request is instantiated on a new thread
  - This provides better performance and scalability under typical web loads



NodeJS Installation

- Download latest version from https://nodejs.org/en/download/
- Choose the appropriate type of installation according to your OS

Downloads

Latest LTS Version: **8.11.3** (includes npm 5.6.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

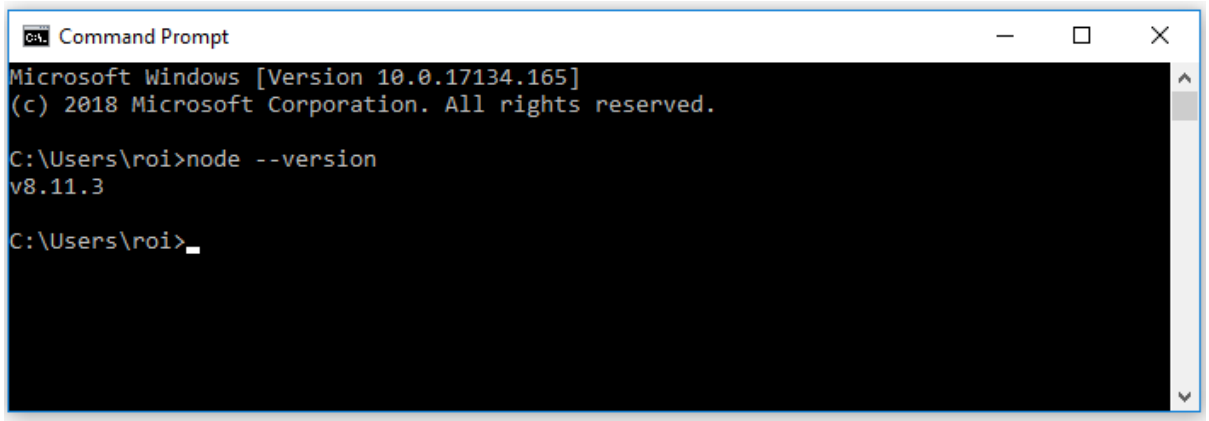
LTS Recommended For Most Users	Current Latest Features

Windows Installer	macOS Installer	Source Code
node-v8.11.3-x64.msi	node-v8.11.3.pkg	node-v8.11.3.tar.gz

Windows Installer (.msi)	32-bit	64-bit	
Windows Binary (.zip)	32-bit	64-bit	
macOS Installer (.pkg)	64-bit		
macOS Binary (.tar.gz)	64-bit		
Linux Binaries (x86/x64)	32-bit	64-bit	
Linux Binaries (ARM)	ARMv6	ARMv7	ARMv8
Source Code	node-v8.11.3.tar.gz		

Verifying Installation

- After installing node it should be available in your PATH
- Verify that the installation was successful by running node --version from the command prompt:



Running node

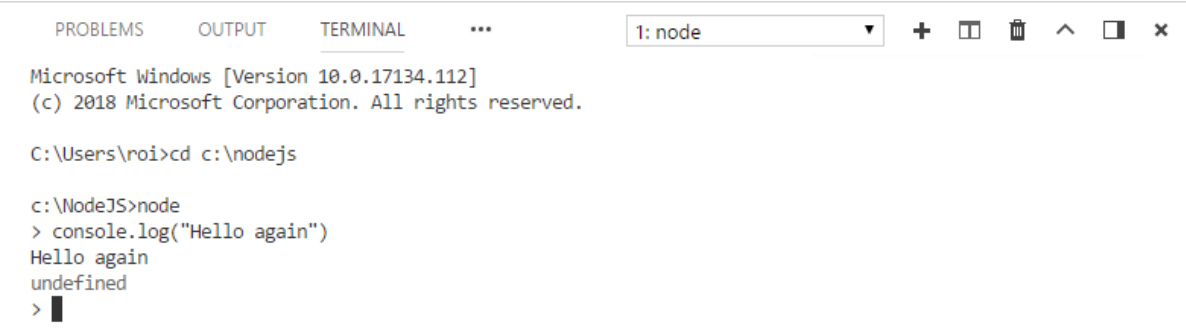
- Start a new cmd window and write ‘node’ in command line
- node.exe acts as javascript interpreter, so you can directly put some code in console

```
Command Prompt - node
Microsoft Windows [Version 10.0.17134.165]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\roi>node
> console.log("hello world")
hello world
undefined
> let x = 5, y = 7
undefined
> x + y
12
> console.log(x + y)
12
undefined
>
```

  - The interpreter displays the return value of each command, and since console.log() returns undefined, you see these “undefined” messages in the output

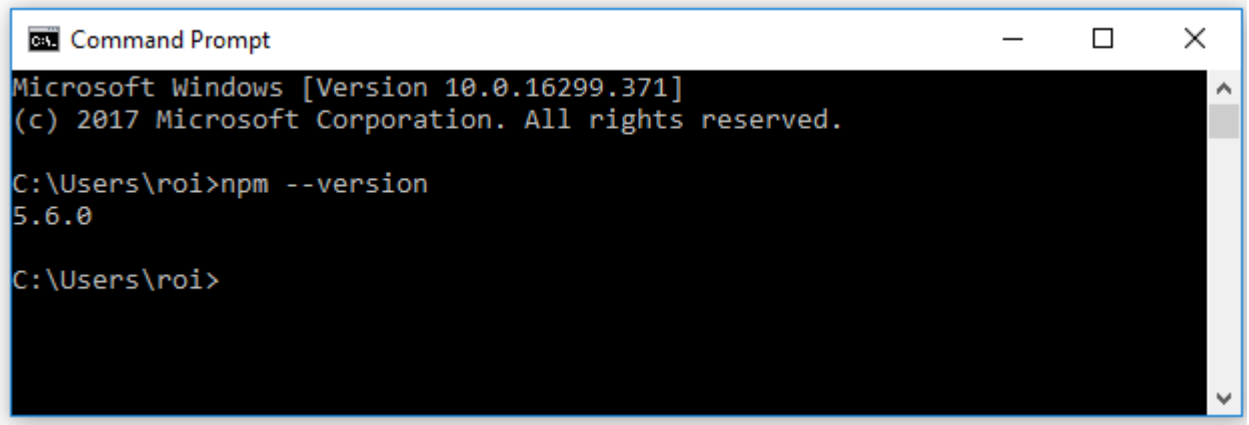
Running node from VS Code

- You can also run Node directly from Visual Studio Code
- Choose View -> Integrated Terminal (or Ctrl+`)
- Go to the folder where your Node server is installed (e.g., C:\NodeJS)
- Type node



Node Package Manager (NPM)

- NPM provides two main functionalities:
  - Online repositories for node.js packages/modules which are searchable on search.nodejs.org
  - Command line utility to install Node.js packages, do version management and dependency management of Node.js packages
- NPM comes bundled with Node.js installable
- To verify the NPM version, open console and type the following command:

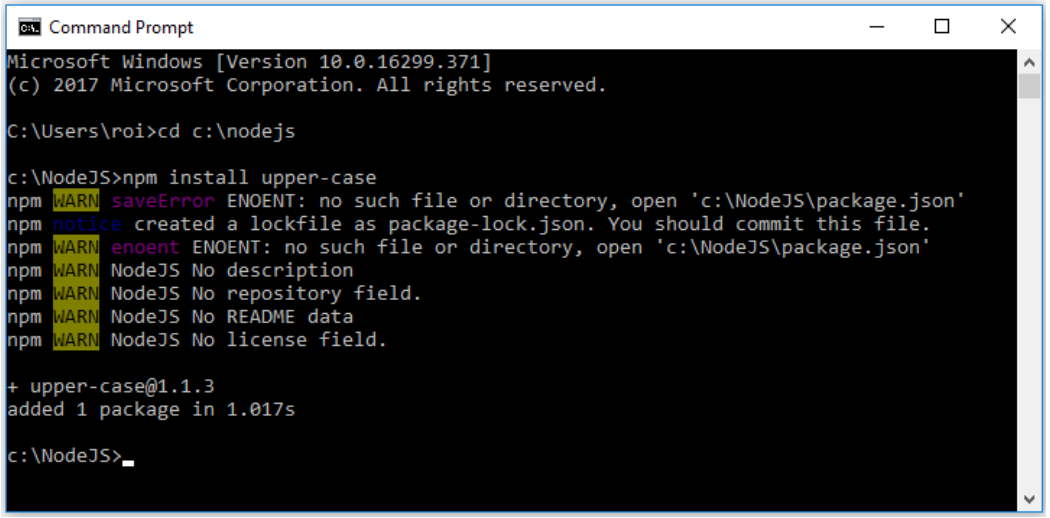


Installing Packages using NPM

- A package in Node.js contains all the files you need for a module
- Modules are JavaScript libraries you can include in your project
- To install any Node.js package, type:

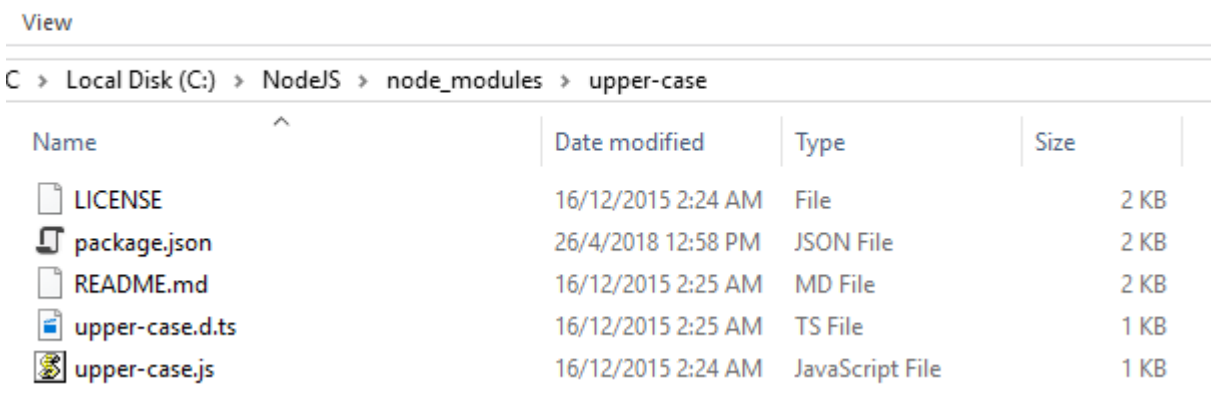
```
npm install <Package Name>
```

- For example, to install the package “upper-case” open a command line and type:



Installing Packages using NPM

- NPM creates a folder named "node_modules“ inside the folder you are currently in, and places the new package there
- By default, npm will also install all the modules listed as dependencies of the module that you’re trying to install



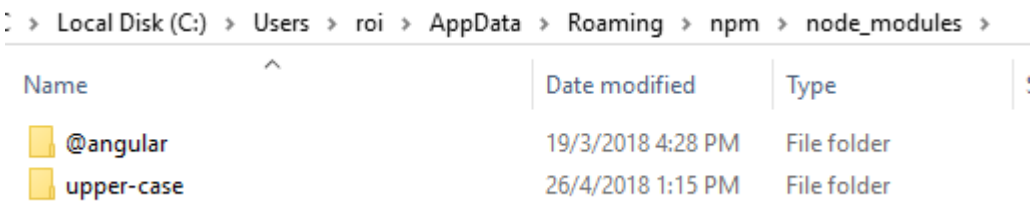
Global vs. Local Installation

- By default, NPM installs any dependency in local mode, i.e., in the node_modules sub folder of the folder you are currently in
- To install a package globally add the **-g** switch:

```
npm install <Package Name> -g
```

- Globally installed packages are stored in the system directory
  - In Windows: %AppData%\npm\node_modules
  - In Linux: /usr/local/lib/node_modules
- For example, to install the upper_case package globally type:

```
c:\NodeJS>npm install upper-case -g
+ upper-case@1.1.3
added 1 package in 0.853s
```



List Packages

- You can use **npm ls** to list down all the locally installed modules
- Or **npm ls -g** to list all the globally installed modules

Command Prompt

```
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\roi>npm ls -g
C:\Users\roi\AppData\Roaming\npm
+-- @angular/cli@1.7.3
| +-- @angular-devkit/build-optimizer@0.3.2
| | +-- loader-utils@1.1.0 deduped
| | +-- source-map@0.5.7
| | +-- typescript@2.6.2
| | `-- webpack-sources@1.1.0 deduped
| +-- @angular-devkit/core@0.3.2
| | +-- ajv@5.5.2
| | | +-- co@4.6.0
| | | +-- fast-deep-equal@1.1.0 deduped
| | | +-- fast-json-stable-stringify@2.0.0 deduped
| | | `-- json-schema-traverse@0.3.1 deduped
| | +-- chokidar@1.7.0
| | | +-- anymatch@1.3.2
| | | | +-- micromatch@2.3.11 deduped
| | | | `-- normalize-path@2.1.1 deduped
| | | +-- async-each@1.0.1
| | | +-- UNMET OPTIONAL DEPENDENCY fsevents@^1.0.0
| | | +-- glob-parent@2.0.0
| | | | `-- is-glob@2.0.1 deduped
| | | +-- inherits@2.0.3 deduped
| | | +-- is-binary-path@1.0.1
| | | | `-- binary-extensions@1.11.0
| | | +-- is-glob@2.0.1
| | | | `-- is-extglob@1.0.0
```

NPM Commands Summary

NPM

Command Line Interface (CLI) commands for npm

Getting Started

`npm init`
Interactively create a package.json file

`npm install`
Install packages based on package.json file in the current folder

`npm search <term>`
Search the registry for packages matching the search terms

`npm update -g <package_name>`
Updates all the packages to the latest version, respecting semver

`npm help <command>`
Show help for the specific command

`npm search <package_name>`
Search for the package

Installing Packages

`npm install <package_name>`
Install the latest version of a package

`npm install <package_name>@<version>`
Install specific version of a package

`npm install -g <package_name>`
Install a package globally, usually for command line use (On *nix requires sudo)

`npm install -S <package_name>`
Install a package and append it in the dependencies section of your package.json

`npm install -D <package_name>`
Install a package and append it in the devDependencies section of your package.json

`npm install -O <package_name>`
Install a package and append it in the optionalDependencies section of your package.json

Uninstalling Packages

`npm uninstall <package_name>`
Uninstall the latest version of a package

`npm uninstall <package_name>@<version>`
Uninstall specific version of a package

`npm uninstall -g <package_name>`
Uninstall the package globally

`npm uninstall -S <package_name>`
Uninstall the package and append it in the dependencies section of your package.json

`npm uninstall -D <package_name>`
Uninstall the package and append it in the devDependencies section of your package.json

`npm uninstall -O <package_name>`
Uninstall the package and append it in the optionalDependencies section of your package.json

Package Details

`npm docs <package_name>`
Show the docs for a package in a web browser

`npm bugs <package_name>`
Show the issues for a package in a web browser

`npm repo <package_name>`
Open package repository page in the browser

`npm ls`
Print all the versions of packages that are installed, as well as their dependencies

package.json

- All npm packages contain a file, usually in the project root, called **package.json**
- This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies
- It must at least specify a name and version, which together form a unique identifier of your package
- Typically it also contains a list of all the packages that your project depends on

```
{
  "name"         : "Todo App",
  "version"      : "1.0.0",
  "description"  : "Simple todo application.",
  "main"         : "todo.js",
  "author"       : "Roi Yehoshua",
  "dependencies" : {
    "express"    : "~3.4.4",
    "mongoose"   : "~3.6.2"
  }
}
```

Creating a Package

- To create a package.json with values that you supply, run **npm init**
- This will initiate a command line questionnaire that will conclude with the creation of a package.json in the directory in which you initiated the command
- To get a default package.json, run **npm init --yes**
  - This will generate a default package.json using information extracted from the current directory
- Let's create a new package called my-first-package
  - Create a folder C:\NodeJS\my-first-package
  - Then run npm init from that folder

Creating a Package

npm

```
Microsoft Windows [Version 10.0.17134.165]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\roi>cd c:\nodejs

c:\NodeJS>mkdir my-first-package

c:\NodeJS>cd my-first-package

c:\NodeJS\my-first-package>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (my-first-package)
```

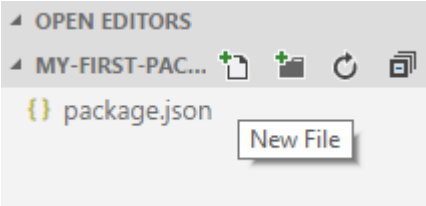
Creating a Package

- Accepting all the defaults by pressing Enter will generate the following package.json:

```
{
  "name": "my-first-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Creating a Package

- Typically, the main entry file to your package will be called app.js
- Open the folder C:\NodeJS\my-first-package in Visual Studio Code by using the menu File > Open Folder…
- Currently your folder contains only package.json
- Create a new file app.js by clicking the New File button next to the package name



- Name your file app.js

Creating a Package

- Type the following code into app.js:

app.js - my-first-package - Visual Studio Code

File Edit Selection View Go Debug Tasks Help

EXPLORER

OPEN EDITORS

app.js

MY-FIRST-PACKAGE

app.js

package.json

OUTLINE

```
1  console.log("This is my first NodeJS package!")
2
3  let x = 5, y = 7;
4  console.log("x + y = ", x + y);
5
```

0 0 Ln 4, Col 32 Spaces: 4 UTF-8 CRLF JavaScript 1

Running Your Server Script

- To run node with your script, open a Console window in VS Code
  - Choose View -> Debug Console
  - Click the TERMINAL tab
- Verify that you're located at the folder of your package (C:\NodeJS\my-first-package)
- Type node app.js

```
PROBLEMS   OUTPUT   TERMINAL   ...                    1: cmd
Microsoft Windows [Version 10.0.17134.165]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\NodeJS\my-first-package>node app.js
This is my first NodeJS package!
x + y =  12

C:\NodeJS\my-first-package>
```

Exercise (1)

- Create a new NodeJS package named display-time
- Create a server script app.js that prints to the console the current date and time in the format dd/mm/yyyy HH:MM
- Run your script both from Visual Studio terminal and from a command prompt

NodeJS Modules

- In the NodeJS module system, each JavaScript file is treated as a separate module
  - NodeJS modules are analogous to JavaScript libraries
- NodeJS has a set of built-in modules that you can use without any installation
- The following table shows some of the most commonly used built-in modules:

Module	Description
http	Makes Node.js act as an HTTP server
https	Makes Node.js act as an HTTPS server
fs	Handles the file system
path	Handles file paths
os	Provides information about the operation system
url	Parses url strings
stream	Handles streaming data
cluster	Splits a single Node process into multiple processes
events	Handles events

Include Modules

- To include a module, use the require() function with the name of the module
- For example, let's another file app2.js to our package, and write the following code:

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

console.log('Platform: ' + os.platform());
console.log('Architecture: ' + os.arch());
```

- Now run this script from terminal:

```
C:\NodeJS\my-first-package>node app2.js
Platform: win32
Architecture: x64
```

Create Your Own Module

- You can create your own modules, and easily include them in your applications
- Use the **exports** keyword to make properties and methods available outside the module file
  - **exports** is a property of an object called **module** which is included in every NodeJS file
- For example, add the file “date_formatter.js”, and add the following code to it:

```
exports.formatDate = function(date, sep) {
    let day = date.getDate();
    let month = date.getMonth();
    let year = date.getFullYear();
    return `${day}${sep}${month}${sep}${year}`;
}

exports.formatTime = function(time, sep) {
    let hour = time.getHours();
    let min = time.getMinutes();
    return `${hour}${sep}${min}`;
}
```

Include Your Own Module

- Now you can include and use the module in any of your Node.js files
- Edit the file app2.js and the following code to it:

```
const dateFormatter = require('./date_formatter');

let now = new Date();
console.log('Current date: ' + dateFormatter.formatDate(now, '/'));
console.log('Current time: ' + dateFormatter.formatTime(now, ':'));
```

  - To load a module located in the same directory use ‘./’
- Now run the file:

```
C:\NodeJS\my-first-package>node app2.js
Platform: win32
Architecture: x64
Current date: 3/8/2018
Current time: 10:59
```

Exporting a Class

- Typically, when defining a class in a module, the class is the only thing you want to export from the module
- In this case you can assign the class directly to the exports object
- For example, add a file named “circle.js” with the following code:

```
module.exports = class Circle {
    constructor(radius) {
        this.radius = radius;
    }

    getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }
}
```

  - Note that in this case you have the write the full name of the property module.exports and not only the keyword exports

Importing a Class

- To import the class, add the following code to app2.js:

```
const Circle = require('./circle.js');
let c1 = new Circle(5);
console.log('Area of circle is: ' + c1.getArea());
```

```
C:\NodeJS\my-first-package>node app2.js
Platform: win32
Architecture: x64
Current date: 3/8/2018
Current time: 20:41
area of circle is: 78.53981633974483
```

Add a Dependency to a Package

- If you need to use modules from other NodeJS packages, you first have to install them in your package's folder using **npm install**
- This will create a node_modules subfolder in your package folder containing the installed package files
- It will also add the package to the dependencies list in package.json
  - This will cause the users of your own package to install all the necessary dependencies when installing your own package
- For example, assume that our package needs to use some advanced mathematical functions from the mathjs package
- First we install the package from the command prompt:

```
C:\NodeJS\my-first-package>npm install mathjs
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN my-first-package@1.0.0 No description
npm WARN my-first-package@1.0.0 No repository field.

+ mathjs@5.0.4
added 9 packages in 3.473s
```

Add a Dependency to a Package

- package.json has been modified to reflect the new dependency:

```
{
  "name": "my-first-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mathjs": "^5.0.4"
  }
}
```

- a caret symbol matches the most recent major release, i.e., ^5.0.4 matches any 5.x.x release
- a tilde symbol matches the most recent minor release, i.e., ~5.0.4 matches any 5.0.x release

Use the New Dependency

- In app2.js we can now use the mathjs module:

```
const math = require('mathjs');

dx = math.derivative('x^2 + x', 'x');
console.log(dx.toString());
```

```
C:\NodeJS\my-first-package>node app2.js
Platform: win32
Architecture: x64
Current date: 3/8/2018
Current time: 21:7
Area of circle is: 78.53981633974483
2 * x + 1
```

Exercise (2)

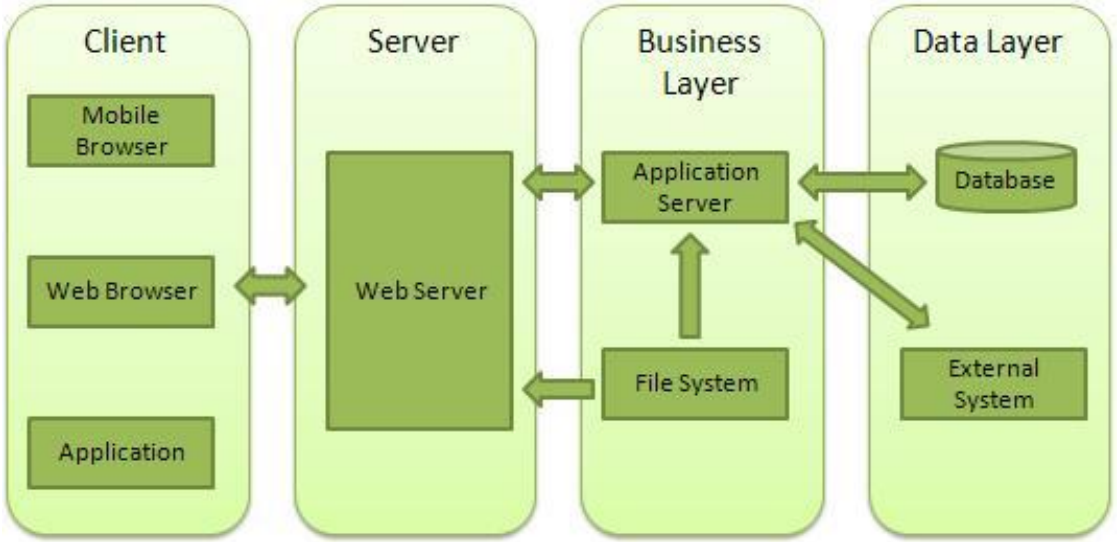
- Create a new NPM package called students_[your id]
- Add the module you've created in the previous exercise in the new package
- Test the package locally
- Publish your package
- Verify the installation of the published package

Exercise (2)

- Create a package named students
- Add a student.js module that exports a class named Student with the properties: id, name, age, and grades, and the following methods:
  - constructor(id, name, age)
  - addGrade(grade) - adds a grade to the student's grades list
  - computeGradesAverage() - returns the student's grades average
    - Use the package simple-statistics for computing the average
- Create another package named test_students
- Add app.js to test_students and import the student module from the students package
  - Hint: use require('../students/student') to import the module
- Create a new Student object, add a few grades and print the student's grades average
- Run app.js in node

Web Servers

- A **Web Server** (or HTTP server) is a software application which handles HTTP requests sent by HTTP clients, like web browsers, and returns web pages/resources in response
- Most of the web servers support server-side scripts, which retrieve data from a database or perform some complex logic and sends a result to the HTTP client



Creating a Web Server with NodeJS

- The **http** module can create an HTTP server that listens to server ports and gives a response back to the client
- Create a new package named my-first-server, and add app.js to it
- Use the **createServer()** method to create an HTTP server:

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

http.createServer(function (req, res) {
    res.write('Hello world!'); // write a response to the client
    res.end(); // end the response
}).listen(8080); // the server listens on port 8080
```

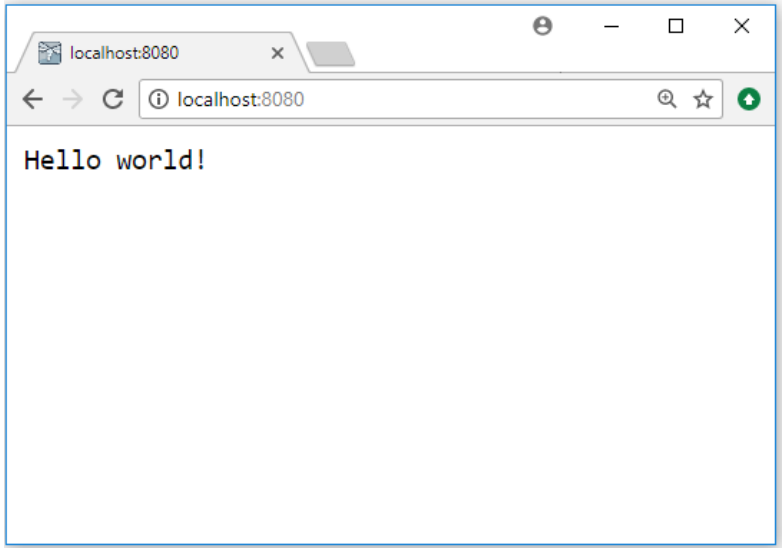
  - The method http.createServer() receives a function, that is invoked when a client sends a request to the server on the specified port
  - This function receives two parameters:
    - req – represents the request object, containing parameters sent from the client
    - res – allows you to write a response back to the client

Creating a Web Server with NodeJS

- Run app.js in node:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                    1: node

Microsoft Windows [Version 10.0.17134.165]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\NodeJS\my-first-server>node app.js
```

- Open http://localhost:8080 in any browser to see the result



Exercise (3)

- Create a NodeJS server that listens on port 9000
- Whenever it receives an HTTP request from a client, it should send back the current time on the server in the format HH:MM:SS
- Test the server using a browser

Express

- **Express** is a minimal and flexible MVC framework for Node.js that provides a thin layer of fundamental web application features
- It is an open source framework developed and maintained by the Node.js foundation
- Many popular web frameworks are based on express
- Main features of Express:
  - Has convenient functions for parsing HTTP arguments and headers
  - Routing – can easily build RESTful APIs with Express
  - Has support for many templating languages like Jade and EJS, that reduce the amount of HTML code you have to write
  - Has support for NoSQL databases out of the box
  - Has a flexible, modular middleware pattern, where special middleware modules/functions are used to process different requests
- https://expressjs.com/

Installing Express

- Express is installed like any other NPM package using npm install
- Create a Node package named first-express-app
- Call npm init to create a package.json file
- Now install express inside your package:

```
C:\NodeJS\first-express-app>npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN first-express-app@1.0.0 No description
npm WARN first-express-app@1.0.0 No repository field.

+ express@4.16.3
added 50 packages in 2.263s
```

Basic Express App

- Create a new file called **app.js** and type the following code:

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

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

app.listen(3000);
```

  - The **express()** function is the top-level function exported by the express module, which creates an express application
  - The app starts a server and listens on port 3000 for connections
  - The app responds with “Hello World!” for requests to the root URL (/)
  - For every other path, it will respond with a **404 Not Found**
  - The req (request) and res (response) are the exact same objects that Node provides

nodemon

- nodemon is a utility that will monitor for any changes in your source and automatically restart your server
- Just use **nodemon** instead of node to run your code, and now your process will automatically restart when your code changes
- To install, run from the terminal:

```
npm install -g nodemon
```

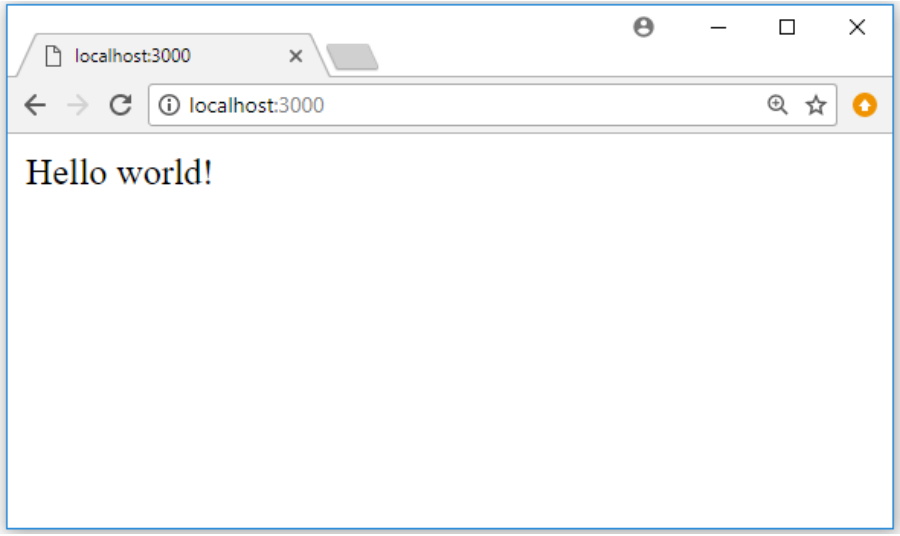
```
C:\NodeJS\first-express-app>npm install -g nodemon
C:\Users\roi\AppData\Roaming\npm\nodemon -> C:\Users\roi\AppData\Roaming\npm\node_modules\nodemon\bin\
nodemon.js

> nodemon@1.18.3 postinstall C:\Users\roi\AppData\Roaming\npm\node_modules\nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
 > https://opencollective.com/nodemon/donate

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\nodemon\node_modules\fsev
ents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"d
arwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ nodemon@1.18.3
added 232 packages in 13.704s
```

Basic Express App

- Run the app using nodemon:

```
C:\NodeJS\first-express-app>nodemon app.js
[nodemon] 1.18.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
```

- Then, load http://localhost:3000/ in a browser to see the output



Basic Routing

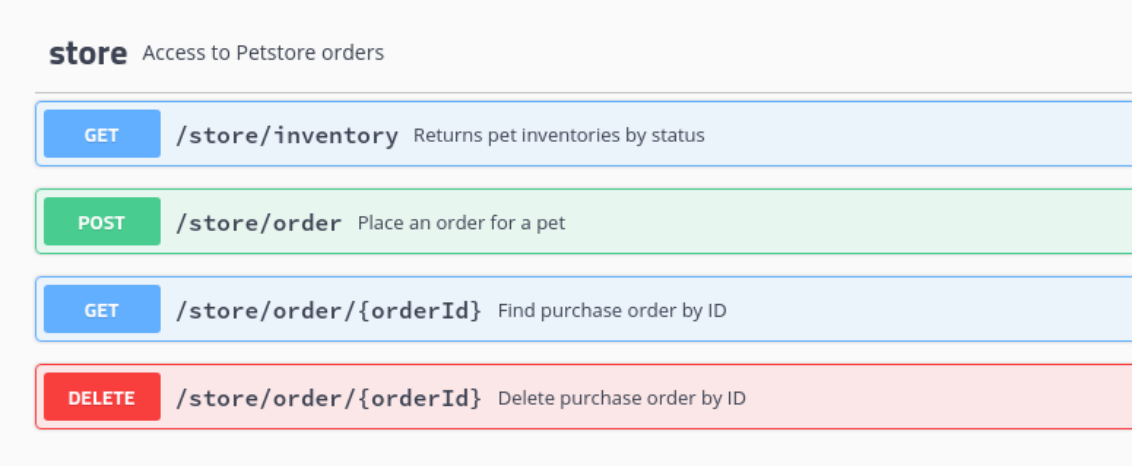
- Routing refers to determining how the server responds to a client request to a specific URI (or path) and a specific HTTP request method
- Each route can have one or more handler functions, which are executed when the route is matched
- Route definition takes the following structure:

```
app.method(path, handler)
```

  - app is an instance of express application
  - *method* is an HTTP request method, in lowercase (e.g., ‘get’, ‘post’)
  - *path* is a path on the server
  - *handler* is the function executed when the route is matched

HTTP Methods

Method	Description
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
POST	The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI
PUT	The PUT method replaces all current representations of the target resource with the request payload
DELETE	The DELETE method deletes the specified resource



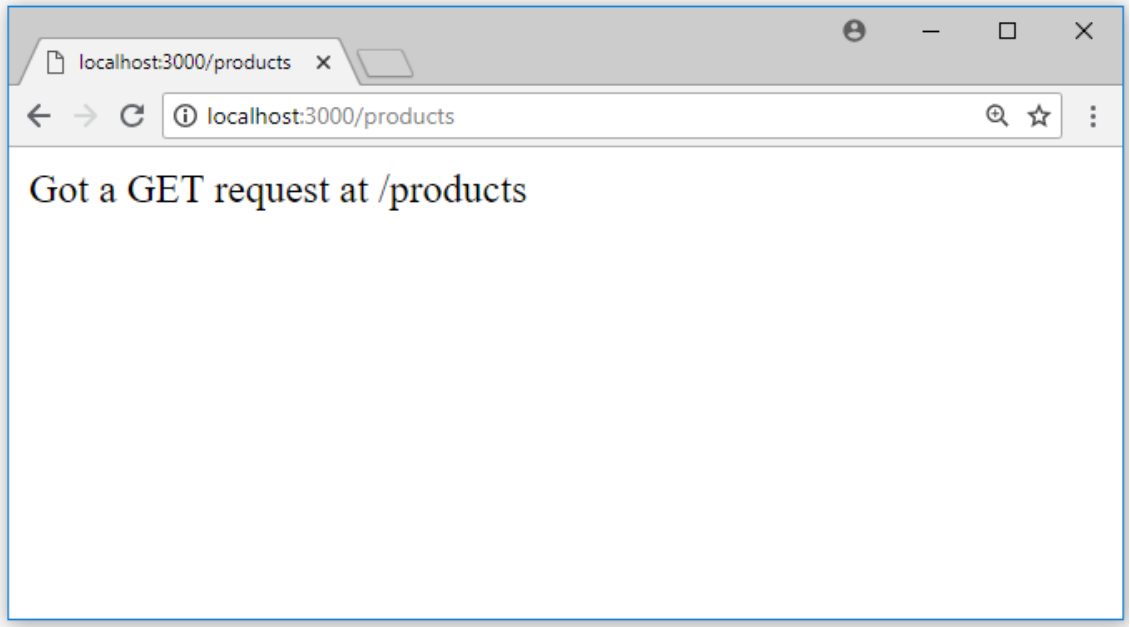
Basic Routing

- The following examples illustrate defining simple routes:

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

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

app.get('/products', (req, res) => {
    res.send("Got a GET request at /products");
});
app.post('/products', (req, res) => {
    res.send("Got a POST request at /products");
});
app.put('/products', (req, res) => {
    res.send("Got a PUT request at /products");
});
app.delete('/products', (req, res) => {
    res.send("Got a DELETE request at /products");
});
app.listen(3000);
```

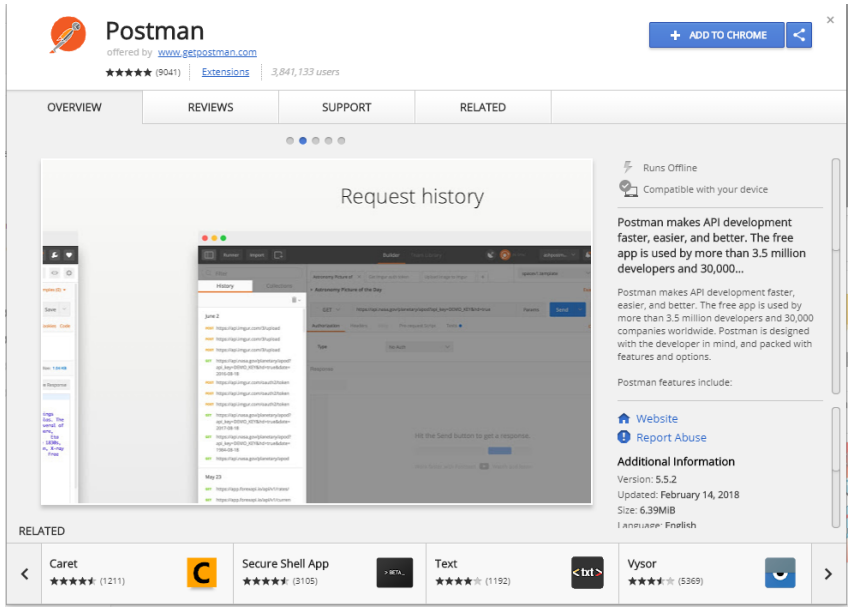
Basic Routing

- You can only test the GET methods in the browser
- For example, enter the URL http://localhost:3000/products to test the GET method of products



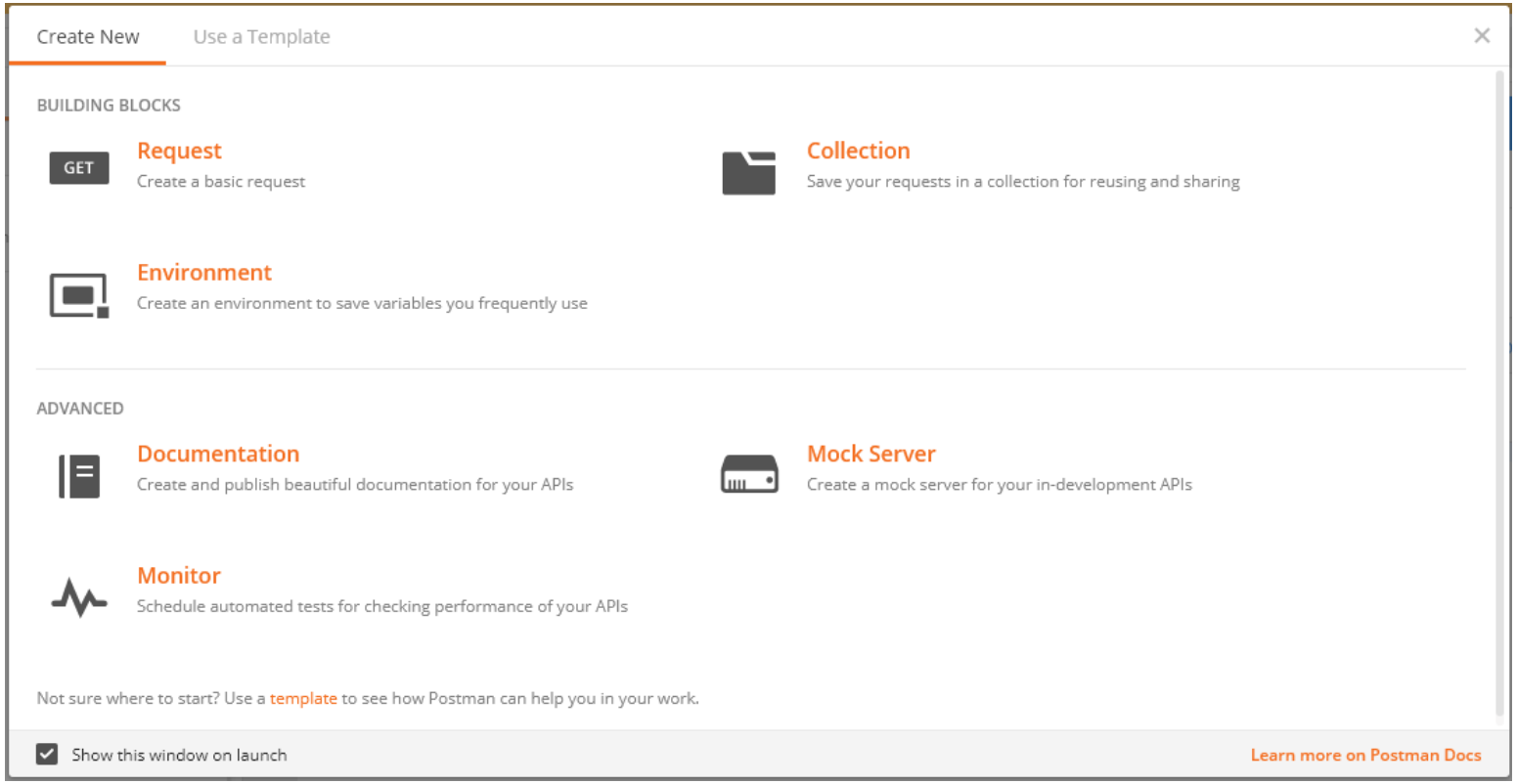
PostMan

- PostMan is a web REST client that allows you to enter and monitor HTTP requests of different types and examine the responses
- Can be installed as an extension to Chrome
  - Search “PostMan – Chrome Web Store” in google, and click the first link
  - Then click the Add To Chrome button



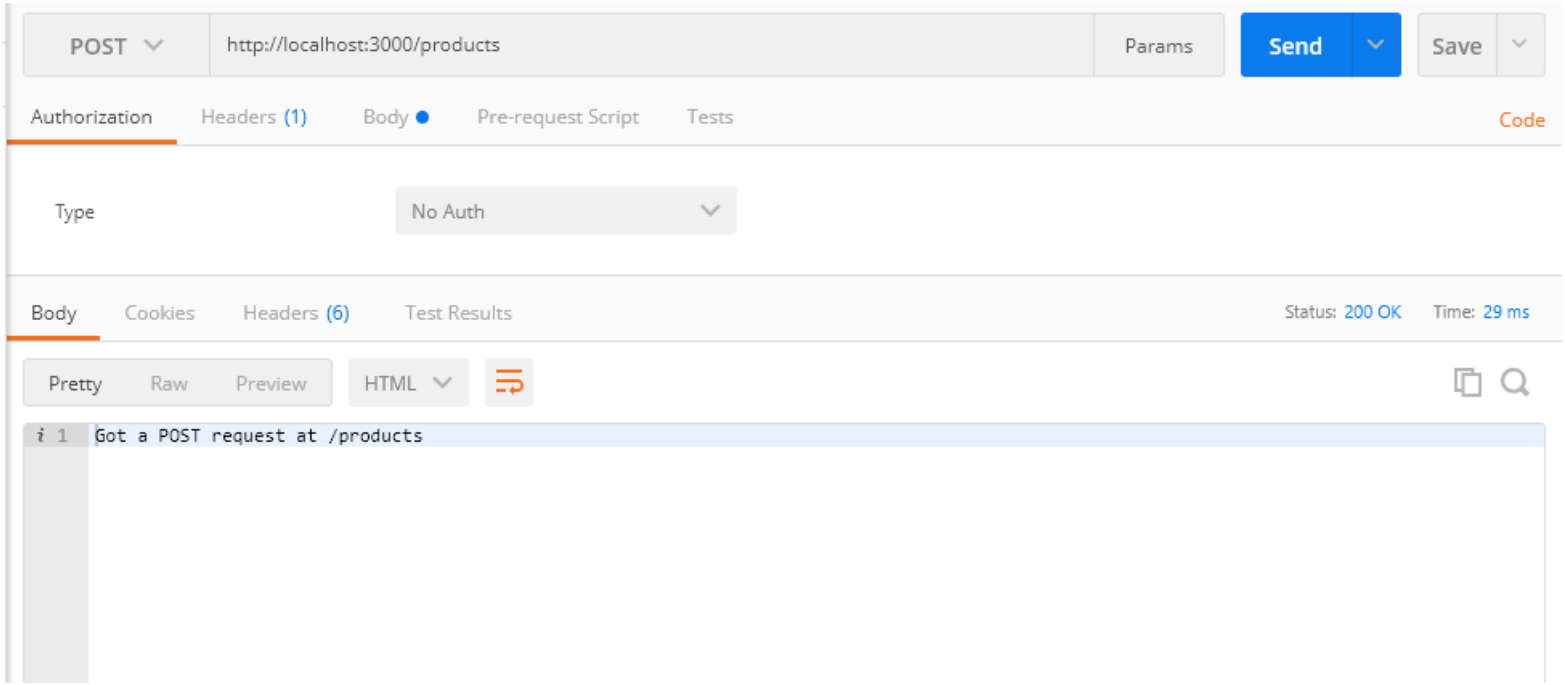
PostMan

- Create a new HTTP request



PostMan

- Choose the HTTP method from the drop-down list
- Enter the URL of the server
- Click Send



Route Paths

- Route paths, in combination with a request method, define the endpoints at which requests can be made
- Route paths can be strings, string patterns, or regular expressions
- The characters $, ?, +, *, and () are subsets of their regular expression counterparts
  - The hyphen (-) and the dot (.) are interpreted literally as part of the path
- If you need to use one of the special characters in a path string, enclose it escaped within ([and])
  - For example, the path string for requests at /data/$book, would be /data/([\$])book

Route Paths Examples

- The following route path will match acd and abcd:

```
app.get('/ab?cd', (req, res) => {
    res.send('ab?cd')
});
```

- The following route path will match abcd, abbcd, abbbcd, and so on:

```
app.get('/ab+cd', (req, res) => {
    res.send('ab+cd')
});
```

- The following route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on:

```
app.get('/ab*cd', (req, res) => {
    res.send('ab*cd')
});
```

- This following route path will match anything with an “a” in it:

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

Route Parameters

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL
- To define routes with route parameters, simply specify the route parameters in the path of the route, preceded by a colon (:)

```
app.get('/users/:userId', (req, res) => {
    res.send("userId: " + req.params.userId);
});
```

  - The captured values are populated in the **req.params** object, with the name of the route parameter specified in the path as their respective keys
- To send a parameter from the client insert its value at the proper place in the URL:

```
Request URL: http://localhost:3000/users/34
req.params: { "userId": "34" }
```

Route Parameters

- Example for sending more than one parameter in the URL:

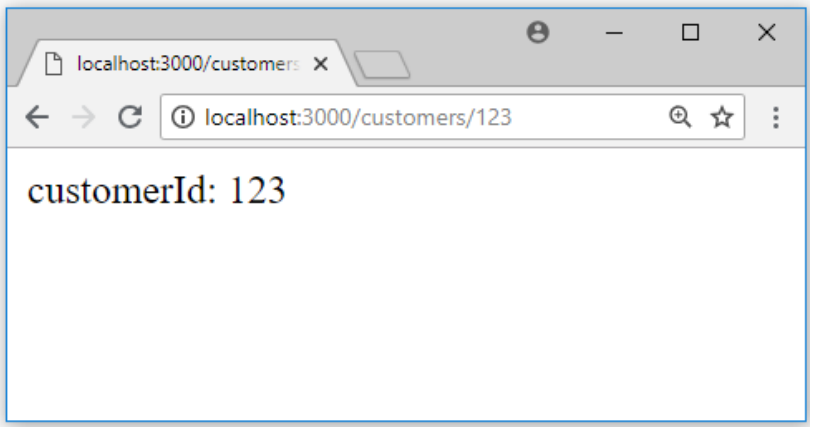
```
app.get('/users/:userId/books/:bookId', (req, res) => {
    res.send(`userId: ${req.params.userId}, bookId: ${req.params.bookId}`);
});
```

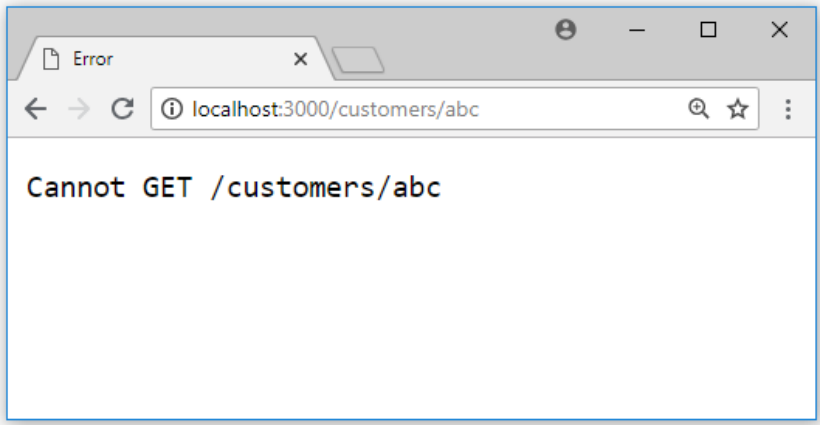
Route Parameters

- To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses (())
- For example, the following route will only match requests that have a numeric id

```
app.get('/customers/:customerId(\\d+)', (req, res) => {
    res.send("customerId: " + req.params.customerId);
});
```

  - Because the regular expression is part of a literal string, be sure to escape any \ characters with an additional backslash, for example \\d+.





Query String Parameters

- Query strings are not part of the route path
- They can be appended to any URL path
- The property **req.query** is an object containing a property for each query string parameter in the route

```
app.get('/users', (req, res) => {
    res.send(`<h2>Welcome ${req.query.firstname} ${req.query.lastname}</h2>`);
});
```

Query Params vs. Route/Path Params

- Route params are typically used to identify a specific resource or resources, while query params are used to sort/filter those resources
- Route params are part of the URL, thus they must be specified, while query params are optional
- For example, suppose you are implementing API endpoints for an entity called Car
- You could structure your endpoints like this:

```
GET /cars
GET /cars/:id
POST /cars
PUT /cars/:id
DELETE /cars/:id
```

- Now if you want to add the capability to filter cars by color, you could add a query parameter to your GET /cars request like this:

```
GET /cars?color=blue
```

Exercise (4)

- Create a web server that handles a repository of products
- Each product should have an id, name, and price
- The server should support the following operations:
  - Get all products – return a string with all the product names separated by a comma
  - Get product by id – get all product details (id, name and price)
  - Get product by name – get all product details (use query string param)
  - Add a new product – URL should have 3 parameters, e.g. /products/add/1/apple/2.5
  - Delete a product by id
- Test all the server methods by using PostMan

Exercise (4)

- Example:

POST http://localhost:3000/products/1/apple/2.5 Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Type No Auth

Body Cookies Headers (6) Test Results Status: 200 OK Time: 60 ms

Pretty Raw Preview HTML

```
1 Product successfully added
```

GET http://localhost:3000/products/1 Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Type No Auth

Body Cookies Headers (6) Test Results Status: 200 OK Time: 29 ms

Pretty Raw Preview HTML

```
1 Id: 1, Name: apple, Price: 2.5
```