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, basic middleware modules for common operations such as database management, and site features and services provided by function specific handlers defined by a user-configurable JS or JSON definition. This offers an adaptable user specified application versus a unique rewrite of code per application.

# 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 compact, high-performance, specialized applications or apps on the server-side, that is, dedicated web servers requiring very little code – **you write the server vs using a pre-compiled binary!** The compact code eases code testing and debug. 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 more. 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. 1-10 hosts or services) including site-specific custom apps, websites, blogs, data logging, web sockets, etc.

The homebrew 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 collectively all of the server elements – a convenience only. It does not directly perform any server operations related to client requests. The setup script requires a JavaScript or 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 and apps.
4. Start the “global” (i.e. homebrew scope) transcript logger.
5. Define shared database connections.
6. Start shared services used by apps, such as notification (i.e. email and SMS) service.
7. Start child “apps” for each “site” server.
8. Start any specified reverse proxy servers.

As such the homebrew script loosely represents a framework for the homebrew server design. The homebrew.js script likely requires no user modification for use over a variety of site applications.

See *User js* section for details and notions 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

By default, the script assumes use of the file ../restricted/config (either .js or .json) for configuration, 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 a global, meaning homebrew or top level scope, transcripting object called scribe. This captures all transcript messages from the server, formats them in a standardized form with a timestamp, (filter) level, originating source tag, and message, and outputs the result to a transcript file and/or in a level specific color to the console to record and debug server responses. The scribe passes in the context to each application, and optionally to each middleware module. This enables everything to log consistently to a single transcript regardless of the source. Alternately, each app or module may use to inherited scribe or perform their own transcripting. Transcript level corresponds to the scribe method used. See *Scribe* module description for details.

### Dependent Databases and Shared Services

The startup script initializes homebrew server wide database connections as specified. References pass in the cfg.db object to each site application so that each child app can share them.

The startup script also starts global dependent services per a configuration file, for example the Notification service that provides email and SMS capability, so that each child app can reference them. The script configures services according to the cfg.shared definition and pass references in the cfg.services object to each site application.

### 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 “site” from the configuration and starts a child app with server context to perform the specific service. It loads any specified required module. It passes the child a site-specific configuration variable that includes global (i.e. homebrew wrapper level) context and configuration information such as shared database connections. The configuration may specify any application per site, such as the provided hbBaseApp.js that suffices for many applications.

### Reverse Proxy Service

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

### Homebrew Context

The homebrew.js script calls each app with a server and app specific context, which represents the JavaScript scope (i.e. this) for the app. The script builds the context sequentially as it runs as a JavaScript object including the following definition:

site: site specific configuration directly from the configuration file

env: the homebrew system environment

headers: (aggregation of a default, global, and site specific)

"x-powered-by": "Raspberry Pi Homebrew NodeJS Server " + VERSION (default)

specified global headers (from the homebrew configuration)

site headers (from the site specific configuration, given precedence)

tag: given in site specific configuration or the key used for the site

db: shared database connections defined by the homebrew configuration

scribe: homebrew transcript instance

services: shared services, such as email and SMS suppport

### Homebrew Internals (hbInternals.js)

The hbInternals.js module encapsulates internal functions used by all parts of the homebrew server. It represents a modular element of homebrew.js as opposed to separate functionality. It implements:

***Events Emitter****: A server level events manager. Any part of the server may attach event handlers and emit events for interaction. Primary use includes commanding the secure proxies to reload renewed certificates.*

***Stat Object****: A server level object to set/get variables data. It enables collecting statistical data about server operations such as the number of server requests, errors, etc.*

***App Server****: An internal web server, not exposed to the open Internet. This server enables primitive communication with the homebrew internal operations, specifically access to the Events and Stat object. For example, curl http://localhost:8081/STAT/sc/served, returns a JSON object identifying the number of requests to site sc. Similarly, curl http://localhost:8081/CMD/HTTPS/renew commands the HTTPS proxy to reload renewed certificate files.*

## 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) to the Internet, 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 only. 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 (or IP:443 for https requests) 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 many *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 multiple 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 potentially for load balancing as an example or connecting to an IoT device as another.

Again, like homebrew.js, for most applications, the hbProxy.js script likely requires no user modification, just configuration.

### Secure Proxy

The Homebrew proxy supports secure service by simply supplying paths to the private key and certificate files. Back ends can operate as either http or https servers, which enables securing open services behind secure proxy access.

Note: Because of the shift to a “secure web by design” where browsers force using https even when an application doesn’t require it, many http services such as IoT devices can benefit from a front end https proxy server.

## Base Server App (hbBaseApp.js)

The hbBaseApp.js script implements an easily configured generic or non-specific server application. You can customize as needed per site to provide unique services simply by configuration without code rewrite. hbBaseApp.js gets its flexibility from the ability to load unique handlers, equivalent to middleware, 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 most necessary elements. Each site may operate from independent hardware as well.

### hbBaseApp.js Versus NodeJS Model

Perhaps technically, the hbBaseApp.js script contradicts the notion of the NodeJS app model. It adds some complexity to implement a single configurable multipurpose app, whereas NodeJS using a framework such as Express makes writing individual apps (appear) trivial, although not nearly so in real application practice. While true, the user can chose to write site-specific apps, as the homebrew.js script and top-level configuration does not require use of the hbBaseApp.js. The homebrew.js script can call any number of unique site apps. Alternately, using the hbBaseApp.js allows creation of many sites with no code development.

#### hbAuth.js

The hbAuth script represents a special hbBaseApp “built-in” handler that provides functions to support user authentication and authorization. This module integrates a number of functions to implement an all-in-one service. It gets loaded automatically to the request/response chain (if <site>.app.options.auth is defined) in order that the route gets properly set and it has precedence over other middleware. It:

* Initializes the user’s database, loads recipes from the definitions tables, adds a wrapper to the users database for simplified calls, and creates a sessions cache at startup.
* Builds an hbSession object and hbIsAuth function and attaches them to the request object for downstream middleware reference.
* “Harvests” ***user/API authentication data*** and ***session ID*** (i.e. hsid) from request headers, post data, the URL query string, or cookies, in that order, to build the hbSession object, defined as:

id: unique homebrew session object ID, AKA hsid.

auth: object containing any submitted user credentials, i.e. username, hash, …

\*api: object containing any submitted api credentials, i.e. key, salt, epoch, hash, …

\*user: object containing user records, i.e. username, identification, credentials, …

* Performs user POST actions.
* Perform user GET items.

\*Note, the api and user keys represent alternate optional keys. For an API response an api key replaces the auth key. The module stores user records in the cache keyed to the hsid. If the hsid key exists in the cache, meaning a prior validated login, it adds the user record to the session object.

##### hbIsAuth()

The hbIsAuth function offers a means for other middleware to validate users, api calls, and authorization permissions against the session information. It returns a truth-y/false-y response.

* **hbIsAuth({check: 'login', user: {…}}).** Validates a user login. Returns the session 'id' when valid or empty string when validation fails.
* **hbIsAuth({check: 'challenge', code:'…',challenge: {…}}).** Validates a challenge code against a user challenge credentials. Returns true/false.
* **hbIsAuth({check: 'api'}).** Validates current session API parameters. Returns true/false.
* **Authorizations:** (Return true/false).
  + **hbIsAuth().** Just checks for a valid user by looking for 'id' key in hbSession.
  + **hbIsAuth({user: <username>}).** Validates a specific user.
  + **hbIsAuth({service: ''}).** Authorizes current user as long as permission != 'DENY'.
  + **hbIsAuth({service: 'permission'}).** Validates a user authorization against a specific service for specified permission, such as read, write, or admin. A user permission of '\*' always returns true for any specified permission.

##### User Post Actions

The user post handler provides functions necessary for defining and managing user accounts. It depends on hbIsAuth. It includes the functions:

account: create a new user account or updates a validated user

POST /user/account/<username> (user data in body)

activate: change account status to active with a vaild challenge code

POST /user/activate/<username>/<code>

admin: admin management for services, admin access only

POST /user/authorize/<admin> (user(s) authorization data)

code: generate a code stored in user.credentials.challenge and sent by SMS

POST /user/code/<username>/<mode> (mode 'text' or default numeric)

list: list user(s) for admin authorization, admin access only

POST /user/list/<admin>/<username> (optional username or all users)

login: validate user local login credentials

POST /user/login/<username>/<auth\_key> (user credentials data)

logout: terminate user access

POST /user/logout/<hsid>

reset: reset a user local login password

POST /user/reset/<username>/<challenge> (user credentials data)

Additional data, indicated in parenthesis, usually included as rqst.body data, or as a header or query string where applicable.

##### User Get Items

TBD

## 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. Handlers implement the Express framework signature known as middleware.

Note: The handlers defined for the homebrew server follow the Express middleware signature. Handlers can be used with custom site apps other than hbBaseApp.js, but assume a few items specific to the homebrew ecosystem and may not directly play with other Express apps. For example, most handlers assume use of the Scribe object for transcripting and therefore a dependency for the Scribe module and/or a Scribe object in the options parameter.

The configuration file defines an array of handlers for each site, which load and initialize in the defined order of the array, just as an Express app entails ordered loading of middleware. 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 “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. Alternately, the configuration may specify a handler route pattern and/or method, which only calls the handler when the current request path matches the route and the request method matches the specified method. Handler action may complete a response to a request, which terminates the request/response flow, 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 with a response 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

As stated, 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(options) {

// handler setup code called once during startup …

return handlerMiddleware(request, response, next) {

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

};

};

exports = module.exports = handler;

Alternately, as with Express, for error-handling middleware, the handler must return a four-argument middleware signature given as

return errorHandlerMiddleware(error, request, response, next) {…};

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

Note: The handlerMiddleware (or errorHandlerMiddleware) function may be anonymous, but naming it specific to the handler makes debug easier as the trace back specifies the function name.

At startup, the app examines the handler signature and either calls the handler with the handler specified options and application context (i.e. this scope) or directly loads the middleware function on the application flow stack. That is, if the configuration function is called with a single configuration object (i.e. *options*) argument, the app calls the middleware with site context to initialize it, which returns the middleware handler function to app.use (or other specified method) to become part of the request/response chain. The app in turn calls the returned middleware handler function for each server request, unless an earlier middleware function completes the request or no route match occurs.

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. The homebrew server package includes some handlers needed to support basic website operations, which you may use as is or modify as needed for use. These include:

#### database

The database handler implements a simple database interface primarily for Internet of Things (IoT) data services. It also supports basic functions needed to deliver dynamic content. Access to and from the database utilizes defined recipes that provide preset queries, input parameter filtering constraints, parameter ordering, JSON conversions, etc.

#### mapURL

The mapURL handler provides URL redirect and rewrite capability for a server. Rules specify what and how to modify requests. Redirects return directly to the client for automatic request. Rewrites pass on to other middleware downstream.

#### static

An obsolete module for serving static content. It offers a few additional features to the express.static middleware such as directory listing support.

#### templates

The templates handler serves dynamic pages built from templates, typically Pug or Markdown. No longer supported, based on using client side built Vue templates.

#### utility

The utility handler implements a number of different site metrics, server diagnostic information, and utility functions such as logging requests. Some operations only work for development mode or a valid user login.

## Supporting Modules

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

#### Cleanup

The cleanup module simply extends the Node process object to enable graceful exit of the code to ensure servers, files, and such properly close.

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

#### CryptoPlus

This module extends the Node crypto module with a number of common functions used by the scripts and middleware modules. Include it by the normal require method. Extended functions occur under the plus object.

#### Extensions2JS

The extensions module adds a number of redumentary properties and methods to the JavaScript language and primitives to simplify code. The module has no exports as all methods and definitions extend primitives. Many of the other modules depend on the Extensions2JS module.

#### Notification

The Notification module supports user notifications via email and SMS. Notification, if loaded as a shared service, passes to each app in the context services variable with the handle notify. It includes functions for each defined notification method, including Site.services.notify.sendMail() and Site.services.notify.sendText(). This enables any app or middleware to send email or text messages over a shared connection without having to implement the respective service connection independently.

#### SafeJSON

The SafeJSON module provides a complex filtering scheme suitable for sanitizing hierarchical JavaScript data objects, including HTML content. In addition to a jsonSafe function It includes lower level functions for escHTML, rexSafe (i.e. regular expression), scalarSafe, and htmlSafe.

#### Scribe

Scribe implements the transcript object, which dumps color coded messages to the console and to a log file. The scribe configuration requires a tag used to identify output messages. When passed a parent Scribe object, it uses that objects streamToLog routine to enable independent Scribe instances to all writing to the same transcript file if desired.

#### WrapSQ3

The WrapSQ3 module provides a wrapper for the SQLite3 bindings with a number of convenience functions. Homebrew assumes the use of SQLite3 databases, but in principal it could use any database that has a wrapper to replicate the methods of WrapSQ3.

## Configuration

Homebrew configuration assumes a JSON file or a JavaScript (.js) file that returns a JSON object. By default, homebrew.js will assume a file located at ../restricted/config, with no filename extension. It first looks 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 default .json or .js file and you do not specify another configuration file on the commandline, 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 unauthorized 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, a JavaScript file offers more flexibility such as dynamic construction based on flag options as well as the use of comments to document configuration. The simplicity of the configuration means you can create a basic JavaScript configuration file with very little knowledge of JavaScript.

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

// JavaScript based configuration file template

var secure = require('./private.js');

var cfg = {

// configuration definition goes here …

// some definitions could use elements defined in secure

};

exports = module.exports = cfg;

Note: A JavaScript file enables the use of a secondary file for defining private information making it easier to sanitize the config.js file when posting to Github or help forums.

See the provided config.js and private\_example.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. The following assumes and describes creation of user js to run the homebrew NodeJS server scripts, although any username may be used.

To create an account for user js.

sudo useradd js

sudo passwd js

Be sure to remove user js from the adm and sudo group to limit admin permissions.

sudo gpasswd -d js sudo

sudo gpasswd -d js adm

Note: You may wish to remove other permissions if defined, such as lpadmin (i.e. line printer admin). Use the groups js command to determine user js group memberships.

Run the scripts from this 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…

lib

api

restricted

configuration file(s)

site database(s)

sites

acme (used for Let’s Encrypt challenges by all sites)

site A root

api (shared javascript api library link)

css

html

images

js (site specific javascript)

index.html

favicon.ico

site B root

…

Tip: The structure and folder names do not matter as long as defined consistent with the configuration file references. Organize each site root tree as desired, but use a separate folder for each web service for security as well as organization.

Tip: For easier maintenance make the api folder a link to a shared library area.

WARNING: For secure operation remember never to place the restricted files under a site root.

Note: JavaScript files do not require executable privileges since you run node, which calls the scripts, but they DO require read and execute privileges for Windows Sharing via Samba.

## 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

Note: A specific version, such a node-8.9.4 will install to respective paths under /usr/local. Classic Linux management creates generic links (i.e. /usr/local/bin/node => /usr/local/bin/node-8.9.4) for easy version management and consistent calls with version changes.

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: bcryptjs, body-parser, colors, compression, cookie-parser, emailjs, express, http-proxy, multer, and sqlite3. The templates module requires async, mustache, pug, and showdown. A Node-red server requires node-red and node-red-admin modules, and dependencies. Some modules may require multiple installs under /home/js/bin and /home/js/.node-red. Future work will require promises.

## Running Homebrew Script

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

node homebrew [<configuration\_file.json>]

NODE\_ENV=production node homebrew [<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, may impact other NodeJS library settings.

The NodeJS 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 running a continuous terminal 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. Running NodeJS this way also captures the STDIO transcripting similar to an interactive shell session.

# 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 (i.e. user js) to operate as a non-privileged service it must use ports of 1024 or greater. Default configuration uses 8080 (proxy), 8081, 8082, … (services). This means you need to forward (default port 80) requests to port 8080 by either of two methods, depending on application.

Tip: By configuration, you can choose any port numbers desired, particularly to avoid conflicts with other servers, just make sure the numbers match with the following instructions.

### Local Network Intranet Server

For a server that runs on the local network providing in-house services, you can configure port forwarding as follows. Run these commands to define the needed routing (as su).

# redirect port 80 to unprivileged port 8080

iptables -t nat -I PREROUTING -p tcp --dport 80 -j REDIRECT --to-ports 8080

# optionally redirect port 443 to unprivileged port 8443

iptables -t nat -I PREROUTING -p tcp --dport 443 -j REDIRECT --to-ports 8443

To include the ability to accept connections from the local loopback (i.e. localhost) add:

# redirect localhost port 80 output to unprivileged port 8080

iptables -t nat -I OUTPUT -p tcp -o lo --dport 80 -j REDIRECT --to-ports 8080

# optionally redirect port 443 to unprivileged port 8443

iptables -t nat -I OUTPUT -p tcp -o lo --dport 443 -j REDIRECT --to-ports 8443

To list rules or flush all rules and start over use, respectively:

iptables -t nat --list

iptables -t nat --flush

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

apt-get install iptables-persistent

The iptables-persistent install will build rules from the existing configuration. Save future configuration changes using:

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

### Internet Server

WARNING: The terms of service for some Internet Service Providers, namely Comcast, prohibit the operation of web servers. Doing so represents a violation that may result in termination of services.

Requests to an Internet server route through your Internet router (i.e. Comcast or CenturyLink modem), and must be rerouted by port forwarding at the router.

For routers that support port forwarding as well as port redirection, it’s simply a matter of forwarding WAN:80 to PROXY:8080, where WAN represents the router external IP address and PROXY represents the IP address of the machine running the proxy.

Note: Instructions for configuring router port forwarding exceeds the scope of this document since every router has its own setup for port forwarding. See your router setup instructions for more information.

If your router does not support port redirection, then forward WAN:80 to PROXY:80 and use the instructions given above in section *Local Network Intranet Server* to redirect port 80 to port 8080.

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://jquery.org](https://jquery.org/), <https://angularjs.org>, <http://www.json.org>, <https://developer.mozilla.org/en-US/docs/Web/JavaScript>, …, 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 book 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 mode. 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. (See *User js* section.) 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

\* Note: the first two lines above represent one single long line with no break.

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.

Note: 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.

#### Contents

<!DOCTYPE html>

<html>

<head>

<title>Kiosk Start Page</title>

<script>

/// ???

</script>

</head>

<body>

<h1>Waiting on server...</h1>

</body>

</html>