Back to Scaling Node.js Applications guides

How to Build a Node.js Application with Express and Pug

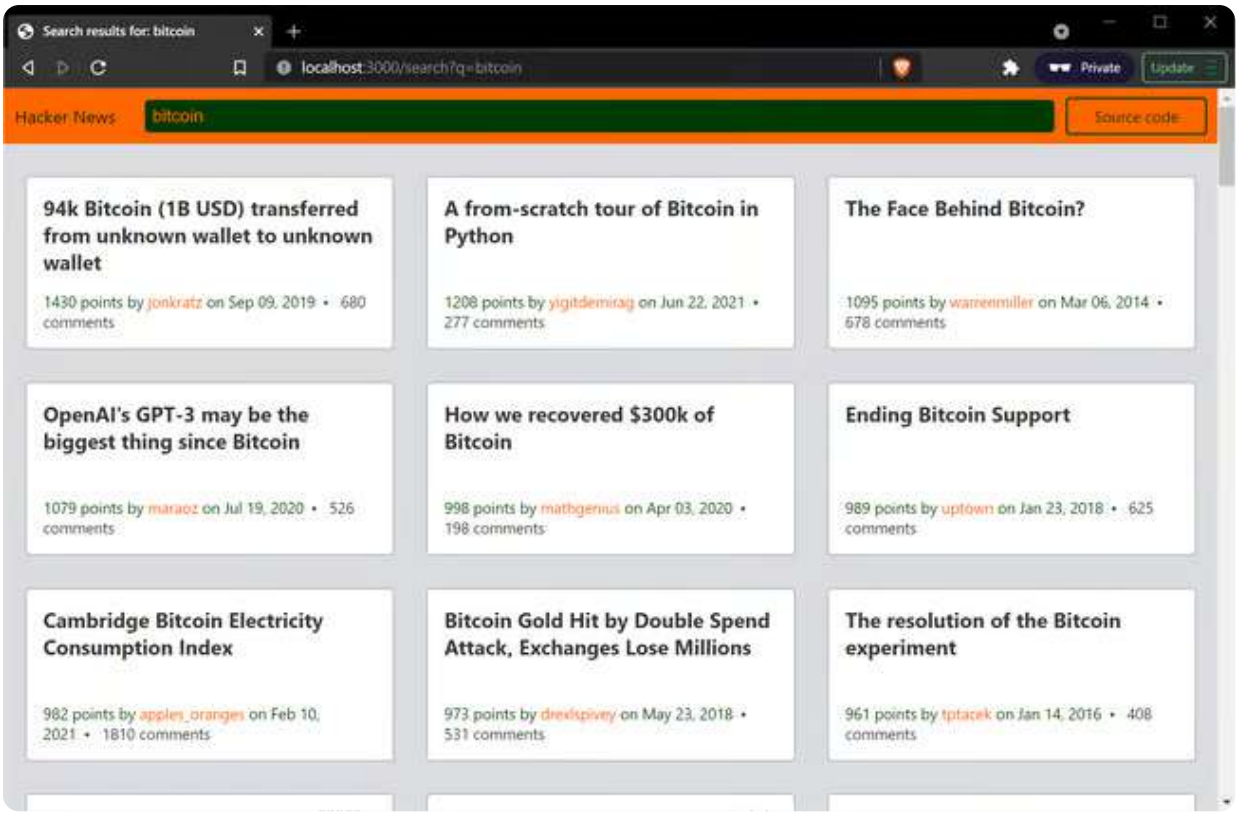
Node.js Express Pug

Ayooluwa Isaiah

UPDATED ON OCTOBER 13, 2023

Express is the most popular web application framework for Node.js. It is easy to use, offers decent performance, and provides access to many of the necessary tools you need to build and deploy a robust web service, either built-in to the framework itself or as additional modules.

This tutorial will walk you through a practical example of building and deploying a Node.js-based web application through Express and Pug, a popular templating engine for Node.js which is often used alongside Express. We'll develop an app that searches Hacker News and presents the results in an HTML document.



Prerequisites

Before you proceed with the remainder of this tutorial, ensure that you have met the following requirements:

- A basic understanding of the JavaScript programming language and the command line.
- A recent version of Node.js and npm installed on your computer or server.

Step 1 — Setting up the Project

You'll set up the project directory and install the necessary dependencies in this step. Open a terminal window, create a new directory for the application, and change into it as shown below:

```
$ mkdir hacker-news
```

```
$ cd hacker-news
```

Next, initialize the project with a `package.json` file using the command below. You can omit the `-y` flag to answer the initialization questionnaire.

```
$ npm init -y
```

At this point, you can install the express package through the following command:

```
$ npm install express
```

You are now all set to start using Express for your Node.js application.

Step 2 — Creating an Express server

This section will take you through the process of setting up a Node.js web server with Express. Open the terminal, and ensure that you're in your project root, then create and open a `server.js` file in your favorite text editor:

```
$ nano server.js
```

Paste the following code into the file and save it:

server.js

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

const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Node.js server');
});

const server = app.listen(process.env.PORT || 3000, () => {
  console.log(`Hacker news server started on port: ${server.address().port}`);
});
```

The code snippet above requires the `express` package that was installed in the previous step. Its exported top-level `express()` function to set up an Express application which is referenced through the `app` variable.

The next portion of the code (starting with `app.get`) represents an Express route definition. Whenever an HTTP GET request is made to the site root ('\'), the provided callback function will be executed with `req` representing the request and `res` designating the response. In this snippet, the `send()` method is called to return the text 'Hello from Node.js server' as the response.

In the concluding block of code, the `listen()` method is used to start the server and listen for incoming connections on the given port. This port can be set through an environmental variable and accessed through the `process.env` object. If a `PORT` property is not found on this object, it defaults to `3000`.

The `listen()` method also accepts a callback function which is executed once this server starts listening for connections. We're using it here to write

some text to the standard output to determine if the server was launched successfully.

Head back to your terminal, and run the command below from the project root to start the server:

```
$ node server.js
```

You should see the following output, indicating that the server was started successfully:

Output

```
Hacker news server started on port: 3000
```

You can also test if the server is fully operational by making a GET request to the site root in a separate terminal through `curl` as shown below. It should yield the string `Hello from Node.js server` as expected.

```
$ curl http://localhost:3000
```

Now that our server is up and running, let's look at how we can automatically restart the server during development so that updates are effected immediately after changes to the source code.

Step 3 — Auto restarting the Node.js server in development

In Node.js, changes to the source code are not reflected in an already existing process until it is killed and restarted. Doing this manually can be

tedious so we'll utilize the PM2 process manager to address this concern. You can install its NPM package to your project through the command below:

```
$ npm install pm2 --save-dev
```

Afterward, open your `package.json` file in your text editor:

```
$ nano package.json
```

Replace the `scripts` property in the file with the snippet shown below:

package.json

```
"scripts": {
  "dev": "npx pm2-dev server.js",
  "start": "npx pm2 start server.js"
}
```

PM2 provides a `pm2-dev` binary for development, and it auto restarts the server when file changes are detected in the project directory. In contrast, the `pm2` binary is meant to be used when deploying the application to production.

After saving the file, kill your existing Node.js application process by pressing `Ctrl-C` in the terminal, then run the command below to start it again through PM2.

```
$ npm run dev
```

You should observe the following output if everything goes well:

Output

```
> hacker-news@1.0.0 dev
> npx pm2-dev server.js

================================================================================
--- PM2 development mode ------------------------------------------------------
Apps started         : server
Processes started    : 1
Watch and Restart    : Enabled
Ignored folder       : node_modules
================================================================================
server-0  | Hacker news server started on port: 3000
```

At this point, your web server will be restarted by PM2 each time it detects a new file or a change to an existing file in the project directory. In the next section, you'll install the Pug template engine and use it to construct your application's HTML templates.

Step 4 — Setting up Node.js templating with Pug

A templating engine is used to produce HTML documents from a data model. It typically processes predefined templates and input data to create a portion of a web page or a complete document. There are several Node.js templating engines that are compatible with Express, but we've opted for Pug here due to its status as the recommended default.

Open a separate terminal instance, and run the command below to install the `pug` package from NPM:

```
$ npm install pug
```

Afterward, open the `server.js` file in your editor, set `pug` as the `view engine`, and save the file. You don't need to require the `pug` package in your code after setting the `view engine` as Express already does so internally.

server.js

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

const app = express();

app.set('view engine', 'pug');
. . .
```

Create a new `views` directory within the project root for organizing Pug template files. This is the default directory where Express expects your template files to be located.

```
$ mkdir views
```

Create a new `default.pug` file within the `views` directory and open it in your text editor:

```
$ nano views/default.pug
```

Paste the following code into the file:

views/default.pug

```
doctype html
html
    head
      meta(charset='UTF-8')
      meta(name='viewport' content='width=device-width, initial-scale=1.0')
      meta(http-equiv='X-UA-Compatible' content='ie=edge')
```

```
    title= title
    link(rel='stylesheet' type="text/css" href='/css/style.css')
  body
    block content
```

Notice how Pug relies on indentation to describe the structure of a template, and that there are no closing tags. Attributes are placed within parentheses, and the `block` keyword is used to define a section in the template that may be replaced by a derived child template. This template also expects a `title` variable to be passed to it on render.

The purpose of this `default.pug` template is to provide the boilerplate for all other templates. Each subsequent template will extend the `default` template and replace the contents of the `content` block as appropriate. Let's demonstrate this concept by creating a template for the homepage of our application.

Create a new `home.pug` file in the `views` directory:

```
$ nano views/home.pug
```

Enter the following code into the file:

views/home.pug

```
extends default

block content
  .home-search
    img(src="/images/hn.png" width="180" height="180")
    form(action='/search' method='GET')
      input.home-input(autofocus='' placeholder='Search Hacker News' type='search'
```

The `extends` keyword is used to inherit from the specified template (`default.pug` in this case), and the `content` block overrides the parent

`content` block. So, essentially, the contents of this file are rendered inside the `body` of the `default.pug` template.

Return to your `server.js` file, and change the handler for the site root route as follows:

server.js

```
app.get('/', (req, res) => {
  res.render('home', {
    title: 'Search Hacker News',
  });
});
```

This calls the `render()` method on the response to render the view described by the `home.pug` template. We don't need to specify the extension in the template name since our `view engine` has been set to `pug`. The second argument to `render()` is an object that defines local variables for the template. In this case, only the page's `title` is defined as a variable (see `default.pug`).

After saving the file, you can open your browser and type `http://localhost:3000` or `http://<your_server_ip>:3000` to view your website. It should present an unstyled webpage with a search bar and a broken image:

In the next section, we'll configure how we can serve static files through Express so that the linked image and CSS file will start working correctly.

Step 5 — Serving static files in Express

In the `default.pug` template, we referenced a `style.css` file but this file does not exist yet. The same thing can be observed in `home.pug` where we also linked to non-existent `hn.png` file. Let's create these files inside `css` and `images` directories within a `public` directory at the project root:

```
$ mkdir -p public/css public/images
```

We now have a `public` directory to store static assets such as images, CSS, and client-side JavaScript files. In this directory, a `css` and `images` directory has been created for organizing stylesheets and image files respectively.

Go ahead and create a new `style.css` file in the `public/css` directory, and open it in your text editor:

```
$ nano public/css/style.css
```

Paste the code in this GitHub gist into the file, and save it. It consists of the styles for the entire application.

Afterward, return to the terminal and use the `wget` command to download the hn.png file to your `public/images` directory as shown below:

```
$ wget https://i.imgur.com/qUbNHtf.png -O public/images/hn.png
```

Our static files are now in the appropriate location, but we still need to use the built-in `express.static` middleware function to specify how they should be handled. Open your `server.js` file, and add the following highlighted lines:

server.js

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

const app = express();

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

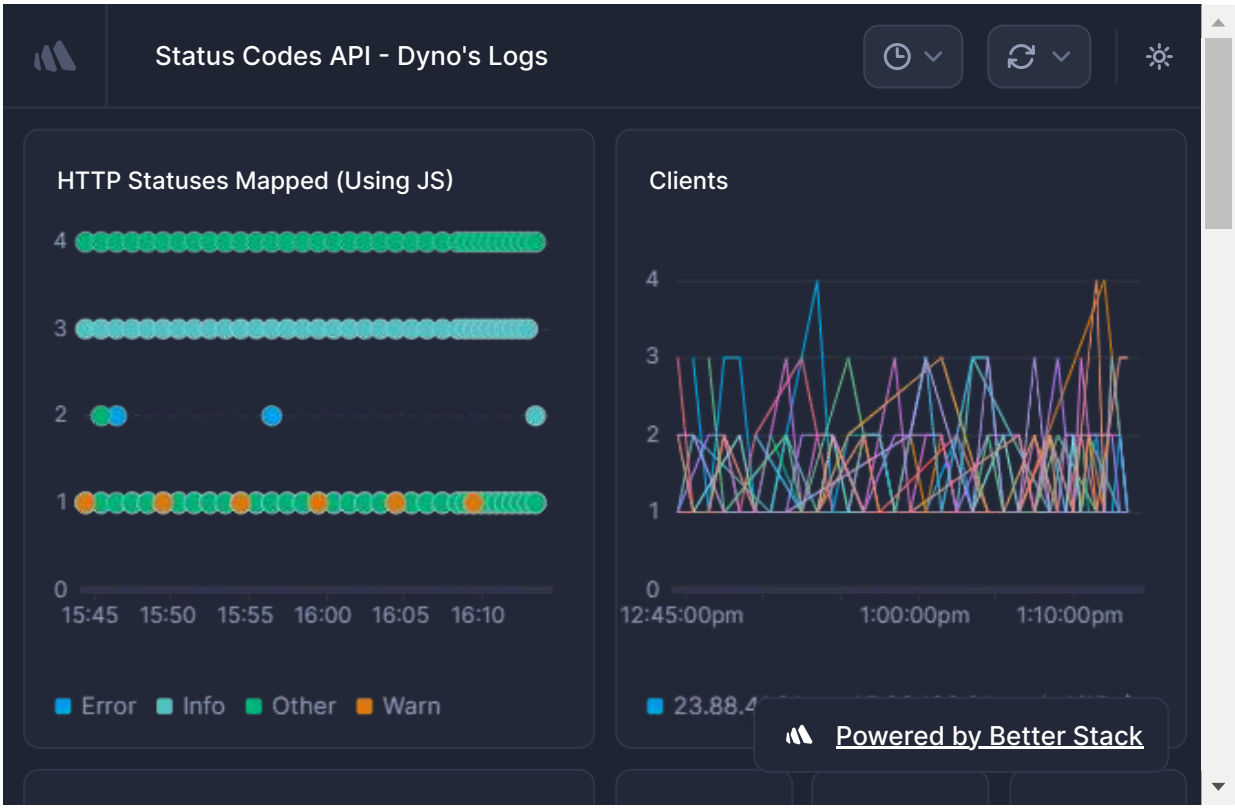
The snippet above instructs Express to look for static files in the `public` directory. Any requests for a file will be served relative to this directory. You can now access the files in the `public` directory in the following manner:

```
http://localhost:3000/images/hn.png
```

```
http://localhost:3000/css/style.css
```

If you reload your application in the browser, the styles should kick in, and the image should load:

We can now move on to create the main functionality of our application, which is searching Hacker News and presenting the results in the browser.



Step 6 — Creating the `/search` route

The form on the application homepage makes a GET request to the `/search` route when it is submitted, with the `q` parameter set to the value of the search query. In this step, we will create this route on the server through `app.get()` to handle form submissions.

Open your `server.js` file, and enter the following code below the handler for the root route:

server.js

```
app.get('/search', (req, res) => {
  const searchQuery = req.query.q;
  if (!searchQuery) {
    res.redirect(302, '/');
    return;
  }

  console.log(searchQuery);
  res.status(200).end();
});
```

The `req.query` object contains a property for each query parameter in the request. For example, the `q` parameter value in a GET request to `/search?q=bitcoin` will be accessible under `req.query.q` and stored in the `searchQuery` variable. If `searchQuery` is "falsy" (such as when it is set to an empty string), the application redirects back to the homepage. Otherwise, the search query is logged to the standard output and a 200 OK response is sent back to the client.

Let's modify the `/search` route so that the search term is used to search for stories on Hacker News through the HN Algolia API. Before we can make

requests to the API, we need to install the axios package first through `npm`:

```
$ npm install axios
```

Afterwards, require the `axios` package in your `server.js` file:

server.js

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

Then create the function below in your `server.js` file just above the `/search` route:

server.js

```
async function searchHN(query) {
  const response = await axios.get(
    `https://hn.algolia.com/api/v1/search?query=${query}&tags=story&hitsPerPage=90`
  );

  return response.data;
}
```

This function accepts a query string which is subsequently used to query the Algolia API which requires no authentication. Afterward, the response is returned to the caller. You can utilize the `searchHN()` function in your `/search` route by making the following modifications:

server.js

```
app.get('/search', async (req, res) => {
  const searchQuery = req.query.q;
  if (!searchQuery) {
    res.redirect(302, '/');
    return;
```

```
  }

  const results = await searchHN(searchQuery);
  res.status(200).json(results);
});
```

We've prefixed the callback function with the `async` keyword so that we may `await` the results from the `searchHN()` method before responding to the client request. Instead of logging the search result to the standard output, it is sent to the browser through the `json()` helper method, which sends the response with the `Content-Type` set to `application/json`, and whose argument is converted to a JSON string through `JSON.stringify()`.

You can test the search route by submitting a query through the form on your application's homepage. You should get the JSON response from Algolia in your browser as in the screenshot below:

Now that we're able to search Hacker News through Algolia's HN API, let's go ahead and prepare the view that will render the results instead of echoing the raw JSON response back to the browser.

Step 7 — Rendering the search results

Let's create a template for rendering the search results in the `views` folder. You can call the file `search.pug`.

```
$ nano views/search.pug
```

Enter the following code into the file:

views/search.pug

```
extends default

block content
  header
    a.logo(href='/') Hacker News
    form(action='/search' method='GET')
      input.search-input(autofocus='' value=`${searchQuery || ''}` placeholder='Ent
    a.button.github-button(href='<https://github.com/finallyayo/hacker-news>') Sour

  .container
    if !searchResults || !searchResults.nbHits
      .result-count
        p No results found for your query:
        |
        strong #{searchQuery}
    else
      ul.search-results
        each story in searchResults.hits
          li.news-article
            a.title-link(target='_blank' rel='noreferrer noopener' href=`${story.ur
              h3.title #{story.title}
            .metadata #{story.points} points by
              |
              |
              span.author #{story.author}
              |
              | on
              |
              time.created-date #{story.created_at}
              |
              |
              a.comments(target='_blank' rel='noreferrer noopener' href=`https://ne
```

This template extends from `default.pug`, and provides its own `content` block, which consists of a search form and a section where the search results are conditionally rendered. The `searchResults` object is the JSON object received from the Algolia API after a successful query, while `searchQuery` represents the user's original query.

The each keyword in Pug provides the ability to iterate over an array so that we can output some HTML for each item in the array. For example, in the code snippet above, each object in the `searchResults.hits` array is aliased into the `story` identifier, and its properties (such as `title`, `author`, or `points`) become accessible through the standard dot notation method.

Go ahead and utilize the `search.pug` template in your `/search` route as shown below:

server.js

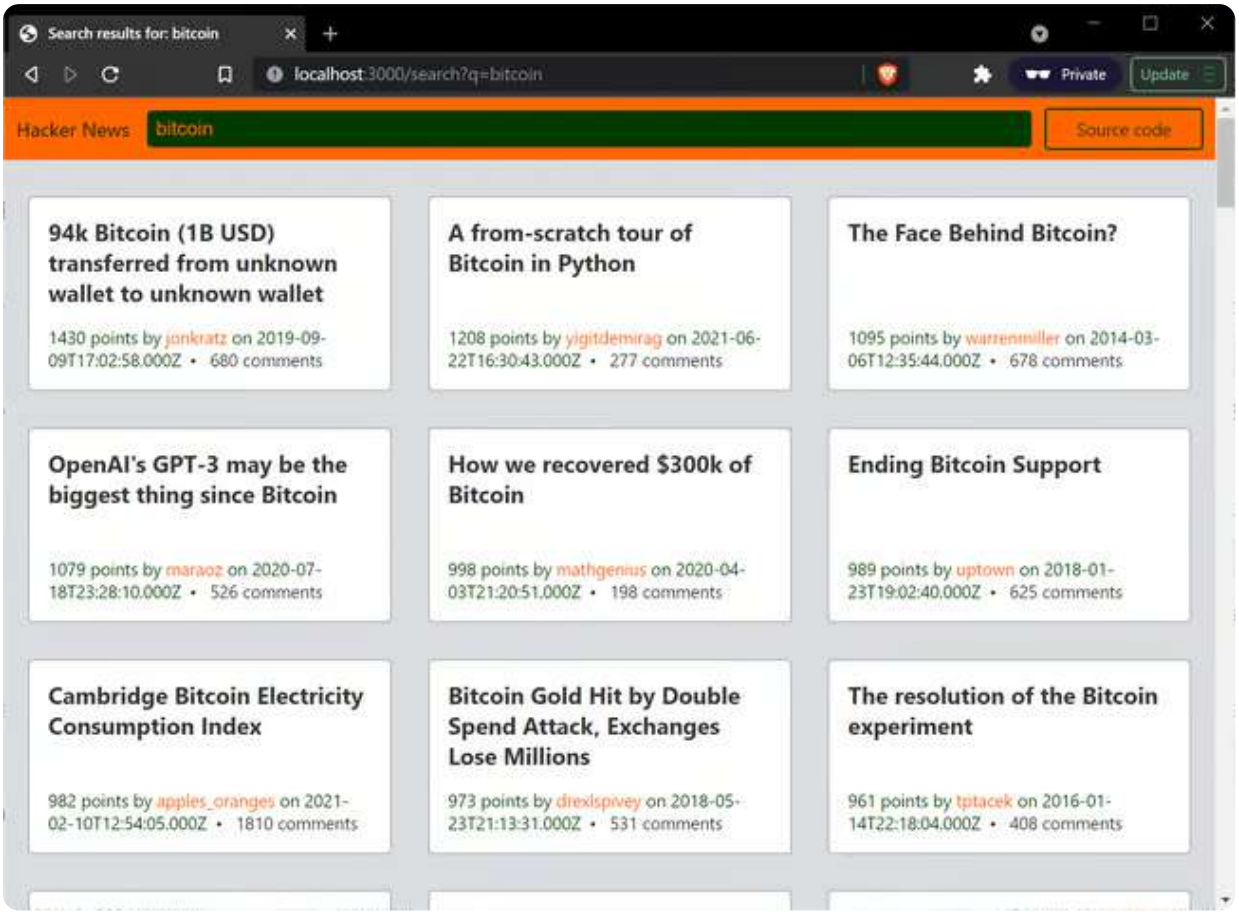
```
app.get('/search', async (req, res) => {
  const searchQuery = req.query.q;
  if (!searchQuery) {
    res.redirect(302, '/');
    return;
  }

  const results = await searchHN(searchQuery);
  res.render('search', {
    title: `Search results for: ${searchQuery}`,
    searchResults: results,
    searchQuery,
  });
});
```

Instead of echoing the JSON object returned by `searchHN()` to the client, we're utilizing the `search.pug` template to get an HTML representation of the result. This is achieved by passing the `results` object as `searchResults` in the second argument to `render()` since its required in the `search.pug` template.

At this point, you will see an adequately constructed search results page when you make a search query at your application's homepage. A search

input is also provided on the results page so that you don't have to return to the home page to modify your query.



Notice how the date on each story is currently formatted. This is how the API returns the published date for each item, but it's not in an appropriate format for consumption on a search results page. In the next section, we'll turn it into a more readable format through the date-fns library.

Step 8 — Formatting the date on each story

Return to your terminal and install the `date-fns` package through `npm`:

```
$ npm install date-fns
```

Once the installation completes, add the following line to your `server.js` file just above the handler for the site root:

server.js

```
app.locals.dateFns = require('date-fns');
```

The `app.locals` object contains properties that are local variables in the application, and they persist for the entire lifetime of the application. This is useful for providing helper methods to templates or other data.

In the `search.pug` file, you can utilize the `dateFns` object described below. Go ahead and replace the following line:

views/search.pug

```
time.created-date #{story.created_at}
```

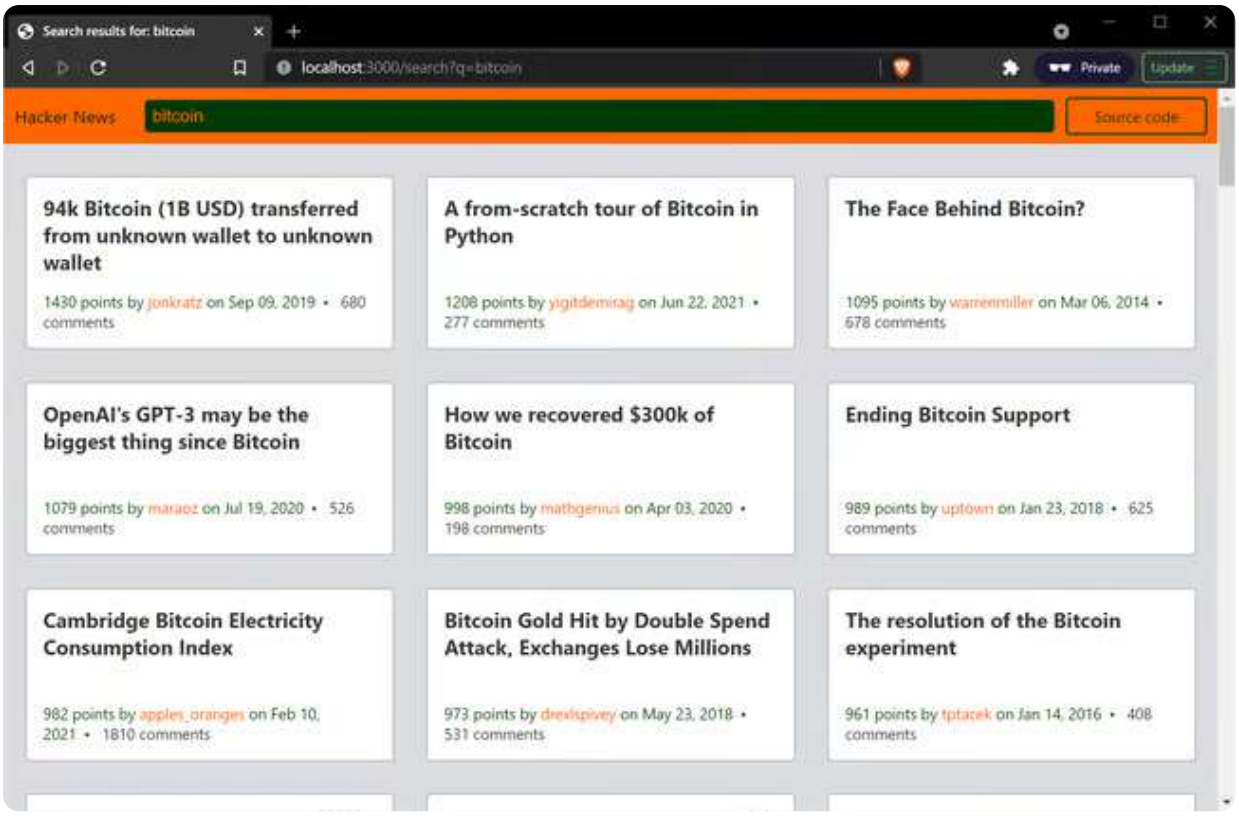
with the snippet below (while maintaining the indentation):

views/search.pug

```
time.created-date #{dateFns.format(new Date(`${story.created_at}`), 'LLL dd, yyyy')
```

The format method is used to return a date string in the specified format (which is Jan 02, 2006) in the above snippet. See the accepted patterns for more details.

Once you've saved the file, you can repeat the search query from the previous step. The published dates should now have more readable formatting.



We've almost completed the app, but let's make sure we're dealing with potential errors correctly.

Step 9 — Handling errors in Express

Errors can happen anytime, so we need to account for them in any application we build. Express simplifies error handling in routes by providing a special middleware function with four arguments. Place the snippet below in your `server.js` file, just above the `server` variable.

server.js

```
app.use(function (err, req, res, next) {
  console.error(err);
  res.set('Content-Type', 'text/html');
  res.status(500).send('<h1>Internal Server Error</h1>');
});
```

This function must be placed after all other `app.use()` calls and after all routes to act as a "catch-all" error handler. The error is logged to the

terminal in the above snippet, and an `Internal Server Error` message is returned to the client.

While this works as-is for synchronous route handlers, it won't work for async functions (such as the handler for the `/search` route). To ensure the error handler also processes async errors, we need to wrap the contents of the function in a `try...catch` block and call `next()` when an error is detected to pass the error to the next middleware function (which is the error handler in this case).

Modify your `/search` route handler as shown below:

server.js

```
app.get('/search', async (req, res, next) => {
  try {
    const searchQuery = req.query.q;
    if (!searchQuery) {
      res.redirect(302, '/');
      return;
    }

    const results = await searchHN(searchQuery);
    res.render('search', {
      title: `Search results for: ${searchQuery}`,
      searchResults: results,
      searchQuery,
    });
  } catch (err) {
    next(err);
  }
});
```

With the `try..catch` block above in place, async errors originating from the callback function will be sent to the error handler middleware for further processing. You can read more about error handling in the Express docs.

Step 10 — Deploying the application to production

With some general error handling in place, we've completed our application, and we can now deploy to production. Return to the terminal and kill the development server by pressing `Ctrl-C`, then run the command below to start the application in production mode:

```
$ npm start
```

You should observe the following output if all goes well:

Output

```
> hacker-news@1.0.0 start
> npx pm2 start server.js

[PM2] Starting /home/ayo/dev/demo/hacker-news/server.js in fork_mode (1
instance)
[PM2] Done.
┌────┬──────────┬─────────────┬─────────┬───────┬──────┬────────┬────┬
│ id │ name     │ namespace   │ version │ mode  │ pid  │ uptime │ ↺
│ status   │ cpu     │ mem     │ user    │ watching │
├────┼──────────┼─────────────┼─────────┼───────┼──────┼────────┼────┼
│ 0  │ server   │ default     │ 1.0.0   │ fork  │ 24985 │ 0s    │ 0  │
online   │ 0%      │ 20.3mb  │ ayo     │ disabled │
└────┴──────────┴─────────────┴─────────┴───────┴──────┴────────┴────┴
```

Using a process manager like PM2 is the recommended way to deploy Node.js apps because it makes it easy to manage the application process, and it can keep it running indefinitely through its automatic restart feature that kicks in if the application crashes unexpectedly. Learn more about PM2 here.

At this point, you should be able to access your Express app by going to `http://localhost:3000` or `http://<your_server_ip>:3000` in your web browser.

Conclusion and next steps

In this article, you've learned Express and Pug basics by using them to build an application that searches Hacker News. We discussed various concepts like routing, middleware, processing static files, and rendering views from Pug templates before closing out with a brief demonstration on error handling and deploying to production with PM2. You should now understand the main aspects of building an Express application and how to apply them in your projects.

Do note that the application we've developed in the course of this tutorial is far from production-ready in a true sense. To keep the tutorial relatively short, we didn't cover concepts like caching, logging, securing your Node.js server or integration with a database (such as PostgreSQL or MongoDB), so ensure to perform adequate research on those topics before deploying any real-word project to production.

Thanks for reading, and happy coding! The final code for this tutorial can be viewed in this GitHub repo.

Article by

Ayooluwa Isaiah

Ayo is the Head of Content at Better Stack. His passion is simplifying and communicating complex technical ideas effectively. His work was featured on several esteemed publications including LWN.net, Digital Ocean, and CSS-Tricks. When he’s not writing or coding, he loves to travel, bike, and play tennis.

Got an article suggestion? Let us know

Next article

How to Get Started with Debugging Node.js Applications →

Make your mark

Join the writer's program

Are you a developer and love writing and sharing your knowledge with the world? Join our guest writing program and get paid for writing amazing technical guides. We'll get them to the right readers that will appreciate them.

Write for us ›

Writer of the month

Woo Jia Hao

Woo Jia Hao is a software developer from Singapore. He is an avid learner who...

Build on top of Better Stack

Write a script, app or project on top of Better Stack and share it with the world. Make a public repository and share it with us at our email.

community@betterstack.com

or submit a pull request and help us build better products for everyone.

See the full list of amazing projects on github

Explained: All Meanings of MTTR and Other Incident Metrics

10 Best API Monitoring Tools in 2023

10 Best Docker Monitoring Tools in 2023

Better Stack

Terms of Use Privacy Policy GDPR System status