A Flexible, User-Configurable, Homebrew,  
NodeJS-Based Web Server

By Dave Campbell

Keywords: web, server, NodeJS, JavaScript, configurable, homebrew, reverse, proxy

# Abstract

NodeJS implements server-side JavaScript and supports development of application specific web servers. This code implements a flexible user-configurable web server platform with a number of embellishments over the basic NodeJS examples. These embellishments include a reverse proxy for handling multiple servers/services on a single IP address, features and services defined by a user-configurableJSON file, and adaptable application based on user specified handlers loaded at startup.

# NodeJS

The NodeJS project, sometimes referred to as simply Node, implements a wrapper and libraries for the Google V8 JavaScript engine, the client-side interpreter integrated into the Chrome browser. This enables building high-performance, compact, and highly specialized applications on the server-side, called apps, i.e. web servers, with very little code – you write the server vs using a pre-compiled binary! The compact code makes it very easy to identify and follow exactly what the server does. This mantra follows the Linux methodology of small simple dedicated code for a single problem, achieving greater functionality by aggregating smaller tools through scripting. This simplifies debug and maintenance of the code.

Secondly, this approach means web developers can work with a single coding language, **JavaScript**, on both client and server sides. This greatly simplifies the learning curve and reduces errors introduced by constantly switching between languages. It also opens the door of web server development to a large body of developers already well versed in client side JavaScript.

The simplicity of the NodeJS servers generally means writing a dedicated sever to handle each specific task. Support for proxies means a single website can handle many services with transparent integration, seemingly appearing as a single site.

NodeJS comes full featured with support of all web protocols, including modern web sockets and streaming. It integrates full system access with file system, database bindings, and other OS features. Libraries include tools for encryption, compression, parsing, dynamically linked C/C++ addons, and many others. These features give NodeJS capabilities rivaling the best of web servers.

NodeJS uses a single-thread, event-driven, asynchronous processing model that yields excellent performance with minimal resources. Handling the asynchronous behavior in code imposes a burden, mostly in form of a learning curve for the paradigm shift, but minimizes system resources. A small Raspberry Pi based system, for example, can handle an impressive, hundreds to potentially thousands of requests per second, suitable for a small office Intranet server.

# Homebrew Web Server Architecture

This web server application suits small sites providing a few services (e.g. ~10 hosts or services) including site-specific custom apps, websites, blogs, data logging, web sockets, etc.

The web server consists of four parts: the server setup script, a reverse proxy, base server application, and function handlers. Breaking the server down into these simple elements partitions the work and makes understanding each element easier and adaptable as necessary. Additionally, it allows individual server functions to reside on independent hardware, if so desired, for greater throughput.

## Server Setup Script (homebrew.js)

The setup script has very basic functionality that executes the actions necessary to configure all of the server elements. It does not directly perform any server operations related to client requests. The setup script requires a JSON configuration object that defines setup actions and behavior. The script performs the following actions in the order listed:

1. Load internal module dependences.
2. Read configuration file.
3. Establish the environment context used by other elements.
4. Start the transcript logger.
5. Load shared module dependences.
6. Start dependent services used by apps such as database servers.
7. Spawn child processes for each “site” server defined in configuration.
8. Start any specified reverse proxy servers.

See *User js* section for details on running the homebrew.js script.

### Module Dependencies

Modules in NodeJS provide the means for partitioning or modularizing code. NodeJS has a fairly well developed mechanism for loading and caching modules to prevent redundant copies and partition each in its own namespace. The setup script loads a number of fundamental modules needed by other parts of the server code. The configuration can optionally specify additional modules to load and individual apps may load additional modules too.

### Configuration

The script assumes use of the file ../restricted/config (either .js or .json) for configuration by default, although you can specify another file on the commandline. This places the configuration in a parallel folder with limited access permissions. See the *Configuration File* section for details.

### Environment Context

The shell environment passes operating system context to the NodeJS apps. Internally, the setup script aggregates a number of data sources in addition to the shell environment, such as configuration data, transcript instance, and database handles, which it passes to each app for context as the cfg variable.

### Transcripting

The startup script implements transcripting for all server, proxy, and child app actions to record and debug server responses. Transcripting outputs all messages with a standardized format including a timestamp, filter level, originating module tag name, and message. The cfg specifies the output log file and filter level. The homebrew transcript object passes as part of the context so that all child apps can transcribe to a master transcript if so desired, or define their own independent transcript. The default homebrew server transcripts everything to a single file.

### Dependent Services

The startup script starts global dependent services as configured, for example database servers, so that each child app can reference them. References pass in the cfg object.

### Site Apps

Site apps represent dedicated action specific web services – the actual web server processes. With setup configuration completed, the startup script loops through each defined application and spawns a child process to run it. It starts by loading any specified required modules. It passes the child a site-specific configuration variable that includes global configuration information as well. The configuration may specify any application per site. The provided hbBaseApp.js suffices for most applications.

### Reverse Proxy Service

Once the startup has completed spawning the child apps it starts any reverse proxies. The proxies essentially serve as a router to redirect requests to the appropriate app. See the *Reverse Proxy (hbProxy.js)* section below for details.

## Reverse Proxy (hbProxy.js)

A reverse proxy, or proxy for short, simply allows multiple servers to operate from a single IP address. (Here for simplicity, I use the terms proxy and reverse proxy interchangeably. Technically, a normal proxy routes many users (clients) through a single portal (i.e. IP), where a reverse proxy operates backwards redistributing requests made to a single portal to many backend services.) It represents little more than a traffic switch and filter. The reverse proxy differentiates requests to independent servers by hostname. It routes requests for each specific host to its designated server.

For example, you could define two hosts blog.example.net and data.example.net to point to the same IP address. Web requests for either host directed to the IP:80 first go to the proxy. The proxy simply examines the hostname associated with the request and forwards it to the appropriate backend server. **It no server exists for the specified hostname, the proxy rejects the request.** This filters *probing* requests made direct to the IP address without a hostname as well as requests to invalid hostnames. **Aliases** in the configuration of each site enable the use of many names for the same service. For example, using blog as a local network reference for the longer blog.example.net.

Although not all applications require the reverse proxy, specifically those involving a single service, it offers basic request filtering and therefore benefits all applications to exclude bogus or unsolicited requests. Note too, the reverse proxy can serve other services not part of the NodeJS server, such as an independent IP camera. The proxy also enables services to run on physically different server hardware.

## Base Server App (hsBaseApp.js)

The hbBaseApp.js script implements an easily configured generic server application. You can replace or alter as needed per site to provide unique services. hbBaseApp.js gets its flexibility from the ability to load unique handlers for each site specified in the configuration file.

The base app and included handlers enable basic web sites supporting both static pages and dynamic content built from database queries. It also serves as a model for developing code for specialized sites as it handles all the necessary elements. Each site may operate from independent hardware as well.

## Handlers ()

The handlers represent a group of function specific scripts to perform a singular task such as a serving static content, performing data queries, user authentication, and so forth. The configuration file defines an array of handlers for each site, which load and initialize in the defined order of the array. The app script may load the same handler twice with different options to give precedence or finer control. For example, you may need to serve multiple root folders with the static handler.

Handlers load with the app.use signature of the Express framework known as middleware. Handlers “chain” together in the specified order. Each handler generally tests the request URL against a “pattern” and takes action when a match occurs or passes the request on down the chain if no match occurs. Handler action may complete a response to a request and terminate the request/response or simply preprocess the request in preparation for a later handler or throw an error. Handlers can perform just about any action possible, including returning fixed file content, creating content dynamically on the fly, signaling events, streaming content including web sockets, or even performing hardware actions such as communications with sensors. A default or specified error handler completes the chain to process all failed or unrecognized requests.

WARNING: All requests must terminate or the server will hang or generate an internal error and may stop functioning. Therefore, always ensure that the logic of a given handler captures and handles all possibilities.

### Handler Signature

Each handler follows the Express Middleware signature. This means a handler module should export a single-argument configuration function that in turn returns a function object with the triple-argument Express middleware signature given as

var handler = function handler(optionsObj) {

// handler setup code called once during startup …

return handler\_middleware(request, response, next) {

// handler code called from setup code context for each server request

};

};

exports = module.exports = handler;

Alternately as error handling middleware, the handler may return a four-argument middleware signature given as

return error\_handler\_middleware(error, request, response, next) {…};

The handler module may define the middleware callback directly as well, if it does not require any configuration.

At startup, the app examines the function signature and either loads the middleware function on the application stack or calls the handler with the application context. That is, the configuration function is called with a configuration object (i.e. optionsObj), which returns the middleware handler function to app.use to become part of the request/response chain. The app in turn calls middleware handler function for each server request, unless an earlier middleware function completes the request.

Tip: See Express documentation for details or review included handler code examples.

Tip: Remember that the Express app calls handlers sequentially so the order defined in the configuration file matters.

### Included Handlers

You can write handlers to perform any desired function. Some handlers needed to support basic website operations, which you may easily modify as needed, include:

#### account.js

The account.js handler provides functions necessary for defining and managing user accounts.

#### database.js

The database.js handler implements a simple database interface primarily for Internet of Things (IoT) data services. It also supports basic functions needed to deliver dynamic content.

#### fileserve.js

The fileserve.js handler provides directory listing and file serving for repositories.

#### info.js

The info.js handler provides diagnostic information for a server and connections, retricted to local network connections.

#### mapURL.js

The mapURL.js handler provides URL redirect and rewrite capability for a server.

#### notify.js

The notify.js handler supports user notifications via email, twitter, and such. Notify creates server instances for the parent app and attaches callback functions to the request object under the hbNotify key. Simply call rqst.hbNotify.sendmail(mailObj,[callback]) or rqst.hbNotify.sendText(textObj,[callback]).

#### static.js

The static.js handler serves static content such as html, images, and JavaScript libraries.

#### templates.js

The templates.js handler serves dynamic pages built from templates, typically Pug or Markdown.

#### utility.js

The utility.js handler implements a number of different site metric options, such as logging requests and returning server information.

## Supporting Modules

The project divides the NodeJS JavaScript code into a number of modules for easier code management and maintenance. The following describes supporting modules used by the main scripts and modules not described above.

### Cleanup.js

The cleanup.js module captures the CTRL-C break signal and performs and orderly shutdown of the server and its child processes. The user provides a callback to cleanup any application specific things prior to stopping the server.

### Crypto.js and CryptoPlus

The Crypto.js and CryptoPlus.js library add various encryption and hash convenience wrapper functions for the NodeJS crypto library.

### Extensions2JS.js

The Extensions2JS.js library defines a number of useful object extensions used in the homebrew coding, such as Date formatting, string padding, object key iteration, and more.

### Laundromat.js

The Laundromat.js library implements generic JSON input filtering. It can filter any data including a hierarchical JSON object. It defines a number of routine filters for numbers, text strings, and such as well as configurable regular expression filtering.

### Scribe

The Scribe.js module provides a transcripting object to log all web server tasks, setup, and actions. The startup script creates a scribe instance in the global namespace that each child app inherits as a reference in the configuration environment, cfg, or apps can establish independent scribe instances. By inheriting the reference all transcripting routes to the same file.

### WrapSQ3.js

The dbjs.js library provides asynchronous wrapper functions for simplifying SQLite database queries. The functions enable simple high-level access calls from web services. See the Web Server Database Interface document for more details.

## Configuration

Configuration assumes a JSON file or a JavaScript (.js) file that returns a JSON object with the default name ../restricted/config. Node will (automatically) first look for a .js file of the name specified to load. If not found it will next look for a .json file. If the startup script does not find either a .json or .js file and you do not specify another configuration file, the startup script throws an error and terminates as it cannot operate without a configuration.

WARNING: For secure operation, never place restricted files (i.e. databases and configuration) or scripts (modules) under a site root where malicious users might access and alter them maliciously to gain server control.

A JSON file follows a very strict verbose syntax (see json.org) and does not allow comments, but does not require knowledge of JavaScript programming, especially for those familiar with JSON support within other languages. On the other hand, you can create a basic JavaScript based configuration file with very little knowledge of JavaScript. A JavaScript file offers more flexibility such as dynamic construction based on flag options as well as the use of comments to document configuration.

A JavaScript configuration (i.e. config.js) should export a single object such as defined by the following template.

// JavaScript based configuration file

var cfg = {

// configuration definition goes here …

};

exports = module.exports = cfg;

See the provided config.json and config.js files for examples and description of configuration fields.

# User js

Another advantage of the NodeJS server involves running as a non-privileged service. This means it can run as a standard user and does not require root or administrator privileges to run, which greatly lowers the security risk should someone breech the server. For this create an account for user js. Be sure to remove user js from the adm and sudo group to limit admin permissions.

sudo gpasswd -d js sudo adm

sudo gpasswd -d js sudo sudo

Run the scripts from the user account. The server assumes, but is not necessarily limited to, the following directory structure:

/home/js

bin

homebrew.js

hbProxy.js

hbBaseApp.js

other JavaScript handlers…

restricted

configuration file(s)

site database(s)

sites

web service A root

static

images

templates

web service B root

…

The structure and folder names do not matter as long as the configuration file properly references the correct locations, which you can specify relative to the bin run directory. I recommend a separate folder for each web service for security as well as organization. Each site root tree may be organized as desired. However, for secure operation remember never to place the restricted files under a site root. Note also that the JavaScript files do not require executable privileges since you run node, which calls the scripts, but DO require read and execute privileges for Windows Sharing.

## Install Node

The repository will likely not have the latest (or even a recent) version of NodeJS. The best option involves getting the appropriate binary from *https://nodejs.org/en/downloads*. I recommend the latest (long-term support) LTS version. Run the following command.

uname –a

Linux tienda 4.4.50+ #970 Mon Feb 20 19:12:50 GMT 2017 armv6l GNU/Linux

This will report the processor version such as armv6l above. Use this to select the correct binary. Then as superuser (su) download and “install” as follows, assuming the correct tar file. (The install simply puts the files in the /usr/local/bin, /usr/local/lib, … folders.)

cd /usr/src/node

wget https://nodejs.org/dist/v6.10.0/node-v6.10.0-linux-armv6l.tar.xz

tar -xf node-v6.10.0-linux-armv6l.tar.xz --directory /usr/local --strip-components 1

You can also build NodeJS from source following the instructions in *Building a Raspberry Pi Office Network Server* document, Appendix E.

### Install Node Modules

After installing node, you must install any modules used using the node package manager, npm. For example,

npm install colors

The packages needed depends on the specific server and modules used, but requires at least the packages needed by the main homebrew scripts including: async, body-parser, colors, express, http-proxy, multer, and sqlite3. The notify module requires email and voice. The templates module requires mustache, pug, and showdown.

## Running Homebrew Script

The homebrew.js script simply runs from a terminal window as (one of the following) commands:

node homebrew.js [<configuration\_file.json>]

NODE\_ENV=production node homebrew.js [<configuration\_file.json>]

The second command sets the NODE\_ENV environment variable before executing the command. By default, NODE\_ENV=development, which enables additional debug and may impact other NodeJS library settings.

The node server must run even in the event that user js logs out (ending the terminal session), usually referred to as a daemon service run as the root user. The terminal multiplexer, tmux, enables doing so for a normal (non-privileged) user as it starts a background terminal service disconnected from the user login. You can configure tmux to start a session at boot as well to automatically start the server anytime the machine runs. For example, the following creates a new window and runs the homebrew.js server script from the /home/js/bin folder using the configuration in /home/js/restricted/config.js. The bash -i appended to the end keeps the bash open in interactive mode after the script ends this is useful for debugging script crashes.

# web server

/usr/local/bin/tmux new-window -c /home/js/bin -n server '/usr/local/bin/nodejs ./homebrew.js ../restricted/config.js; bash -i'

See the tmux command for details. See *Building a Raspberry Pi Office Network Server* document, Appendix D.

## Ports

Linux treats ports less than 1024, such as the default http ports of 80 and 443, as privileged ports, meaning they require root permissions to operate. In order for the NodeJS server to operate as non-privileged service it must use ports of 1024 or greater. Default configuration uses 8000 (proxy), 8001, 8002, … (services). This means you need to forward (default port 80) requests to port 8000 by either of two methods. For routers that support port redirection as well as port it’s simply a matter of forwarding WAN:80 to PROXY:8000, where WAN represents the router external IP and PROXY represents the machine running the proxy. If your router does not support port redirection, then forward WAN:80 to PROXY:80 and use iptables command to redirect port 80 to port 8000 by the following steps.

Run the commands to define the needed routing (as su).

# redirect port 80 to unprivileged port 8080

iptables -A PREROUTING -t nat -p tcp --dport 80 -j REDIRECT --to-port 8080

# optionally redirect port 443 to unprivileged port 8443

#iptables -A PREROUTING -t nat -p tcp --dport 443 -j REDIRECT --to-port 8443

To restore the IP routing every time the interface starts up, install the following:

sudo apt-get install iptables-persistent

The iptables-persistent install will build rules from the existing configuration. Save future configuration changes using (as su).

iptables-save > /etc/iptables/rules.v4

1. Mastering JavaScript

There are many excellent resources available for learning JavaScript. The follow identifies and briefly describes several.

## W3 Schools

The W3 Schools site ([www.w3schools.com](http://www.w3schools.com)) provides interactive training for all kinds of tools and languages related to web sites, including JavaScript. The short tutorials focus on client-side JavaScript but offer a fairly comprehensive JavaScript training since server-side JavaScript differs very little, mainly in modularity and additional functionality not supported by clients. W3 Schools has tutorials on related subjects as well, such as jQuery.

## Mixu’s Node Book PDF

While not really a book, but rather a PDF, Mixu’s Node Book provides a detailed understanding of the inner workings of NodeJS, how it does what it does, to give great insight into understanding Node and the asynchronous programming paradigm.

## Code Web Sites

Supporting organizations, including <https://www.javascript.com>, <https://nodejs.org>, <https://angularjs.org>, [https://jquery.org](https://jquery.org/), <http://www.json.org>, …, maintain websites and wikis documenting their respective libraries and tools. These sites generally provide valuable documentation and reference material.

## O’reilly Books and Media

O’reilly Books and Media specialize in technical publications, particularly software language references. As such they offer a number of books for learning and using JavaScript with a few noted below.

### Learning JavaScript

This books outlines basic JavaScript concepts, practices, and code writing examples. It gives entry-level instruction in learning JavaScript.

### Learning NodeJS

Learning NodeJS deals with JavaScript specific to NodeJS and covers asynchronous programming methods related and used with NodeJS. It focuses on learning about NodeJS programming issues and methods to deal with them. Definitely required reading for building NodeJS apps.

### Learning jQuery

jQuery implements a very popular cross-platform library that simplifies writing very terse client-side JavaScript.

### Learning AngularJS

AngularJS represents a popular cross-platform library for building dynamic web content. It plays well with NodeJS to build highly functional content compatible with NodeJS.

1. Kiosk App (Message Board)

The message board application, commonly called a kiosk, functions as an autonomous web client (browser) operating in full screen. It represents an extension to the web server because it requires some specific client side setup. The setup here assumes that the kiosk ***client*** runs on the same machine as the web ***server***, although this is not required, it minimizes the required hardware as a single machine can perform both tasks.

## Chromium Browser Setup

Install the chromium browser and X-windows utilities as

sudo apt-get install chromium-browser x11-xserver-utils unclutter

You may use another browser, such as IceWeasel (i.e. Firefox on RPi) as long as you can define it to operate in full screen mode from the commandline.

## Kiosk User Setup

Setup a kiosk user account that has limited permissions. This limits vulnerability since the account gets configured for autologin. Then edit the LightDM configuration …

sudo nano /etc/lightdm/lightdm.conf

Add the following lines under the [SeatDefaults] section.

autologin-user=kiosk

autologin-user-timeout=0

This will automatically login user kiosk when the machine reboots. Then edit the user kiosk configuration …

sudo nano /home/kiosk/.config/lxsession/LXDE-pi/autostart

Add the following lines.

@chromium-browser --noerrdialogs --kiosk --incognito file:///home/kiosk/Documents/index.html

@xset s noblank

@xset s off

@xset -dpms

@unclutter -idle 0.1 -root

This will autostart Chromium in full-screen or kiosk mode after the kiosk login completes. You can replace the file:///home/kiosk/Documents/index.html reference with the link of your choice. This configuration uses a local file to allow the local web server time to start before making a call to the actual message board content.

### /home/kiosk/Documents/index.html

This file performs two functions:

* It displays a simple notice informing the user that it is waiting on the server.
* It polls the server for content. Once it responds, it redirects to the provided site content.