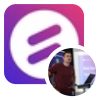


György Márk Varga for Shiwaforce
Posted on Feb 5, 2024 • Edited on Jul 7, 2024

Notion Like text editor with AI autocomplete and Neon database in Next.js using shadcn/ui

#webdev #typescript #tutorial #react

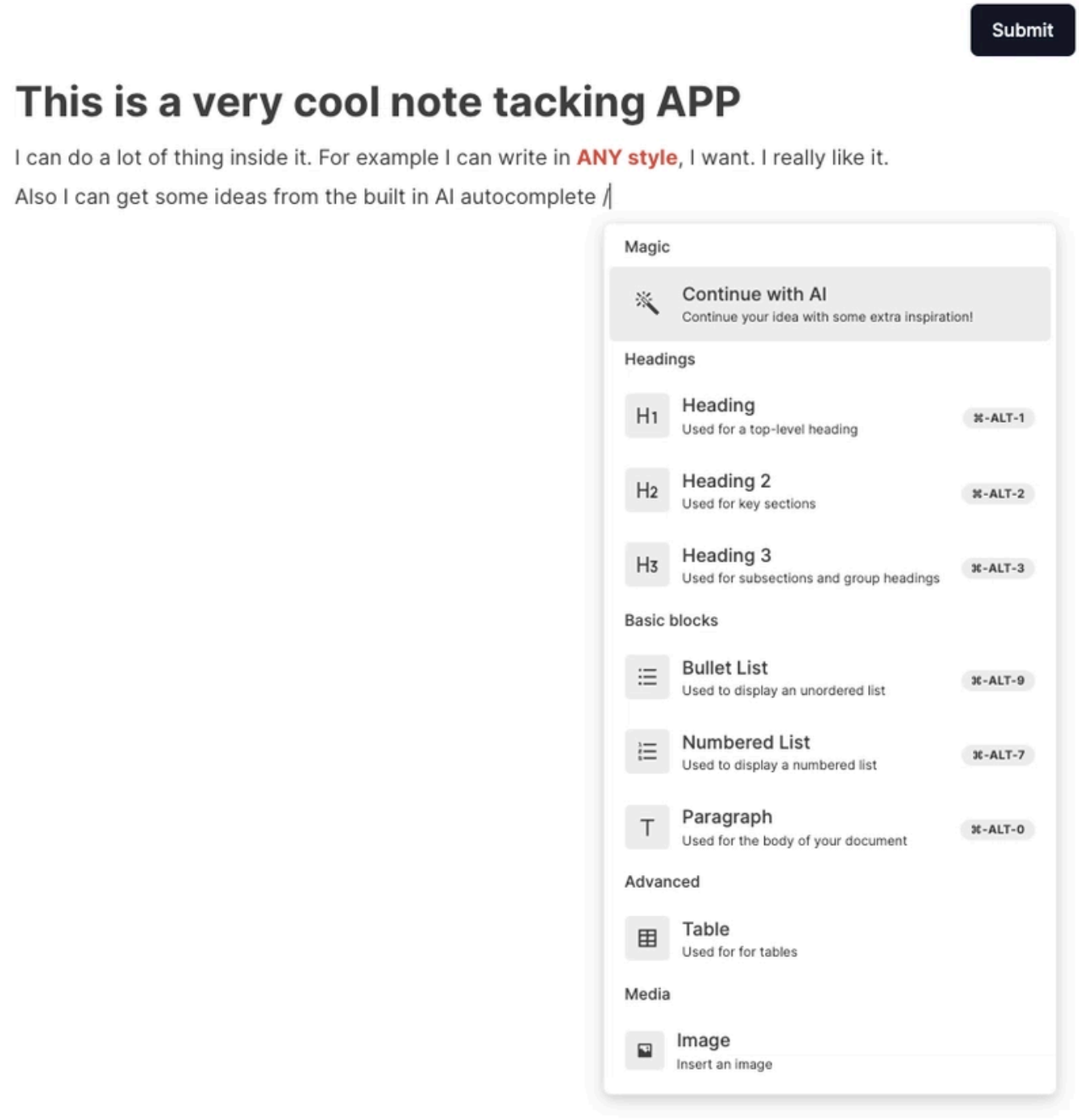
Nowadays, there are many text editors that can be integrated into React applications. We can easily get lost in them, the selection is so wide.

If we are looking for inspiration in web products, we can still browse through many options. Many of you here are probably familiar with Notion and its associated document editor, which is very easy and user-friendly to use. I think this is one of the best. It's not just me, many people see it this way, which is why there are quite a few downloadable and/or open source solutions available as npm packages, with which we can integrate a good text editor experience into our web application.

I could go out with two projects that I have already worked with. One is Novel and the other is Blocknote. Both are based on already existing open-source projects, none other than Tip-Tap and Prosemirror. However, they are much easier to use. How easily these can be used is also evident from the fact that, in the case of Blocknote, we can already snap them into our React application with these lines of code:

```
import { BlockNoteEditor } from "@blocknote/core";
import { BlockNoteView, useCreateBlockNote } from "@blocknote/react";
import "@blocknote/core/style.css";

export default function App() {
  // Creates a new editor instance.
  const editor = useCreateBlockNote();

  // Renders the editor instance using a React component.
  return <BlockNoteView editor={editor} theme={"light"} />;
}
```

What are we going to build?



Well, let's not run so far ahead. Let's see what we will create together in this article and what we will get to know along the way:
We are making an application where you can take notes and view them in a list and in details in read-only mode as you can see in the picture above. The editing interface is supplemented with a small AI autocomplete, which is integrated into Blocknote based on the open source Novel editor. Let's see which main technologies we will use for this:

- Blocknote
- OpenAI API
- Scadcn UI
- Vercel

Among other things, we will learn how to stream responses in the form of an edge function, which is available through the AI package provided by Vercel. We will also look at how and in what form we save a formatted document to a Postgres database (Neon), so it will be worth staying with me until the end of the article.

Implementation

Creating the project

Let's start the project and get to know the tools used in detail along the way.
We can create the project called 'note-blocks' by issuing the following command:

```
npx create-next-app@latest
```

We will work with the following configuration (everything remains at default):

```
Ok to proceed? (y) y
✔ What is your project named? … note-blocks
✔ Would you like to use TypeScript? … No / Yes
✔ Would you like to use ESLint? … No / Yes
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like to use `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to customize the default import alias (@/*)? … No / Yes
Creating a new Next.js app in /Users/gyuri/Desktop/note-blocks.
```

Then, let's quickly "clean up" the lines of code left behind after creating the Next.js project. Let's start with `globals.css`. It's enough if we just leave this:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Creating pages

Then go to `page.tsx` inside the app directory. And there we can convert the entire file to this:

```
import Link from "next/link";

export default function Home() {
  return (
    <main className="flex min-h-screen flex-col items-center justify-between p-24">
      <div className="z-10 max-w-5xl w-full items-center justify-between font-mono t
        <Link href="/new">
          New Note
        </Link>
      </div>
    </main>
  )
}
```

Don't apply any design yet, that moment will come later. First, let's create a Next page, where we will arrive when we click on the “New Note” link. We create a folder called "new" in the "app" directory, then inside it a `page.tsx` file with this content:

```
export default function Page() {
  return (
    <div>
      Editor
    </div>
  )
}
```

It's not too much, and it's not pretty, but we'll talk about it later. For the time being, the placeholder page will open if you click on the "New Note" link.

Adding and using the Blocknote editor

It follows that, similar to the method described at the beginning of the article, we drag the Blocknote editor onto this page. To do this, we need to install it in our application with npm install:

```
npm install @blocknote/core @blocknote/react
```

operated on the server side, in this regard you can read the relevant part of Blocknote's documentation here.

We can also insert the following lines into the Editor component:

```
"use client";
import { BlockNoteEditor } from "@blocknote/core";
import { BlockNoteView, useCreateBlockNote } from "@blocknote/react";
import "@blocknote/core/style.css";

export default function Editor() {
  const editor = useCreateBlockNote();

  return <BlockNoteView editor={editor} />;
}
```

Make sure that there is `"use client"` at the top of the file, which indicates to Next.js that this is a client component.

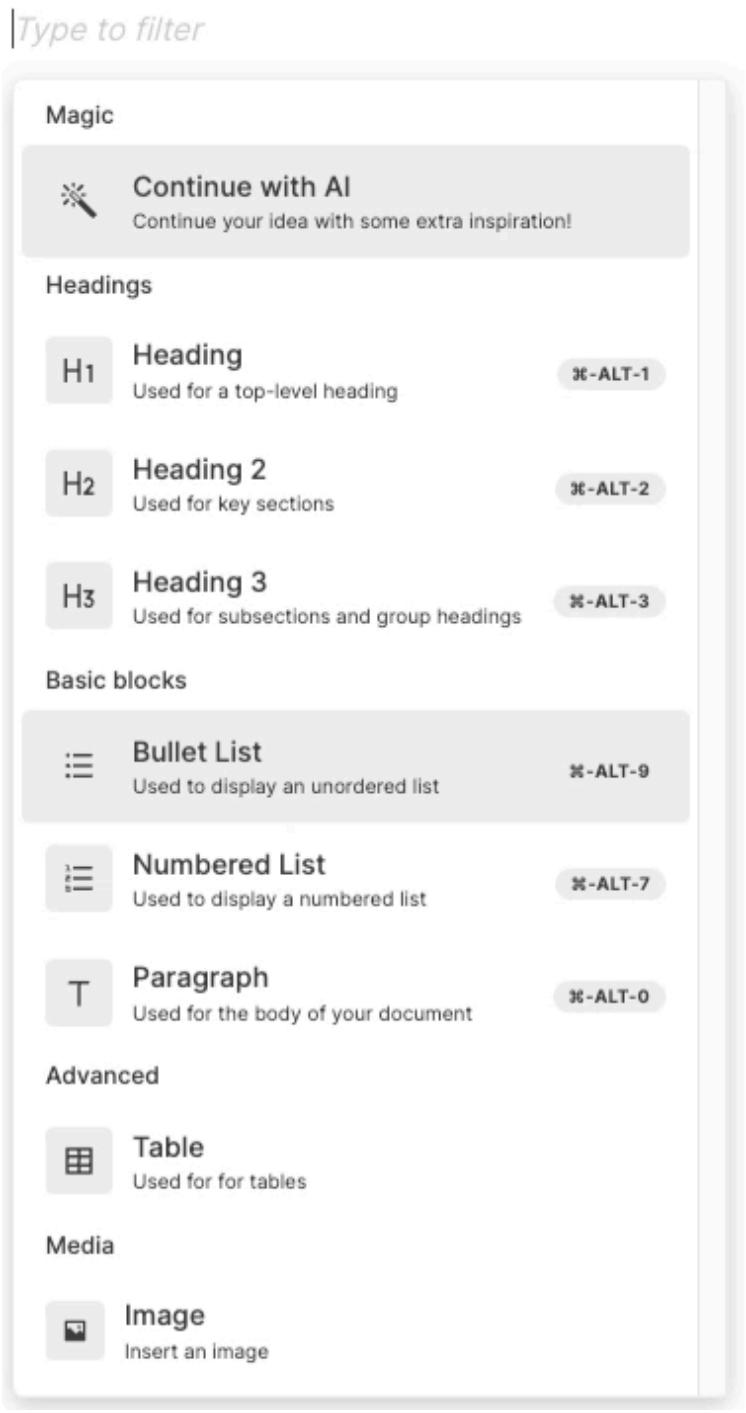
And page.tsx is transformed as follows:

```
import dynamic from "next/dynamic";

const Editor = dynamic(() => import("./Editor"), { ssr: false });

export default function Page() {
  return (
    <div>
      <Editor />
    </div>
  )
}
```

Here we import our client-side Editor component in such a way that it is represented on the client side (`'ssr: false'`). So when we're done with that, we can already see

After looking at it and testing it, we quickly realized that this editor is very good, but how are we going to do the rest of the application?

Let's move nicely in line. First, let's see how we can add AI autocomplete to our little editor, first of all, let's see how we can add a menu item to our text editor. This menu item appears when we press a "/" in our editor or click on the "+" sign at the beginning of the line. So we want our Toolbar to look like this:



Implementing the magic AI function

We need to modify the code in the `Editor.tsx` file for the desired behaviour:

```
"use client";
import { BlockNoteEditor } from "@blocknote/core";
import { BlockNoteView, getDefaultReactSlashMenuItems, ReactSlashMenuItem, useCreate
import "@blocknote/core/style.css";
import { ImMagicWand } from "react-icons/im";

const insertMagicAi = (editor: BlockNoteEditor) => {
  console.log('Magic AI insertion incoming!')
};

  const insertMagicItem = (editor: BlockNoteEditor) => ({
    title: 'Insert Magic Text',
    onItemClick: async () => {
      const prevText = editor._tiptapEditor.state.doc.textBetween(
          Math.max(0, editor._tiptapEditor.state.selection.from - 5000),
          editor._tiptapEditor.state.selection.from - 1,
          '\n'
      );
      insertMagicAi(editor);
    },
    aliases: ['autocomplete', 'ai'],
    group: 'AI',
    icon: <ImMagicWand size={18} />,
    subtext: 'Continue your note with AI-generated text',
  });

  const getCustomSlashMenuItems = (
      editor: BlockNoteEditor
  ): DefaultReactSuggestionItem[] => [
    ...getDefaultReactSlashMenuItems(editor),
    insertMagicItem(editor),
  ];

export default function Editor() {
  const editor: BlockNoteEditor | null = useCreateBlockNote();

  return <BlockNoteView editor={editor} theme={"light"} />;
}
```

What this code addition currently does is to add a new item to the `"slashMenuItems"` list, which will be responsible for the AI autocomplete operation. But how are we going to do this function? Of course, we call the OpenAI API. If you haven't registered

Let's get into it. First, we need an endpoint that can stream the response, which OpenAI gives us. Here, the part of the code that can be found in the Novel open source project will be perfect for us. Let's create an "api" folder inside the "app" directory and a "generate" folder inside that "api" folder, and then a `route.ts` file in it. This folder structure tells Next.js that this will be an api endpoint at this URI: `/api/generate`. The following content will be added to the "route.ts" file, don't be alarmed if the packages that we will use in it are not installed at first, we will explain them and install everything:

```
import OpenAI from 'openai';
import { OpenAIStream, StreamingTextResponse } from 'ai';
import { kv } from '@vercel/kv';
import { Ratelimit } from '@upstash/ratelimit';

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY || '',
});

export const runtime = 'edge';

export async function POST(req: Request): Promise<Response> {
  if (!process.env.OPENAI_API_KEY || process.env.OPENAI_API_KEY === '') {
    return new Response(
      'Missing OPENAI_API_KEY - make sure to add it to your .env file.',
      {
        status: 400,
      }
    );
  }
  if (
    process.env.NODE_ENV != 'development' &&
    process.env.KV_REST_API_URL &&
    process.env.KV_REST_API_TOKEN
  ) {
    const ip = req.headers.get('x-forwarded-for');
    const ratelimit = new Ratelimit({
      redis: kv,
      limiter: Ratelimit.slidingWindow(50, '1 d'),
    });
```

```
    );

    if (!success) {
      return new Response('You have reached your request limit for the day.', {
        status: 429,
        headers: {
          'X-RateLimit-Limit': limit.toString(),
          'X-RateLimit-Remaining': remaining.toString(),
          'X-RateLimit-Reset': reset.toString(),
        },
      });
    }
  }

  let { prompt } = await req.json();

  const response = await openai.chat.completions.create({
    model: 'gpt-3.5-turbo',
    messages: [
      {
        role: 'system',
        content:
          'You are an AI writing assistant that continues existing text based on con
          'Give more weight/priority to the later characters than the beginning ones
          'Limit your response to no more than 200 characters, but make sure to cons
      },
      {
        role: 'user',
        content: prompt,
      },
    ],
    temperature: 0.7,
    top_p: 1,
    frequency_penalty: 0,
    presence_penalty: 0,
    stream: true,
    n: 1,
  });

  const stream = OpenAIStream(response);

  return new StreamingTextResponse(stream);
}
```

It is not as hard as it seems at first. First, let's look at the package imports one by one at the top. First, we import the OpenAi package, which we can load into our program like this:

```
npm install openai
```

This provides us with an API client interface.
The next line shows a mysterious package called "ai". Vercel provides this for us. It will help us properly stream our response from OpenAI to the endpoint caller. Let's install this:

```
npm install ai
```

Then, in the next two import lines, we will also import the serverless Redis component called KV from Vercel, and then also a rate limiter, which will be responsible for ensuring that endless requests do not arrive here. We also install these:

```
npm install @vercel/kv
npm install @upstash/ratelimit
```

Once we have this, we can see that we don't encounter any more errors in the code lines (hopefully) and we can see exactly what the "soul" of our application is doing:

```
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY || '',
});
```

This section initializes the OpenAI object with the API key obtained from the environment variables `process.env.OPENAI_API_KEY`. You can also copy your own key from the OpenAI developer interface and insert it into the `.env` file created in the

Just in case this was missed, in the next line of code we check whether we have defined this key:

```
if (!process.env.OPENAI_API_KEY || process.env.OPENAI_API_KEY === '') {
  return new Response(...);
}
```

And the next part uses the users IP address to control the requests to prevent overuse if you have the right .env variables for the requests. We will create these .env variables later, don't worry about it for now.

```
 if (
    process.env.NODE_ENV != 'development' &&
    process.env.KV_REST_API_URL &&
    process.env.KV_REST_API_TOKEN
  ) {
const ip = req.headers.get('x-forwarded-for');
const { success, limit, reset, remaining } = await ratelimit.limit(
  `noteblock_ratelimit_${ip}`
);
}
```

This line reads the user-supplied prompt (initial text) from the request:

```
let { prompt } = await req.json();
```

We use the GPT-3.5-turbo model to generate the answers:

```
const response = await openai.chat.completions.create({...});
```

The response is sent back to the user as a stream (a continuous stream of data), which enables more efficient data transmission, especially for larger responses:

```
const stream = OpenAIStream(response);
return new StreamingTextResponse(stream);
```

Now that we have an overview of exactly what this API endpoint does, we can move on to the next part, which is actually putting this endpoint to use. We go back to our previously edited `TextEditor.tsx` file and insert this function above the `insertMagic()` function:

```
const { complete } = useCompletion({
   id: 'hackathon_starter',
   api: '/api/generate',
   onResponse: (response) => {
     if (response.status === 429) {
       return;
     }
     if (response.body) {
       const reader = response.body.getReader();
       let decoder = new TextDecoder();
       reader.read().then(function processText({ done, value }) {
         if (done) {
           return;
         }
         let chunk = decoder.decode(value, { stream: true });
         editor?._tiptapEditor.commands.insertContent(chunk);
         reader.read().then(processText);
       });
     } else {
       console.error('Response body is null');
     }
   },
```

```
  });
```

`useCompletion()` must be imported from the AI module provided by Vercel:

```
import { useCompletion } from "ai/react";
```

This utilizes a custom React hook, useCompletion, to handle API responses and errors. It makes an API call, handles rate-limiting responses (status 429), and processes the response stream. If the response contains data, it decodes and inserts this content into our text editor.

Once we have this, add the `insertMagicAi()` function as follows:

```
const insertMagicAi = (editor: BlockNoteEditor) => {
  const prevText = editor._tiptapEditor.state.doc.textBetween(
      Math.max(0, editor._tiptapEditor.state.selection.from - 5000),
      editor._tiptapEditor.state.selection.from - 1,
      '\n'
  );
  complete(prevText);
};
```

Here you can see that we call our previously written `complete()` and getting the previous text. We extract the last text context from our editor (5000 characters to be exact) and continue the AI-supported generation of the text based on that.

If we test the functionality on the interface now, we can already see (if we did everything right) that AI autocomplete works.

Using a database to store the notes

After registration at Neon, we can very easily create a free database for our application.

Once we have that, we can look at our database url, which we can copy one by one into our .env file.
However, Prisma itself is not installed yet. We can install it by issuing the following command standing on the root directory:

```
npm install prisma --save-dev
```

Then start the Prisma CLI:

```
npx prisma
```

After that, we initialize the files necessary for Prisma to work:

```
npx prisma init
```

We can see that a Prisma folder has been created in the root directory, which contains a `schema.prisma` file, the content of which can be replaced with this:

```
datasource db {
  provider     = "postgresql"
  url          = env("DATABASE_URL")
  relationMode = "prisma"
}

generator client {
```

In this file, we specify that we will use postgres and what our database access path will be. Our next task will be to create the schema, which describes how our database will look like. We stay in the same `schema.prisma` file and insert this model:

```
model Note {
  id        String   @id @default(cuid())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  document  Json?    @db.Json
}
```

This defines a schema, which is a note object. In our Notion-like editor, we will store the text in the document field, in Json form. Fortunately, Postgres already supports this.
To implement the changes, run this command, which will "push" our schema model into the Neon database:

```
npx prisma db push
```

We'd be fine with that. We've come a long way, let's see what's left to build the app:

- We need to save the note to the database using a Save button, Server Action will come in handy here
- We need to list our notes on the main page
- In terms of appearance, we need to get our site in order - TailwindCSS and the Schadcn UI library will help us here

Let's create the save function first. For this, we need to be able to access the Prisma client from anywhere in the application. In the prisma folder where we also have the `prisma.schema` ts file, create a `client.tsx` file that will look like this:

```
import { PrismaClient } from '@prisma/client';
```

```
  }
}
interface CustomNodeJsGlobal extends NodeJS.Global {
  prisma: PrismaClient;
}
declare const global: CustomNodeJsGlobal;

const prisma = global.prisma || new PrismaClient();

if (process.env.NODE_ENV !== 'production') global.prisma = prisma;

export default prisma;
```

Adding the saving action

Next, create a file in the root folder called `actions.ts`. Our server action, which is responsible for saving the note, will be placed here.

```
'use server'
import prisma from "@/prisma/client";

type Note = {
  document: object
}
export async function createNote(note: Note) {
  return prisma.note.create({
    data: note,
  });
}
```

Let's integrate this aciton next to our Editor by connecting a Save button. This requires a button. How surprising, right? First, let's put a very simple, design-free, smooth, natural HTML button on the interface. We will do this in `TextEditor.tsx`. First, import our server action at the top of the file:

```
import { createNote } from "@/app/actions";
```

At the bottom of the code line, add the following:

```
const handleSubmitNote = async () => {
   const note = {
     document: editor.document
   }
   await createNote(note)
 }
 return (
   <div>
     <BlockNoteView editor={editor} theme={"light"}/>
     <button onClick={() => handleSubmitNote()}>Submit</button>
   </div>
 );
```

In the `handleSubmitNote()` function, we call our server action, which allows us to save our given note. Let's test the function. After we hit "Submit" we don't see anything, since we haven't developed what happens after the Save. For now, to check our work, let's look into the database. We can do that by starting Prisma Studio, enter in the terminal that:

```
prisma studio
```

A database viewing system is already opened on a port, in which the content of the table is entered when clicking on the appropriate model, and thus we can see (if we have done it well) that there will be a line containing our note.

Designing our application

We would be ready with the main functions of our application. In the following, let's make it all a bit more fancy.
For this - as I have already described - we will use TailwindCSS with ShadCN. Tailwind

```
shadcn-ui@latest init
```

We'll go through everything on the default, except for the theme, which can actually be anything you like.

```
✔ Would you like to use TypeScript (recommended)? … no / yes
✔ Which style would you like to use? › Default
✔ Which color would you like to use as base color? › Gray
✔ Where is your global CSS file? … app/globals.css
✔ Would you like to use CSS variables for colors? … no / yes
✔ Are you using a custom tailwind prefix eg. tw-? (Leave blank if not) …
✔ Where is your tailwind.config.js located? … tailwind.config.js
✔ Configure the import alias for components: … @/components
✔ Configure the import alias for utils: … @/lib/utils
✔ Are you using React Server Components? … no / yes
✔ Write configuration to components.json. Proceed? … yes

✔ Writing components.json...
✔ Initializing project...
✔ Installing dependencies...

Success! Project initialization completed.
```

We will definitely need a button from this component library. We can install it as follows:

```
shadcn-ui@latest add button
```

Then on the "main page" we will be able to use it in `app/page.tsx`, which will look like this:

```
import Link from "next/link";
import { Button } from "@/components/ui/button"
export default function Home() {
  return (
    <main className="flex min-h-screen flex-col items-center justify-between p-24">
```

```
            New Note
          </Link>
        </Button>
      </div>
    </main>
  )
}
```

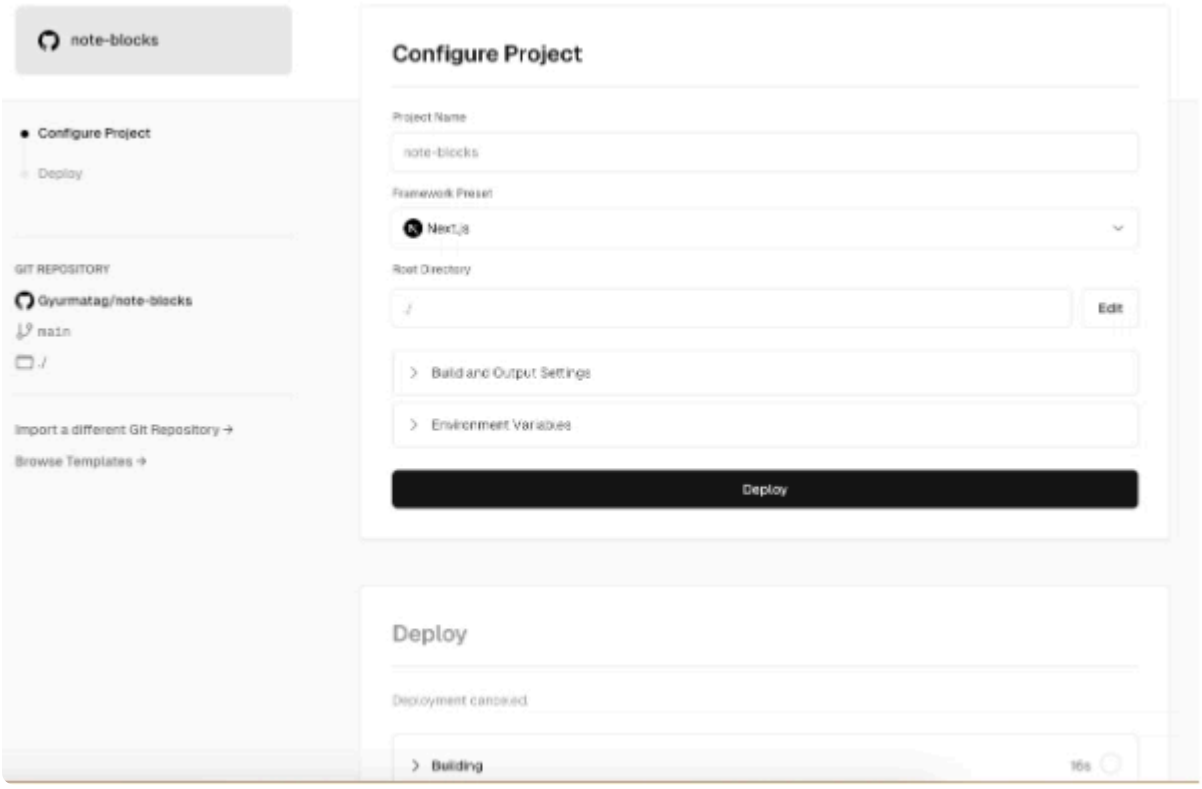
You can see that we have slightly modified the Tailwind layout and added our UI component. However, we have included the Link component from Next due to optimized routing.
Then go to our Note editor interface and make the design changes there as well. We will modify the Editor, so go into that component (`TextEditor.tsx`) and modify the HTML part as follows:

```
<div className="flex flex-col items-center min-h-screen px-4 w-full">
   <div className="w-full max-w-4xl mx-auto m-5">
     <BlockNoteView
         editor={editor}
         slashMenu={false}
     >
       <SuggestionMenuController
           triggerCharacter={'/'}
           getItems={async (query) =>
               filterSuggestionItems(getCustomSlashMenuItems(editor), query)
           }
       />
     </BlockNoteView>
     <div className="flex justify-end">
         <Button
             className="mt-4 mr-4 px-4 py-2"
             onClick={() => handleSubmitNote()}
         >
             Submit
         </Button>
     </div>
   </div>
 </div>
```

... we are done with the appearance of the application and we are ready with the core functions, which you can see in the title. You will find this project in [this Github repository](), whose code you can use and use as inspiration for your own projects!

Ship it!

Now let's deploy our system. For this, nothing else will be needed except to have our code on Github and to have a Vercel account. After we have uploaded our project to Github (or it is enough if we have a fork of the Github project I shared), we can now deploy the project. On the Vercel dashboard, go to the `Add New...` button, select Project, then select `note-blocks` from our Github repos. Then this configuration screen greets us:



Thank God we don't have to do anything because it recognizes everything automatically. We only need to set the Environment variables. We simply copy these from our .env file.
If we are satisfied with this, we also press the deploy button. After that, the application is activated and our app becomes publicly available.

Let's build in public

open for those of you who want to fork my Github project and do a PR with this function. This way you can also practice Next.js Server Actions and the operation of Server components!
What is also missing and will be necessary (especially in production) is the proper configuration of the rate limiter. To do this, all we need is to connect the @vercel/kv package to our project within the Vercel interface, the description of which can be found HERE. This will also be very easy to solve.

Let's connect

If you have any questions about the article or noticed any problems with it feel free to write them down in the comment section or reach ot to me on my X account.
Happy coding! :)

Top comments (7)

Munna kumar • Jun 24 '24

its not working

György Márk Varga Shiwaforce • Jun 25 '24

Can you please explain what is not working with your implementation? I will be happy to help.

Munna kumar • Jun 25 '24

import { BlockNoteEditor, PartialBlock } from "@blocknote/core";
import { useCreateBlockNote } from "@blocknote/react";
import { BlockNoteView } from "@blocknote/mantine"
import "@blocknote/mantine/style.css";

interface editorProps {
initialContent?: string;
editable?: boolean
}

const TextEditor: React.FC = ({ initialContent, editable }) => {

```
const editor = useCreateBlockNote({
    initialContent: initialContent ? (JSON.parse(initialContent) as Part
}) as BlockNoteEditor;

return (
    <BlockNoteView
        onChange={() => { console.log(JSON.stringify(editor.domElement.i
        editor={editor}
        editable={editable}
        theme={"light"}
```

}

export default TextEditor;
how to get the value where i use this component

György Márk Varga Shiwaforce • Jun 26 '24

Probably you can check the source code which I provided in the article. But generally speaking you can get the value from the editor like this, please examine the handleSubmitNote function!

```
const handleSubmitNote = async () => {
  const note = {
    document: editor.topLevelBlocks
  }
  await createNote(note)
}

return (
  <div className="flex flex-col items-center min-h-screen px-4 w-full
    <div className="relative w-full max-w-4xl mx-auto mb-4">
      <Button
        className="absolute right-0 top-0 mt-4 mr-4 px-4 py-2"
        onClick={() => handleSubmitNote()}
      >
        Submit
      </Button>
      <BlockNoteView
        className="w-full bg-white p-4 pt-16"
        editor={editor}
        theme={"light"}
      />
    </div>
  </div>
);
```

Please let me know if this resolved your question.

@m5553 I updated the article for the latest version of Blocknote and Neon database.
There were some syntax changes.

Vishnu Vardhan Gowd Vaka • Dec 12 '24

TypeError: Cannot read properties of undefined (reading 'SideMenu')

György Márk Varga Shiwaforce • Dec 23 '24

Hello @vishnuvardhanvaka !
Thank you for your comment. In which part do you have this type error? I cannot reproduce it in a freshly cloned repository.
Maybe try to delete the node_modules folder and run npm install again!
Let me know if you need any further assistance! :)

Tired of jumping between terminals, dashboards, and code?

Check out this demo showcasing how tools like Cursor can connect to Heroku through the MCP, letting you trigger actions like deployments, scaling, or provisioning—all without leaving your editor.

Learn More

Shiwaforce

More from Shiwaforce

#cloudflare #typescript #resend #reactemail

Make a real-time, offline first application with Instant

#webdev #nextjs #instantdb #realtime

Realtime Chart with Supabase and Tremor

#webdev #nextjs #supabase #tremor