Edit Manage Stats

Ingo Steinke
Posted on 21 Jun 2021 • Updated on 23 Dec 2021

MERN-Stack Setup: Building a Reading List Web App with Node, Preact, and Tailwind CSS

#typescript #showdev #tailwindcss #javascript

Building a Reading List web app with Node, Preact, and Tailwind CSS (2 Part Series)

This is a work in progress, updated on 1st July 2021. I will only highlight a few code examples and helpful links in this article. You can see the full code and commit messages on GitHub. I will add a link to a live demo in the last part of the article series once (if ever) I have released a working draft.

Table of Contents

Motivation

I finally took some time for proper research (and some lazy trial and error based on tutorials and example code) to set up a proper full-stack application.

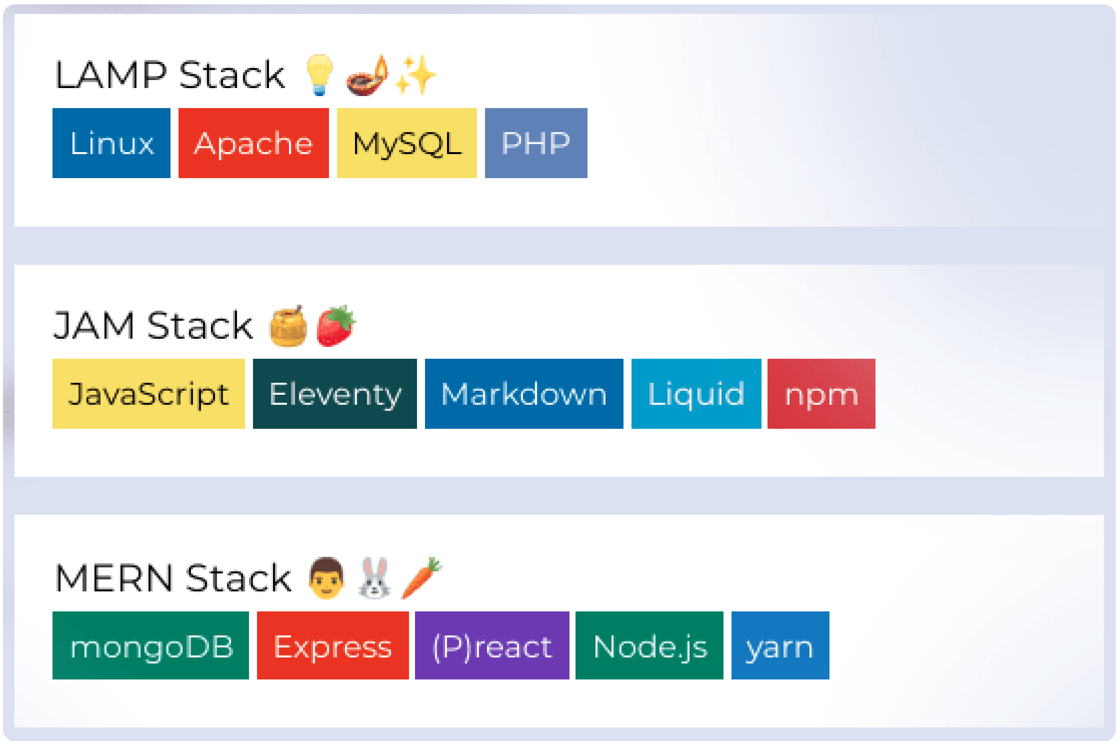
Beginning with a general example / starter setup useful for several projects, forking the actual side project (**bookstack-reading-list-app**) once things get more specific.

Why bother?

In the past, I had worked with legacy code or did not take the time for proper research, which even made me suspect I did not like React or even JavaScript single page applications at all.

It might have been easy to go back to the classic LAMP stack using PHP and SQL, and base my app on the Symfony framework.

But as many other developers pointed out, there is more than one way to do things, especially when it comes to React, and I wanted to keep an open mind and be able to work with different web development tech stacks.



LAMP Stack 💡🪔✨

Linux Apache MySQL PHP

JAM Stack 🍯🍓

JavaScript Eleventy Markdown Liquid npm

MERN Stack 👨🐰🥕

mongoDB Express (P)react Node.js yarn

Finding out: learning, experience, and hopefully some fun...

When I am free to decide everything on my own (thus having to do so) I swing back and forth between joy of discovery and getting stressed out. So many technologies and buzzwords, so much apparent over-complexity.

Fueled by my stress, I find the energy not to simply reproduce some outdated tutorials from 2018, but to try and build only what is actually reasonable now.

... and save me some time in future projects

Besides learning, experience, and hopefully some fun, the result should be an actual project to show, plus a scaffold that might save me some time in future projects. Time that I already invested when researching and building my side project.

How to Build an SPA that does not suck

Trying to use Tailwind, TypeScript, and React in a proper way, or at least in a way that fits my experience and requirements seems a good way to make up my own mind about technology that seems to elicit a lot of controversy in the web developer community.

Coding Horror 😱

Like Stephen Hay said at Beyond Tellerrand conference in 2019:

Stephen Hay

I don’t care what Airbnb is doing.

(And neither should you.)

"I don't care what AirBnB is doing and neither should you," because arbitrary syntax rules like AirBnB's version of ES6, bloated syntax horror like ReactPropTypes or styled components are just some of the things that made me "hate React" in projects of the past.

Styled Components vs. Utility CSS 💅🛠️

While **styled components** are one possible way of modularization, I still do not like the fact that useful core CSS features - classes, cascade and inheritance are rather avoided than used properly. But on the other hand, trying to write proper CSS in complex projects often resulted in messy and unmaintanable style sheets in the past, so maybe there is still another solution.

Thanks to my friend Andy Weisner (Retinafunk) for convincing me that **Utility CSS** can be nice and useful even (or especially) for experienced web developers.

I will elaborate my possibly controversial stance in another article, so we can leave this topic for now.

Slow Pace Applications 🐌

Many single page applications turn out to be "slow pace applications" built with a back-end development mindset that fails to care about usability, accessibility, and page speed / web performance. Try to fix the mess by adding pre-rendering which optimizes for largest content paint on the cost of first input delay and making built time so slow that it feels like developing a monolithic Java application.

Thanks to Anna Backs and Christina Zenzes for the term "slow pace application" and for giving me back some hope that it is still possible to write an SPA that does not suck!

How not to JavaScript – Anleitung zum Unglücklichsein | Anna & Christina a...

Focus on Performance and Minimalism

Why I don't "create react app": in the beginning, I intended to use tailwind with create-react-app, but after trying to solve conflicts and incompatibilities, I rebased the starter on retinafunk's tailwind-jit-starter and decided to drop old technology like webpack in favor of smaller, more elegant solutions, which in consequence lead to the decision to drop React and use preact, focussing on performance and minimalism.

"Zero Configuration" unless ...

After trying parcel, which claims to be a "zero configuration" bundler, we ended up using snowpack.

Parcel 📦

I started with parcel 1, after too many deprecation warnings of required node dependencies, tried parcel 2 although it is still labelled beta (but then again, open vpn used to be "beta" for ages, while working fine in production).

At that step, my starter project had become a combination of at least three different projects, so copying and pasting from one tutorial is not guaranteed to work in another setup. I got a lot of warnings about missing types and missing support for the "experimental syntax 'jsx'".

> "Support for the experimental syntax 'jsx' isn't currently enabled"

The suggestions how to fix were misleading in my case, as the crucial part was using `.jsx` or `.tsx` file extensions for every file that contains JSX. As I never actually used preact before, it was also unclear to me that the seemingly unused `import { h } from 'preact'` actually makes sense once JSX is internally transpiled to an `h()` function.

Snowpack ❄️

Switching from parcel to snowpack, we were able to simplify the setup according to snowpack's preact typescript template.

Now the preact typescript server was running, we had to bring back our tailwind configuration from the original template,

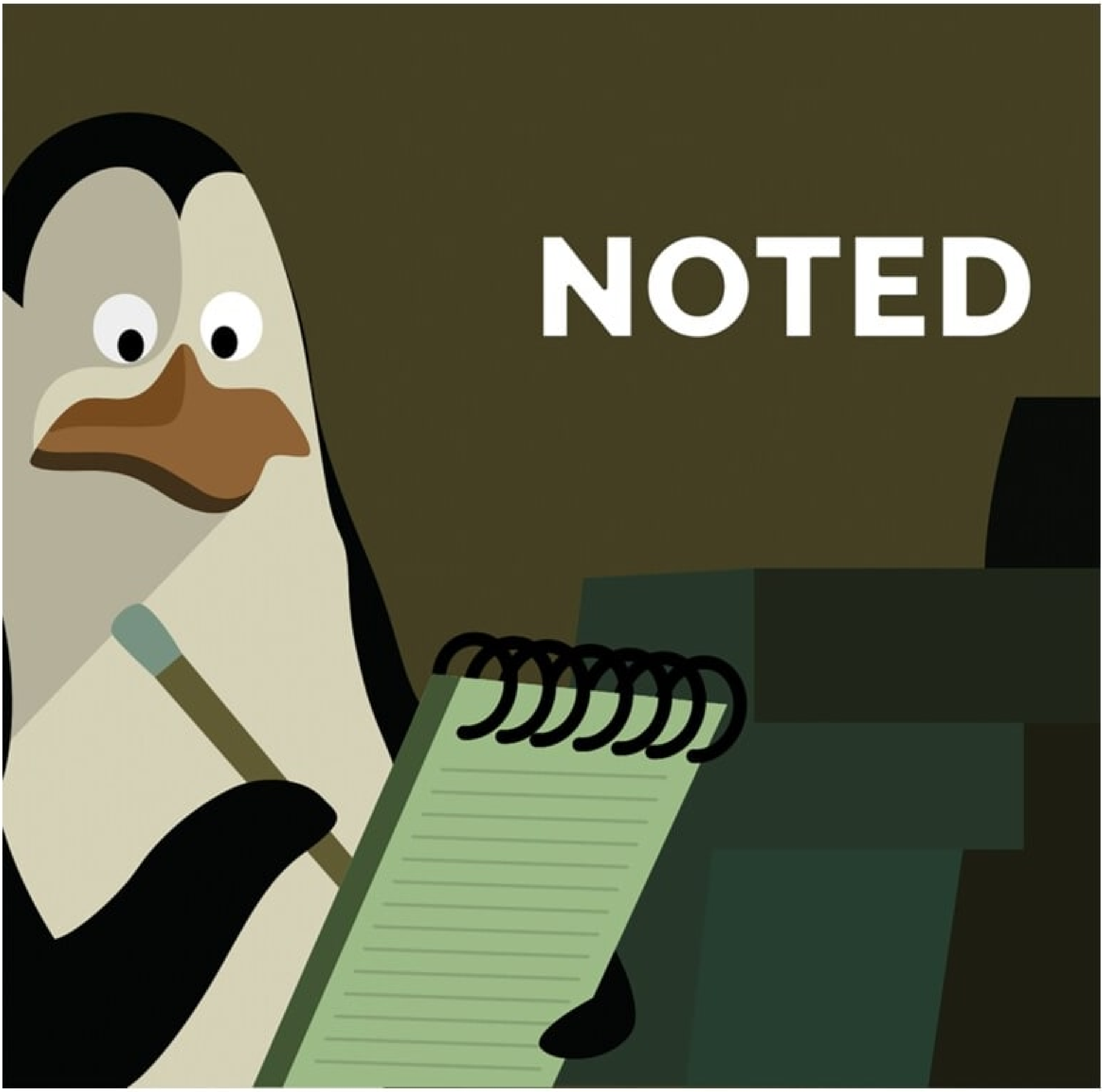
- remove non-tailwind CSS
- configure `postcss` to use tailwind
- adapt build and include paths

Now we have a front-end app based on 3 templates:

- Preact Default using preact router (probably created by `preact cli default`)?
- snowpack preact type script starter (maybe this one but I already forgot which one I actually used),
- Tailwind-JIT starter by retinafunk.

Taking Notes

While I did this not long ago, I still don't remember every details anymore. It can be useful to take notes (or write articles) to actually learn by coding instead of taking the resulting code to copy and paste for a future project.



Front-End Performance, Accessibility and Search Engine Optimization

Google currently favors pages that don't waste the users' loading time (and maybe even costly bandwidth) and which follow their usability guidelines (see [Optimizing Speed and Usability for Google's Core Web Vitals](#)).

Here are a few things we can do right from the start.

Purge Tailwind CSS 🪠

To be able to deploy only a minimal subset of the CSS library, so that only styles are exported which are actually used, we have to make sure `purge` finds all files.

Properly Using Conditional Class Names

How to make sure to write dynamic / conditional styles in a detectable way?

So we must avoid string concatenation to create class names. For example `<div class="text-{{ error ? 'red' : 'green' }}-600">` fails to expose the class names to the purge algorithm, thus `.text-red-600` and `.text-green-600` will be missing in the exported style sheets unless they are used somewhere else by coincidence.

On the other hand, writing the full class names still allows us to use conditional class names:

```
<div class="{{ error ? 'text-red-600' : 'text-green-600' }}"></div>
```

We also want to load CSS in HTML `<head>`, not in JavaScript: this might unblock load speed by allowing parallel download of CSS and JavaScript, and it also allows us to define styled static page elements outside of our (p)react app, like a footer with links to external or static resources.

This also allows search engines which do not run JavaScript, to find some content apart from the `<noscript>` text, which we also want to change into something that our customers might find helpful on search engine result page.

We can define "critical" ("above the fold") CSS and load web fonts in the head as well. Now our HTML markup looks something like this:

src/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Tailwind JIT MERN App Starter</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    /* critical "above the fold" styles to prevent layout shift */
    html, body {
      font-family: 'Alegreya', serif;
      background-color: #7fb71e;
      color: #fff;
    }
    </style>
    <link rel="preconnect" href="https://fonts.gstatic.com">
    <link href="https://fonts.googleapis.com/css2?family=Alegreya:wght@400;700&
    <link rel="stylesheet" href="index.css" />
    <script type="module" src="index.js" defer></script>
    <!-- add meta information like favicon, twitter cards etc. here -->
  </head>
  <body>
    <div id="root"></div>
    <footer class="bg-black text-white">
      Static footer content
      <a href="https://example.com">example.com</a>
    </footer>
    <aside>
      <noscript>You need to enable JavaScript to run this app.</noscript>
    </aside>
  </body>
</html>
```

You need to enable JavaScript to run this app?

This is technically true, but probably not helpful to anyone. There should at least be any information about the app, maybe even a statically rendered snapshot of some highlighted content, which will help real people and could benefit search engine optimization as well.

Finishing the Basic Front-End Setup

Cleanup, fix typescript warnings (missing types for components), configure production build, add tests and ensure accessibility and performance.

"Could not find a declaration file for module" is another misleading error message, as we don't need to provide `.d.ts` files when we can provide type information directly in our code.

Converting preact default example code to typescript was easy. Functional components without any properties only need a definition as a function component. Once provided, our editors already tells us to rename the file to `.tsx` which does not break any import statement, as file extensions have to be left out anyway (so everything still works after implicit transpilation from `tsx` back to `jsx` internally).

Simple property type definitions can be provided inline, without creating interfaces. Our TypeScript code is much more compact than the same definition would be using `React.PropTypes`.

```
import { FunctionComponent } from 'preact';
const Profile: FunctionComponent<{ user?: string }> = ({ user }) => {
  // ...
}

export default Profile;
```

Also had to make sure that the application will work in a sub folder as part of a monorepo, as some configurations were said to work "only in the root directory" of a project (which does not make sense, as I never save anything to "/", and any application should not care where its root folder is mounted).

Building the back-end using JavaScript

Thanks to node.js, we can use modern JavaScript on the server-side. So I wonder:

While many tutorials still suggest to set up Babel to use ES6 syntax in node, which I could not believe is still true in 2021, and luckily it isn't! But what about TypeScript, or do node (or deno) offer native typing without any extensions?

ES6 and TypeScript in NodeJS without Babel

While deno supports TypeScript "out of the box", I was still curious how to use modern syntax in `node`. While the web is full of answers stating to "simply" start node with the `--harmony` flag, little was written about how to achieve this descriptively inside our `package.json`.

But, no surprise, this is all just outdated code? Node already supports ES6 by default, and modern modules once we specify `"type": "module"` in our `package.json`? So what about types(cript) then?

Trying to use Typescript in the Back-End...

```
yarn add -D typescript
yarn add -D ts-node \@types/node \@types/express \@types/cors \@types/mongoose
yarn tsc --init --target esnext --rootDir src --outDir build
```

"Nodemon will detect and run .ts files with ts-node automatically now" triggered by the file extension, so we have to rename `server.js` to `server.ts`.

... before finally giving up

After trying for several days (!) to use mongoose schemata, express and node together with TypeScript, I gave up for the moment, as literally no back-end tutorial seems to use TS and it confused me a lot to get the types and interfaces right.

As I want to concentrate on the front-end anyway, I will use TypeScript in the front-end only in the first version.

Node Express: Built-in Middleware

Any common use case is probably built into latest express core. Apart from `CORS` (see below), a classic tutorial boilerplate code to install `bodyParser` as an extra module is also obsolete for common cases. Thanks to Taylor Beeston for pointing it out.

Put Your Helmet On 👷

To avoid the obvious security warning in our browser, we can add a classic wildcard CORS header:

```
res.header("Access-Control-Allow-Origin", "*");
```

... . But we do not have to stop here, nor do we have to do this manually, do we? The useful helmetjs should be part of any proper node express setup.

```
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
```

Do we really need to restart the server after each change? Sure there is some sort of preview watcher available? Nodemon would have done this, but we can also use Rollup instead.

Setting Up a Database

One reason for using MongoDB is to go full-stack using only JavaScript. Some tutorials suggest installing mongodb using homebrew on a Mac. As npm does not seem to offer mongo, why not try docker - that might even already be the solution how to manage that both locally for development now, and on AWS in production later.

Mongoose ServerSelectionError: ECONNREFUSED

I still can't believe how hard it can be to connect to a local database. Unfamiliar to the current syntax (many code on the web,like from a 3 year old tutorial is already deprecated) and struggling with misleading error messages, I missed the point that you have use MongoDB's docker service name instead of localhost in a `docker-compose` setup.

At least my full-stack setup got easier again on the client side:

Connecting Client and Server locally

... by adding a `fetch()` method to the front-end. Copy and paste boilerplate code: `const asJson = r => r.json();`. Reading the warning that `r` implicitly "has the 'any' type" makes me wonder, what type to expect anyway. First google result:

> "The response of a fetch() request is a Stream object, which means that when we call the json() method, a Promise is returned since the reading of the stream will happen asynchronously."

Still wondering, without any real TS experience, how to specify it? Simply write new Stream, or more simply, `asJson< r: stream>` or rather `r: Stream` as this is surely no simple type? But, not so quickly: "Cannot find name 'Stream'. Did you mean 'MSStream'"?

Then again it does not seem to be common practice to type every single variable. At least I found a lot of TypeScript examples that rely on tolerating `implicit any`.

After solving the type issue, we can replace `fetch` by a `fetchRetry` wrapper so our app does not fail if there is a glitch in the network. I still don't understand why there is no retry option in the native Fetch API yet.

A Universal App in Production

A first glance at our simple server looks too naive in many ways. While in a classic LAMP stack we would probably use Apache or nginx as a server and provide nothing but configuration to control some headers and rewrites and use a back-end language for the logic, it seems as if we wanted to reinvent the wheel when using node.

Currently we only need to call one undefined or invalid route to make the server crash completely after putting out a detailed error message to a potential attacker.

Why do we even need a node server? We need a layer to ensure authentication and authorization, and probably simplify query syntax to provide either a REST API or a GraphQL middleware.

If we use an existing cloud infrastructure like AWS, they will probably handle load balancing and caching in a better way than any hand-coded setup could do.

For a production setup we could use nginx to serve both client and server app on the same domain, same (default public) port, routing all `/api` requests to the back-end, like Asim describes in the article on how to deploy a React + Node app to production on AWS

Universal / Shared Code 🤝

Why use back-end JavaScript at all? PHP has improved a lot in the past 10 years, and there are other popular languages like Go and Python, so why bother? By using the same language, JavaScript, both for client and server inside the same repository, we can possibly avoid redundant code and share common business logic between front-end and back-end.

Don't repeat yourself?

What struck me was that I was not able to find much about how to avoid redundancy by providing a common data model instead of writing at least 2x, mongoose in the back-end and JavaScript in the front-end application. Either nobody cares, or there is a good reason not to do it. I can imagine that this would introduce technical debt, coupling or even unintended downsides concerning front-end business logic. But why do I have to find out by trial and error? Hope to provide an answer and share my results of research and experiment after doing so...

Going Full-Stack: Setting Up a Monorepo 🚄

Time to add the back-end application to our "universal" setup. After moving our front-end code in a folder of its own, I learned that you can't simple make a script defined in a top-level `package.json` execute another script defined in another `package.json` which is inside a sub folder (is it not called directory anymore?)

We will solve this using `workspaces` to create a so-called **monorepo** later. Once we are inside our sub folder, the front-end app still works as it used to, like when called by

```
cd client && yarn start
```

Yarn Workspaces 🧵

Yarn provides a useful feature called workspaces, which is a more elegant and platform-agnostic alternative to my `start.sh` shell script.

In a new top-level `package.json`, we can refer to our existing `client/package.json` and `server/package.json` by their name (not by their path, so be sure to match the `name` defined in the existing package files.

```
/package.json
```

```
  "workspaces": [
    "client",
    "server"
  ],
  "scripts": {
    "client": "yarn workspace client start",
    "server": "yarn workspace server start",
    "database": "yarn workspace server start-db",
    "start": "concurrently --kill-others-on-fail \"yarn database\"  \"yarn server
  },
  "dependencies": {
    "concurrently": "^6.2.0"
  }
```

Concurrently allows us to start both of them simultaneously, as well as our third component, the database server, via the build target `start-db` that does nothing else but run an existing docker setup.

`/server/package.json`

```
    "scripts": {
      "start": "node src/api-service/index.js",
      "start-db": "docker-compose up",
```

Resilience against Race Conditions 🏁

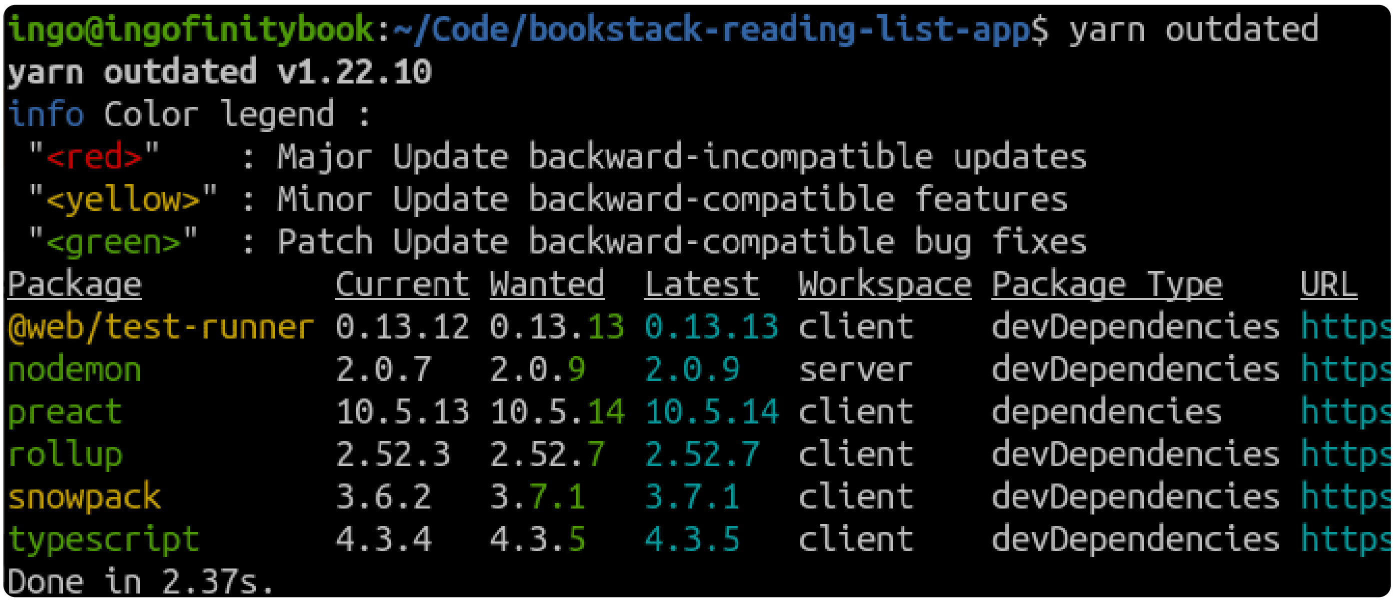
This kind of setup can only work by chance, as it creates a race condition by failing to define, that the client depends on the server, which, in turn, depends on the database.

But if we plan to deploy our app in the cloud, having each component running independently from the others, each component must be so resilient to check whether the service is available, and wait before retrying if not.

Monorepo Top Level Commands

A monorepo can be controlled by using `yarn` in the top level directory of our repository for tasks like installation and updates.

When I want to upgrade my dependencies, I only have to type `yarn outdated` and it will list the suggested upgrades both for server and client application:



```
ingo@ingofinitybook:~/Code/bookstack-reading-list-app$ yarn outdated
yarn outdated v1.22.10
info Color legend :
 "<red>"    : Major Update backward-incompatible updates
 "<yellow>" : Minor Update backward-compatible features
 "<green>"  : Patch Update backward-compatible bug fixes
Package          Current Wanted  Latest  Workspace Package Type    URL
@web/test-runner 0.13.12 0.13.13 0.13.13 client    devDependencies https
nodemon          2.0.7   2.0.9   2.0.9   server    devDependencies https
preact           10.5.13 10.5.14 10.5.14 client    dependencies    https
rollup           2.52.3  2.52.7  2.52.7  client    devDependencies https
snowpack         3.6.2   3.7.1   3.7.1   client    devDependencies https
typescript       4.3.4   4.3.5   4.3.5   client    devDependencies https
Done in 2.37s.
```

I will continue to show and comment my work in a series of articles, so follow me to stay tuned!

Acknowledgements 🙏

I want to say thanks to several people and communities:

Retinafunk (Andy Weisner)

Retinafunk's tailwind-jit-starter as well as Andy's support and suggestions saved me from abandoning tailwind before I even made it work.

Anna Backs and Christina Zenzes

Anna's and Christina's talk "Slow Pace Application" (a follow-up to "Anleitung zum Unglücklichsein: How not to JavaScript" ("The Pursuit of Unhappiness: How not to JavaScript", one of the best talks about JavaScript performance optimization ever! Why does everybody else have to be so serious?)

StackOverflow

While I'm still not happy about the smart Alecs that keep closing and downvoting my questions, while at the same time failing to delete outdated "works for me" answers, I can't help to say that StackOverflow does provide many valuable answers to every day dev problems. Hopefully that will not change after the acquisition by Prosus, but if it does, we will get over it.

dev.to

Without abitrary gatekeeping (like StackOverflow) nor pay-for-free-content (like medium.com), there is finally a community for in-depth coding discussion and exchange of ideas. Thanks for everyone who take their time to read, write and edit articles. Thanks for reading! Hope that I can add something, too.

About the Cover Image

... and also a big shout out to all the friendly and open-minded people I met while travelling in the UK in 2018 and 2019, very lucky to have done this before Brexit and Covid, so I have some nice memories to remember while sitting at my home office desk. This article's cover image was taken in the little town of Settle in Yorkshire, UK and you can see it here in my flickr photostream and read more about the trip in my blog.

About me: I am Ingo Steinke, a creative web developer, creating sustainable software and websites. If you like my talks and articles, feel free to support me on patreon, buy me a coffee or hire me as a freelancer.

Building a Reading List web app with Node, Preact, and Tailwind CSS (2 Part Series)

1 **MERN-Stack Setup: Building a Reading List Web App with Node,...**

2 No need for Virtual DOM and Controlled Components

Discussion (5)

Ingo Steinke • Jun 21 '21

Please note: this is a work in progress, first published on 21 June 2021. I will only highlight a few code examples and helpful links in this article. You can see the full code and commit messages on GitHub. I will add a link to a live demo once (if ever) I have published a working draft. There will be updates to this article in the future.

Ingo Steinke • Jun 29 '21 • Edited on Jul 1

I updated and edited the article to be more concise (and no more fragments in German, I promise) and show more code examples. The article is now part of a series: Building a reading list web app with Node, Preact, and Tailwind CSS. I will continue to show and comment my work in a series of articles, so follow me to stay tuned!

InHuOfficial • Jun 21 '21

Interesting account “as you are doing it”. I did think something was wrong with me half way through as you still had a German section in there...really threw me off lol.

Look forward to seeing the end product and conclusions! ❤️

Ingo Steinke • Jun 21 '21 • Edited on Jul 1

thanks Graham! Thanks for reading! Just thought I should turn it into an article series.

InHuOfficial • Jun 21 '21

It was interesting to see where the pain points were and how you attacked stuff!

Code of Conduct • Report abuse

Ingo Steinke

Web Development, Sustainability, Art and Music, Nature and Travel, Sustainability

LOCATION
Germany

WORK
Creative Web Developer at Ingo Steinke

JOINED
21 Sep 2019

More from Ingo Steinke

Animated Gradient Text Color
#webdev #showdev #css #tutorial

CSS :has(.parent-selectors)
#css #webdev #todayilearned #javascript

Using JSDoc to write better JavaScript Code
#javascript #webdev #programming #typescript