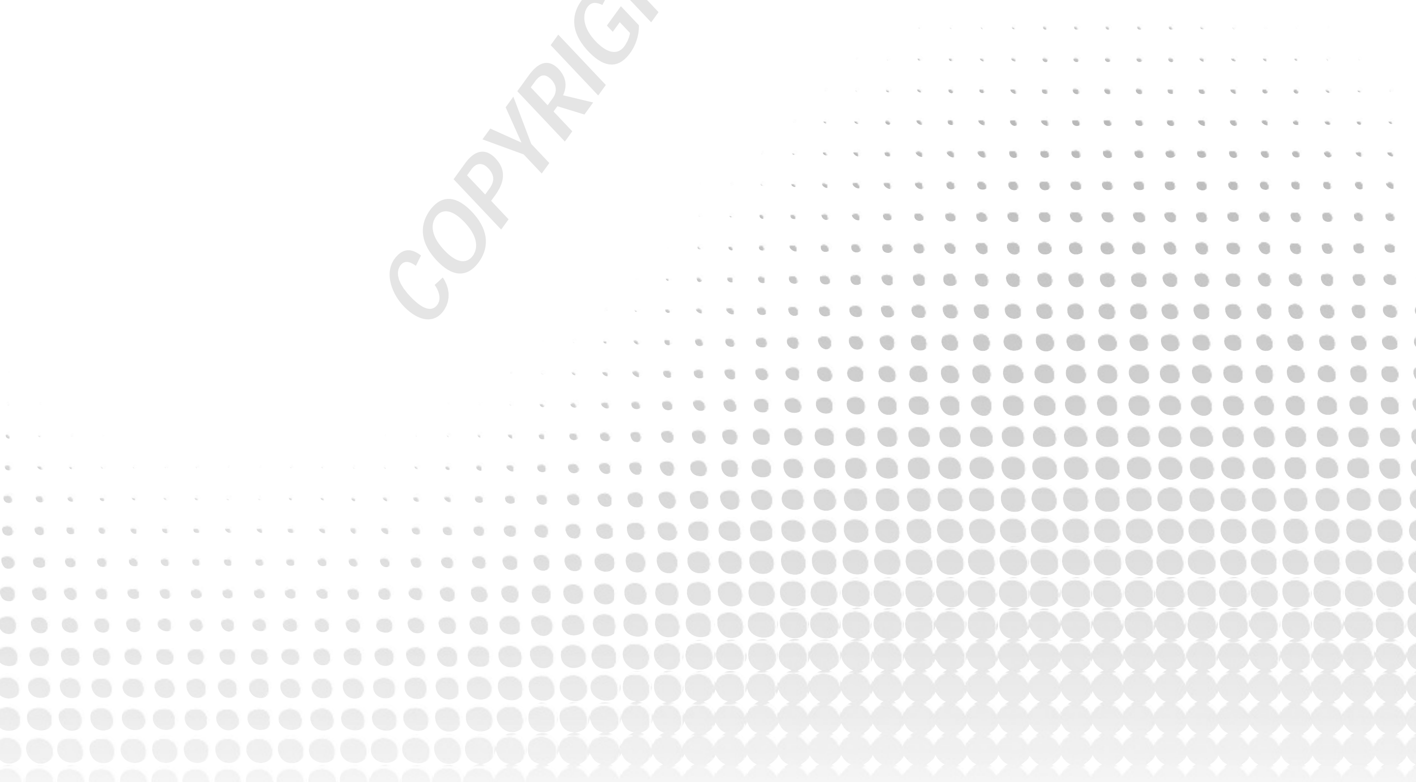
PART I
Introduction and Setup

- **CHAPTER 1:** Installing Node
- **CHAPTER 2:** Introducing Node

COPYRIGHTED MATERIAL



1

Installing Node

WHAT'S IN THIS CHAPTER?

- Getting Node up and running
- Installing Node Package Manager (NPM)
- Using NPM to install, uninstall, and update packages

At the European JSConf in 2009, Ryan Dahl, a young programmer, presented a project he had been working on. This project was a platform that combined Google's V8 JavaScript engine, an event loop, and a low-level I/O API. This project was not like other server-side JavaScript platforms where all the I/O primitives were event-driven and there was no way around it. By leveraging the power and simplicity of JavaScript, this project turned the difficult task of writing event-driven server-side applications into an easy one. The project received a standing ovation and has since then been met with unprecedented growth, popularity, and adoption.

The project was named Node.js and is now known to developers simply as Node. Node provides a purely event-driven, non-blocking infrastructure for building highly concurrent software.

> **NOTE** *Node allows you to easily construct fast and scalable network services.*

Ever since its introduction, Node has received attention from some of the biggest players in the industry. They have used Node to deploy networked services that are fast and scalable. Node is so attractive for several reasons.

One reason is JavaScript. JavaScript is the most widely used programming language on the planet. Most web programmers are used to writing JavaScript in the browser, and the server is a natural extension of that.

The other reason is Node's simplicity. Node's core functionalities are kept to a minimum and all the existing APIs are quite elegant, exposing the minimum amount of complexity to the programmers. When you want to build something more complex, you can easily pick, install, and use several of the available third-party modules.

Another reason Node is attractive is because of how easy it is to get started using it. You can download and install it very easily and then get it up and running in a matter of minutes.

The typical way to install Node on your development machine is by following the steps on the `http://nodejs.org` website. Node installs out of the box on Windows, Linux, Macintosh, and Solaris.

INSTALLING NODE ON WINDOWS

Node supports the Windows operating system since version 0.6.0. To install Node on Windows, point your browser to `http://nodejs.org/#download` and download the node-v*.msi Windows installer by clicking on the link. You should then be prompted with a security dialog box, as shown in Figure 1-1.

Click on the Run button, and you will be prompted with another security dialog box asking for confirmation. If you agree, the Node install wizard begins (see Figure 1-2).

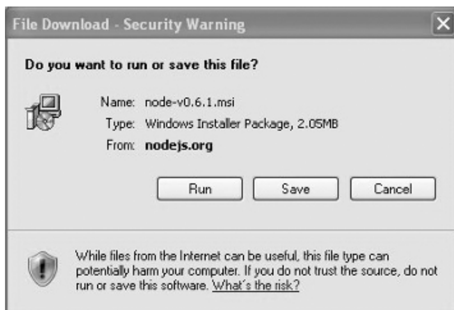


FIGURE 1-1

FIGURE 1-2

When you click Run, the Installation Wizard starts (see Figure 1-3).

Click on the Next button and Node will start installing. A few moments later you will get the confirmation that Node was installed (see Figure 1-4).

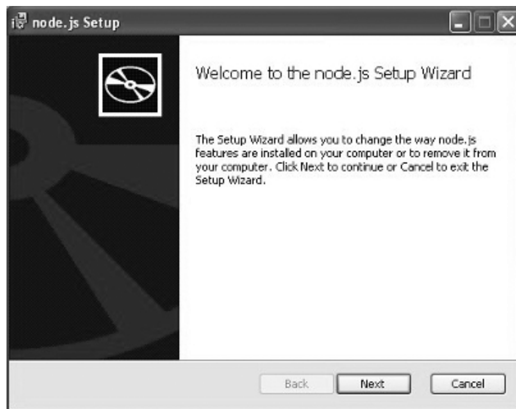


FIGURE 1-3

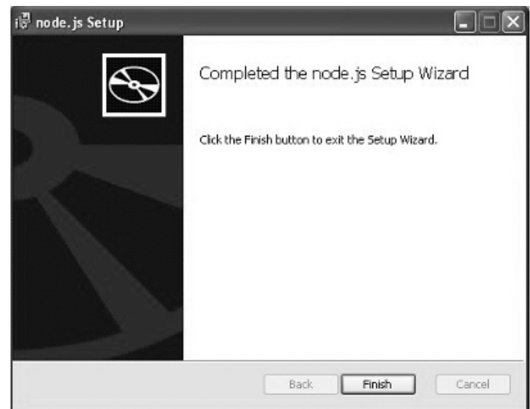


FIGURE 1-4

INSTALLING ON MAC OS X

If you use Mac OS X you can install Node using an Install Wizard. To start, head to `http://nodejs.org/#download` and download the node-v*.pkg Macintosh installer by clicking on the link. Once the download is finished, click on the downloaded file to run it. You will then get the first wizard dialog box, as seen in Figure 1-5.

Choose to continue and install. The wizard will then ask you for the system user password, after which the installation will start. A few seconds later you'll get the confirmation window stating that Node is installed on your system (see Figure 1-6).

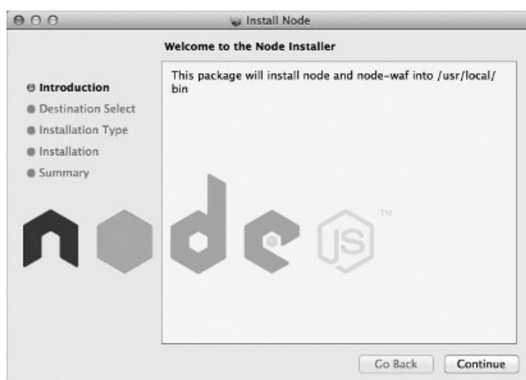


FIGURE 1-5

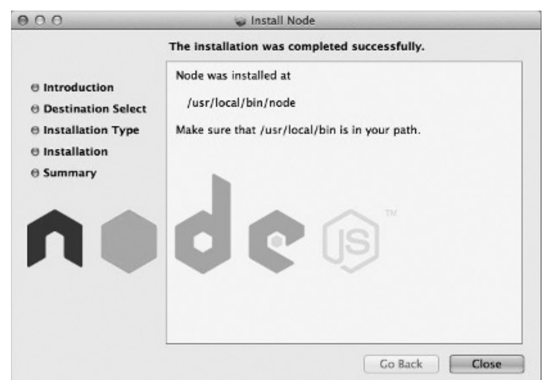


FIGURE 1-6

INSTALLING NODE USING THE SOURCE CODE

If you have a UNIX system, you can install Node by compiling the source code. First you need to select which version of Node you will be installing, then you will download the source code and build, install, and run Node.

> **NOTE** *Node depends on several third-party code libraries, but fortunately most of them are already distributed along with the program. If you are building from source code, you should need only two things:*
>
> - *python (version 2.4 or higher) — The build tools distributed with Node run on python.*
> - *libssl-dev — If you plan to use SSL/TLS encryption in your networking, you'll need this. libssl is the library used in the openssl tool. On Linux and UNIX systems it can usually be installed with your favorite package manager. The libssl library comes pre-installed on Mac OS X.*

Choosing the Node Version

Two different versions of Node are usually available for download on the `nodejs.org` website: the latest stable and the latest unstable.

In Node, the minor version numbering denotes the stability of the version. Stable versions have an even minor version (0.2, 0.4, 0.6), and unstable versions have an odd minor version (0.1, 0.3, 0.5, 0.7).

Not only might an unstable version be functionally unstable, but the API might also be mutating. The stable versions should not change the public API. For each stable branch, a new patch should include only bug fixes, whereas APIs sometimes change in the unstable branch.

Unless you want to test a new feature that is only available in the latest unstable release, you should always choose the latest stable version. The unstable versions are a battleground for the Node Core Team to test new developments in the platform.

More and more projects and companies successfully use Node in production (some of the most relevant are on the `nodejs.org` home page), but you might have to put some effort into keeping up with the API changes on each new minor stable release. That's the price of using a new technology.

Downloading the Node Source Code

After you choose a version to download, copy the source code *tarball* URL from the `http://nodejs.org` website and download it. If you're running in a UNIX system, you probably have *wget* installed, which means that you can download it by using a shell prompt and typing the following:

```
$ wget http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz
```

If you don't have *wget* installed, you may also use the *curl* utility:

```
$ curl -O http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz
```

If you don't have either of these tools installed, you have to find another way to download the *tarball* file into your local directory — such as by using a browser or transferring it via the local network.

(The examples in this book use the latest stable version at the time of writing, which is 0.6.12.)

Building Node

Now that you have the source code in your computer, you can build the Node executable. First you need to unpack the source files like this:

```
$ tar xfz node-v0.6.12.tar.gz
```

Then step into the source directory:

```
$ cd node-v0.6.12
```

Configure it:

```
$ ./configure
```

You should get a successful output like this:

```
'configure' finished successfully (9.278s)
```

Then you are ready to compile it:

```
$ make
```

You should get a successful output like this:

```
'build' finished successfully (0.734s)
```

Installing Node

When you have built Node, you can install it by running the following command:

```
$ make install
```

This will copy the Node executable into `/usr/local/bin/node`.

If you have a permissions problem when issuing this command, run it as the root user or using sudo:

```
$ sudo make install
```

Running Node

Now you are ready to start using Node. First you can simply experiment running Node as a *command-line interface* (CLI). For that you need only to call the Node executable with no arguments like this:

```
$ node
```

This will start the CLI, which will then wait for you to input an expression. Just to test the installation and see Node actually doing something, you can type:

```
> console.log('Hello World!');
Hello World!
> undefined
```

You can also run a JavaScript script from a file. For instance, if you create a file with this content:

```
console.log('Hello World!');
```

Name the file `hello_world.js`. Then run the file by passing the file path as first argument to the *`Node`* executable while inside a shell prompt. For example:

```
$ node hello_world.js
Hello World!
```

You can quit the CLI by typing Ctrl+D or Ctrl+C.

SETTING UP AND USING NODE PACKAGE MANAGER

You can only get so far using the language features and the core functions. That's why most programming platforms have a system in place that allows you to download, install, and manage third-party modules. In Node, you have *Node Package Manager* (NPM).

NPM is three things — a third-party package repository, a way to manage packages installed in your computer, and a standard to define dependencies on other packages. NPM provides a public registry service that contains all the packages that programmers publish in NPM. NPM also provides a command-line tool to download, install, and manage these packages. You can also use the standard package descriptor format to specify which third-party modules your module or application depends on.

You don't need to know about NPM to start using Node, but it will become necessary once you want to use third-party modules. Because Node provides only low-level APIs, including third-party modules is almost always necessary to fulfill any complex application without having to do it all yourself. As you will see, NPM allows you to download and play with modules without installing packages globally, which makes it ideal for playing around and trying things.

NPM and Node once required separate installs, but since Node version 0.6.0, NPM is already included.

Using NPM to Install, Update, and Uninstall Packages

NPM is a powerful package manager and can be used in many ways. NPM maintains a centralized repository of public modules, which you can browse at `http://search.npmjs.org`. A Node opensource module author may choose, as most do, to publish the module to NPM, and in the installation instructions should be the NPM module name you can use to remotely download and install it.

This section covers the most typical uses of NPM, which are installing and removing packages. This should be just enough for you to start managing your application dependencies on third-party modules published on NPM. However, first you need to understand the differences between global and local modes of operation and how these affect module lookups.

Using the Global versus the Local Mode

NPM has two main modes of operation: global and local. These two modes change target directories for storing packages and have deep implications for how Node loads modules.

The local mode is the default mode of operation in NPM. In this mode, NPM works on the local directory level, never making system-wide changes. This mode is ideal for installing the modules your application depends on without affecting other applications you might also have installed locally.

The global mode is more suited for installing modules that should always be available globally, like the ones that provide command-line utilities and that are not directly used by applications.

Always use the default local mode if you are in doubt. If module authors intend for one specific module to be installed globally, generally they will tell you so in the installation instructions.

The Global Mode

If you installed Node using the default directory, while in the global mode, NPM installs packages into `/usr/local/lib/node_modules`. If you type the following in the shell, NPM will search for, download, and install the latest version of the package named `sax` inside the directory `/usr/local/lib/node_modules/sax`:

```
$  npm install -g sax
```

> **NOTE** *If your current shell user doesn't have enough permissions, you will need to login as the root user or run this command within sudo:*
>
> ```
> $ sudo npm install -g sax
> ```

If you then have the requirement for this package in any Node script:

```
var sax = require('sax');
```

Node will pick up this module in this directory (unless you have it installed locally, in which case Node prefers the local version).

The local mode is the default mode, and you have to explicitly activate the global mode in NPM by using the `-g` flag.

The Local Mode

The local mode is the default mode of operation in NPM and the recommended dependency resolution mechanism. In this mode NPM installs everything inside the current directory — which can be your application root directory — and never touches any global settings. This way you can choose, application by application, which modules and which versions to include without polluting global module space. This means that you can have two applications that depend on different versions of the same module without them conflicting.

In this mode, NPM works with the `node_modules` directory under the current working directory. If you are inside `/home/user/apps/my_app`, NPM will use `/home/user/apps/my_app/node_modules` as file storage for the modules.

If you run a Node application installed in the directory `/home/user/apps/my_app`, Node will search this `/home/user/apps/my_app/node_modules` directory first (before trying the parent directories and finally searching for it inside the global one). This means that, when Node is resolving a module dependency, a module installed using the local mode will always take precedence over a module installed globally.

Installing a Module

Using the following command, you can download and install the latest version of a package:

```
$ npm install <package name>
```

For example, to download and install the latest version of the `sax` package, you should first change your current directory to your application root directory and then type:

```
$ npm install sax
```

This will create the `node_modules` directory if it doesn't already exist and install the `sax` module under it.

You can also choose which version of a specific module to install by specifying it like this:

```
$ npm install <package name>@<version spec>
```

You can use a specific version number in the `<version spec>` placeholder. For instance, if you want to install version 0.2.5 of the `sax` module, you need to type:

```
$ npm install sax@0.2.5
```

Under `<version spec>` you can also specify a version range. For instance, if you want to install the latest release of the 0.2 branch, you can type:

```
$ npm install sax@0.2.x
```

If you want to install the latest version before 0.3 you should type:

```
$ npm install sax@"<0.3"
```

You can get more sophisticated and combine version specifications like this:

```
$ npm install sax@">=0.1.0 <0.3.1"
```

Uninstalling a Module

When you use the following command, NPM will try to find and uninstall the package with the specified name.

```
$ npm uninstall <package name>
```

If you want to remove a globally installed package, you should use:

```
$ npm uninstall -g <package name>
```

Updating a Module

You can also update an installed module by using the following command:

```
$ npm update <package name>
```

This command will fetch the latest version of the package and update it. If the package does not exist, it will install it.

You can also use the global switch (`-g`) to update a globally installed module like this:

```
$ npm update -g <package name>
```

Using the Executable Files

It's possible that a module includes one or more executable files. If you choose to install a module globally and you used the default installation directory settings, NPM installs the executables inside `/usr/local/bin`. This path is usually included in the default executable `PATH` environment variable.

If you installed a package locally, NPM installs any executables inside the `./node_modules/.bin` directory.

Resolving Dependencies

NPM not only installs the packages you request but also installs the packages that those packages depend on. For instance, if you request to install package A and this package depends on package B and C, Node will fetch packages B and C and install them inside `./node_modules/A/node_modules`.

For instance, if you locally installed the *nano* package like this:

```
$ npm install nano
```

NPM outputs something like:

```
nano@0.9.3 ./node_modules/nano
├── underscore@1.1.7
└── request@2.1.1
```

This shows that the *nano* package depends on the *underscore* and *request* packages and indicates which versions of these packages were installed. If you peek inside the `./node_modules/nano/node_modules` directory, you'll see these packages installed there:

```
$ ls node_modules/nano/node_modules
request  underscore
```

Using *package.json* to Define Dependencies

When coding a Node application, you can also include a `package.json` file at the root. The `package.json` file is where you can define some of your application metadata, such as the name, authors, repository, contacts, and so on. This is also where you should specify extraneous dependencies.

You don't need to use it to publish the application to NPM — you may want to keep that application private — but you can still use `package.json` to specify the application dependencies.

The `package.json` is a JSON-formatted file that can contain a series of attributes, but for the purposes of declaring the dependencies you only need one: `dependencies`. An application that depends on the *sax*, *nano*, and *request* packages could have a `package.json` file like this:

```
{
  "name" : "MyApp",
  "version" : "1.0.0",
  "dependencies" : {
    "sax" : "0.3.x",
    "nano" : "*",
    "request" : ">0.2.0"
  }
}
```

Here you are specifying that your application or module depends on *sax* version 0.3, on any version of *nano*, and on any version of *request* greater than 0.2.0.

> **NOTE** *You may also find that if you omit the* `name` *and* `version` *fields, NPM will not work. This happens because NPM was initially conceived to describe public packages, not private applications.*

Then, on the application root, type:

```
$ npm install
```

NPM will then analyze the dependencies and your `node_modules` directory and automatically download and install any missing packages.

You can also update all the locally installed packages to the latest version that meets your dependency specifications, like this:

```
$ npm update
```

In fact you can always use this last command, because it will also make NPM fetch any missing packages.

SUMMARY

You've learned how to install *Node* and the *Node Package Manager*. You can now use NPM to install, uninstall, and remove third-party packages. You can also use the `package.json` file together with NPM to manage the third-party packages you depend on.

Now that you have Node and NPM installed, you are ready to experiment with them. But first you need some background on Node and the event-driven programming style.