Express Tutorial Part 7: Deploying to production

Now you've created (and tested) an awesome LocalLibrary website, you're going to want to install it on a public web server so that it can be accessed by library staff and members over the Internet. This article provides an overview of how you might go about finding a host to deploy your website, and what you need to do in order to get your site ready for production.

Prerequisites:
Complete all previous tutorial topics, including Express Tutorial Part 6: Working with forms.

Objective:
To learn where and how you can deploy an Express app to production.

Overview

Once your site is finished (or finished "enough" to start public testing) you're going to need to host it somewhere more public and accessible than your personal development computer.

Up to now, you've been working in a development environment, using Express/Node as a web server to share your site to the local browser/network, and running your website with (insecure) development settings that expose debugging and other private information. Before you can host a website externally you're first going to have to:

- Choose an environment for hosting the Express app.
- Make a few changes to your project settings.
- Set up a production-level infrastructure for serving your website.

This tutorial provides some guidance on your options for choosing a hosting site, a brief overview of what you need to do in order to get your Express app ready for production, and a worked example of how to install the LocalLibrary website onto the Heroku cloud hosting service.

Bear in mind that you don't have to use Heroku — there are other hosting services available. We've also provided a separate tutorial to show how to Install LocalLibrary on PWS/Cloud Foundry.

What is a production environment?

The production environment is the environment provided by the server computer where you will run your website for external consumption. The environment includes:

- Computer hardware on which the website runs.
- Operating system (e.g. Linux or Windows).
- Programming language runtime and framework libraries on top of which your website is written.

- Web server infrastructure, possibly including a web server, reverse proxy, load balancer, etc.
- Databases on which your website is dependent.

The server computer could be located on your premises and connected to the Internet by a fast link, but it is far more common to use a computer that is hosted "in the cloud". What this actually means is that your code is run on some remote computer (or possibly a "virtual" computer) in your hosting company's data center(s). The remote server will usually offer some guaranteed level of computing resources (e.g. CPU, RAM, storage memory, etc.) and Internet connectivity for a certain price.

This sort of remotely accessible computing/networking hardware is referred to as *Infrastructure as a Service (IaaS)*. Many IaaS vendors provide options to preinstall a particular operating system, onto which you must install the other components of your production environment. Other vendors allow you to select more fully-featured environments, perhaps including a complete Node setup.

> **Note:** Pre-built environments can make setting up your website very easy because they reduce the configuration, but the available options may limit you to an unfamiliar server (or other components) and may be based on an older version of the OS. Often it is better to install components yourself so that you get the ones that you want, and when you need to upgrade parts of the system, you have some idea of where to start!

Other hosting providers support Express as part of a *Platform as a Service* (*PaaS*) offering. When using this sort of hosting you don't need to worry about most of your production environment (servers, load balancers, etc.) as the host platform takes care of those for you. That makes deployment quite easy because you just need to concentrate on your web application and not any other server infrastructure.

Some developers will choose the increased flexibility provided by IaaS over PaaS, while others will appreciate the reduced maintenance overhead and easier scaling of PaaS. When you're getting started, setting up your website on a PaaS system is much easier, so that is what we'll do in this tutorial.

> **Tip:** If you choose a Node/Express-friendly hosting provider they should provide instructions on how to set up an Express website using different configurations of web server, application server, reverse proxy, etc. For example, there are many step-by-step guides for various configurations in the Digital Ocean Node community docs.

Choosing a hosting provider

There are numerous hosting providers that are known to either actively support or work well with *Node* (and *Express*). These vendors provide different types of environments (IaaS, PaaS), and different levels of computing and network resources at different prices.

> **Tip:** There are a lot of hosting solutions, and their services and pricing can change over time. While we introduce a few options below, it is worth performing your own Internet search before selecting a hosting provider.

Some of the things to consider when choosing a host:

- How busy your site is likely to be and the cost of data and computing resources required to meet that demand.
- Level of support for scaling horizontally (adding more machines) and vertically (upgrading to more powerful machines) and the costs of doing so.

- Where the supplier has data centers, and hence where access is likely to be the fastest.
- The host's historical uptime and downtime performance.
- Tools provided for managing the site — are they easy to use and are they secure (e.g. SFTP vs FTP).
- Inbuilt frameworks for monitoring your server.
- Known limitations. Some hosts will deliberately block certain services (e.g. email). Others offer only a certain number of hours of "live time" in some price tiers, or only offer a small amount of storage.
- Additional benefits. Some providers will offer free domain names and support for SSL certificates that you would otherwise have to pay for.
- Whether the "free" tier you're relying on expires over time, and whether the cost of migrating to a more expensive tier means you would have been better off using some other service in the first place!

The good news when you're starting out is that there are quite a few sites that provide computing environments for "free", albeit with some conditions. For example, Heroku provides a free but resource-limited *PaaS* environment "forever", while Amazon Web Services, Microsoft Azure, and the open source option PWS/Cloud Foundry provide free credit when you first join.

Many providers also have a "basic" tier that provides more useful levels of computing power and fewer limitations. Digital Ocean is an example of a popular hosting provider that offers a relatively inexpensive basic computing tier (in the $5 per month lower range at time of writing).

> **Note:** Remember that price is not the only selection criterion. If your website is successful, it may turn out that scalability is the most important consideration.

Getting your website ready to publish

The main things to think about when publishing your website are web security and performance. At the bare minimum, you will want to remove the stack traces that are included on error pages during development, tidy up your logging, and set the appropriate headers to avoid many common security threats.

In the following subsections, we outline the most important changes that you should make to your app.

> **Tip:** There are other useful tips in the Express docs — see Production best practices: performance and reliability and Production Best Practices: Security.

Set NODE_ENV to 'production'

We can remove stack traces in error pages by setting the `NODE_ENV` environment variable to *production* (it is set to '*development*' by default). In addition to generating less-verbose error messages, setting the variable to *production* caches view templates and CSS files generated from CSS extensions. Tests indicate that setting `NODE_ENV` to *production* can improve app performance by a factor of three!

This change can be made either by using `export`, an environment file, or the OS initialization system.

> **Note:** This is actually a change you make in your environment setup rather than your app, but important enough to note here! We'll show how this is set for our hosting example below.

Log appropriately

Logging calls can have an impact on a high-traffic website. In a production environment, you may need to log website activity (e.g. tracking traffic or logging API calls) but you should attempt to minimize the amount of logging added for debugging purposes.

One way to minimize "debug" logging in production is to use a module like debug that allows you to control what logging is performed by setting an environment variable. For example, the code fragment below shows how you might set up "author" logging. The debug variable is declared with the name 'author', and the prefix "author" will be automatically displayed for all logs from this object.

```
var debug = require('debug')('author');

// Display Author update form on GET
exports.author_update_get = function(req, res, next) {

    req.sanitize('id').escape().trim();
    Author.findById(req.params.id, function(err, author
        if (err) {
            debug('update error:' + err);
            return next(err);
        }
        //On success
        res.render('author_form', { title: 'Update Auth
    });

};
```

You can then enable a particular set of logs by specifying them as a comma-separated list in the `DEBUG` environment variable. You can set the variables for displaying author and book logs as shown (wildcards are also supported).

```
#Windows
set DEBUG=author,book

#Linux
export DEBUG="author,book"
```

Challenge: Calls to `debug` can replace logging you might previously have done using `console.log()` or `console.error()`. Replace any `console.log()` calls in your code with logging via the debug module. Turn the logging on and off in your development environment by setting the DEBUG variable and observe the impact this has on logging.

If you need to log website activity you can use a logging library like *Winston* or *Bunyan*. For more information on this topic see: Production best practices: performance and reliability.

Use gzip/deflate compression for responses

Web servers can often compress the HTTP response sent back to a client, significantly reducing the time required for the client to get and load the page. The compression method used will depend on the decompression methods the client says it supports in the request (the response will be sent uncompressed if no compression methods are supported).

Add this to your site using compression middleware. Install this at the root of your project by running the following command:

```
npm install compression
```

Open **./app.js** and require the compression library as shown. Add the compression library to the middleware chain with the `use()` method (this should appear before any routes you want compressed — in this case, all of them!)

```
var catalogRouter = require('./routes/catalog'); //Impo
var compression = require('compression');

// Create the Express application object
var app = express();

...

app.use(compression()); //Compress all routes

app.use(express.static(path.join(__dirname, 'public')))

app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/catalog', catalogRouter);  // Add catalog rou

...
```

Note: For a high-traffic website in production you wouldn't use this middleware. Instead, you would use a reverse proxy like *Nginx*.

Use Helmet to protect against well known vulnerabilities

Helmet is a middleware package. It can set appropriate HTTP headers that help protect your app from well-known web vulnerabilities (see the docs for more information on what headers it sets and vulnerabilities it protects against).

Install this at the root of your project by running the following command:

```
npm install helmet
```

Open **./app.js** and require the *helmet* library as shown. Then add the module to the middleware chain with the `use()` method.

```
var compression = require('compression');
var helmet = require('helmet');

// Create the Express application object
var app = express();

app.use(helmet());
...
```

Note: The command above adds a *subset* of the available headers. They make sense for most sites. You can add/disable specific headers as needed by following the instructions on npm.

Example: Installing LocalLibrary on Heroku

This section provides a practical demonstration of how to install *LocalLibrary* on the Heroku PaaS cloud.

Why Heroku?

Heroku is one of the longest-running and popular cloud-based PaaS services. It originally supported only Ruby apps, but now can be used to host apps from many programming environments, including Node (and hence Express)!

We are choosing to use Heroku for several reasons:

- Heroku has a free tier that is *really* free (albeit with some limitations).
- As a PaaS, Heroku takes care of a lot of the web infrastructure for us. This makes it much easier to get started because you don't worry about servers, load balancers, reverse proxies, restarting your website on a crash, or any of the other web infrastructure that Heroku provides.
- While it does have limitations, they will not affect this particular application. For example:
  - Heroku's free-tier only provides short-lived storage. User-uploaded files are not safely stored on Heroku itself.
  - The free tier will sleep an inactive web app if there are no requests within a half-hour period. The site may take several seconds to respond if the dyno is asleep.
  - The free tier limits your site to a certain amount of hours of runtime each month (time "asleep" is not used in the calculation). This is fine for a low use or demonstration site. It's not suitable if 100% uptime is required.
  - Other limitations are listed in Limits (Heroku docs).
- If it works and you end up loving it, you'll want to upgrade. Scaling your app on Heroku is very easy.

While Heroku is perfect for hosting this demonstration it may not be perfect for your real website. Heroku makes things easy to set up and scale. If you need more speed or uptime or add-on features, expect to pay for them.

How does Heroku work?

Heroku runs websites within one or more "Dynos". These are isolated, virtualized Unix containers that provide the environment required to run an application. The dynos are completely isolated and have an *ephemeral* file system (a short-lived file system that is cleaned and emptied each time the dyno restarts). The one thing dynos share by default are the application configuration variables. Internally, Heroku uses a load balancer to distribute web traffic to all "web" dynos. Since nothing is shared between them, Heroku can scale an app horizontally by adding more dynos. You may also need to scale your database to accept additional connections.

Because the file system is ephemeral you can't directly install services required by your application. Databases, queues, caching systems, storage, email services, etc. are considered "add-ons." Heroku web applications use backing services provided by Heroku or 3rd parties. Once attached to your web application, the add-on services are accessed in your web application via environment variables. For each additional service, charges may apply.

In order to execute your application Heroku needs to be configured to set up the appropriate environment for your application's dependencies and be told how to start. For Node apps, all the information it needs is obtained from your **package.json** file.

Developers interact with Heroku using a special client app/terminal, which is much like a Unix bash script. This allows you to upload code stored in a git repository, inspect the running processes, see logs, set configuration variables, and much more.

Let's get our application on Heroku. First we'll initialize a git repository for our Express web application. Next, we'll make some minor changes to the package.json. Once we've done that we'll set up a Heroku account, install the Heroku client on our local machine and use it to upload our application.

That's all the overview you need in order to get started (see Getting Started on Heroku with Node.js for a more comprehensive guide).

Creating an application repository in Github

Heroku is integrated with **git,** the source code version control system. The Heroku client you install will use git to synchronize changes you upload. The Heroku client creates a new "remote" repository named *heroku.* It connects to a repository of your code on the Heroku cloud. During development, you use git to store changes on your "master" repository. When you want to deploy your site, you sync your changes to the Heroku repository.

> **Note:** If you're accustomed to following good software development practices you may already be using git or some other SCM system. If you already have a git repository, skip this step.

There are a lot of ways of to work with git. One easy workflow is to first set up an account on GitHub, create a new repository there and then clone it to your local machine:

1. Visit https://github.com/ and create an account.
2. Once you are logged in, click the **+** link in the top toolbar and select **New repository**.
3. Fill in all the fields on this form. While these are not compulsory, they are strongly recommended.
    - Enter a new repository name (e.g. *express-locallibrary-tutorial*), and description (e.g. "Local Library website written in Express (Node)".
    - Choose **Node** in the *Add .gitignore* selection list.
    - Choose your preferred license in the *Add license* selection list.
    - Check **Initialize this repository with a README**.
4. Press **Create repository**.

5. Click the green "**Clone or download**" button on your new repo page.
6. Copy the URL value from the text field inside the dialog box that appears (it should be something like: **https://github.com/*<your_git_user_id>*/express-locallibrary-tutorial.git**).

Now that the repository ("repo") is created we are going to want to clone it on our local computer:

1. Install *git* for your local computer (you can find versions for different platforms here).
2. Open a command prompt/terminal and clone your repository using the URL you copied above:

   ```
   git clone https://github.com/<your_git_user_id>/expr
   ```

   This will create the repository below the current point.
3. Navigate into the new repo.

   ```
   cd express-locallibrary-tutorial
   ```

The final step is to copy in your application and then add the files to your repo using git:

1. Copy your Express application into this folder (excluding **/node_modules**, which contains dependency files that you should fetch from NPM as needed).
2. Open a command prompt/terminal and use the `add` command to add all files to git.

3.
   ```
   git add -A
   ```

4. Use the status command to check all files that you are about to add are correct (you want to include source files, not binaries, temporary files etc.). It should look a bit like the listing below.

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   ...
```

5. When you're satisfied, commit the files to your local repository:

```
git commit -m "First version of application mov
```

6. Then synchronize your local repository to the Github website, using the following:

```
git push origin master
```

When this operation completes, you should be able to go back to the page on Github where you created your repo, refresh the page, and see that your whole application has now been uploaded. You can continue to update your repository as files change using this add/commit/push cycle.

> **Tip:** This is a good point to make a backup of your "vanilla" project — while some of the changes we're going to be making in the following sections might be useful for deployment on any platform (or development) others might not.
>
> The *best* way to do this is to use *git* to manage your revisions. With *git* you can not only go back to a particular past version, but you can maintain this

> in a separate "branch" from your production changes and cherry-pick any changes to move between production and development branches. Learning Git is well worth the effort, but is beyond the scope of this topic.
>
> The *easiest* way to do this is to just copy your files into another location. Use whichever approach best matches your knowledge of git!

Update the app for Heroku

This section explains the changes you'll need to make to our *LocalLibrary* application to get it to work on Heroku.

Set node version

The **package.json** contains everything needed to work out your application dependencies and what file should be launched to start your site. Heroku detects the presence of this file, and will use it to provision your app environment.

The only useful information missing in our current **package.json** is the version of node. We can find the version of node we're using for development by entering the command:

```
>node --version
v10.15.1
```

Open **package.json**, and add this information as an **engines > node** section as shown (using the version number for your system).

```
{
  "name": "express-locallibrary-tutorial",
  "version": "0.0.0",
  "engines": {
    "node": "10.15.1"
  },
  "private": true,
  ...
```

Database configuration

So far in this tutorial, we've used a single database that is hard-coded into **app.js**. Normally we'd like to be able to have a different database for production and development, so next we'll modify the LocalLibrary website to get the database URI from the OS environment (if it has been defined), and otherwise use our development database.

Open **app.js** and find the line that sets the MongoDB connection variable. It will look something like this:

```
var mongoDB = 'mongodb+srv://your_user:your_passwo
```

Replace the line with the following code that uses `process.env.MONGODB_URI` to get the connection string from an environment variable named `MONGODB_URI` if has been set (use your own database URL instead of the placeholder below.)

```
// Set up mongoose connection
var dev_db_url = 'mongodb+srv://cooluser:coolpassword@c
var mongoDB = process.env.MONGODB_URI || dev_db_url;
```

Get dependencies and re-test

Before we proceed, let's test the site again and make sure it wasn't affected by any of our changes.

First, we will need to fetch our dependencies (you will recall we didn't copy the **node_modules** folder into our git tree). You can do this by running the following command in your terminal at the root of the project:

```
npm install
```

Now run the site (see Testing the routes for the relevant commands) and check that the site still behaves as you expect.

Save changes to Github

Next, let's save all our changes to Github. In the terminal (whilst inside our repository), enter the following commands:

```
git add -A
git commit -m "Added files and changes required fo
git push origin master
```

We should now be ready to start deploying *LocalLibrary* on Heroku.

Get a Heroku account

To start using Heroku you will first need to create an account (skip ahead to Create and upload the website if you've already got an account and installed the Heroku client):

- Go to www.heroku.com and click the **SIGN UP FOR FREE** button.
- Enter your details and then press **CREATE FREE ACCOUNT**. You'll be asked to check your account for a sign-up email.
- Click the account activation link in the signup email. You'll be taken back to your account on the web browser.

- Enter your password and click **SET PASSWORD AND LOGIN**.
- You'll then be logged in and taken to the Heroku dashboard: https://dashboard.heroku.com/apps.

Install the client

Download and install the Heroku client by following the instructions on Heroku here.

After the client is installed you will be able to run commands. For example to get help on the client:

```
heroku help
```

Create and upload the website

To create the app we run the "create" command in the root directory of our repository. This creates a git remote ("pointer to a remote repository") named *heroku* in our local git environment.

```
heroku create
```

> **Note:** You can name the remote if you like by specifying a value after "create". If you don't then you'll get a random name. The name is used in the default URL.

We can then push our app to the Heroku repository as shown below. This will upload the app, get all its dependencies, package it in a dyno, and start the site.

```
git push heroku master
```

If we're lucky, the app is now "running" on the site. To open your browser and run the new website, use the command:

```
heroku open
```

Note: This may result in the Heroku page error page. This is will be remedy in the next section.

Note: The site will be running using our development database. Create some books and other objects, and check out whether the site is behaving as you expect. In the next section, we'll set it to use our new database.

Setting configuration variables

You will recall from a preceding section that we need to set NODE_ENV to 'production' in order to improve our performance and generate less-verbose error messages. We do this by entering the following command:

```
>heroku config:set NODE_ENV='production'
Setting NODE_ENV and restarting limitless-tor-1892
NODE_ENV: production
```

We should also use a separate database for production, setting its URI in the **MONGODB_URI** environment variable. You can set up a new database and database-user exactly as we did originally, and get its URI. You can set the URI as shown (obviously, using your own URI!)

```
>heroku config:set MONGODB_URI='mongodb+srv://cooluser:
Setting MONGODB_URI and restarting limitless-tor-18923.
MONGODB_URI: mongodb+srv://cooluser:coolpassword@cluste
```

You can inspect your configuration variables at any time using the `heroku config` command — try this now:

```
>heroku config
=== limitless-tor-18923 Config Vars
MONGODB_URI: mongodb+srv://cooluser:coolpassword@c
NODE_ENV:    production
```

Heroku will restart your app when it updates the variables. If you check the home page now it should show zero values for your object counts, as the changes above mean that we're now using a new (empty) database.

> **Note:** If you got the Heroku error page in the last section. Now try rerunning the last section.

Managing addons

Heroku uses independent add-ons to provide backing services to apps — for example, email or database services. We don't use any addons in this website, but they are an important part of working with Heroku, so you may want to check out the topic Managing Add-ons (Heroku docs).

Debugging

The Heroku client provides a few tools for debugging:

```
heroku logs  # Show current logs
heroku logs --tail # Show current logs and keep up
heroku ps   #Display dyno status
```

Summary

That's the end of this tutorial on setting up Express apps in production, and also the series of tutorials on working with Express. We hope you've found them useful. You can check out a fully worked-through version of the source code on Github here.

See also

- Production best practices: performance and reliability (Express docs)
- Production Best Practices: Security (Express docs)
- Heroku
  - Getting Started on Heroku with Node.js (Heroku docs)
  - Deploying Node.js Applications on Heroku (Heroku docs)
  - Heroku Node.js Support (Heroku docs)
  - Optimizing Node.js Application Concurrency (Heroku docs)
  - How Heroku works (Heroku docs)
  - Dynos and the Dyno Manager (Heroku docs)
  - Configuration and Config Vars (Heroku docs)
  - Limits (Heroku docs)
- Digital Ocean
  - Express tutorials
  - Node.js tutorials

Previous

Overview: Express Nodejs

In this module

Last modified: Apr 28, 2020, by MDN contributors

- ▶ CSS first steps
- ▶ CSS building blocks
- ▶ Styling text
- ▶ CSS layout

JavaScript — Dynamic client-side scripting

- ▶ JavaScript first steps
- ▶ JavaScript building blocks
- ▶ Introducing JavaScript objects
- ▶ Asynchronous JavaScript
- ▶ Client-side web APIs

Web forms — Working with user data

- ▶ Core forms learning pathway
- ▶ Advanced forms articles

Accessibility — Make the web usable by everyone

- ▶ Accessibility guides
- ▶ Accessibility assessment

Tools and testing

- ▶ Cross browser testing
- ▶ Git and GitHub
- ▶ Understanding client-side web development tools

Server-side website programming

▶ First steps

▶ Django web framework (Python)

▼ Express Web Framework (node.js/JavaScript)

Express Web Framework (Node.js/JavaScript) overview

Express/Node introduction

Setting up a Node (Express) development environment

Express tutorial: The Local Library website

Express Tutorial Part 2: Creating a skeleton website

Express Tutorial Part 3: Using a database (with Mongoose)

Express Tutorial Part 4: Routes and controllers

Express Tutorial Part 5: Displaying library data

Express Tutorial Part 6: Working with forms

Express Tutorial Part 7: Deploying to production

Further resources

▶ Common questions

How to contribute

Sign up now