Developer Docs (GitHub-Connected)

1. Table of contents
    1.1. Introduction
2. Walkthroughs
    2.1. Installing Slate
    2.2. Adding Event Handlers
    2.3. Defining Custom Elements
    2.4. Applying Custom Formatting
    2.5. Executing Commands
    2.6. Saving to a Database
    2.7. Using the Bundled Source
3. Concepts
    3.1. Interfaces
    3.2. Nodes
    3.3. Locations
    3.4. Transforms
    3.5. Operations
    3.6. Commands
    3.7. Editor
    3.8. Plugins
    3.9. Rendering
    3.10. Serializing
    3.11. Normalizing
    3.12. Using TypeScript
    3.13. Migrating
4. API
    4.1. Transforms API
    4.2. Node Types APIs
        4.2.1. Editor API

4.2.2. Element API

4.2.3. Node API

4.2.4. NodeEntry API

4.2.5. Text API

4.3. Location Types APIs

4.3.1. Location API

4.3.2. Path API

4.3.3. PathRef API

4.3.4. Point API

4.3.5. PointEntry API

4.3.6. PointRef API

4.3.7. Range API

4.3.8. RangeRef API

4.3.9. Span API

4.4. Operation API

5. Libraries

5.1. Slate React

5.2. Slate History

5.3. Slate Hyperscript

6. General

6.1. Resources

6.2. Contributing

6.3. Changelog

6.4. FAQ

1. Table of contents

1.1. Introduction

Code Examples

```
const greeting = 'Hello, world!'
console.log(greeting)
```

This is a GitHub-connected repository.

Slate is a *completely* customizable framework for building rich text editors.

Slate lets you build rich, intuitive editors like those in Medium, Dropbox Paper or Google Docs—which are becoming table stakes for applications on the web—without your codebase getting mired in complexity.

It can do this because all of its logic is implemented with a series of plugins, so you aren't ever constrained by what *is* or *isn't* in "core". You can think of it like a pluggable implementation of `contenteditable` built on top of React. It was inspired by libraries like Draft.js, Prosemirror and Quill.

```
 Slate is currently in beta. Its core API is usable now, but you might need
to pull request fixes for advanced use cases. Some of its APIs are not
"finalized" and will (breaking) change over time as we find better solutions.
```

Why?

Why create Slate? Well... *(Beware: this section has a few of* *my* *opinions!)*

Before creating Slate, I tried a lot of the other rich text libraries out there—**Draft.js**, **Prosemirror**, **Quill**, etc. What I found was that while getting simple examples to work was easy enough, once you started trying to build something like **Medium**, **Dropbox Paper** or **Google Docs**, you ran into deeper issues...

- **The editor's "schema" was hardcoded and hard to customize.** Things like bold and italic were supported out of the box, but what about comments, or embeds, or even more domain-specific needs?
- **Transforming the documents programmatically was very convoluted.** Writing as a user may have worked, but making programmatic changes, which is critical for building advanced behaviors, was needlessly complex.
- **Serializing to HTML, Markdown, etc. seemed like an afterthought.** Simple things like transforming a document to HTML or Markdown involved writing lots of boilerplate code, for what seemed like very common use cases.
- **Re-inventing the view layer seemed inefficient and limiting.** Most editors rolled their own views, instead of using existing technologies like React, so you had to learn a whole new system with new "gotchas".
- **Collaborative editing wasn't designed for in advance.** Often the editor's internal representation of data made it impossible to use for a realtime, collaborative editing use case without basically rewriting the editor.
- **The repositories were monolithic, not small and reusable.** The code bases for many of the editors often didn't expose the internal tooling that could have been re-used by developers, leading to having to reinvent the wheel.
- **Building complex, nested documents was impossible.** Many editors were designed around simplistic "flat" documents, making things like tables, embeds and captions difficult to reason about and sometimes impossible.

Of course not every editor exhibits all of these issues, but if you've tried using another editor you might have run into similar problems. To get around the limitations of their APIs and achieve the user experience you're after, you have to resort to very hacky things. And some experiences are just plain impossible to achieve.

If that sounds familiar, you might like Slate.

Which brings me to how Slate solves all of that...

Principles

Slate tries to solve the question of "[Why?]" with a few principles:

1. **First-class plugins.** The most important part of Slate is that plugins are first-class entities. That means you can *completely* customize the editing experience, to build complex editors like Medium's or Dropbox's, without having to fight against the library's assumptions.
2. **Schema-less core.** Slate's core logic assumes very little about the schema of the data you'll be editing, which means that there are no assumptions baked into the library that'll trip you up when you need to go beyond the most basic use cases.
3. **Nested document model.** The document model used for Slate is a nested, recursive tree, just like the DOM itself. This means that creating complex components like tables or nested block quotes are possible for advanced use cases. But it's also easy to keep it simple by only using a single level of hierarchy.
4. **Parallel to the DOM.** Slate's data model is based on the DOM—the document is a nested tree, it uses selections and ranges, and it exposes all the standard event handlers. This means that advanced behaviors like tables or nested block quotes are possible. Pretty much anything you can do in the DOM, you can do in Slate.
5. **Intuitive commands.** Slate documents are edited using "commands", that are designed to be high-level and extremely intuitive to write and read, so that custom functionality is as expressive as possible. This greatly increases your ability to reason about your code.
6. **Collaboration-ready data model.** The data model Slate uses—specifically how operations are applied to the document—has been designed to allow for collaborative editing to be layered on top, so you won't need to rethink everything if you decide to make your editor collaborative.
7. **Clear "core" boundaries.** With a plugin-first architecture, and a schema-less core, it becomes a lot clearer where the boundary is between "core" and "custom", which means that the core experience doesn't get bogged down in edge cases.

Demo

Check out the live demo of all of the examples!

Examples

To get a sense for how you might use Slate, check out a few of the examples:

- Plain text — showing the most basic case: a glorified `<textarea>`.
- Rich text — showing the features you'd expect from a basic editor.
- Markdown preview — showing how to add key handlers for Markdown-like shortcuts.
- Links — showing how to wrap text in inline nodes with associated data.
- Images — showing how to use void (text-less) nodes to add images.
- Hovering toolbar — showing how a contextual hovering menu can be implemented.
- Tables — showing how to nest blocks to render more advanced components.
- Paste HTML — showing how to use an HTML serializer to handle pasted HTML.
- Mentions — showing how to use inline void nodes for simple @-mentions.

Each example includes a **View Source** link to the code that implements it. And we have other examples too.

If you have an idea for an example that shows a common use case, pull request it!

Documentation

If you're using Slate for the first time, check out the Getting Started walkthroughs and the Concepts to familiarize yourself with Slate's architecture and mental models.

- Walkthroughs
- Concepts
- FAQ
- Resources

If even that's not enough, you can always read the source itself, which is heavily commented.

There are also translations of the documentation into other languages:

- 中文

If you're maintaining a translation, feel free to pull request it here!

Contributing!

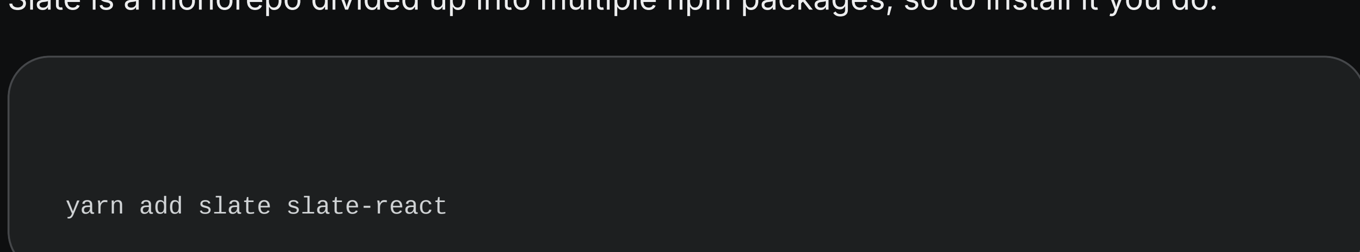
All contributions are super welcome! Check out the **Contributing instructions** for more info!

Slate is **MIT-licensed**.

2. Walkthroughs

2.1. Installing Slate

Slate is a monorepo divided up into multiple npm packages, so to install it you do:



```
yarn add slate slate-react
```

You'll also need to be sure to install Slate's peer dependencies:

```
yarn add react react-dom
```

Note, if you'd rather use a pre-bundled version of Slate, you can `yarn add slate` *and retrieve the bundled* `dist/slate.js` *file! Check out the* **Using the Bundled Source** *guide for more information.*

Once you've installed Slate, you'll need to import it.

```
// Import React dependencies.
import React, { useState } from 'react'
// Import the Slate editor factory.
import { createEditor } from 'slate'

// Import the Slate components and React plugin.
import { Slate, Editable, withReact } from 'slate-react'
```

Before we use those imports, let's start with an empty `<App>` component:

```
// Define our app...
const App = () => {
  return null
}
```

The next step is to create a new `Editor` object. We want the editor to be stable across renders, so we use the `useState` hook **without a setter**:

```
const App = () => {
  // Create a Slate editor object that won't change across renders.
  const [editor] = useState(() => withReact(createEditor()))
  return null
}
```

Of course we haven't rendered anything, so you won't see any changes.

> If you are using TypeScript, you will also need to extend the `Editor` with `ReactEditor` and add annotations as per the documentation on **TypeScript**. The example below also includes the custom types required for the rest of this example.

```
// TypeScript users only add this code
import { BaseEditor, Descendant } from 'slate'
import { ReactEditor } from 'slate-react'

type CustomElement = { type: 'paragraph'; children: CustomText[] }
type CustomText = { text: string }

declare module 'slate' {
  interface CustomTypes {
    Editor: BaseEditor & ReactEditor
    Element: CustomElement
    Text: CustomText
  }
}
```

Next up is to render a `<Slate>` context provider.

The provider component keeps track of your Slate editor, its plugins, its value, its selection, and any changes that occur. It **must** be rendered above any `<Editable>` components. But it can also provide the editor state to other components like toolbars, menus, etc. using the `useSlate` hook.

```
const initialValue = []

const App = () => {
  const [editor] = useState(() => withReact(createEditor()))
  // Render the Slate context.
  return <Slate editor={editor} value={initialValue} />
}
```

You can think of the `<Slate>` component as providing a context to every component underneath it.

```
Slate Provider's "value" prop is only used as initial state for
editor.children. If your code relies on replacing editor.children you should
```

```
do so by replacing it directly instead of relying on the "value" prop to do
this for you. See Slate PR 4540 for a more in-depth discussion.
```

This is a slightly different mental model than things like `<input>` or `<textarea>`, because richtext documents are more complex. You'll often want to include toolbars, or live previews, or other complex components next to your editable content.

By having a shared context, those other components can execute commands, query the editor's state, etc.

Okay, so the next step is to render the `<Editable>` component itself:

```
const initialValue = []

const App = () => {
  const [editor] = useState(() => withReact(createEditor()))
  return (
    // Add the editable component inside the context.
    <Slate editor={editor} value={initialValue}>
      <Editable />
    </Slate>
  )
}
```

The `<Editable>` component acts like `contenteditable`. Anywhere you render it will render an editable richtext document for the nearest editor context.

There's only one last step. So far we've been using an empty `[]` array as the initial value of the editor, so it has no content. Let's fix that by defining an initial value.

The value is just plain JSON. Here's one containing a single paragraph block with some text in it:

```
// Add the initial value.
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const [editor] = useState(() => withReact(createEditor()))

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable />
    </Slate>
  )
}
```

There you have it!

That's the most basic example of Slate. If you render that onto the page, you should see a paragraph with the text `A line of text in a paragraph.` And when you type, you should see the text change!

2.2. Adding Event Handlers

Okay, so you've got Slate installed and rendered on the page, and when you type in it, you can see the changes reflected. But you want to do more than just type a plaintext string.

What makes Slate great is how easy it is to customize. Just like other React components you're used to, Slate allows you to pass in handlers that are triggered on certain events.

Let's use the `onKeyDown` handler to change the editor's content when we press a key.

Here's our app from earlier:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable />
    </Slate>
  )
}
```

Now we add an `onKeyDown` handler:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        // Define a new handler which prints the key that was pressed.
        onKeyDown={event => {
          console.log(event.key)
        }}
      />
    </Slate>
  )
}
```

Cool, now when a key is pressed in the editor, its corresponding keycode is logged in the console.

Now we want to make it actually change the content. For the purposes of our example, let's implement turning all ampersand, `&`, keystrokes into the word `and` upon being typed.

Our `onKeyDown` handler might look like this:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        onKeyDown={event => {
          if (event.key === '&') {
            // Prevent the ampersand character from being inserted.
            event.preventDefault()
            // Execute the `insertText` method when the event occurs.
            editor.insertText('and')
          }
        }}
      />
    </Slate>
  )
}
```

With that added, try typing `&`, and you should see it suddenly become `and` instead!

This offers a sense of what can be done with Slate's event handlers. Each one will be called with the `event` object, and you can use your `editor` to perform commands in response. Simple!

2.3. Defining Custom Elements

In our previous example, we started with a paragraph, but we never actually told Slate anything about the `paragraph` block type. We just let it use its internal default renderer, which uses a plain old `<div>`.

But that's not all you can do. Slate lets you define any type of custom blocks you want, like block quotes, code blocks, list items, etc.

We'll show you how. Let's start with our app from earlier:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        onKeyDown={event => {
          if (event.key === '&') {
            event.preventDefault()
            editor.insertText('and')
          }
        }}
      />
    </Slate>
  )
}
```

Now let's add "code blocks" to our editor.

The problem is, code blocks won't just be rendered as a plain paragraph, they'll need to be rendered differently. To make that happen, we need to define a "renderer" for `code` element nodes.

Element renderers are just simple React components, like so:

```
// Define a React component renderer for our code blocks.
const CodeElement = props => {
  return (
    <pre {...props.attributes}>
      <code>{props.children}</code>
    </pre>
  )
}
```

Easy enough.

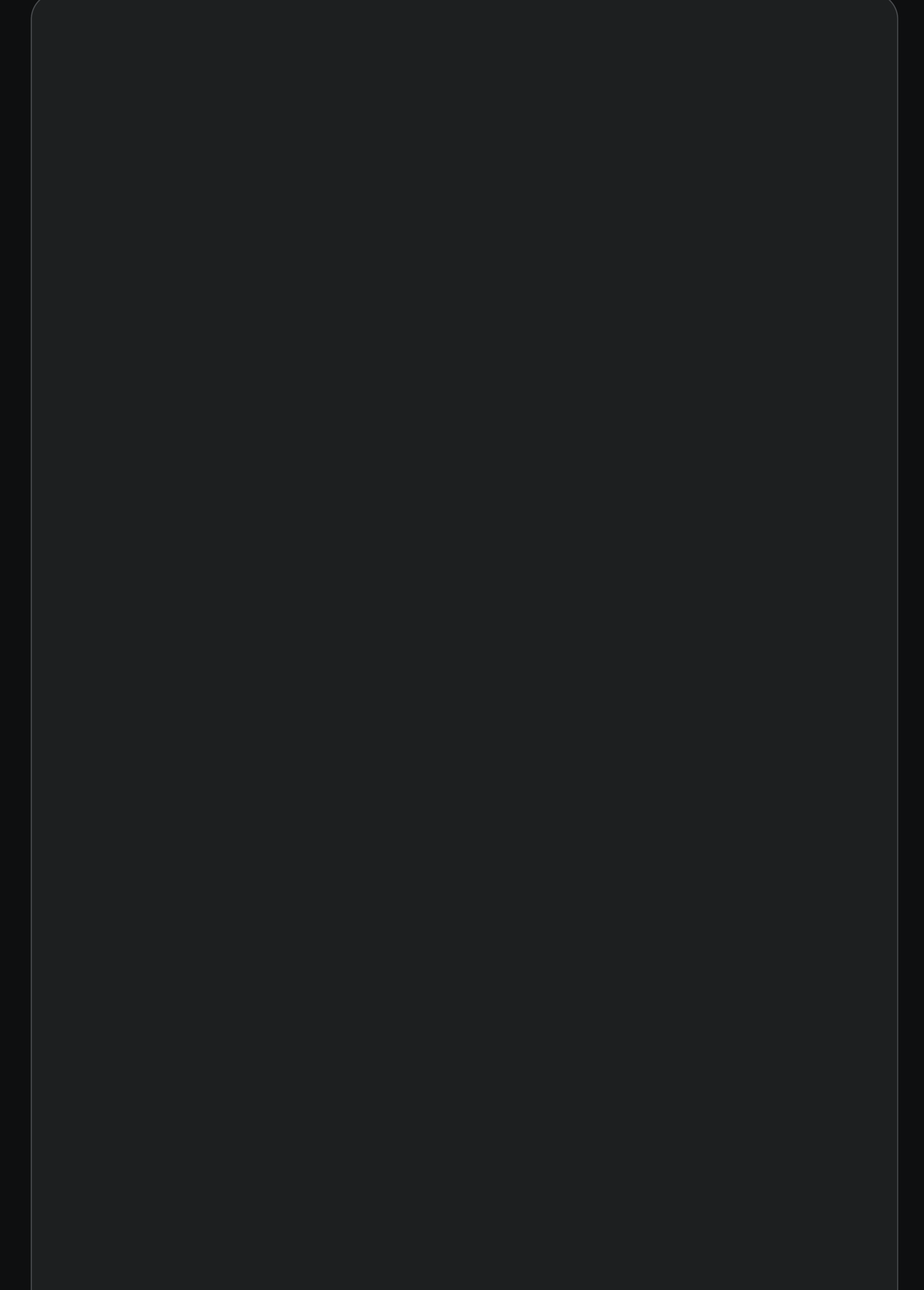
See the `props.attributes` reference? Slate passes attributes that should be rendered on the top-most element of your blocks, so that you don't have to build them up yourself. You **must** mix the attributes into your component.

And see that `props.children` reference? Slate will automatically render all of the children of a block for you, and then pass them to you just like any other React component would, via `props.children`. That way you don't have to muck around with rendering the proper text nodes or anything like that. You **must** render the children as the lowest leaf in your component.

And here's a component for the "default" elements:

```
const DefaultElement = props => {
  return <p {...props.attributes}>{props.children}</p>
}
```

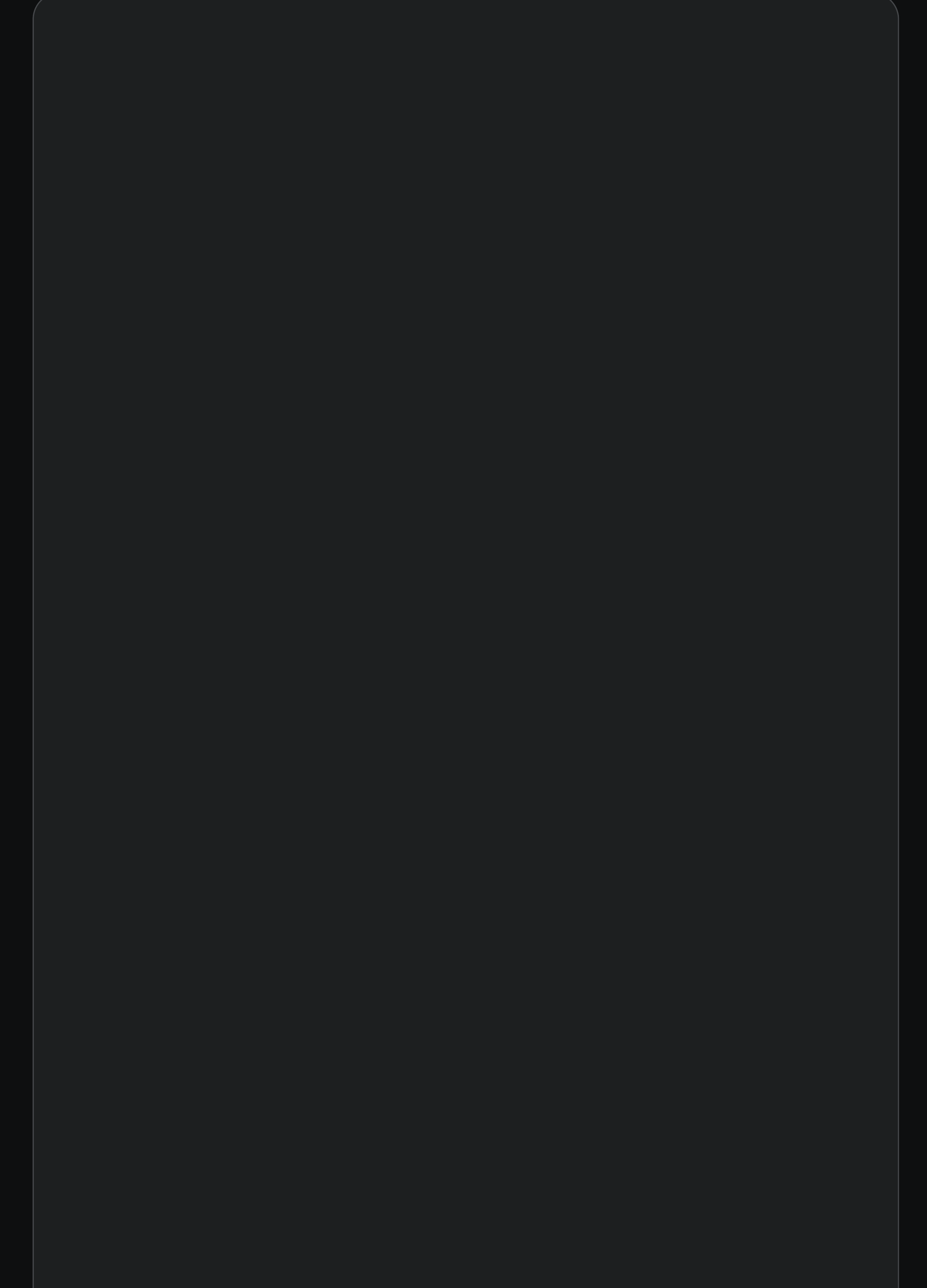
Now, let's add that renderer to our `Editor`:



```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  // Define a rendering function based on the element passed to `props`. We use
  // `useCallback` here to memoize the function for subsequent renders.
  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        // Pass in the `renderElement` function.
        renderElement={renderElement}
        onKeyDown={event => {
          if (event.key === '&') {
            event.preventDefault()
            editor.insertText('and')
          }
        }}
      />
    </Slate>
  )
}

const CodeElement = props => {
  return (
    <pre {...props.attributes}>
      <code>{props.children}</code>
    </pre>
  )
```

```
}

const DefaultElement = props => {
  return <p {...props.attributes}>{props.children}</p>
}
```

Okay, but now we'll need a way for the user to actually turn a block into a code block. So let's change our `onKeyDown` function to add a ``Ctrl-``` shortcut that does just that:



```
// Import the `Editor` and `Transforms` helpers from Slate.
import { Editor, Transforms } from 'slate'

const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        onKeyDown={event => {
          if (event.key === '`' && event.ctrlKey) {
            // Prevent the "`" from being inserted by default.
            event.preventDefault()
            // Otherwise, set the currently selected blocks type to "code".
            Transforms.setNodes(
              editor,
              { type: 'code' },
              { match: n => Editor.isBlock(editor, n) }
            )
          }
        }}
      />
    </Slate>
  )
}
```

```
const CodeElement = props => {
  return (
    <pre {...props.attributes}>
      <code>{props.children}</code>
    </pre>
  )
}

const DefaultElement = props => {
  return <p {...props.attributes}>{props.children}</p>
}
```

Now, if you press ``Ctrl-``` the block your cursor is in should turn into a code block! Magic!

But we forgot one thing. When you hit ``Ctrl-``` again, it should change the code block back into a paragraph. To do that, we'll need to add a bit of logic to change the type we set based on whether any of the currently selected blocks are already a code block:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        onKeyDown={event => {
          if (event.key === '`' && event.ctrlKey) {
            event.preventDefault()
            // Determine whether any of the currently selected blocks are code blocks.
            const [match] = Editor.nodes(editor, {
              match: n => n.type === 'code',
            })
            // Toggle the block type depending on whether there's already a match.
            Transforms.setNodes(
              editor,
              { type: match ? 'paragraph' : 'code' },
              { match: n => Editor.isBlock(editor, n) }
            )
          }
        }}
      />
    </Slate>
  )
}
```

And there you have it! If you press ``Ctrl-``` while inside a code block, it should turn back into a paragraph!

2.4. Applying Custom Formatting

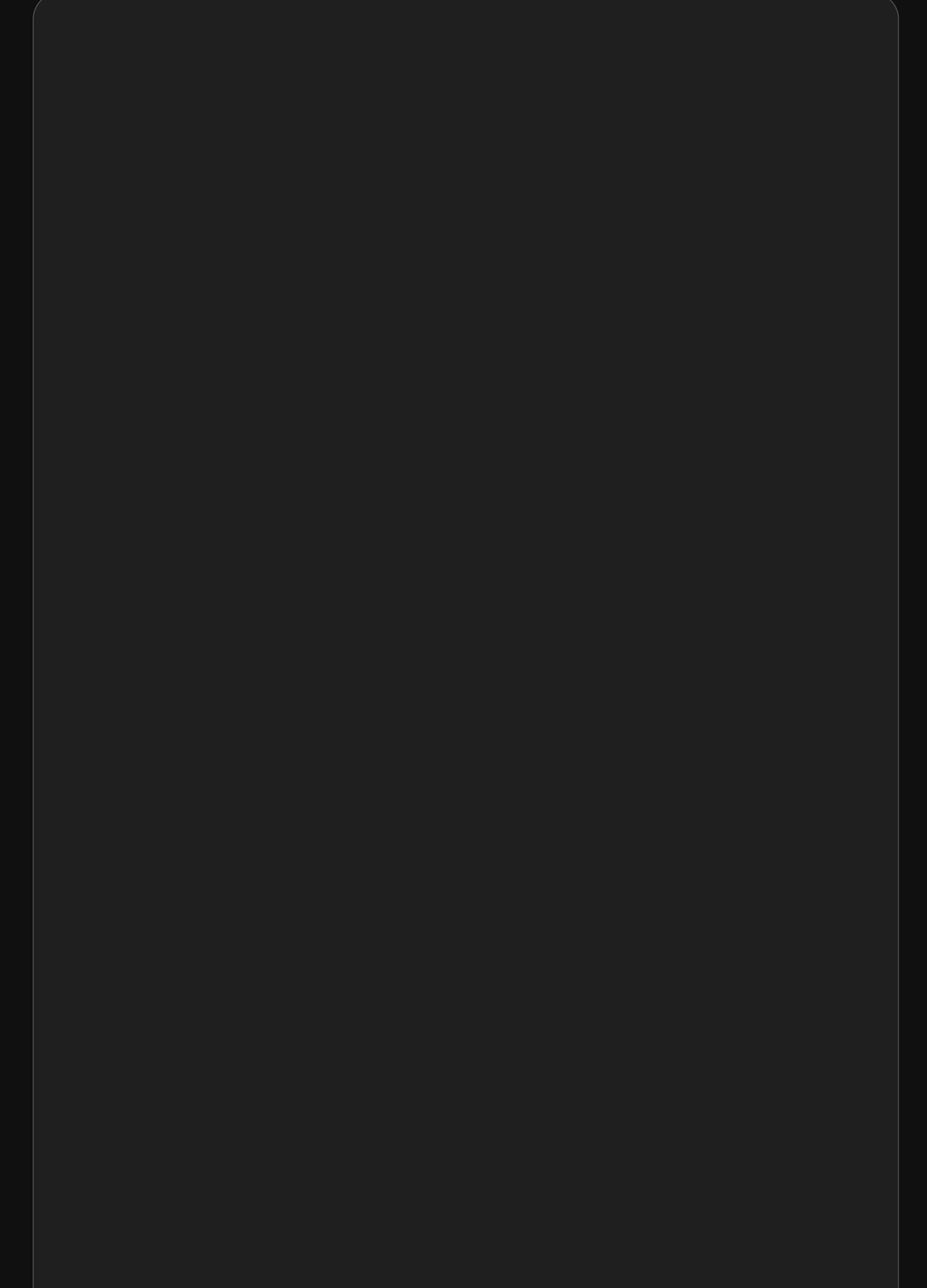
In the previous guide we learned how to create custom block types that render chunks of text inside different containers. But Slate allows for more than just "blocks".

In this guide, we'll show you how to add custom formatting options, like **bold**, *italic*, `code` or ~~strikethrough~~.

So we start with our app from earlier:

```
const renderElement = props => {
  switch (props.element.type) {
    case 'code':
      return <CodeElement {...props} />
    default:
      return <DefaultElement {...props} />
  }
}

const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        onKeyDown={event => {
          if (event.key === '`' && event.ctrlKey) {
            event.preventDefault()
            const { selection } = editor
            const [match] = Editor.nodes(editor, {
              match: n => n.type === 'code',
            })
            Transforms.setNodes(
              editor,
              { type: match ? 'paragraph' : 'code' },
              { match: n => Editor.isBlock(editor, n) }
            )
          }
        }}
      />
    </Slate>
  )
}
```

And now, we'll edit the `onKeyDown` handler to make it so that when you press `control-B`, it will add a `bold` format to the currently selected text:



```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        onKeyDown={event => {
          if (!event.ctrlKey) {
            return
          }

          switch (event.key) {
            // When "`" is pressed, keep our existing code block logic.
            case '`': {
              event.preventDefault()
              const [match] = Editor.nodes(editor, {
                match: n => n.type === 'code',
              })
              Transforms.setNodes(
                editor,
                { type: match ? 'paragraph' : 'code' },
                { match: n => Editor.isBlock(editor, n) }
              )
              break
            }
```

```
          // When "B" is pressed, bold the text in the selection.
          case 'b': {
            event.preventDefault()
            Transforms.setNodes(
              editor,
              { bold: true },
              // Apply it to text nodes, and split the text node up if the
              // selection is overlapping only part of it.
              { match: n => Text.isText(n), split: true }
            )
            break
          }
        }
      }}
    />
  </Slate>
  )
}
```

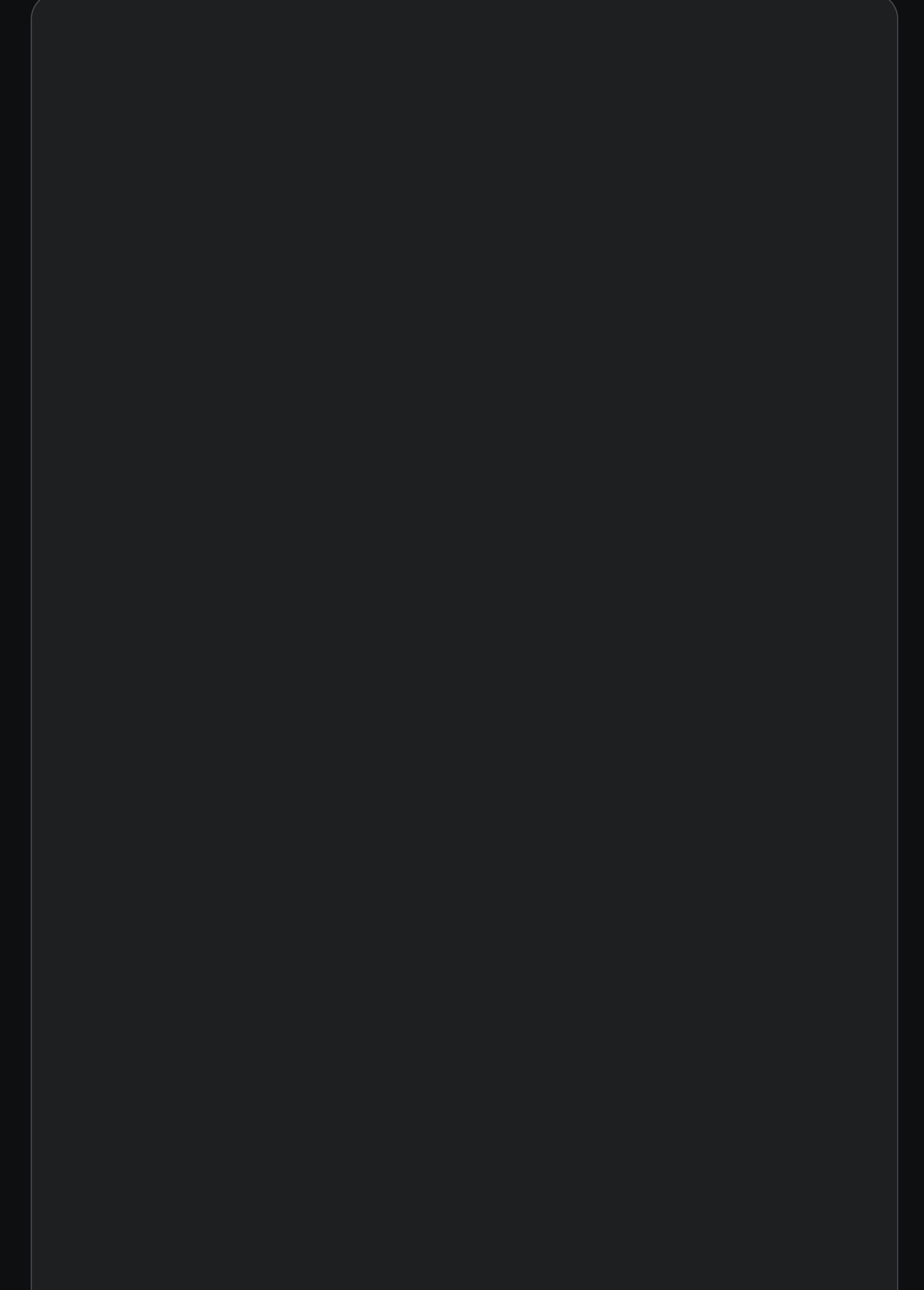
Okay, so we've got the hotkey handler setup... but! If you happen to now try selecting text and hitting `Ctrl-B`, you won't notice any change. That's because we haven't told Slate how to render a "bold" mark.

For every format you add, Slate will break up the text content into "leaves", and you need to tell Slate how to read it, just like for elements. So let's define a `Leaf` component:

```
// Define a React component to render leaves with bold text.
const Leaf = props => {
  return (
    <span
      {...props.attributes}
      style={{ fontWeight: props.leaf.bold ? 'bold' : 'normal' }}
    >
      {props.children}
    </span>
  )
}
```

Pretty familiar, right?

And now, let's tell Slate about that leaf. To do that, we'll pass in the `renderLeaf` prop to our editor.



```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  // Define a leaf rendering function that is memoized with `useCallback`.
  const renderLeaf = useCallback(props => {
    return <Leaf {...props} />
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        // Pass in the `renderLeaf` function.
        renderLeaf={renderLeaf}
        onKeyDown={event => {
          if (!event.ctrlKey) {
            return
          }

          switch (event.key) {
            case '`': {
              event.preventDefault()
              const [match] = Editor.nodes(editor, {
                match: n => n.type === 'code',
              })
              Transforms.setNodes(
                editor,
```

```
          { type: match ? null : 'code' },
          { match: n => Editor.isBlock(editor, n) }
        )
        break
      }

      case 'b': {
        event.preventDefault()
        Transforms.setNodes(
          editor,
          { bold: true },
          { match: n => Text.isText(n), split: true }
        )
        break
      }
    }
  }}
  />
</Slate>
)
}

const Leaf = props => {
  return (
    <span
      {...props.attributes}
      style={{ fontWeight: props.leaf.bold ? 'bold' : 'normal' }}
    >
      {props.children}
    </span>
  )
}
```

Now, if you try selecting a piece of text and hitting `Ctrl-B` you should see it turn bold! Magic!

2.5. Executing Commands

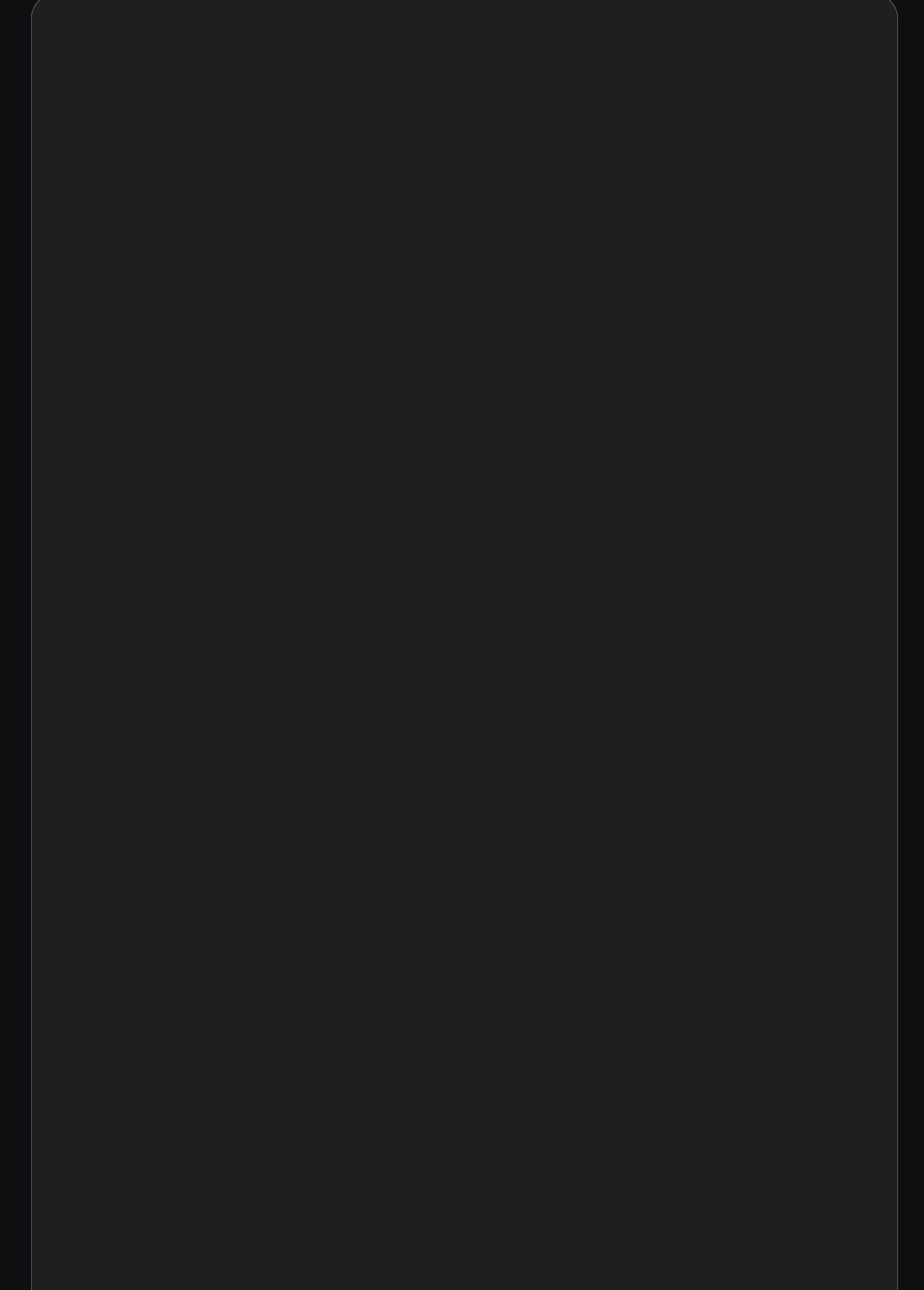
Up until now, everything we've learned has been about how to write one-off logic for your specific Slate editor. But one of the most powerful things about Slate is that it lets you model your specific rich text "domain" however you'd like, and write less one-off code.

In the previous guides we've written some useful code to handle formatting code blocks and bold marks. And we've hooked up the `onKeyDown` handler to invoke that code. But we've always done it using the built-in `Editor` helpers directly, instead of using "commands".

Slate lets you augment the built-in `editor` object to handle your own custom rich text commands. And you can even use pre-packaged "plugins" which add a given set of functionality.

Let's see how this works.

We'll start with our app from earlier:

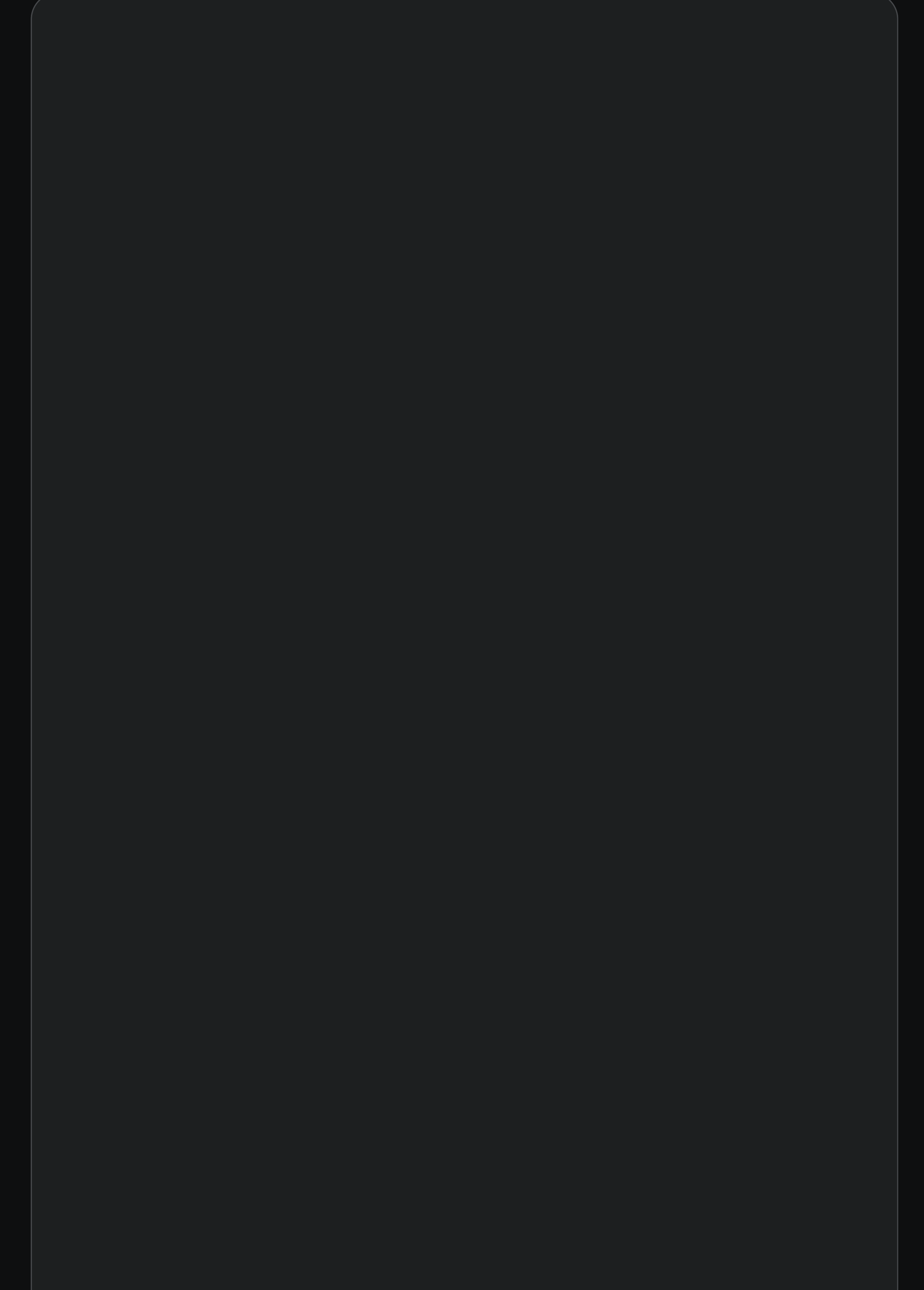


```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  const renderLeaf = useCallback(props => {
    return <Leaf {...props} />
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        renderLeaf={renderLeaf}
        onKeyDown={event => {
          if (!event.ctrlKey) {
            return
          }

          switch (event.key) {
            case '`': {
              event.preventDefault()
              const [match] = Editor.nodes(editor, {
                match: n => n.type === 'code',
              })
              Transforms.setNodes(
                editor,
                { type: match ? null : 'code' },
                { match: n => Editor.isBlock(editor, n) }
```

```
            )
            break
          }

          case 'b': {
            event.preventDefault()
            Transforms.setNodes(
              editor,
              { bold: true },
              { match: n => Text.isText(n), split: true }
            )
            break
          }
        }
      }}
    />
  </Slate>
  )
}
```

It has the concept of "code blocks" and "bold formatting". But these things are all defined in one-off cases inside the `onKeyDown` handler. If you wanted to reuse that logic elsewhere you'd need to extract it.

We can instead implement these domain-specific concepts by creating custom helper functions:

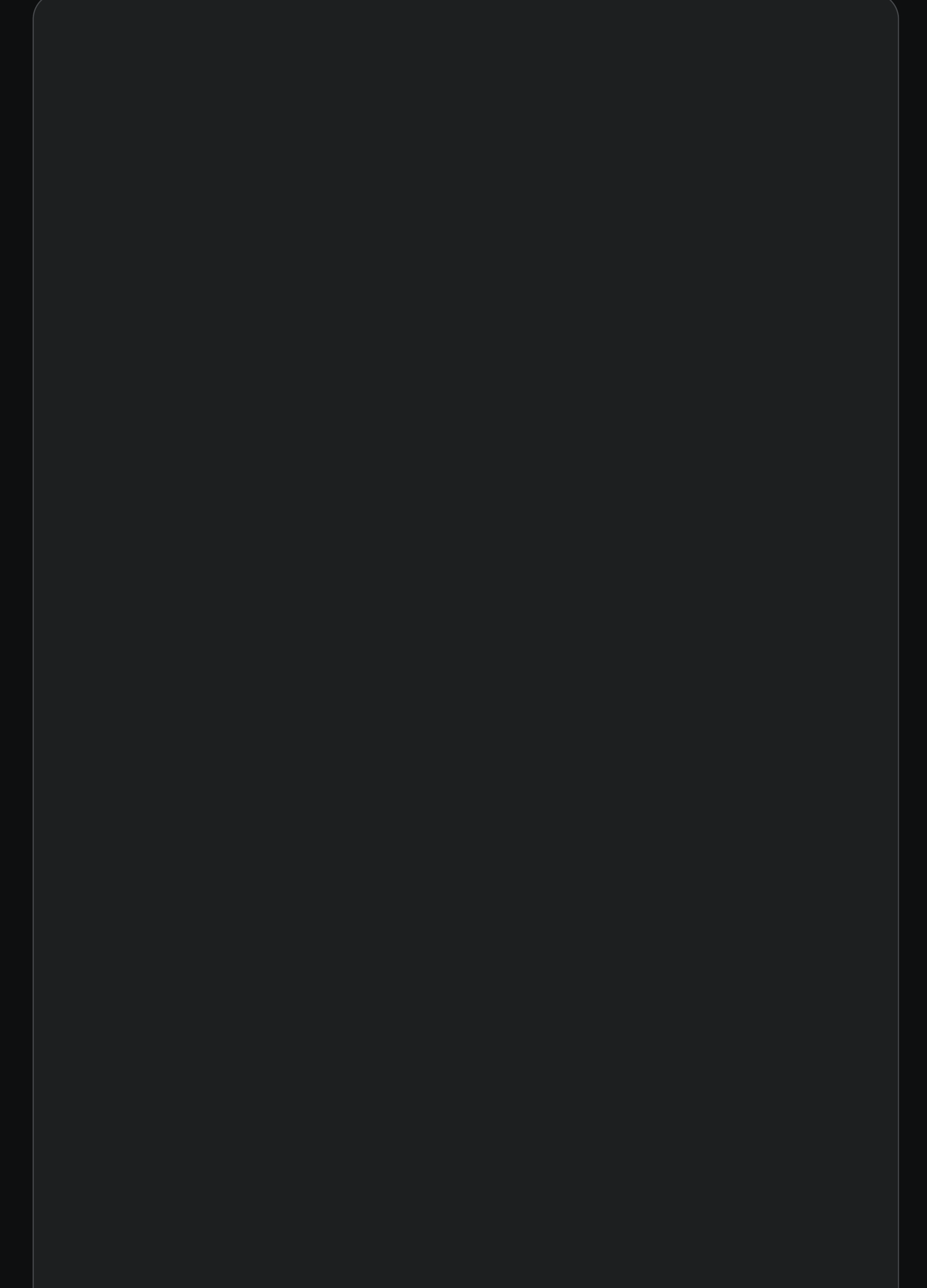


```
// Define our own custom set of helpers.
const CustomEditor = {
  isBoldMarkActive(editor) {
    const [match] = Editor.nodes(editor, {
      match: n => n.bold === true,
      universal: true,
    })

    return !!match
  },

  isCodeBlockActive(editor) {
    const [match] = Editor.nodes(editor, {
      match: n => n.type === 'code',
    })

    return !!match
  },

  toggleBoldMark(editor) {
    const isActive = CustomEditor.isBoldMarkActive(editor)
    Transforms.setNodes(
      editor,
      { bold: isActive ? null : true },
      { match: n => Text.isText(n), split: true }
    )
  },

  toggleCodeBlock(editor) {
    const isActive = CustomEditor.isCodeBlockActive(editor)
    Transforms.setNodes(
      editor,
      { type: isActive ? null : 'code' },
      { match: n => Editor.isBlock(editor, n) }
    )
  },
}

const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
```

```
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  const renderLeaf = useCallback(props => {
    return <Leaf {...props} />
  }, [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable
        renderElement={renderElement}
        renderLeaf={renderLeaf}
        onKeyDown={event => {
          if (!event.ctrlKey) {
            return
          }

          // Replace the `onKeyDown` logic with our new commands.
          switch (event.key) {
            case '`': {
              event.preventDefault()
              CustomEditor.toggleCodeBlock(editor)
              break
            }

            case 'b': {
              event.preventDefault()
              CustomEditor.toggleBoldMark(editor)
              break
            }
          }
        }}
      />
    </Slate>
```

```
)
```

Now our commands are clearly defined and you can invoke them from anywhere we have access to our `editor` object. For example, from hypothetical toolbar buttons:



```jsx
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  const renderElement = useCallback(props => {
    switch (props.element.type) {
      case 'code':
        return <CodeElement {...props} />
      default:
        return <DefaultElement {...props} />
    }
  }, [])

  const renderLeaf = useCallback(props => {
    return <Leaf {...props} />
  }, [])

  return (
    // Add a toolbar with buttons that call the same methods.
    <Slate editor={editor} value={initialValue}>
      <div>
        <button
          onMouseDown={event => {
            event.preventDefault()
            CustomEditor.toggleBoldMark(editor)
          }}
        >
          Bold
        </button>
        <button
          onMouseDown={event => {
            event.preventDefault()
            CustomEditor.toggleCodeBlock(editor)
          }}
        >
          Code Block
        </button>
```

```
        </div>
        <Editable
          editor={editor}
          renderElement={renderElement}
          renderLeaf={renderLeaf}
          onKeyDown={event => {
            if (!event.ctrlKey) {
              return
            }

            switch (event.key) {
              case '`': {
                event.preventDefault()
                CustomEditor.toggleCodeBlock(editor)
                break
              }

              case 'b': {
                event.preventDefault()
                CustomEditor.toggleBoldMark(editor)
                break
              }
            }
          }}
        />
      </Slate>
    )
  }
```

That's the benefit of extracting the logic.

And there you have it! We just added a ton of functionality to the editor with very little work. And we can keep all of our command logic tested and isolated in a single place, making the code easier to maintain.

2.6. Saving to a Database

Now that you've learned the basics of how to add functionality to the Slate editor, you might be wondering how you'd go about saving the content you've been editing, such that you can come back to your app later and have it load.

In this guide, we'll show you how to add logic to save your Slate content to a database for storage and retrieval later.

Let's start with a basic editor:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable />
    </Slate>
  )
}
```

That will render a basic Slate editor on your page, and when you type things will change. But if you refresh the page, everything will be reverted back to its original value—nothing saves!

What we need to do is save the changes you make somewhere. For this example, we'll just be using **Local Storage**, but it will give you an idea for where you'd need to add your own database hooks.

So, in our `onChange` handler, we need to save the `value` if anything besides the selection was changed:

```
const initialValue = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate
      editor={editor}
      value={initialValue}
      onChange={value => {
        const isAstChange = editor.operations.some(
          op => 'set_selection' !== op.type
        )
        if (isAstChange) {
          // Save the value to Local Storage.
          const content = JSON.stringify(value)
          localStorage.setItem('content', content)
        }
      }}
    >
      <Editable />
    </Slate>
  )
}
```

Now whenever you edit the page, if you look in Local Storage, you should see the `content` value changing.

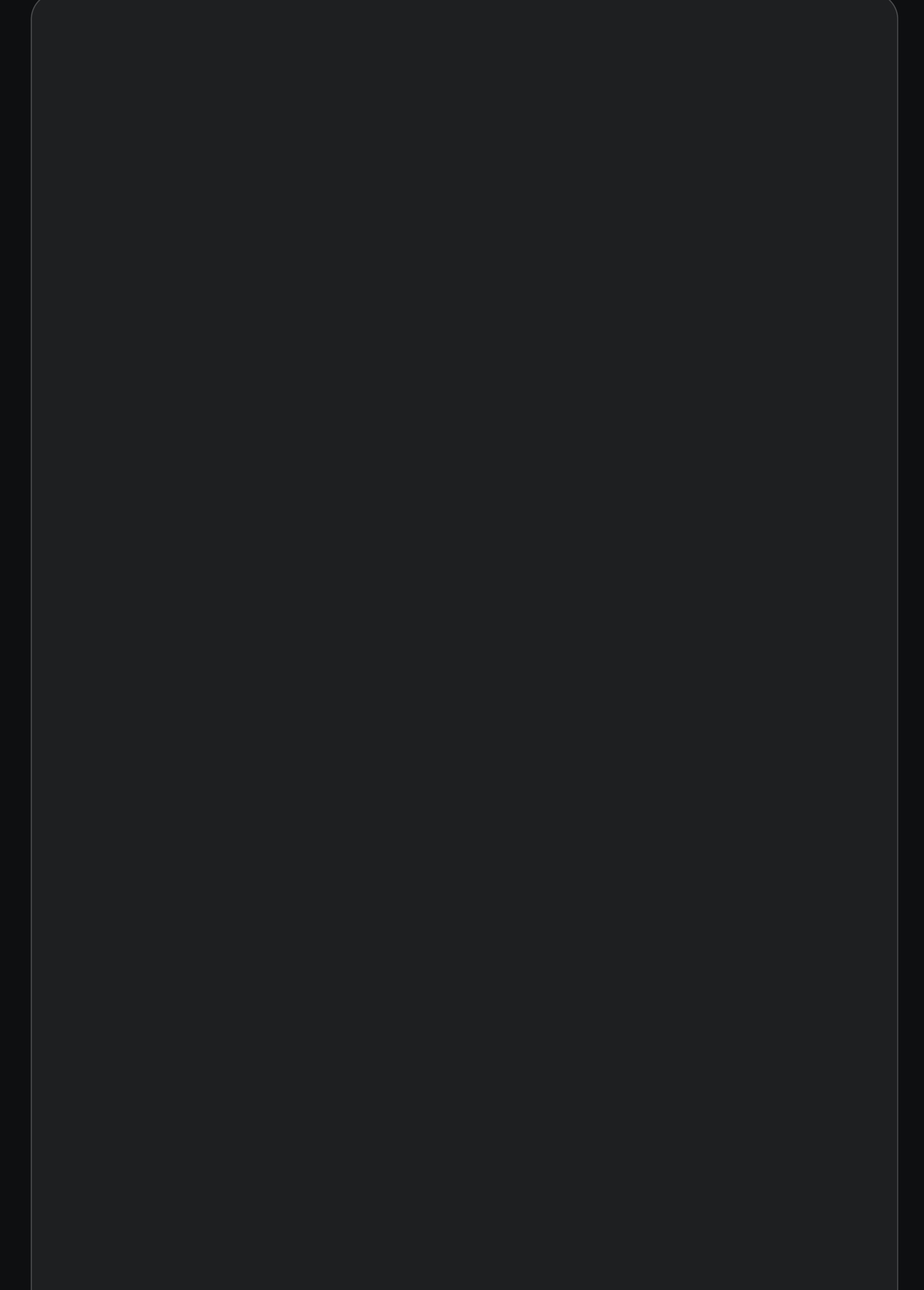
But... if you refresh the page, everything is still reset. That's because we need to make sure the initial value is pulled from that same Local Storage location, like so:

```
const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])
  // Update the initial content to be pulled from Local Storage if it exists.
  const initialValue = useMemo(
    JSON.parse(localStorage.getItem('content')) || [
      {
        type: 'paragraph',
        children: [{ text: 'A line of text in a paragraph.' }],
      },
    ],
    []
  )

  return (
    <Slate
      editor={editor}
      value={initialValue}
      onChange={value => {
        const isAstChange = editor.operations.some(
          op => 'set_selection' !== op.type
        )
        if (isAstChange) {
          // Save the value to Local Storage.
          const content = JSON.stringify(value)
          localStorage.setItem('content', content)
        }
      }}
    >
      <Editable />
    </Slate>
  )
}
```

Now you should be able to save changes across refreshes!

Success—you've got JSON in your database.

But what if you want something other than JSON? Well, you'd need to serialize your value differently. For example, if you want to save your content as plain text instead of JSON, we can write some logic to serialize and deserialize plain text values:



```
// Import the `Node` helper interface from Slate.
import { Node } from 'slate'

// Define a serializing function that takes a value and returns a string.
const serialize = value => {
  return (
    value
      // Return the string content of each paragraph in the value's children.
      .map(n => Node.string(n))
      // Join them all with line breaks denoting paragraphs.
      .join('\n')
  )
}

// Define a deserializing function that takes a string and returns a value.
const deserialize = string => {
  // Return a value array of children derived by splitting the string.
  return string.split('\n').map(line => {
    return {
      children: [{ text: line }],
    }
  })
}

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])
  // Use our deserializing function to read the data from Local Storage.
  const initialValue = useMemo(
    deserialize(localStorage.getItem('content')) || '',
    []
  )

  return (
    <Slate
      editor={editor}
      value={initialValue}
      onChange={value => {
        const isAstChange = editor.operations.some(
          op => 'set_selection' !== op.type
        )
        if (isAstChange) {
          // Serialize the value and save the string value to Local Storage.
          localStorage.setItem('content', serialize(value))
```

```
          }
        }}
      >
        <Editable />
      </Slate>
    )
  }
```

That works! Now you're working with plain text.

You can emulate this strategy for any format you like. You can serialize to HTML, to Markdown, or even to your own custom JSON format that is tailored to your use case.

```
 Note that even though you *can* serialize your content however you like,
there are tradeoffs. The serialization process has a cost itself, and certain
formats may be harder to work with than others. In general we recommend
writing your own format only if your use case has a specific need for it.
Otherwise, you're often better leaving the data in the format Slate uses.
```

If you want to update the editor's content in response to events from outside of slate, you need to change the children property directly. The simplest way is to replace the value of editor.children `editor.children = newValue` and trigger a re-rendering (e.g. by calling `editor.onChange()` in the example above). Alternatively, you can use slate's internal operations to transform the value, for example:

```
/**
* resetNodes resets the value of the editor.
* It should be noted that passing the `at` parameter may cause a "Cannot resolve a DOM
*/
resetNodes<T extends Node>(
  editor: Editor,
  options: {
    nodes?: Node | Node[],
    at?: Location
  } = {}
): void {
  const children = [...editor.children]

  children.forEach((node) => editor.apply({ type: 'remove_node', path: [0], node }))

  if (options.nodes) {
    const nodes = Node.isNode(options.nodes) ? [options.nodes] : options.nodes

    nodes.forEach((node, i) => editor.apply({ type: 'insert_node', path: [i], node: no
  }

  const point = options.at && Point.isPoint(options.at)
    ? options.at
    : Editor.end(editor, [])

  if (point) {
    Transforms.select(editor, point)
  }
}
```

2.7. Using the Bundled Source

For most folks, you'll want to install Slate via `npm`, in which case you can follow the regular Installing Slate guide.

But, if you'd rather install Slate by simply adding a `<script>` tag to your application, this guide will help you. To make the "bundled" use case simpler, each version of Slate ships with a bundled source file called `slate.js`.

To get a copy of `slate.js`, download the version of slate you want from npm:

```
npm install slate@latest
```

And then look in the `node_modules` folder for the bundled `slate.js` file:

```
node_modules/
  slate/
    dist/
      slate.js
      slate.min.js
```

A minified version called `slate.min.js` is also included for convenience.

Before you can add `slate.js` to your page, you need to bring your own copy of `react`, `react-dom` and `react-dom-server`, like so:

```
<script src="./vendor/react.js"></script>
<script src="./vendor/react-dom.js"></script>
<script src="./vendor/react-dom-server.js"></script>
```

This ensures that Slate isn't bundling its own copy of React, which would greatly increase the file size of your application.

Then you can add `slate.js` after those includes:

```
<script src="./vendor/slate.js"></script>
```

To make things easier, for quick prototyping, you can also use the **`unpkg.com`** delivery network that makes working with bundled npm modules easier. In that case, your includes would look like:

```
<script src="https://unpkg.com/react/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom.production.min.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom-server.browser.production.min.js">
<script src="https://unpkg.com/slate/dist/slate.js"></script>
<script src="https://unpkg.com/slate-react/dist/slate-react.js"></script>
```

That's it, you're ready to go!

3. Concepts

3.1. Interfaces

Slate works with pure JSON objects. All it requires is that those JSON objects conform to certain interfaces. For example, a text node in Slate must obey the `Text` interface:

```
interface Text {
  text: string
}
```

Which means it must have a `text` property with a string of content.

But **any** other custom properties are also allowed, and completely up to you. This lets you tailor your data to your specific domain and use case, adding whatever formatting logic you'd like, without Slate getting in the way.

This interface-based approach separates Slate from most other rich text editors which require you to work with their hand-rolled "model" classes and makes it much easier to reason about. It also means that it avoids startup time penalties related to "initializing" the data model.

Custom Properties

To take another example, the `Element` node interface in Slate is:

```
interface Element {
  children: Node[]
}
```

This is a very permissive interface. All it requires is that the `children` property gets defined containing the element's child nodes.

But you can extend elements (or any other interface) with your custom properties that are specific to your domain. For example, you might have "paragraph" and "link" elements:

```
const paragraph = {
  type: 'paragraph',
  children: [...],
}

const link = {
  type: 'link',
  url: 'https://example.com',
  children: [...]
}
```

The `type` and `url` properties are your custom API. Slate sees that they exist, but doesn't use them. However, when Slate renders a link element, you'll receive an object with the custom properties attached so that you can render it as:

```
<a href={element.url}>{element.children}</a>
```

When getting started with Slate, it's important to understand all of the interfaces it defines. There are a handful of interfaces that are discussed in each of the guides.

Helper Functions

In addition to the typing information, each interface in Slate also exposes a series of helper functions that make them easier to work with.

For example, when working with nodes:

```
import { Node } from 'slate'

// Get the string content of an element node.
const string = Node.string(element)

// Get the node at a specific path inside a root node.
const descendant = Node.get(value, path)
```

Or, when working with ranges:

```
import { Range } from 'slate'

// Get the start and end points of a range in order.
const [start, end] = Range.edges(range)

// Check if a range is collapsed to a single point.
if (Range.isCollapsed(range)) {
  // ...
}
```

There are many helper functions available for all common use cases when working with different interfaces. When getting started it helps to read through all of them so you can often simplify complex logic into just a handful of lines of code.

Custom Helpers

In addition to the built-in helper functions, you might want to define your custom helper functions and expose them on your custom namespaces.

For example, if your editor supports images, you might want a helper that determines if an element is an image element:

```
const isImageElement = element => {
  return element.type === 'image' && typeof element.url === 'string'
}
```

You can define these as one-off functions easily. But you might also bundle them up into namespaces, just like the core interfaces do, and use them instead.

For example:

```
import { Element } from 'slate'

// You can use `MyElement` everywhere to have access to your extensions.
export const MyElement = {
  ...Element,
  isImageElement,
  isParagraphElement,
  isQuoteElement,
}
```

This makes it easy to reuse domain-specific logic alongside the built-in Slate helpers.

3.2. Nodes

Couldn't convert your doc.

3.3. Locations

Locations are how you refer to specific places in the document when inserting, deleting, or doing anything else with a Slate editor. There are a few different kinds of location interfaces, each for different use cases.

`Path`

Paths are the lowest-level way to refer to a location. Each path is a simple array of numbers that refers to a node in the document tree by its indexes in each of its ancestor nodes down the tree:

```
type Path = number[]
```

For example, in this document:

```
const editor = {
  children: [
    {
      type: 'paragraph',
      children: [
        {
          text: 'A line of text!',
        },
      ],
    },
  ],
}
```

The leaf text node would have a path of: `[0, 0]`.

The Editor itself has a path of `[]`. For example, to select the whole contents of the editor, call

`Transforms.select(editor, [])`

`Point`

Points are slightly more specific than paths, and contain an `offset` into a specific text node. Their interface is:

```
interface Point {
  path: Path
  offset: number
}
```

For example, with that same document, if you wanted to refer to the very first place you could put your cursor it would be:

```
const start = {
  path: [0, 0],
  offset: 0,
}
```

Or, if you wanted to refer to the end of the sentence:

```
const end = {
  path: [0, 0],
  offset: 15,
}
```

It can be helpful to think of points as being "cursors" (or "carets") of a selection.

> 🤖 Points *always* refer to text nodes! Since they are the only ones with strings that can have cursors.

Range

Ranges are a way to refer not just to a single point in the document, but to a wider span of content between two points. (An example of a range is when you make a selection.) Their interface is:

```
interface Range {
  anchor: Point
  focus: Point
}
```

```
 The terms "anchor" and "focus" are borrowed from the DOM, see Anchor and Focus.
```

An anchor and focus are established by a user interaction. The anchor point isn't always *before* the focus point in the document. Just like in the DOM, the ordering of an anchor and selection point depend on whether the range is backwards or forwards.

Here's how Mozilla Developer Network explains it:

> A user may make a selection from left to right (in document order) or right to left (reverse of document order). The anchor is where the user began the selection and the focus is where the user ends the selection. If you make a selection with a desktop mouse, the anchor is placed where you pressed the mouse button and the focus is placed where you released the mouse button. Anchor and focus should not be confused with the start and end positions of a selection, since anchor can be placed before the focus or vice versa, depending on the direction you made your selection. — Selection, MDN

One important distinction is that the anchor and focus points of ranges **always reference the leaf-level text nodes** in a document and never reference elements. This behavior is different than the DOM, but it simplifies working with ranges as there are fewer edge cases for you to handle.

```
 For more info, check out the Range API reference.
```

Selection

Ranges are used in many places in Slate's API when you need to refer to a span of content between two points. One of the most common though is the user's current "selection".

The selection is a special range that is a property of the top-level `Editor`. For example, say someone has the whole sentence currently selected:

```
const editor = {
  selection: {
    anchor: { path: [0, 0], offset: 0 },
    focus: { path: [0, 0], offset: 15 },
  },
  children: [
    {
      type: 'paragraph',
      children: [
        {
          text: 'A line of text!',
        },
      ],
    },
  ],
}
```

> 🤖 The selection concept is also borrowed from the DOM, see **Selection, MDN**, although in a greatly-simplified form because Slate doesn't allow for multiple ranges inside a single selection, which makes things a lot easier to work with.

There isn't a special `Selection` interface. It's just an object that happens to respect the more general-purpose `Range` interface instead.

3.4. Transforms

Slate's data structure is immutable, so you can't modify or delete nodes directly. Instead, Slate comes with a collection of "transform" functions that let you change your editor's value.

Slate's transform functions are designed to be very flexible, to make it possible to represent all kinds of changes you might need to make to your editor. However, that flexibility can be hard to understand at first.

Typically, you'll apply a single operation to zero or more Nodes. For example, here's how you flatten the syntax tree,
by applying `unwrapNodes` to every parent of block Elements:

```
Transforms.unwrapNodes(editor, {
  at: [], // Path of Editor
  match: node =>
    !Editor.isEditor(node) &&
    node.children?.every(child => Editor.isBlock(editor, child)),
  mode: 'all', // also the Editor's children
})
```

Non-standard operations (or debugging/tracing which Nodes will be affected by a set of NodeOptions) may require using
`Editor.nodes` to create a JavaScript Iterator of NodeEntries and a for..of loop to act.
For example, to replace all image elements with their alt text:

```
const imageElmnts = Editor.nodes(editor, {
  at: [], // Path of Editor
  match: (node, path) => 'image' === node.type,
  // mode defaults to "all", so this also searches the Editor's children
})
for (const nodeEntry of imageElmnts) {
  const altText =
    nodeEntry[0].alt ||
    nodeEntry[0].title ||
    /\/([^/]+)$/.exec(nodeEntry[0].url)?.[1] ||
    '☹'
  Transforms.select(editor, nodeEntry[1])
  Editor.insertFragment(editor, [{ text: altText }])
}
```

Check out the **Transforms** reference for a full list of Slate's transforms.

Selection Transforms

Selection-related transforms are some of the simpler ones. For example, here's how you set the selection to a new range:

```
Transforms.select(editor, {
  anchor: { path: [0, 0], offset: 0 },
  focus: { path: [1, 0], offset: 2 },
})
```

But they can be more complex too.

For example, it's common to need to move a cursor forwards or backwards by varying distances—by character, by word, by line. Here's how you'd move the cursor backwards by three words:

```
Transforms.move(editor, {
  distance: 3,
  unit: 'word',
  reverse: true,
})
```

 For more info, check out the Selection Transforms API Reference

Text Transforms

Text transforms act on the text content of the editor. For example, here's how you'd insert a string of text as a specific point:

```
Transforms.insertText(editor, 'some words', {
  at: { path: [0, 0], offset: 3 },
})
```

Or you could delete all of the content in an entire range from the editor:

```
Transforms.delete(editor, {
  at: {
    anchor: { path: [0, 0], offset: 0 },
    focus: { path: [1, 0], offset: 2 },
  },
})
```

 For more info, check out the Text Transforms API Reference

Node Transforms

Node transforms act on the individual element and text nodes that make up the editor's value. For example you could insert a new text node at a specific path:

```
Transforms.insertNodes(
  editor,
  {
    text: 'A new string of text.',
  },
  {
    at: [0, 1],
  }
)
```

Or you could move nodes from one path to another:

```
Transforms.moveNodes(editor, {
  at: [0, 0],
  to: [0, 1],
})
```

 For more info, check out the [Node Transforms API Reference]

The `at` Option

Many transforms act on a specific location in the document. By default, they will use the user's current selection. But this can be overridden with the `at` option.

For example when inserting text, this would insert the string at the user's current cursor:

```
Transforms.insertText(editor, 'some words')
```

Whereas this would insert it at a specific point:

```
Transforms.insertText(editor, 'some words', {
  at: { path: [0, 0], offset: 3 },
})
```

The `at` option is very versatile, and can be used to implement more complex transforms very easily. Since it is a `Location` it can always be either a `Path`, `Point`, or `Range`. And each of those types of locations will result in slightly different transformations.

For example, in the case of inserting text, if you specify a `Range` location, the range will first be deleted, collapsing to a single point where your text is then inserted.

So to replace a range of text with a new string you can do:

```
Transforms.insertText(editor, 'some words', {
  at: {
    anchor: { path: [0, 0], offset: 0 },
    focus: { path: [0, 0], offset: 3 },
  },
})
```

Or, if you specify a `Path` location, it will expand to a range that covers the entire node at that path. Then, using the range-based behavior it will delete all of the content of the node, and replace it with your text.

So to replace the text of an entire node with a new string you can do:

```
Transforms.insertText(editor, 'some words', {
  at: [0, 0],
})
```

These location-based behaviors work for all the transforms that take an `at` option. It can be hard to wrap your head around at first, but it makes the API very powerful and capable of expressing many subtly different transforms.

The `match` Option

Many of the node-based transforms take a `match` function option, which restricts the transform to only apply to nodes for which the function returns `true`. When combined with `at`, `match` can also be very powerful.

For example, consider a basic transform that moves a node from one path to another:

```
Transforms.moveNodes(editor, {
  at: [2],
  to: [5],
})
```

Although it looks like it simply takes a path and moves it to another place. Under the hood two things are happening...

First, the `at` option is expanded to be a range representing all of the content inside the node at `[2]`. Which might look something like:

```
at: {
  anchor: { path: [2, 0], offset: 0 },
  focus: { path: [2, 2], offset: 19 }
}
```

Second, the `match` option is defaulted to a function that only matches the specific path, in this case `[2]`:

```
match: (node, path) => Path.equals(path, [2])
```

Then Slate iterates over the range and moves any nodes that pass the matcher function to the new location. In this case, since `match` is defaulted to only match the exact `[2]` path, that node is moved.

But what if you wanted to move the children of the node at `[2]` instead?

You might consider looping over the node's children and moving them one at a time, but this gets very complex to manage because as you move the nodes the paths you're referring to become outdated.

Instead, you can take advantage of the `at` and `match` options to match all of the children:

```
Transforms.moveNodes(editor, {
  // This will again be expanded to a range of the entire node at `[2]`.
  at: [2],
  // Matches nodes with a longer path, which are the children.
  match: (node, path) => path.length === 2,
  to: [5],
})
```

Here we're using the same `at` path (which is expanded to a range), but instead of letting it match just that path by default, we're supplying our own `match` function which happens to match only the children of the node.

Using `match` can make representing complex logic a lot simpler.

For example, consider wanting to add a bold mark to any text nodes that aren't already italic:

```
Transform.setNodes(
  editor,
  { bold: true },
  {
    // This path references the editor, and is expanded to a range that
    // will encompass all the content of the editor.
    at: [],
    // This only matches text nodes that are not already italic.
    match: (node, path) => Text.isText(node) && node.italic !== true,
  }
)
```

When performing transforms, if you're ever looping over nodes and transforming them one at a time, consider seeing if `match` can solve your use case, and offload the complexity of managing loops to Slate instead.

The `match` function can examine the children of a node, in `node.children`, or use `Node.parent` to examine its parent.

Transforms and Normalization

Sequences of Transforms may need to be wrapped in **Editor.withoutNormalizing** if the node tree should *not* be normalized between Transforms.

See **Normalization - Implications for Other Code**;

3.5. Operations

Operations are the granular, low-level actions that occur while invoking transforms. A single transform could result in many low-level operations being applied to the editor.

Slate's core defines all of the possible operations that can occur on a richtext document. For example:

```
editor.apply({
  type: 'insert_text',
  path: [0, 0],
  offset: 15,
  text: 'A new string of text to be inserted.',
})

editor.apply({
  type: 'remove_node',
  path: [0, 0],
  node: {
    text: 'A line of text!',
  },
})

editor.apply({
  type: 'set_selection',
  properties: {
    anchor: { path: [0, 0], offset: 0 },
  },
  newProperties: {
    anchor: { path: [0, 0], offset: 15 },
  },
})
```

Under the covers Slate converts complex transforms into the low-level operations and applies them to the editor automatically. So you rarely have to think about operations unless you're implementing collaborative editing.

🤖 Slate's editing behaviors being defined as operations is what makes things like collaborative editing possible, because each change is easily define-able, apply-able, compose-able and even undo-able!

3.6. Commands

While editing richtext content, your users will be doing things like inserting text, deleting text, splitting paragraphs, adding formatting, etc. Under the cover these edits are expressed using transforms and operations. But at a high level we talk about them as "commands".

Commands are the high-level actions that represent a specific intent of the user. They are represented as helper functions on the `Editor` interface. A handful of helpers are included in core for common richtext behaviors, but you are encouraged to write your own that model your specific domain.

For example, here are some of the built-in commands:

```
Editor.insertText(editor, 'A new string of text to be inserted.')

Editor.deleteBackward(editor, { unit: 'word' })

Editor.insertBreak(editor)
```

But you can (and will!) also define your own custom commands that model your domain. For example, you might want to define a `formatQuote` command, or an `insertImage` command, or a `toggleBold` command depending on what types of content you allow.

Commands always describe an action to be taken as if the **user** was performing the action. For that reason, they never need to define a location to perform the command, because they always act on the user's current selection.

> 🤖 The concept of commands is loosely based on the DOM's built-in `execCommand` APIs. However Slate defines its own simpler (and extendable!) version of the API, because the DOM's version is too opinionated and inconsistent.

Under the covers, Slate takes care of converting each command into a set of low-level "operations" that are applied to produce a new value. This is what makes collaborative

editing implementations possible. But you don't have to worry about that, because it happens automatically.

Custom Commands

When defining custom commands, you can create your own namespace:

```
const MyEditor = {
  ...Editor,

  insertParagraph(editor) {
    // ...
  },
}
```

When writing your own commands, you'll often make use of the `Transforms` helpers that ship with Slate.

Transforms

Transforms are a specific set of helpers that allow you to perform a wide variety of specific changes to the document, for example:

```
// Set a "bold" format on all of the text nodes in a range.
Transforms.setNodes(
  editor,
  { bold: true },
  {
    at: range,
    match: node => Text.isText(node),
    split: true,
  }
)

// Wrap the lowest block at a point in the document in a quote block.
Transforms.wrapNodes(
  editor,
  { type: 'quote', children: [] },
  {
    at: point,
    match: node => Editor.isBlock(editor, node),
    mode: 'lowest',
  }
)

// Insert new text to replace the text in a node at a specific path.
Transforms.insertText(editor, 'A new string of text.', { at: path })

// ...there are many more transforms!
```

The transform helpers are designed to be composed together. So you might use a handful of them for each command.

3.7. Editor

All of the behaviors, content and state of a Slate editor is rolled up into a single, top-level `Editor` object. It has an interface of:

```
interface Editor {
  // Current editor state
  children: Node[]
  selection: Range | null
  operations: Operation[]
  marks: Omit<Text, 'text'> | null
  // Schema-specific node behaviors.
  isInline: (element: Element) => boolean
  isVoid: (element: Element) => boolean
  normalizeNode: (entry: NodeEntry) => void
  onChange: () => void
  // Overrideable core actions.
  addMark: (key: string, value: any) => void
  apply: (operation: Operation) => void
  deleteBackward: (unit: 'character' | 'word' | 'line' | 'block') => void
  deleteForward: (unit: 'character' | 'word' | 'line' | 'block') => void
  deleteFragment: () => void
  insertBreak: () => void
  insertSoftBreak: () => void
  insertFragment: (fragment: Node[]) => void
  insertNode: (node: Node) => void
  insertText: (text: string) => void
  removeMark: (key: string) => void
}
```

It is slightly more complex than the others, because it contains all of the top-level functions that define your custom, domain-specific behaviors.

The `children` property contains the document tree of nodes that make up the editor's content.

The `selection` property contains the user's current selection, if any.
Don't set it directly; use **Transforms.select**

The `operations` property contains all of the operations that have been applied since the last "change" was flushed. (Since Slate batches operations up into ticks of the event loop.)

The `marks` property stores formatting to be applied when the editor inserts text. If `marks` is `null`, the formatting will be taken from the current selection. Don't set it directly; use `Editor.addMark` and `Editor.removeMark`.

Overriding Behaviors

In previous guides we've already hinted at this, but you can override any of the behaviors of an editor by overriding its function properties.

For example, if you want to define link elements that are inline nodes:

```
const { isInline } = editor

editor.isInline = element => {
  return element.type === 'link' ? true : isInline(element)
}
```

Or maybe you want to override the `insertText` behavior to "linkify" URLs:

```
const { insertText } = editor

editor.insertText = text => {
  if (isUrl(text)) {
    // ...
    return
  }

  insertText(text)
}
```

Or you can even define custom "normalizations" that take place to ensure that links obey certain constraints:

```
const { normalizeNode } = editor

editor.normalizeNode = entry => {
  const [node, path] = entry

  if (Element.isElement(node) && node.type === 'link') {
    // ...
    return
  }

  normalizeNode(entry)
}
```

Whenever you override behaviors, be sure to call the existing functions as a fallback mechanism for the default behavior. Unless you really do want to completely remove the default behaviors (which is rarely a good idea).

> For more info, check out the [Editor Instance Methods to Override API Reference]

Helper Functions

The `Editor` interface, like all Slate interfaces, exposes helper functions that are useful when implementing certain behaviors. There are many, many editor-related helpers. For example:

```
// Get the start point of a specific node at path.
const point = Editor.start(editor, [0, 0])

// Get the fragment (a slice of the document) at a range.
const fragment = Editor.fragment(editor, range)
```

There are also many iterator-based helpers, for example:

```
// Iterate over every node in a range.
for (const [node, path] of Editor.nodes(editor, { at: range })) {
  // ...
}

// Iterate over every point in every text node in the current selection.
for (const point of Editor.positions(editor)) {
  // ...
}
```

 For more info, check out the **Editor Static Methods API Reference**

3.8. Plugins

You've already seen how the behaviors of Slate editors can be overridden. These overrides can also be packaged up into "plugins" to be reused, tested and shared. This is one of the most powerful aspects of Slate's architecture.

A plugin is simply a function that takes an `Editor` object and returns it after it has augmented it in some way.

For example, a plugin that marks image nodes as "void":

```
const withImages = editor => {
  const { isVoid } = editor

  editor.isVoid = element => {
    return element.type === 'image' ? true : isVoid(element)
  }

  return editor
}
```

And then to use the plugin, simply:

```
import { createEditor } from 'slate'

const editor = withImages(createEditor())
```

This plugin composition model makes Slate extremely easy to extend!

Helper Functions

In addition to the plugin functions, you might want to expose helper functions that are used alongside your plugins. For example:

```
import { Editor, Element } from 'slate'

const MyEditor = {
  ...Editor,
  insertImage(editor, url) {
    const element = { type: 'image', url, children: [{ text: '' }] }
    Transforms.insertNodes(editor, element)
  },
}

const MyElement = {
  ...Element,
  isImageElement(value) {
    return Element.isElement(element) && element.type === 'image'
  },
}
```

Then you can use `MyEditor` and `MyElement` everywhere and have access to all your helpers in one place.

3.9. Rendering

One of the best parts of Slate is that it's built with React, so it fits right into your existing application. It doesn't re-invent its own view layer that you have to learn. It tries to keep everything as React-y as possible.

To that end, Slate gives you control over the rendering behavior of your custom nodes and properties in your richtext domain.

You can define these behaviors by passing "render props" to the top-level `<Editable>` component.

For example if you wanted to render custom element components, you'd pass in the `renderElement` prop:

```
import { createEditor } from 'slate'
import { Slate, Editable, withReact } from 'slate-react'

const MyEditor = () => {
  const editor = useMemo(() => withReact(createEditor()), [])
  const renderElement = useCallback(({ attributes, children, element }) => {
    switch (element.type) {
      case 'quote':
        return <blockquote {...attributes}>{children}</blockquote>
      case 'link':
        return (
          <a {...attributes} href={element.url}>
            {children}
          </a>
        )
      default:
        return <p {...attributes}>{children}</p>
    }
  }, [])

  return (
    <Slate editor={editor}>
      <Editable renderElement={renderElement} />
    </Slate>
  )
}
```

 Be sure to mix in `props.attributes` and render `props.children` in your custom components! The attributes must be added to the top-level DOM element inside the component, as they are required for Slate's DOM helper functions to work. And the children are the actual text content of your document which Slate manages for you automatically.

You don't have to use simple HTML elements, you can use your own custom React components too:

```
const renderElement = useCallback(props => {
  switch (props.element.type) {
    case 'quote':
      return <QuoteElement {...props} />
    case 'link':
      return <LinkElement {...props} />
    default:
      return <DefaultElement {...props} />
  }
}, [])
```

Leaves

When text-level formatting is rendered, the characters are grouped into "leaves" of text that each contain the same formatting applied to them.

To customize the rendering of each leaf, you use a custom `renderLeaf` prop:

```
const renderLeaf = useCallback(({ attributes, children, leaf }) => {
  return (
    <span
      {...attributes}
      style={{
        fontWeight: leaf.bold ? 'bold' : 'normal',
        fontStyle: leaf.italic ? 'italic' : 'normal',
      }}
    >
      {children}
    </span>
  )
}, [])
```

Notice though how we've handled it slightly differently than `renderElement`. Since text formatting tends to be fairly simple, we've opted to ditch the `switch` statement and just toggle on/off a few styles instead. (But there's nothing preventing you from using custom components if you'd like!)

One disadvantage of text-level formatting is that you cannot guarantee that any given format is "contiguous"—meaning that it stays as a single leaf. This limitation with respect to leaves is similar to the DOM, where this is invalid:

```
<em>t<strong>e</em>x</strong>t
```

Because the elements in the above example do not properly close themselves they are invalid. Instead, you would write the above HTML as follows:

```
<em>t</em><strong><em>e</em>x</strong>t
```

If you happened to add another overlapping section of `<strike>` to that text, you might have to rearrange the closing tags yet again. Rendering leaves in Slate is similar—you can't guarantee that even though a word has one type of formatting applied to it that that leaf will be contiguous, because it depends on how it overlaps with other formatting.

Of course, this leaf stuff sounds pretty complex. But, you do not have to think about it much, as long as you use text-level formatting and element-level formatting for their intended purposes:

- Text properties are for **non-contiguous**, character-level formatting.
- Element properties are for **contiguous**, semantic elements in the document.

Decorations

Decorations are another type of text-level formatting. They are similar to regular old custom properties, except each one applies to a `Range` of the document instead of being associated with a given text node.

However, decorations are computed at **render-time** based on the content itself. This is helpful for dynamic formatting like syntax highlighting or search keywords, where changes to the content (or some external data) has the potential to change the formatting.

Decorations are different from Marks in that they are not stored on editor state.

Toolbars, Menus, Overlays, and more!

In addition to controlling the rendering of nodes inside Slate, you can also retrieve the current editor context from inside other components using the `useSlate` hook.

That way other components can execute commands, query the editor state, or anything else.

A common use case for this is rendering a toolbar with formatting buttons that are highlighted based on the current selection:

```
const MyEditor = () => {
  const editor = useMemo(() => withReact(createEditor()), [])
  return (
    <Slate editor={editor}>
      <Toolbar />
      <Editable />
    </Slate>
  )
}

const Toolbar = () => {
  const editor = useSlate()
  return (
    <div>
      <Button active={isBoldActive(editor)}>B</Button>
      <Button active={isItalicActive(editor)}>I</Button>
    </div>
  )
}
```

Because the `<Toolbar>` uses the `useSlate` hook to retrieve the context, it will re-render whenever the editor changes, so that the active state of the buttons stays in sync.

3.10. Serializing

Slate's data model has been built with serialization in mind. Specifically, its text nodes are defined in a way that makes them easier to read at a glance, but also easy to serialize to common formats like HTML and Markdown.

And, because Slate uses plain JSON for its data, you can write serialization logic very easily.

Plaintext

For example, taking the value of an editor and returning plaintext:

```
import { Node } from 'slate'

const serialize = nodes => {
  return nodes.map(n => Node.string(n)).join('\n')
}
```

Here we're taking the children nodes of an `Editor` as a `nodes` argument, and returning a plaintext representation where each top-level node is separated by a single `\n` new line character.

For an input of:

```
const nodes = [
  {
    type: 'paragraph',
    children: [{ text: 'An opening paragraph...' }],
  },
  {
    type: 'quote',
    children: [{ text: 'A wise quote.' }],
  },
  {
    type: 'paragraph',
    children: [{ text: 'A closing paragraph!' }],
  },
]
```

You'd end up with:

```
An opening paragraph...
A wise quote.
A closing paragraph!
```

Notice how the quote block isn't distinguishable in any way, that's because we're talking about plaintext. But you can serialize the data to anything you want—it's just JSON after all.

HTML

For example, here's a similar `serialize` function for HTML:

```
import escapeHtml from 'escape-html'
import { Text } from 'slate'

const serialize = node => {
  if (Text.isText(node)) {
    let string = escapeHtml(node.text)
    if (node.bold) {
      string = `<strong>${string}</strong>`
    }
    return string
  }

  const children = node.children.map(n => serialize(n)).join('')

  switch (node.type) {
    case 'quote':
      return `<blockquote><p>${children}</p></blockquote>`
    case 'paragraph':
      return `<p>${children}</p>`
    case 'link':
      return `<a href="${escapeHtml(node.url)}">${children}</a>`
    default:
      return children
  }
}
```

This one is a bit more aware than the plaintext serializer above. It's actually *recursive* so that it can keep iterating deeper through a node's children until it gets to the leaf text nodes. And for each node it receives, it converts it to an HTML string.

It also takes a single node as input instead of an array, so if you passed in an editor like:

```
const editor = {
  children: [
    {
      type: 'paragraph',
      children: [
        { text: 'An opening paragraph with a ' },
        {
          type: 'link',
          url: 'https://example.com',
          children: [{ text: 'link' }],
        },
        { text: ' in it.' },
      ],
    },
    {
      type: 'quote',
      children: [{ text: 'A wise quote.' }],
    },
    {
      type: 'paragraph',
      children: [{ text: 'A closing paragraph!' }],
    },
  ],
  // `Editor` objects also have other properties that are omitted here...
}
```

You'd receive back (line breaks added for legibility):

```
<p>An opening paragraph with a <a href="https://example.com">link</a> in it.</p>
<blockquote><p>A wise quote.</p></blockquote>
<p>A closing paragraph!</p>
```

It's really that easy!

Deserializing

Another common use case in Slate is doing the reverse—deserializing. This is when you have some arbitrary input and want to convert it into a Slate-compatible JSON structure. For example, when someone pastes HTML into your editor and you want to ensure it gets parsed with the proper formatting for your editor.

Slate has a built-in helper for this: the `slate-hyperscript` package.

The most common way to use `slate-hyperscript` is for writing JSX documents, for example when writing tests. You might use it like so:

```
/** @jsx jsx */
import { jsx } from 'slate-hyperscript'

const input = (
  <fragment>
    <element type="paragraph">A line of text.</element>
  </fragment>
)
```

And the JSX feature of your compiler (Babel, TypeScript, etc.) would turn that `input` variable into:

```
const input = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text.' }],
  },
]
```

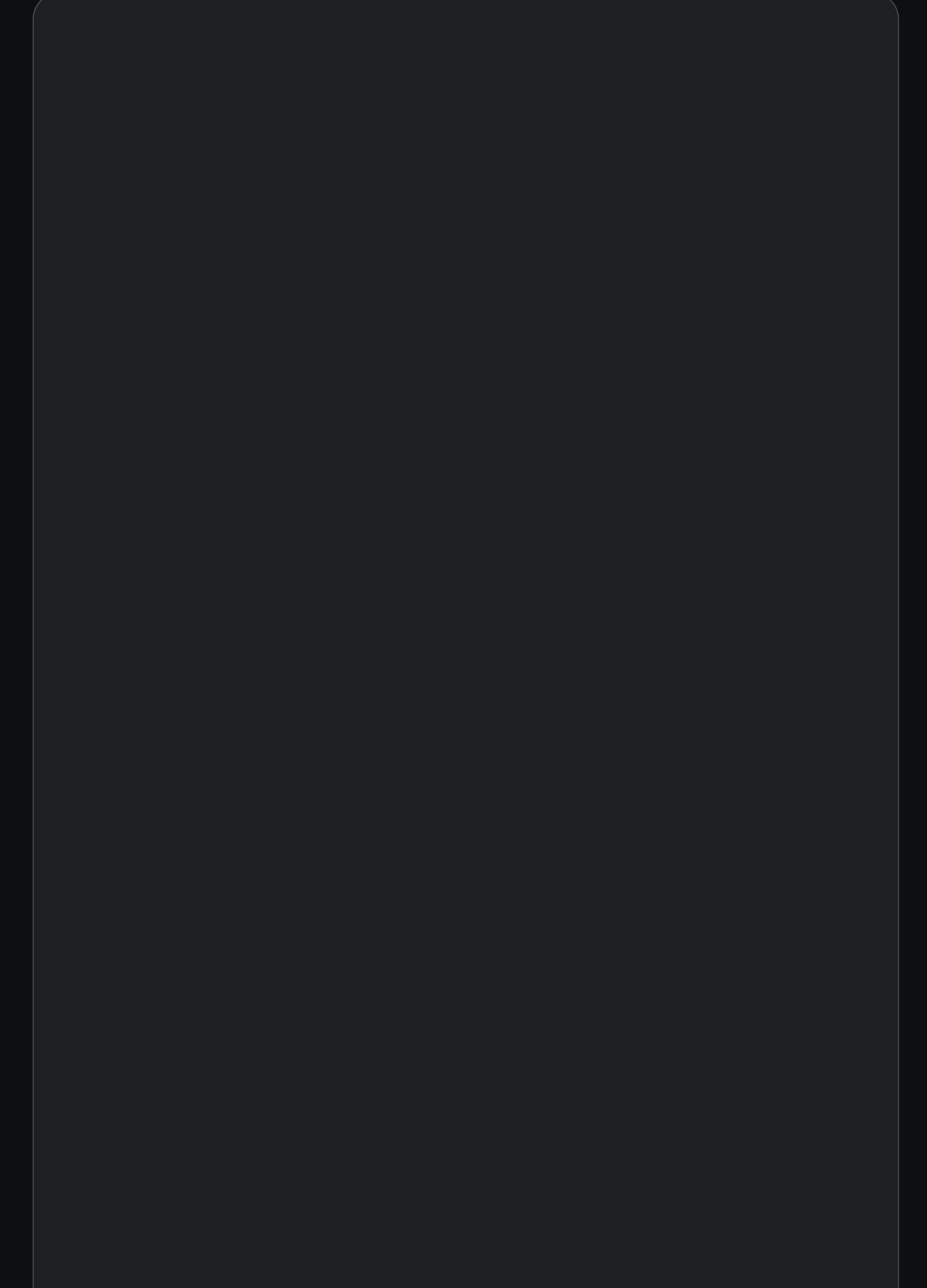
This is great for test cases, or places where you want to be able to write a lot of Slate objects in a very readable form.

However! This doesn't help with deserialization.

But `slate-hyperscript` isn't only for JSX. It's just a way to build *trees of Slate content*. Which happens to be exactly what you want to do when you're deserializing something like

HTML.

For example, here's a `deserialize` function for HTML:



```
import { jsx } from 'slate-hyperscript'

const deserialize = (el, markAttributes = {}) => {
  if (el.nodeType === Node.TEXT_NODE) {
    return jsx('text', markAttributes, el.textContent)
  } else if (el.nodeType !== Node.ELEMENT_NODE) {
    return null
  }

  const nodeAttributes = { ...markAttributes }

  // define attibutes for text nodes
  switch (el.nodeName) {
    case 'strong':
      nodeAttributes.bold = true
  }

  const children = Array.from(el.childNodes)
    .map(node => deserialize(el, nodeAttributes))
    .flat()

  if (children.length === 0) {
    children.push(jsx('text', nodeAttributes, ''))
  }

  switch (el.nodeName) {
    case 'BODY':
      return jsx('fragment', {}, children)
    case 'BR':
      return '\n'
    case 'BLOCKQUOTE':
      return jsx('element', { type: 'quote' }, children)
    case 'P':
      return jsx('element', { type: 'paragraph' }, children)
    case 'A':
      return jsx(
        'element',
        { type: 'link', url: el.getAttribute('href') },
        children
      )
    default:
      return children
```

```
    }
```

It takes in an `el` HTML element object and returns a Slate fragment. So if you have an HTML string, you can parse and deserialize it like so:

```
const html = '...'
const document = new DOMParser().parseFromString(html, 'text/html')
deserialize(document.body)
```

With this input:

```
<p>An opening paragraph with a <a href="https://example.com">link</a> in it.</p>
<blockquote><p>A wise quote.</p></blockquote>
<p>A closing paragraph!</p>
```

You'd end up with this output:

```
const fragment = [
  {
    type: 'paragraph',
    children: [
      { text: 'An opening paragraph with a ' },
      {
        type: 'link',
        url: 'https://example.com',
        children: [{ text: 'link' }],
      },
      { text: ' in it.' },
    ],
  },
  {
    type: 'quote',
    children: [
      {
        type: 'paragraph',
        children: [{ text: 'A wise quote.' }],
      },
    ],
  },
  {
    type: 'paragraph',
    children: [{ text: 'A closing paragraph!' }],
  },
]
```

And just like the serializing function, you can extend it to fit your exact domain model's needs.

3.11. Normalizing

Slate editors can edit complex, nested data structures. And for the most part this is great. But in certain cases inconsistencies in the data structure can be introduced—most often when allowing a user to paste arbitrary richtext content.

"Normalizing" is how you can ensure that your editor's content is always of a certain shape. It's similar to "validating", except instead of just determining whether the content is valid or invalid, its job is to fix the content to make it valid again.

Built-in Constraints

Slate editors come with a few built-in constraints out of the box. These constraints are there to make working with content *much* more predictable than standard `contenteditable`. All of the built-in logic in Slate depends on these constraints, so unfortunately you cannot omit them. They are...

1. **All `Element` nodes must contain at least one `Text` descendant** — even Void Elements. If an element node does not contain any children, an empty text node will be added as its only child. This constraint exists to ensure that the selection's anchor and focus points (which rely on referencing text nodes) can always be placed inside any node. With this, empty elements (or void elements) wouldn't be selectable.
2. **Two adjacent texts with the same custom properties will be merged.** If two adjacent text nodes have the same formatting, they're merged into a single text node with a combined text string of the two. This exists to prevent the text nodes from only ever expanding in count in the document, since both adding and removing formatting results in splitting text nodes.
3. **Block nodes can only contain other blocks, or inline and text nodes.** For example, a `paragraph` block cannot have another `paragraph` block element *and* a `link` inline element as children at the same time. The type of children allowed is determined by the first child, and any other non-conforming children are removed. This ensures that common richtext behaviors like "splitting a block in two" function consistently.
4. **Inline nodes cannot be the first or last child of a parent block, nor can it be next to another inline node in the children array.** If this is the case, an empty text node will be added to correct this to be in compliance with the constraint.

5. **The top-level editor node can only contain block nodes.** If any of the top-level children are inline or text nodes they will be removed. This ensures that there are always block nodes in the editor so that behaviors like "splitting a block in two" work as expected.
6. **Nodes must be JSON-serializable.** For example, avoid using `undefined` in your data model. This ensures that operations are also JSON-serializable, a property which is assumed by collaboration libraries.
7. **Property values must not be `null`.** Instead, you should use an optional property, e.g. `foo?: string` instead of `foo: string | null`. This limitation is due to `null` being used in operations to represent the absence of a property.

These default constraints are all mandated because they make working with Slate documents *much* more predictable.

```
 Although these constraints are the best we've come up with now, we're
always looking for ways to have Slate's built-in constraints be less
constraining if possible—as long as it keeps standard behaviors easy to
reason about. If you come up with a way to reduce or remove a built-in
constraint with a different approach, we're all ears!
```

Adding Constraints

The built-in constraints are fairly generic. But you can also add your own constraints on top of the built-in ones that are specific to your domain.

To do this, you extend the `normalizeNode` function on the editor. The `normalizeNode` function gets called every time an operation is applied that inserts or updates a node (or its descendants), giving you the opportunity to ensure that the changes didn't leave it in an invalid state, and correcting the node if so.

For example here's a plugin that ensures `paragraph` blocks only have text or inline elements as children:

```
import { Transforms, Element, Node } from 'slate'

const withParagraphs = editor => {
  const { normalizeNode } = editor

  editor.normalizeNode = entry => {
    const [node, path] = entry

    // If the element is a paragraph, ensure its children are valid.
    if (Element.isElement(node) && node.type === 'paragraph') {
      for (const [child, childPath] of Node.children(editor, path)) {
        if (Element.isElement(child) && !editor.isInline(child)) {
          Transforms.unwrapNodes(editor, { at: childPath })
          return
        }
      }
    }

    // Fall back to the original `normalizeNode` to enforce other constraints.
    normalizeNode(entry)
  }

  return editor
}
```

This example is fairly simple. Whenever `normalizeNode` gets called on a paragraph element, it loops through each of its children ensuring that none of them are block elements. And if one is a block element, it gets unwrapped, so that the block is removed and its children take its place. The node is "fixed".

But what if the child has nested blocks?

Multi-pass Normalizing

One thing to understand about `normalizeNode` constraints is that they are **multi-pass**.

If you check the example above again, you'll notice the `return` statement:

```
if (Element.isElement(child) && !editor.isInline(child)) {
  Transforms.unwrapNodes(editor, { at: childPath })
  return
}
```

You might at first think this is odd, because with the `return` there, the original `normalizeNodes` will never be called, and the built-in constraints won't get a chance to run their own normalizations.

But, there's a slight "trick" to normalizing.

When you do call `Editor.unwrapNodes`, you're actually changing the content of the node that is currently being normalized. So even though you're ending the current normalization pass, by making a change to the node you're kicking off a *new* normalization pass. This results in a sort of *recursive* normalizing.

This multi-pass characteristic makes it *much* easier to write normalizations, because you only ever have to worry about fixing a single issue at once, and not fixing *every* possible issue that could be putting a node in an invalid state.

To see how this works in practice, let's start with this invalid document:

```
<editor>
  <paragraph a>
    <paragraph b>
      <paragraph c>word</paragraph>
    </paragraph>
  </paragraph>
</editor>
```

The editor starts by running `normalizeNode` on `<paragraph c>`. And it is valid, because it contains only text nodes as children.

But then, it moves up the tree, and runs `normalizeNode` on `<paragraph b>`. This paragraph is invalid, since it contains a block element (`<paragraph c>`). So that child

block gets unwrapped, resulting in a new document of:

```
<editor>
  <paragraph a>
    <paragraph b>word</paragraph>
  </paragraph>
</editor>
```

And in performing that fix, the top-level `<paragraph a>` changed. It gets normalized, and it is invalid, so `<paragraph b>` gets unwrapped, resulting in:

```
<editor>
  <paragraph a>word</paragraph>
</editor>
```

And now when `normalizeNode` runs, no changes are made, so the document is valid!

> For the most part you don't need to think about these internals. You can just know that anytime `normalizeNode` is called and you spot an invalid state, you can fix that single invalid state and trust that `normalizeNode` will be called again until the node becomes valid.

Empty Children Early Constraint Execution

One special normalization executes before all other normalizations and this can be important to keep in mind when writing your normalizers.

Before any of the other normalizations can execute, Slate iterates through all `Element` nodes and makes sure they have at least one child. If it does not, an empty `Text` descendant is created.

This can trip you up when you have custom handling when an `Element` has no children. For example, if a table element has no rows, you may wish to remove the table; however,

this will never happen because a `Text` node would automatically be created before that normalization could run.

Incorrect Fixes

One pitfall to avoid is creating an infinite normalization loop. This can happen if you check for a specific invalid structure, but then **don't** actually fix that structure with the change you make to the node. This results in an infinite loop because the node continues to be flagged as invalid, but it is never fixed properly.

For example, consider a normalization that ensured `link` elements have a valid `url` property:

```
// WARNING: this is an example of incorrect behavior!
const withLinks = editor => {
  const { normalizeNode } = editor

  editor.normalizeNode = entry => {
    const [node, path] = entry

    if (
      Element.isElement(node) &&
      node.type === 'link' &&
      typeof node.url !== 'string'
    ) {
      // ERROR: null is not a valid value for a url
      Transforms.setNodes(editor, { url: null }, { at: path })
      return
    }

    normalizeNode(entry)
  }

  return editor
}
```

This fix is incorrectly written. It wants to ensure that all `link` elements have a `url` property string. But to fix invalid links it sets the `url` to `null`, which is still not a string!

In this case you'd either want to unwrap the link, removing it entirely. *Or* expand your validation to accept an "empty" `url == null` as well.

Implications for Other Code

Sequences of Transforms may need to be wrapped in **Editor.withoutNormalizing** if the node tree should *not* be normalized between Transforms.

This is frequently the case when you `unwrapNodes` followed by `wrapNodes`.

For example, you might write a function to change the type of a block as follows:

```
const LIST_TYPES = ['numbered-list', 'bulleted-list']

function changeBlockType(editor, type) {
  Editor.withoutNormalizing(editor, () => {
    const isActive = isBlockActive(editor, type)
    const isList = LIST_TYPES.includes(type)

    Transforms.unwrapNodes(editor, {
      match: n =>
        LIST_TYPES.includes(
          !Editor.isEditor(n) && SlateElement.isElement(n) && n.type
        ),
      split: true,
    })
    const newProperties = {
      type: isActive ? 'paragraph' : isList ? 'list-item' : type,
    }
    Transforms.setNodes(editor, newProperties)

    if (!isActive && isList) {
      const block = { type: type, children: [] }
      Transforms.wrapNodes(editor, block)
    }
  })
}
```

3.12. Using TypeScript

Slate supports typing of one Slate document model (ie. one set of custom `Editor`, `Element` and `Text` types). If you need to support more than one document model, see the section Multiple Document Models.

Warning: You must define `CustomTypes`, annotate `useState`, and annotate the editor's initial state when using TypeScript or Slate will display typing errors.

Migrating from 0.47.x

When migrating from 0.47.x, read the guide below first. Also keep in mind these common migration issues:

- When referring to `node.type`, you may see the error `Property 'type' does not exist on type 'Node'`. To fix this, you need to add code like `Element.isElement(node) && node.type === 'paragraph'`. This is necessary because a `Node` can be an `Element` or `Text` and `Text` does not have a `type` property.
- Be careful when you define the CustomType for `Editor`. Make sure to define the CustomType for `Editor` as `BaseEditor & ...`. It should not be `Editor & ...`

Defining `Editor`, `Element` and `Text` Types

To define a custom `Element` or `Text` type, extend the `CustomTypes` interface in the `slate` module like this.

```
// This example is for an Editor with `ReactEditor` and `HistoryEditor`
import { BaseEditor } from 'slate'
import { ReactEditor } from 'slate-react'
import { HistoryEditor } from 'slate-history'

type CustomElement = { type: 'paragraph'; children: CustomText[] }
type CustomText = { text: string; bold?: true }

declare module 'slate' {
  interface CustomTypes {
    Editor: BaseEditor & ReactEditor & HistoryEditor
    Element: CustomElement
    Text: CustomText
  }
}
```

Annotations in the Editor

Annotate the editor's initial value w/ `Descendant[]`.

```
import React, { useMemo, useState } from 'react'
import { createEditor, Descendant } from 'slate'
import { Slate, Editable, withReact } from 'slate-react'

const initialValue: Descendant[] = [
  {
    type: 'paragraph',
    children: [{ text: 'A line of text in a paragraph.' }],
  },
]

const App = () => {
  const editor = useMemo(() => withReact(createEditor()), [])

  return (
    <Slate editor={editor} value={initialValue}>
      <Editable />
    </Slate>
  )
}
```

Best Practices for `Element` and `Text` Types

While you can define types directly in the `CustomTypes` interface, best practice is to define and export each type separately so that you can reference individual types like a `ParagraphElement`.

Using best practices, the custom types might look something like:

```typescript
// This example is for an Editor with `ReactEditor` and `HistoryEditor`
import { BaseEditor } from 'slate'
import { ReactEditor } from 'slate-react'
import { HistoryEditor } from 'slate-history'

export type CustomEditor = BaseEditor & ReactEditor & HistoryEditor

export type ParagraphElement = {
  type: 'paragraph'
  children: CustomText[]
}

export type HeadingElement = {
  type: 'heading'
  level: number
  children: CustomText[]
}

export type CustomElement = ParagraphElement | HeadingElement

export type FormattedText = { text: string; bold?: true }

export type CustomText = FormattedText

declare module 'slate' {
  interface CustomTypes {
    Editor: CustomEditor
    Element: CustomElement
    Text: CustomText
  }
}
```

In this example, `CustomText` is equal to `FormattedText` but in a real editor, there can be more types of text like text in a code block which may not allow formatting for example.

Why Is The Type Definition Unusual

Because it gets asked often, this section explains why Slate's type definition is atypical.

Slate needs to support a feature called type discrimination which is available when using union types (e.g. `ParagraphElement | HeadingElement`). This allows a user to narrow a type. If presented with code like `if (node.type === 'paragraph') { ... }` the inside of the block, will narrow the type of node to `ParagraphElement`.

Slate also needs a way to allow developers to get their custom types into Slate. This is done through declaration merging which is a feature of an `interface`.

Slate combines a union type and an interface in order to use both features.

For more information see **Proposal: Add Custom TypeScript Types to Slate**

Multiple Document Models

At the moment, Slate supports types for a single document model at a time. For example, it cannot support two different Rich Text Editor with different document schemas.

Slate's TypeScript support was designed this way because typing for one document schema was better than none. The goal is to eventually support typing for multiple editor definitions and there is currently an in progress PR built by the creator of Slate.

One workaround for supporting multiple document models is to create each editor in a separate package and then import them. This hasn't been tested but should work.

Extending Other Types

Currently there is also support for extending other types but these haven't been tested as thoroughly as the ones documented above:

- `Selection`
- `Range`
- `Point`

Feel free to extend these types but extending these types should be considered experimental. Please report bugs on GitHub issues.

TypeScript Examples

For some examples of how to use types, see `packages/slate-react/src/custom-types.ts` in the slate repository.

3.13. Migrating

Migrating from earlier versions of Slate to the `0.50.x` versions is not a simple task. The entire framework was re-considered from the ground up. This has resulted in a **much** better set of abstractions, which will result in you writing less code. But the migration process is not simple.

It's highly recommended that after reading this guide you read through the original **Walkthroughs** and the other **Concepts** to see how all of the new concepts get applied.

Major Differences

Here's an overview of the *major* differences in the `0.50.x` version of Slate from an architectural point of view.

JSON!

The data model is now comprised of simple JSON objects. Previously, it used **Immutable.js** data structures. This is a huge change, and one that unlocks many other things. Hopefully it will also increase the average performance when using Slate. It also makes it much easier to get started for newcomers. This will be a large change to migrate from, but it will be worth it.

Interfaces

The data model is interface-based. Previously each model was an instance of a class. Now, not only is the data plain objects, but Slate only expects that the objects implement an interface. So custom properties that used to live in `node.data` can now live at the top-level of the nodes.

Namespaces

A lot of helper functions are exposed as a collection of helper functions on a namespace. For example, `Node.get(root, path)` or `Range.isCollapsed(range)`. This ends up making code much clearer because you can always quickly see what interface you're working with.

TypeScript

The codebase now uses TypeScript. Working with pure JSON as a data model, and using an interface-based API are two things that have been made easier by migrating to TypeScript. You don't need to use it yourself, but if you do you'll get a lot more security when using the APIs. (And if you use VS Code you'll get nice autocompletion regardless!)

Fewer Concepts

The number of interfaces and commands has been reduced. Previously `Selection`, `Annotation`, and `Decoration` used to all be separate classes. Now they are simply objects that implement the `Range` interface. Previously `Block` and `Inline` were separate; now they are objects that implement the `Element` interface. Previously there was a `Document` and `Value`, but now the top-level `Editor` contains the children nodes of the document itself.

The number of commands has been reduced too. Previously we had commands for every type of input, like `insertText`, `insertTextAtRange`, `insertTextAtPath`. These have been merged into a smaller set of more customizable commands, eg. `insertText` which can take `at: Path | Range | Point`.

Fewer Packages

In an attempt to decrease the maintenance burden, and because the new abstraction and APIs in Slate's core packages make things much easier, the total number of packages has been reduced. Things like `slate-plain-serializer`, `slate-base64-serializer`, etc. have been removed and can be implemented in userland easily if needed. Even the `slate-html-deserializer` can now be implemented in userland (in ~10 LOC leveraging `slate-hyperscript`). And internal packages like `slate-dev-environment`, `slate-dev-test-utils`, etc. are no longer exposed because they are implementation details.

Commands

A new "command" concept has been introduced. (The old "commands" are now called "transforms".) This new concept expresses the semantic intent of a user editing the document. And they allow for the right abstraction to tap into user behaviors—for example to change what happens when a user presses enter, or backspace, etc. Instead of using `keydown` events you should likely override command behaviors instead.

Commands are triggered by calling the `editor.*` core functions. And they travel through a middleware-like stack, but built from composed functions. Any plugin can override the behaviors to augment an editor.

Plugins

Plugins are now plain functions that augment the `Editor` object they receive and return it again. For example, they can augment the command execution by composing the `editor.exec` function or listen to operations by composing `editor.apply`. Previously they relied on a custom middleware stack, and they were just bags of handlers that got merged onto an editor. Now we're using plain old function composition (aka wrapping) instead.

Elements

Block-ness and inline-ness is now a runtime choice. Previously it was baked into the data model with the `object: 'block'` or `object: 'inline'` attributes. Now, it checks whether an "element" is inline or not at runtime. For example, you might check to see that `element.type === 'link'` and treat it as inline.

More React-ish

Rendering and event-handling are no longer a plugin's concern. Previously plugins had full control over the rendering and event-handling logic in the editor. This creates a bad incentive to start putting **all** rendering logic in plugins, which puts Slate in a position of being a wrapper around all of React, which is very hard to do well. Instead, the new architecture has plugins focused purely on the richtext aspects, and leaves the rendering and event handling aspects to React.

Context

Previously the `<Editor>` component was doing double duty as a sort of "controller" object and also the `contenteditable` DOM element. This led to a lot of awkwardness in how other components worked with Slate. In the new version, there is a new `<Slate>` context provider and a simpler `<Editable>` `contenteditable`-like component. By putting the `<Slate>` provider higher up in your component tree, you can share the editor directly with toolbars, buttons, etc. using the `useSlate` hook.

Hooks

In addition to the `useSlate` hook, there are a handful of other hooks. For example the `useSelected` and `useFocused` hooks help with knowing when to render selected states (often for void nodes). And since they use React's Context API they will automatically re-render when their state changes.

`beforeinput`

We now use the `beforeinput` event almost exclusively. Instead of relying on a series of shims and the quirks of React synthetic events, we're now using the standardized `beforeinput` event as our baseline. It is fully supported in Safari and Chrome, will soon be supported in the new Chromium-based Edge, and is currently being worked on in Firefox. In the meantime there are a few patches to make Firefox work. Internet Explorer is no longer supported in core out of the box.

History-less

The core history logic has now finally been extracted into a standalone plugin. This makes it much easier for people to implement their own custom history behaviors. And it ensures that plugins have enough control to augment the editor in complex ways, because the history requires it.

Mark-less

Marks have been removed from the Slate data model. Now that we have the ability to define custom properties right on the nodes themselves, you can model marks as custom properties of text nodes. For example bold can be modelled simply as a `bold: true` property.

Annotation-less

Similarly, annotations have been removed from Slate's core. They can be fully implemented now in userland by defining custom operations and rendering annotated ranges using decorations. But most cases should be using custom text node properties or decorations anyways. There were not that many use cases that benefited from annotations.

Reductions

One of the goals was to dramatically simplify a lot of the logic in Slate to make it easier to maintain and iterate on. This was done by refactoring to better base abstractions that can be built on, by leveraging modern DOM APIs, and by migrating to simpler React patterns.

To give you a sense for the change in total lines of code:

```
slate                         8,436  ->  3,958  (47%)
slate-react                   3,905  ->  1,954  (50%)

slate-base64-serializer          38  ->      0
slate-dev-benchmark             340  ->      0
slate-dev-environment           102  ->      0
slate-dev-test-utils             44  ->      0
slate-history                     0  ->    211
slate-hotkeys                    62  ->      0
slate-html-serializer           253  ->      0
slate-hyperscript               447  ->    345
slate-plain-serializer           56  ->      0
slate-prop-types                 62  ->      0
slate-react-placeholder          62  ->      0

total                        13,807  ->  6,468  (47%)
```

It's quite a big difference! And that doesn't even include the dependencies that were shed in the process too.

4. API

4.1. Transforms API

Transforms are helper functions operating on the document. They can be used in defining your own commands.

- Node options
- Static methods
  - Node transforms
  - Selection transforms
  - Text transforms
  - Editor transforms

Node options

All transforms support a parameter `options`. This includes options specific to the transform, and general `NodeOptions` to specify which Nodes in the document that the transform is applied to.

```
interface NodeOptions {
  at?: Location
  match?: (node: Node, path: Location) => boolean
  mode?: 'highest' | 'lowest'
  voids?: boolean
}
```

- The `at` option selects a **Location** in the editor. It defaults to the user's current selection. **Learn more about the `at` option**
- The `match` option filters the set of Nodes with a custom function. **Learn more about the `match` option**
- The `mode` option also filters the set of nodes.
- When `voids` is false, **void Elements** are filtered out.

Static methods

Node transforms

Transforms that operate on nodes.

`Transforms.insertFragment(editor: Editor, fragment: Node[], options?)`

Insert of fragment of nodes at the specified location in the document. If no location is specified, insert at the current selection.

Options: `{at?: Location, hanging?: boolean, voids?: boolean}`

`Transforms.insertNodes(editor: Editor, nodes: Node | Node[], options?)`

Atomically inserts `nodes` at the specified location in the document. If no location is specified, inserts at the current selection. If there is no selection, inserts at the end of the document.

Options supported: `NodeOptions & {hanging?: boolean, select?: boolean}`.

For example, to insert at the very end, without replacing the current selection and regardless of block nesting, use

```
Transforms.insertNodes(
  editor,
  { type: targetType, children: [{ text: '' }] },
  { at: [editor.children.length] }
)
```

`Transforms.removeNodes(editor: Editor, options?)`

Remove nodes at the specified location in the document. If no location is specified, remove the nodes in the selection.

Options supported: `NodeOptions & {hanging?: boolean}`

`Transforms.mergeNodes(editor: Editor, options?)`

Merge a node at the specified location with the previous node at the same depth. If no location is specified, use the selection. Resulting empty container nodes are removed.

Options supported: `NodeOptions & {hanging?: boolean}`

`Transforms.splitNodes(editor: Editor, options?)`

Split nodes at the specified location. If no location is specified, split the selection.

Options supported: `NodeOptions & {height?: number, always?: boolean}`

`Transforms.wrapNodes(editor: Editor, element: Element, options?)`

Wrap nodes at the specified location in the `element` container. If no location is specified, wrap the selection.

Options supported: `NodeOptions & {split?: boolean}`.

- `options.mode`: `'all'` is also supported.

- `options.split` indicates that it's okay to split a node in order to wrap the location. For example, if `ipsum` was selected in a `Text` node with `lorem ipsum dolar`, `split: true` would wrap the word `ipsum` only, resulting in splitting the `Text` node. If `split: false`, the entire `Text` node `lorem ipsum dolar` would be wrapped.

`Transforms.unwrapNodes(editor: Editor, options?)`

Unwrap nodes at the specified location. If necessary, the parent node is split. If no location is specified, use the selection.

Options supported: `NodeOptions & {split?: boolean}`. For `options.mode`, `'all'` is also supported.

`Transforms.setNodes(editor: Editor, props: Partial<Node>, options?)`

Set properties of nodes at the specified location. If no location is specified, use the selection.

Options supported: `NodeOptions & {hanging?: boolean, split?: boolean}`. For `options.mode`, `'all'` is also supported.

`Transforms.unsetNodes(editor: Editor, props: string | string[], options?)`

Unset properties of nodes at the specified location. If no location is specified, use the selection.

Options supported: `NodeOptions & {split?: boolean}`. For `options.mode`, `'all'` is also supported.

`Transforms.liftNodes(editor: Editor, options?)`

Lift nodes at the specified location upwards in the document tree. If necessary, the parent node is split. If no location is specified, use the selection.

Options supported: `NodeOptions`. For `options.mode`, `'all'` is also supported.

`Transforms.moveNodes(editor: Editor, options)`

Move the nodes from an origin to a destination. A destination must be specified in the `options`. If no origin is specified, move the selection.

Options supported: `NodeOptions & {to: Path}`. For `options.mode`, `'all'` is also supported.

Selection transforms

Transforms that operate on the document's selection.

`Transforms.collapse(editor: Editor, options?)`

Collapse the selection to a single point.

Options: `{edge?: 'anchor' | 'focus' | 'start' | 'end'}`

`Transforms.select(editor: Editor, target: Location)`

Set the selection to a new value specified by `target`. When a selection already exists, this method is just a proxy for `setSelection` and will update the existing value.

For example, to set the selection to the entire contents of the editor:

```
Transforms.select(editor, {
  anchor: Editor.start(editor, []),
  focus: Editor.end(editor, []),
})
```

`Transforms.deselect(editor: Editor)`

Unset the selection.

`Transforms.move(editor: Editor, options?)`

Move the selection's point forward or backward.

Options: `{distance?: number, unit?: 'offset' | 'character' | 'word' | 'line', reverse?: boolean, edge?: 'anchor' | 'focus' | 'start' | 'end'}`

`Transforms.setPoint(editor: Editor, props: Partial<Point>, options?)`

Set new properties on one of the selection's points.

Options: `{edge?: 'anchor' | 'focus' | 'start' | 'end'}`

`Transforms.setSelection(editor: Editor, props: Partial<Range>)`

Set new properties on an active selection. Since the value is a `Partial<Range>`, this method can only handle updates to an existing selection. If there is no active selection the operation will be void. Use `select` if you'd like to create a selection when there is none.

Text transforms

Transforms that operate on text.

`Transforms.delete(editor: Editor, options?)`

Delete text in the document.

Options: `{at?: Location, distance?: number, unit?: 'character' | 'word' | 'line' | 'block', reverse?: boolean, hanging?: boolean, voids?: boolean}`

`Transforms.insertText(editor: Editor, text: string, options?)`

Insert a string of text at the specified location in the document. If no location is specified, insert at the current selection.

Options: `{at?: Location, voids?: boolean}`

Editor transforms

`Transforms.transform(editor: Editor, transform: Transform)`

Transform the `editor` by an `operation`.

4.2. Node Types APIs

The `Node` union type represents all of the different types of nodes that occur in a Slate document tree.

```
type Node = Editor | Element | Text

type Descendant = Element | Text
type Ancestor = Editor | Element
```

- Node
- NodeEntry
- Editor
- Element
- Text

4.2.1. Editor API

The `Editor` object stores all the state of a Slate editor. It can be extended by plugins to add helpers and implement new behaviors. It's a type of `Node` and its path is `[]`.

```
interface Editor {
  children: Node[]
  selection: Range | null
  operations: Operation[]
  marks: Omit<Text, 'text'> | null

  // Schema-specific node behaviors.
  isInline: (element: Element) => boolean
  isVoid: (element: Element) => boolean
  normalizeNode: (entry: NodeEntry) => void
  onChange: () => void

  // Overrideable core actions.
  addMark: (key: string, value: any) => void
  apply: (operation: Operation) => void
  deleteBackward: (unit: 'character' | 'word' | 'line' | 'block') => void
  deleteForward: (unit: 'character' | 'word' | 'line' | 'block') => void
  deleteFragment: () => void
  insertBreak: () => void
  insertFragment: (fragment: Node[]) => void
  insertNode: (node: Node) => void
  insertText: (text: string) => void
  removeMark: (key: string) => void
}
```


- Instantiation methods
- Static methods
  - Retrieval methods
  - Manipulation methods
  - Check methods
  - Normalization methods

    - Ref methods
- Instance methods
    - Schema-specific methods to override
    - Element Type Methods
    - Normalize Method
    - Callback Method
    - Mark Methods
    - getFragment Method
    - Delete Methods
    - Insert Methods
    - Operation Handling Method

Instantiation methods

`createEditor() => Editor`

Note: This method is imported directly from Slate and is not part of the Editor object.

Creates a new, empty `Editor` object.

Static methods

Retrieval methods

`Editor.above<T extends Ancestor>(editor: Editor, options?) => NodeEntry<T> | undefined`

Get the matching ancestor above a location in the document.

Options:

- `at?: Location = editor.selection`: Where to start at which is `editor.selection` by default.
- `match?: NodeMatch = () => true`: Narrow the match
- `mode?: 'highest' | 'lowest' = 'lowest'`: If `lowest` (default), returns the lowest matching ancestor. If `highest`, returns the highest matching ancestor.

- `voids?: boolean = false`: When `false` ignore void objects.

`Editor.after(editor: Editor, at: Location, options?) => Point | undefined`

Get the point after a location.

If there is no point after the location (e.g. we are at the bottom of the document) returns `undefined`.

Options: `{distance?: number, unit?: 'offset' | 'character' | 'word' | 'line' | 'block', voids?: boolean}`

`Editor.before(editor: Editor, at: Location, options?) => Point | undefined`

Get the point before a location.

If there is no point before the location (e.g. we are at the top of the document) returns `undefined`.

Options: `{distance?: number, unit?: 'offset' | 'character' | 'word' | 'line' | 'block', voids?: boolean}`

`Editor.edges(editor: Editor, at: Location) => [Point, Point]`

Get the start and end points of a location.

`Editor.end(editor: Editor, at: Location) => Point`

Get the end point of a location.

`Editor.first(editor: Editor, at: Location) => NodeEntry`

Get the first node at a location.

`Editor.fragment(editor: Editor, at: Location) => Descendant[]`

Get the fragment at a location.

`Editor.last(editor: Editor, at: Location) => NodeEntry`

Get the last node at a location.

`Editor.leaf(editor: Editor, at: Location, options?) => NodeEntry`

Get the leaf text node at a location.

Options: `{depth?: number, edge?: 'start' | 'end'}`

`Editor.levels<T extends Node>(editor: Editor, options?) => Generator<NodeEntry<T>, void, undefined>`

Iterate through all of the levels at a location.

Options: `{at?: Location, match?: NodeMatch, reverse?: boolean, voids?: boolean}`

`Editor.marks(editor: Editor) => Omit<Text, 'text'> | null`

Get the marks that would be added to text at the current selection.

`Editor.next<T extends Descendant>(editor: Editor, options?) => NodeEntry<T> | undefined`

Get the matching node in the branch of the document after a location.

Note: If you are looking for the next Point, and not the next Node, you are probably looking for the method `Editor.after`

Options: `{at?: Location, match?: NodeMatch, mode?: 'all' | 'highest' | 'lowest', voids?: boolean}`

`Editor.node(editor: Editor, at: Location, options?) => NodeEntry`

Get the node at a location.

Options: `depth?: number, edge?: 'start' | 'end'`

`Editor.nodes<T extends Node>(editor: Editor, options?) => Generator<NodeEntry<T>, void, undefined>`

At any given `Location` or `Span` in the editor provided by `at` (default is the current selection), the method returns a Generator of `NodeEntry` objects that represent the nodes that include `at`. At the top of the hierarchy is the `Editor` object itself.

Options: `{at?: Location | Span, match?: NodeMatch, mode?: 'all' | 'highest' | 'lowest', universal?: boolean, reverse?: boolean, voids?: boolean}`

`options.match`: Provide a value to the `match?` option to limit the `NodeEntry` objects that are returned.

`options.mode`:

- `'all'` (default): Return all matching nodes
- `'highest'`: in a hierarchy of nodes, only return the highest level matching nodes
- `'lowest'`: in a hierarchy of nodes, only return the lowest level matching nodes

`Editor.parent(editor: Editor, at: Location, options?) => NodeEntry<Ancestor>`

Get the parent node of a location.

Options: `{depth?: number, edge?: 'start' | 'end'}`

`Editor.path(editor: Editor, at: Location, options?) => Path`

Get the path of a location.

Options: `{depth?: number, edge?: 'start' | 'end'}`

`Editor.point(editor: Editor, at: Location, options?) => Point`

Get the start or end point of a location.

Options: `{edge?: 'start' | 'end'}`

`Editor.positions(editor: Editor, options?) => Generator<Point, void, undefined>`

Iterate through all of the positions in the document where a `Point` can be placed. The first `Point` returns is always the starting point followed by the next `Point` as determined by the `unit` option.

Read `options.unit` to see how this method iterates through positions.

Note: By default void nodes are treated as a single point and iteration will not happen inside their content unless you pass in true for the voids option, then iteration will occur.

Options:

- `at?: Location = editor.selection`: The `Location` in which to iterate the positions of.
- `unit?: 'offset' | 'character' | 'word' | 'line' | 'block' = 'offset'`:
    - `offset`: Moves to the next offset `Point`. It will include the `Point` at the end of a `Text` object and then move onto the first `Point` (at the 0th offset) of the next `Text` object. This may be counter-intuitive because the end of a `Text` and the beginning of the next `Text` might be thought of as the same position.
    - `character`: Moves to the next `character` but is not always the next `index` in the string. This is because Unicode encodings may require multiple bytes to create one character. Unlike `offset`, `character` will not count the end of a `Text` and the beginning of the next `Text` as separate positions to return. Warning: The character offsets for Unicode characters does not appear to be reliable in some cases like a Smiley Emoji will be identified as 2 characters.
    - `word`: Moves to the position immediately after the next `word`. In `reverse` mode, moves to the position immediately before the previous `word`.
    - `line` | `block`: Starts at the beginning position and then the position at the end of the block. Then starts at the beginning of the next block and then the end of the next block.
- `reverse?: boolean = false`: When `true` returns the positions in reverse order. In the case of the `unit` being `word`, the actual returned positions are different (i.e. we will get the start of a word in reverse instead of the end).
- `voids?: boolean = false`: When `true` include void Nodes.

`Editor.previous<T extends Node>(editor: Editor, options?) => NodeEntry<T> | undefined`

Get the matching node in the branch of the document before a location.

Note: If you are looking for the previous Point, and not the previous Node, you are probably looking for the method `Editor.before`

Options: `{at?: Location, match?: NodeMatch, mode?: 'all' | 'highest' | 'lowest', voids?: boolean}`

`Editor.range(editor: Editor, at: Location, to?: Location) => Range`

Get a range of a location.

```
Editor.start(editor: Editor, at: Location) => Point
```

Get the start point of a location.

```
Editor.string(editor: Editor, at: Location, options?) => string
```

Get the text string content of a location.

Note: by default the text of void nodes is considered to be an empty string, regardless of content, unless you pass in true for the voids option

Options: : `{voids?: boolean}`

```
Editor.void(editor: Editor, options?) => NodeEntry<Element> | undefined
```

Match a void node in the current branch of the editor.

Options: `{at?: Location, mode?: 'highest' | 'lowest', voids?: boolean}`

Manipulation methods

```
Editor.addMark(editor: Editor, key: string, value: any) => void
```

Add a custom property to the leaf text nodes in the current selection.

If the selection is currently collapsed, the marks will be added to the `editor.marks` property instead, and applied when text is inserted next.

```
Editor.deleteBackward(editor: Editor, options?) => void
```

Delete content in the editor backward from the current selection.

Options: `{unit?: 'character' | 'word' | 'line' | 'block'}`

```
Editor.deleteForward(editor: Editor, options?) => void
```

Delete content in the editor forward from the current selection.

Options: `{unit?: 'character' | 'word' | 'line' | 'block'}`

`Editor.deleteFragment(editor: Editor) => void`

Delete the content in the current selection.

`Editor.insertBreak(editor: Editor) => void`

Insert a block break at the current selection.

`Editor.insertFragment(editor: Editor, fragment: Node[]) => void`

Inserts a fragment *at the current selection*.

If the selection is currently expanded, it will be deleted first. To atomically insert nodes (including at the very beginning or end), use Transforms.insertNodes.

`Editor.insertNode(editor: Editor, node: Node) => void`

Inserts a node *at the current selection*.

If the selection is currently expanded, it will be deleted first. To atomically insert a node (including at the very beginning or end), use Transforms.insertNodes.

`Editor.insertText(editor: Editor, text: string) => void`

Inserts text *at the current selection*.

If the selection is currently expanded, it will be deleted first.

`Editor.removeMark(editor: Editor, key: string) => void`

Remove a custom property from all of the leaf text nodes in the current selection.

If the selection is currently collapsed, the removal will be stored on `editor.marks` and applied to the text inserted next.

`Editor.unhangRange(editor: Editor, range: Range, options?) => Range`

Convert a range into a non-hanging one.

A "hanging" range is one created by the browser's "triple-click" selection behavior. When triple-clicking a block, the browser selects from the start of that block to the start of the

next block. The range thus "hangs over" into the next block. If `unhangRange` is given such a range, it moves the end backwards until it's in a non-empty text node that precedes the hanging block.

Note that `unhangRange` is designed for the specific purpose of fixing triple-clicked blocks, and therefore currently has a number of caveats:

- It does not modify the start of the range; only the end. For example, it does not "unhang" a selection that starts at the end of a previous block.
- It only does anything if the start block is fully selected. For example, it does not handle ranges created by double-clicking the end of a paragraph (which browsers treat by selecting from the end of that paragraph to the start of the next).

Options:

- `voids?: boolean = false`: Allow placing the end of the selection in a void node.

Check methods

`Editor.hasBlocks(editor: Editor, element: Element) => boolean`

Check if a node has block children.

`Editor.hasInlines(editor: Editor, element: Element) => boolean`

Check if a node has inline and text children.

`Editor.hasTexts(editor: Editor, element: Element) => boolean`

Check if a node has text children.

`Editor.isBlock(editor: Editor, value: any) => value is Element`

Check if a value is a block `Element` object.

`Editor.isEditor(value: any) => value is Editor`

Check if a value is an `Editor` object.

`Editor.isEnd(editor: Editor, point: Point, at: Location) => boolean`

Check if a point is the end point of a location.

`Editor.isEdge(editor: Editor, point: Point, at: Location) => boolean`

Check if a point is an edge of a location.

`Editor.isEmpty(editor: Editor, element: Element) => boolean`

Check if an element is empty, accounting for void nodes.

`Editor.isInline(editor: Editor, value: any) => value is Element`

Check if a value is an inline `Element` object.

`Editor.isNormalizing(editor: Editor) => boolean`

Check if the editor is currently normalizing after each operation.

`Editor.isStart(editor: Editor, point: Point, at: Location) => boolean`

Check if a point is the start point of a location.

`Editor.isVoid(editor: Editor, value: any) => value is Element`

Check if a value is a void `Element` object.

Normalization methods

`Editor.normalize(editor: Editor, options?) => void`

Normalize any dirty objects in the editor.

Options: `{force?: boolean}`

`Editor.withoutNormalizing(editor: Editor, fn: () => void) => void`

Call a function, deferring normalization until after it completes.
See [Normalization - Implications for Other Code](#);

Ref Methods

`Editor.pathRef(editor: Editor, path: Path, options?) => PathRef`

Create a mutable ref for a `Path` object, which will stay in sync as new operations are applied to the editor.

Options: `{affinity?: 'backward' | 'forward' | null}`

`Editor.pathRefs(editor: Editor) => Set<PathRef>`

Get the set of currently tracked path refs of the editor.

`Editor.pointRef(editor: Editor, point: Point, options?) => PointRef`

Create a mutable ref for a `Point` object, which will stay in sync as new operations are applied to the editor.

Options: `{affinity?: 'backward' | 'forward' | null}`

`Editor.pointRefs(editor: Editor) => Set<PointRef>`

Get the set of currently tracked point refs of the editor.

`Editor.rangeRef(editor: Editor, range: Range, options?) => RangeRef`

Create a mutable ref for a `Range` object, which will stay in sync as new operations are applied to the editor.

Options: `{affinity?: 'backward' | 'forward' | 'outward' | 'inward' | null}`

`Editor.rangeRefs(editor: Editor) => Set<RangeRef>`

Get the set of currently tracked range refs of the editor.

Instance Methods

Schema-specific instance methods to override

Replace these methods to modify the original behavior of the editor when building **Plugins**. When modifying behavior, call the original method when appropriate. For example, a plugin that marks image nodes as "void":

```
const withImages = editor => {
  const { isVoid } = editor

  editor.isVoid = element => {
    return element.type === 'image' ? true : isVoid(element)
  }

  return editor
}
```

Element type methods

Use these methods so that Slate can identify certain elements as inlines or voids.

`isInline(element: Element) => boolean`

Check if a value is an inline `Element` object.

`isVoid(element: Element) => boolean`

Check if a value is a void `Element` object.

Normalize method

`normalizeNode(entry: NodeEntry) => void`

Normalize a Node according to the schema.

Callback method

`onChange() => void`

Called when there is a change in the editor.

Mark methods

`addMark(key: string, value: any) => void`

Add a custom property to the leaf text nodes in the current selection. If the selection is currently collapsed, the marks will be added to the `editor.marks` property instead, and applied when text is inserted next.

`removeMark(key: string) => void`

Remove a custom property from the leaf text nodes in the current selection.

getFragment method

`getFragment() => Descendant`

Returns the fragment at the current selection. Used when cutting or copying, as an example, to get the fragment at the current selection.

Delete methods

When a user presses backspace or delete, it invokes the method based on the selection. For example, if the selection is expanded over some text and the user presses the backspace key, `deleteFragment` will be called but if the selecttion is collapsed, `deleteBackward` will be called.

`deleteBackward(options?: {unit?: 'character' | 'word' | 'line' | 'block'}) => void`

Delete content in the editor backward from the current selection.

`deleteForward(options?: {unit?: 'character' | 'word' | 'line' | 'block'}) => void`

Delete content in the editor forward from the current selection.

`deleteFragment() => void`

Delete the content of the current selection.

Insert methods

`insertFragment(fragment: Node[]) => void`

Insert a fragment at the current selection. If the selection is currently expanded, delete it first.

`insertBreak() => void`

Insert a block break at the current selection. If the selection is currently expanded, delete it first.

`insertSoftBreak() => void`

Insert a soft break at the current selection. If the selection is currently expanded, delete it first.

`insertNode(node: Node) => void`

Insert a node at the current selection. If the selection is currently expanded, delete it first.

`insertText(text: string) => void`

Insert text at the current selection. If the selection is currently expanded, delete it first.

Operation handling method

`apply(operation: Operation) => void`

Apply an operation in the editor.

4.2.2. Element API

`Element` objects are a type of `Node` in a Slate document that contain other `Element` nodes or `Text` nodes.

```
interface Element {
  children: Node[]
}
```


- Behavior Types
  - Block vs. Inline
  - Void vs Not Void
    - Rendering Void Elements
- Static methods
  - Retrieval methods
  - Check methods

Element Behavior Types

Element nodes behave differently depending on the **Slate editor's configuration**. An element can be:

- "block" or "inline" as defined by `editor.isInline`
- either "void" or "not void" as defined by `editor.isVoid`

Block vs. Inline

A "block" element can only be siblings with other "block" elements. An "inline" node can be siblings with `Text` nodes or other "inline" elements.

Void vs Not Void

In a not "void" element, Slate handles the rendering of its `children` (e.g. in a paragraph where the `Text` and `Inline` children are rendered by Slate). In a "void" element, the `children` are rendered by the `Element`'s render code.

Rendering Void Elements

Void Elements must

- always have one empty child text node (for selection)
- render using `attributes` and `children` (so, their outermost HTML element **can't** be an HTML void element)
- set `contentEditable={false}` (for Firefox)

Typical rendering code will resemble this `thematic-break` (horizontal rule) element:

```
return (
  <div {...attributes} contentEditable={false}>
    {children}
    <hr />
  </div>
)
```

Static methods

Retrieval methods

`Element.matches(element: Element, props: Partial<Element>) => boolean`

Check if an element matches a set of `props`. Note: This checks custom properties, but it does not ensure that any children are equivalent.

Check methods

`Element.isElement(value: any) => value is Element`

Check if a `value` implements the `Element` interface.

`Element.isElementList(value: any) => value is Element[]`

Check if a `value` is an array of `Element` objects.

4.2.3. Node API

- Static methods
  - Retrieval methods
  - Text methods
  - Check methods

Static methods

Retrieval methods

`Node.ancestor(root: Node, path: Path) => Ancestor`

Get the node at a specific `path`, asserting that it is an ancestor node. If the specified node is not an ancestor node, throw an error.

`Node.ancestors(root: Node, path: Path, options?) => Generator<NodeEntry<Ancestor>>`

Return a generator of all the ancestor nodes above a specific path. By default, the order is bottom-up, from lowest to highest ancestor in the tree, but you can pass the `reverse: true` option to go top-down.

Options: `{reverse?: boolean}`

`Node.child(root: Node, index: number) => Descendant`

Get the child of a node at the specified `index`.

`Node.children(root: Node, path: Path, options?) => Generator<NodeEntry<Descendant>>`

Iterate over the children of a node at a specific path.

Options: `{reverse?: boolean}`

`Node.common(root: Node, path: Path, another: Path) => NodeEntry`

Get an entry for the common ancestor node of two paths.

`Node.descendant(root: Node, path: Path) => Descendant`

Get the node at a specific path, asserting that it's a descendant node.

`Node.descendants(root: Node, options?) => Generator<NodeEntry<Descendant>>`

Return a generator of all the descendant node entries inside a root node. Each iteration will return a `NodeEntry` tuple consisting of `[Node, Path]`.

Options: `{from?: Path, to?: Path, reverse?: boolean, pass?: (node: NodeEntry => boolean)}`

`Node.elements(root: Node, options?) => Generator<ElementEntry>`

Return a generator of all the element nodes inside a root node. Each iteration will return an `ElementEntry` tuple consisting of `[Element, Path]`. If the root node is an element, it will be included in the iteration as well.

Options: `{from?: Path, to?: Path, reverse?: boolean, pass?: (node: NodeEntry => boolean)}`

`Node.first(root: Node, path: Path) => NodeEntry`

Get the first node entry in a root node from a `path`.

`Node.fragment(root: Node, range: Range) => Descendant[]`

Get the sliced fragment represented by the `range`.

`Node.get(root: Node, path: Path) => Node`

Get the descendant node referred to by a specific `path`. If the path is an empty array, get the root node itself.

`Node.last(root: Node, path: Path) => NodeEntry`

Get the last node entry in a root node at a specific `path`.

`Node.leaf(root: Node, path: Path) => Text`

Get the node at a specific `path`, ensuring it's a leaf text node. If the node is not a leaf text node, throw an error.

`Node.levels(root: Node, path: Path, options?) => Generator<NodeEntry>`

Return a generator of the nodes in a branch of the tree, from a specific `path`. By default, the order is top-down, from the lowest to the highest node in the tree, but you can pass the `reverse: true` option to go bottom-up.

Options: `{reverse?: boolean}`

`Node.nodes(root: Node, options?) => Generator<NodeEntry>`

Return a generator of all the node entries of a root node. Each entry is returned as a `[Node, Path]` tuple, with the path referring to the node's position inside the root node.

Options: `{from?: Path, to?: Path, reverse?: boolean, pass?: (node: NodeEntry => boolean)}`

`Node.parent(root: Node, path: Path) => Ancestor`

Get the parent of a node at a specific `path`.

Text methods

Methods related to Text.

`Node.string(root: Node) => string`

Get the concatenated text string of a node's content. Note that this will not include spaces or line breaks between block nodes. This is not intended as a user-facing string, but as a string for performing offset-related computations for a node.

`Node.texts(root: Node, options?) => Generator<NodeEntry<Text>>`

Return a generator of all leaf text nodes in a root node.

Options: `{from?: Path, to?: Path, reverse?: boolean, pass?: (node: NodeEntry => boolean)}`

Check methods

Methods used to check some attribute of a Node.

`Node.has(root: Node, path: Path) => boolean`

Check if a descendant node exists at a specific `path`.

`Node.isNode(value: any) => value is Node`

Check if a `value` implements the `Node` interface.

`Node.isNodeList(value: any) => value is Node[]`

Check if a `value` is a list of `Node` objects.

`Node.matches(root: Node, props: Partial<Node>) => boolean`

Check if a node matches a set of `props`.

4.2.4. NodeEntry API

`NodeEntry` objects are returned when iterating over the nodes in a Slate document tree. They consist of an array with two elements: the `Node` and its `Path` relative to the root node in the document.

They are generics meaning that sometimes they will return a subset of `Node` types like an `Element` or `Text`.

```
type NodeEntry<T extends Node = Node> = [T, Path]
```

4.2.5. Text API

`Text` objects represent the nodes that contain the actual text content of a Slate document along with any formatting properties. They are always leaf nodes in the document tree as they cannot contain any children.

```
interface Text {
  text: string
}
```


- Static methods
  - Retrieval methods
  - Check methods

Static methods

Retrieval methods

`Text.matches(text: Text, props: Partial<Text>) => boolean`

Check if `text` matches a set of `props`.

The way the check works is that it makes sure that (a) all the `props` exist in the `text`, and (b) if it exists, that it exactly matches the properties in the `text`.

If a `props.text` property is passed in, it will be ignored.

If there are properties in `text` that are not in `props`, those will be ignored when it comes to testing for a match.

`Text.decorations(node: Text, decorations: Range[]) => Text[]`

Get the leaves for a text node, given `decorations`.

Check methods

`Text.equals(text: Text, another: Text, options?) => boolean`

Check if two text nodes are equal.

Options: `{loose?: boolean}`

- `loose?`: When `true`, it checks if the properties of the `Text` object are equal except for the `text` property (i.e. the `String` value of the `Text`). When `false` (default), checks all properties including `text`.

`Text.isText(value: any) => value is Text`

Check if a `value` implements the `Text` interface.

`Text.isTextList(value: any): value is Text[]`

Check if `value` is an `Array` of only `Text` objects.

4.3. Location Types APIs

The `Location` interface is a union of the ways to refer to a specific location in a Slate document: paths, points or ranges. Methods will often accept a `Location` instead of requiring only a `Path`, `Point` or `Range`.

```
type Location = Path | Point | Range
```

- Location
- Path
- PathRef
- Point
- PointEntry
- PointRef
- Range
- RangeRef
- Span

4.3.1. Location API

The Location interface is a union of the ways to refer to a specific location in a Slate document: paths, points or ranges. Methods will often accept a Location instead of requiring only a Path, Point or Range.

```
type Location = Path | Point | Range
```


- Static methods
  - Check methods

Static methods

Check methods

`Location.isLocation(value: any) => value is Location`

Check if a value implements the `Location` interface.

4.3.2. Path API

`Path` arrays are a list of indexes that describe a node's exact position in a Slate node tree. Although they are usually relative to the root `Editor` object, they can be relative to any `Node` object.

```
type Path = number[]
```


- Static methods
  - Retrieval methods
  - Check methods
  - Transform method

Static methods

Retrieval methods

`Path.ancestors(path: Path, options: { reverse?: boolean } = {}) => Path[]`

Get a list of ancestor paths for a given path.

The paths are sorted from deepest to shallowest ancestor. However, if the `reverse: true` option is passed, they are reversed.

`Path.common(path: Path, another: Path) => Path`

Get the common ancestor path of two paths.

`Path.compare(path: Path, another: Path) => -1 | 0 | 1`

Compare a path to another, returning an integer indicating whether the path was before, at, or after the other.

Note: Two paths of unequal length can still receive a `0` result if one is directly above or below the other. If you want exact matching, use [[Path.equals]] instead.

`Path.levels(path: Path, options?) => Path[]`

Get a list of paths at every level down to a path. Note: this is the same as `Path.ancestors`, but includes the path itself.

The paths are sorted from shallowest to deepest. However, if the `reverse: true` option is passed, they are reversed.

Options: `{reverse?: boolean}`

`Path.next(path: Path) => Path`

Given a path, gets the path to the next sibling node. The method does not ensure that the returned `Path` is valid in the document.

`Path.parent(path: Path) => Path`

Given a path, return a new path referring to the parent node above it. If the `path` argument is equal to `[]`, throws an error.

`Path.previous(path: Path) => Path`

Given a path, get the path to the previous sibling node. The method will throw an error if there are no previous siblings (e.g. if the Path is currently `[1, 0]`, the previous path would be `[1, -1]` which is illegal and will throw an error).

`Path.relative(path: Path, ancestor: Path) => Path`

Given two paths, one that is an ancestor to the other, returns the relative path from the `ancestor` argument to the `path` argument. If the `ancestor` path is not actually an ancestor or equal to the `path` argument, throws an error.

Check methods

Check some attribute of a path. Always returns a boolean.

`Path.endsAfter(path: Path, another: Path) => boolean`

Check if a path ends after one of the indexes in another.

`Path.endsAt(path: Path, another: Path) => boolean`

Check if a path ends at one of the indexes in another.

`Path.endsBefore(path: Path, another: Path) => boolean`

Check if a path ends before one of the indexes in another.

`Path.equals(path: Path, another: Path) => boolean`

Check if a path is exactly equal to another.

`Path.hasPrevious(path: Path) => boolean`

Check if the path of previous sibling node exists

`Path.isAfter(path: Path, another: Path) => boolean`

Check if a path is after another.

`Path.isAncestor(path: Path, another: Path) => boolean`

Check if a path is an ancestor of another.

`Path.isBefore(path: Path, another: Path) => boolean`

Check if a path is before another.

`Path.isChild(path: Path, another: Path) => boolean`

Check if a path is a child of another.

`Path.isCommon(path: Path, another: Path) => boolean`

Check if a path is equal to or an ancestor of another.

`Path.isDescendant(path: Path, another: Path) => boolean`

Check if a path is a descendant of another.

`Path.isParent(path: Path, another: Path) => boolean`

Check if a path is the parent of another.

`Path.isPath(value: any) => value is Path`

Check is a value implements the `Path` interface.

`Path.isSibling(path: Path, another: Path) => boolean`

Check if a path is a sibling of another.

Transform method

`Path.transform(path: Path, operation: Operation, options?) => Path | null`

Transform a path by an operation.

Options: `{ affinity?: 'forward' | 'backward' | null }`

4.3.3. PathRef API

`PathRef` objects keep a specific path in a document synced over time as new operations are applied to the editor. It is created using the `Editor.pathRef` method. You can access their property `current` at any time for the up-to-date `Path` value. When you no longer need to track this location, call `unref()` to free the resources. The `affinity` refers to the direction the `PathRef` will go when a user inserts content at the current position of the `Path`.

```
interface PathRef {
  current: Path | null
  affinity: 'forward' | 'backward' | null
  unref(): Path | null
}
```


- Static methods
  - Transform methods

Static methods

Transform methods

`PathRef.transform(ref: PathRef, op: Operation)`

Transform the path refs current value by an `op`.
The editor calls this as needed, so normally you won't need to.

4.3.4. Point API

`Point` objects refer to a specific location in a text node in a Slate document. Its `path` refers to the location of the node in the tree, and its `offset` refers to distance into the node's string of text. Points may only refer to `Text` nodes.

```
interface Point {
  path: Path
  offset: number
}
```


- Static methods
  - Retrieval methods
  - Check methods
  - Transform methods

Static methods

Retrieval methods

`Point.compare(point: Point, another: Point) => -1 | 0 | 1`

Compare a `point` to `another`, returning an integer indicating whether the point was before, at or after the other.

Check methods

`Point.isAfter(point: Point, another: Point) => boolean`

Check if a `point` is after `another`.

`Point.isBefore(point: Point, another: Point) => boolean`

Check if a `point` is before `another`.

`Point.equals(point: Point, another: Point) => boolean`

Check if a `point` is exactly equal to `another`.

`Point.isPoint(value: any) => value is Point`

Check if a `value` implements the `Point` interface.

Transform methods

`Point.transform(point: Point, op: Operation, options?) => Point | null`

Transform a `point` by an `op`.

Options: `{affinity?: 'forward' | 'backward' | null}`

4.3.5. PointEntry API

`PointEntry` objects are returned when iterating over `Point` objects that belong to a range.

```
type PointEntry = [Point, 'anchor' | 'focus']
```

4.3.6. PointRef API

`PointRef` objects keep a specific point in a document synced over time as new operations are applied to the editor. It is created using the `Editor.pointRef` method. You can access their property `current` at any time for the up-to-date `Point` value. When you no longer need to track this location, call `unref()` to free the resources. The `affinity` refers to the direction the `PointRef` will go when a user inserts content at the current position of the `Point`.

```
interface PointRef {
  current: Point | null
  affinity: 'forward' | 'backward' | null
  unref(): Point | null
}
```


- Instance methods
- Static methods
  - Transform methods

Instance methods

`unRef() => Point`

Call this when you no longer need to sync this point.

It also returns the current value.

Static methods

Transform methods

`PointRef.transform(ref: PointRef, op: Operation)`

Transform the point refs current value by an `op`.

The editor calls this as needed, so normally you won't need to.

4.3.7. Range API

`Range` objects are a set of points that refer to a specific span of a Slate document. They can define a span inside a single node or they can span across multiple nodes. The editor's `selection` is stored as a range.

```
interface Range {
  anchor: Point
  focus: Point
}
```

- Static methods
  - Retrieval methods
  - Check methods
  - Transform methods

Static methods

Retrieval methods

`Range.edges(range: Range, options?) => [Point, Point]`

Get the start and end points of a `range`, in the order in which they appear in the document.

Options: `{reverse?: boolean}`

`Range.end(range: Range) => Point`

Get the end point of a `range` according to the order in which it appears in the document.

`Range.intersection(range: Range, another: Range) => Range | null`

Get the intersection of one `range` with `another`. If the two ranges do not overlap, return `null`.

`Range.points(range: Range) => Generator<PointEntry>`

Iterate through the two point entries in a `Range`. First it will yield a `PointEntry` representing the `anchor`, then it will yield a `PointEntry` representing the `focus`.

`Range.start(range: Range) => Point`

Get the start point of a `range` according to the order in which it appears in the document.

Check methods

Check some attribute of a Range. Always returns a boolean.

`Range.equals(range: Range, another: Range) => boolean`

Check if a `range` is exactly equal to `another`.

`Range.includes(range: Range, target: Path | Point | Range) => boolean`

Check if a `range` includes a path, a point, or part of another range.

For clarity the definition of `includes` can mean partially includes. Another way to describe this is if one Range intersectns the other Range.

`Range.isBackward(range: Range) => boolean`

Check if a `range` is backward, meaning that its anchor point appears *after* its focus point in the document.

`Range.isCollapsed(range: Range) => boolean`

Check if a `range` is collapsed, meaning that both its anchor and focus points refer to the exact same position in the document.

`Range.isExpanded(range: Range) => boolean`

Check if a `range` is expanded. This is the opposite of `Range.isCollapsed` and is provided for legibility.

`Range.isForward(range: Range) => boolean`

Check if a `range` is forward. This is the opposite of `Range.isBackward` and is provided for legibility.

`Range.isRange(value: any) => value is Range`

Check if a `value` implements the `Range` interface.

Transform methods

`Range.transform(range: Range, op: Operation, options) => Range | null`

Transform a `range` by an `op`.

Options: `{affinity: 'forward' | 'backward' | 'outward' | 'inward' | null}`

4.3.8. RangeRef API

`RangeRef` objects keep a specific range in a document synced over time as new operations are applied to the editor. It is created using the `Editor.rangeRef` method. You can access their property `current` at any time for the up-to-date `Range` value. When you no longer need to track this location, call `unref()` to free the resources. The `affinity` refers to the direction the `RangeRef` will go when a user inserts content at the edges of the `Range`. `inward` means that the `Range` tends to stay the same size when content is inserted at its edges, and `outward` meaans that the `Range` tends to grow when content is inserted at its edges.

```
interface RangeRef {
  current: Range | null
  affinity: 'forward' | 'backward' | 'outward' | 'inward' | null
  unref(): Range | null
}
```

For example:

```
const selectionRef = Editor.rangeRef(editor, editor.selection, {
  affinity: 'inward',
})
// Allow the user to do stuff which might change the selection
Transforms.unwrapNodes(editor)
Transforms.select(editor, selectionRef.unRef())
```


- Instance methods
- Static methods
  - Transform methods

Instance methods

`unRef() => Range`

Call this when you no longer need to sync this range.

It also returns the current value.

Static methods

Transform methods

`RangeRef.transform(ref: RangeRef, op: Operation)`

Transform the range refs current value by an `op`.

The editor calls this as needed, so normally you won't need to.

4.3.9. Span API

A `Span` is a low-level way to refer to a `Range` using `Element` as the end points instead of a `Point` which requires the use of leaf text nodes.

```
type Span = [Path, Path]
```

- Static methods
  - Check methods

Static Methods

Check Methods

`Span.isSpan(value: any) => value is Span`

Check if a `value` implements the `Span` interface.

4.4. Operation API

`Operation` objects define the low-level instructions that Slate editors use to apply changes to their internal state. Representing all changes as operations is what allows Slate editors to easily implement history, collaboration, and other features.

The `Operation` *API reference needs to be added.*

5. Libraries

5.1. Slate React

This sub-library contains the React-specific logic for Slate.

Components

React components for rendering Slate editors

`RenderElementProps`

`RenderElementProps` are passed to the `renderElement` handler.

`RenderLeafProps`

`RenderLeafProps` are passed to the `renderLeaf` handler.

`Editable`

The main Slate editor.

Event Handling

By default, the `Editable` component comes with a set of event handlers that handle typical rich-text editing behaviors (for example, it implements its own `onCopy`, `onPaste`, `onDrop`, and `onKeyDown` handlers).

In some cases you may want to extend or override Slate's default behavior, which can be done by passing your own event handler(s) to the `Editable` component.

Your custom event handler can control whether or not Slate should execute its own event handling for a given event after your handler runs depending on the return value of your event handler as described below.

```jsx
import {Editable} from 'slate-react';

function MyEditor() {
  const onClick = event => {
    // Implement custom event logic...

    // When no value is returned, Slate will execute its own event handler when
    // neither isDefaultPrevented nor isPropagationStopped was set on the event
  };

  const onDrop = event => {
    // Implement custom event logic...

    // No matter the state of the event, treat it as being handled by returning
    // true here, Slate will skip its own event handler
    return true;
  };

  const onDragStart = event => {
    // Implement custom event logic...

    // No matter the status of the event, treat event as *not* being handled by
    // returning false, Slate will execute its own event handler afterward
    return false;
  };

  return (
    <Editable
      onClick={onClick}
      onDrop={onDrop}
      onDragStart={onDragStart}
      {/*...*/}
    />
  )
}
```

`DefaultElement(props: RenderElementProps)`

The default element renderer.

`DefaultLeaf(props: RenderLeafProps)`

The default custom leaf renderer.

`Slate(editor: ReactEditor, value: Node[], children: React.ReactNode, onChange: (value: Node[]) => void, [key: string]: any)`

A wrapper around the provider to handle `onChange` events, because the editor is a mutable singleton so it won't ever register as "changed" otherwise.

Hooks

React hooks for Slate editors

`useFocused`

Get the current `focused` state of the editor.

`useReadOnly`

Get the current `readOnly` state of the editor.

`useSelected`

Get the current `selected` state of an element.

`useSlate`

Get the current editor object from the React context. Re-renders the context whenever changes occur in the editor.

`useSlateStatic`

Get the current editor object from the React context. A version of useSlate that does not re-render the context. Previously called `useEditor`.

ReactEditor

A React and DOM-specific version of the `Editor` interface. All about translating between the DOM and Slate.

`findKey(editor: ReactEditor, node: Node)`

Find a key for a Slate node.

`findPath(editor: ReactEditor, node: Node)`

Find the path of Slate node.

`isFocused(editor: ReactEditor)`

Check if the editor is focused.

`isReadOnly(editor: ReactEditor)`

Check if the editor is in read-only mode.

`blur(editor: ReactEditor)`

Blur the editor.

`focus(editor: ReactEditor)`

Focus the editor.

`deselect(editor: ReactEditor)`

Deselect the editor.

`hasDOMNode(editor: ReactEditor, target: DOMNode, options: { editable?: boolean } = {})`

Check if a DOM node is within the editor.

`insertData(editor: ReactEditor, data: DataTransfer)`

Insert data from a `DataTransfer` into the editor. This is a proxy method to call in this order `insertFragmentData(editor: ReactEditor, data: DataTransfer)` and then `insertTextData(editor: ReactEditor, data: DataTransfer)`.

`insertFragmentData(editor: ReactEditor, data: DataTransfer)`

Insert fragment data from a `DataTransfer` into the editor. Returns true if some content has been effectively inserted.

`insertTextData(editor: ReactEditor, data: DataTransfer)`

Insert text data from a `DataTransfer` into the editor. Returns true if some content has been effectively inserted.

`setFragmentData(editor: ReactEditor, data: DataTransfer, originEvent?: 'drag' | 'copy' | 'cut')`

Sets data from the currently selected fragment on a `DataTransfer`.

`toDOMNode(editor: ReactEditor, node: Node)`

Find the native DOM element from a Slate node.

`toDOMPoint(editor: ReactEditor, point: Point)`

Find a native DOM selection point from a Slate point.

`toDOMRange(editor: ReactEditor, range: Range)`

Find a native DOM range from a Slate `range`.

`toSlateNode(editor: ReactEditor, domNode: DOMNode)`

Find a Slate node from a native DOM `element`.

`findEventRange(editor: ReactEditor, event: any)`

Get the target range from a DOM `event`.

`toSlatePoint(editor: ReactEditor, domPoint: DOMPoint)`

Find a Slate point from a DOM selection's `domNode` and `domOffset`.

`toSlateRange(editor: ReactEditor, domRange: DOMRange | DOMStaticRange | DOMSelection, options?: { exactMatch?: boolean } = {})`

Find a Slate range from a DOM range or selection.

Plugins

React-specific plugins for Slate editors

`withReact(editor: Editor)`

Adds React and DOM specific behaviors to the editor.

When used with `withHistory`, `withReact` should be applied outside. For example:

```
const editor = useMemo(() => withReact(withHistory(createEditor())), [])
```

Utils

Private convenience modules

5.2. Slate History

This sub-library tracks changes to the Slate value state over time, and enables undo and redo functionality.

`History`

`History` objects hold all of the operations that are applied to a value, so they can be undone or redone as necessary.

`HistoryEditor`

`HistoryEditor` contains helpers for history-enabled editors.

`withHistory`

The `withHistory` plugin keeps track of the operation history of a Slate editor as operations are applied to it, using undo and redo stacks.

When used with `withReact`, `withHistory` should be applied inside. For example:

```
const editor = useMemo(() => withReact(withHistory(createEditor())), [])
```

5.3. Slate Hyperscript

This package contains a hyperscript helper for creating Slate documents with JSX!

6. General

6.1. Resources

A few resources that are helpful for building with Slate.

Libraries

These libraries are helpful when developing with Slate:

- `is-hotkey` is a simple way to check whether an `onKeyDown` handler should fire for a given hotkey, handling cross-platform concerns like cmd vs. ctrl keys for you automatically.

Extensions and Plugins

These extensions and plugins add additional features and capabilities to Slate:

- Plate Rich text editor plugin system for Slate & React
- `slate-angular` Angular-based view layer, which is a useful supplement to Slate for building a rich text editor using Angular.
- `slate-yjs` Collaborative editing utilities for Slate leveraging Yjs
- `slate-collaborative` Collaborative editing utilities for Slate leveraging Automerge

Products

These products use Slate, and can give you an idea of what's possible:

- Archbee
- Cake
- Chatterbug
- Clause
- CoCalc
- GitBook
- Discord
- Grafana
- GraphCMS
- Guilded
- Guru
- Kitemaker
- Living Spec
- Netlify CMS
- Prezly
- Sanity.io
- Slite
- Taskade
- TRPG Engine
- Yuque
- Thoughts

Editors

These pre-packaged editors are built on top of Slate, and can be helpful to see how you might structure your code:

- Accord Project Markdown Editor is a WYSIWYG editor for CommonMark.
- Canner Editor is a rich text editor.
- Chatterslate helps teach language grammar and more at Chatterbug.
- CoCalc Collaborative Calculation editor in the Cloud

- French Press Editor is a customizeable editor with offline support.
- Nossas Editor is a drop-in WYSIWYG editor.
- React Force Slate Editor is a light-weight medium-style editor with no editor chrome.
- React Page is a self-contained, customizable inline WYSIWYG editor library.
- Slate Plugins Next provides an editor with configurable and extendable plugins.

(Or, if you have their exact use case, can be a drop-in editor for you.)

6.2. Contributing

Want to contribute to Slate? That would be awesome!

- Contributing
  - Reporting Bugs
  - Asking Questions
  - Submitting Pull Requests
  - Repository Setup
  - Running Examples
  - Running Tests
  - Testing Input Methods
  - Publishing Releases
    - Publishing Normal @latest Release
    - Publishing @next Release
    - Publishing @experimental Release
    - Running Prerelease Script

Reporting Bugs

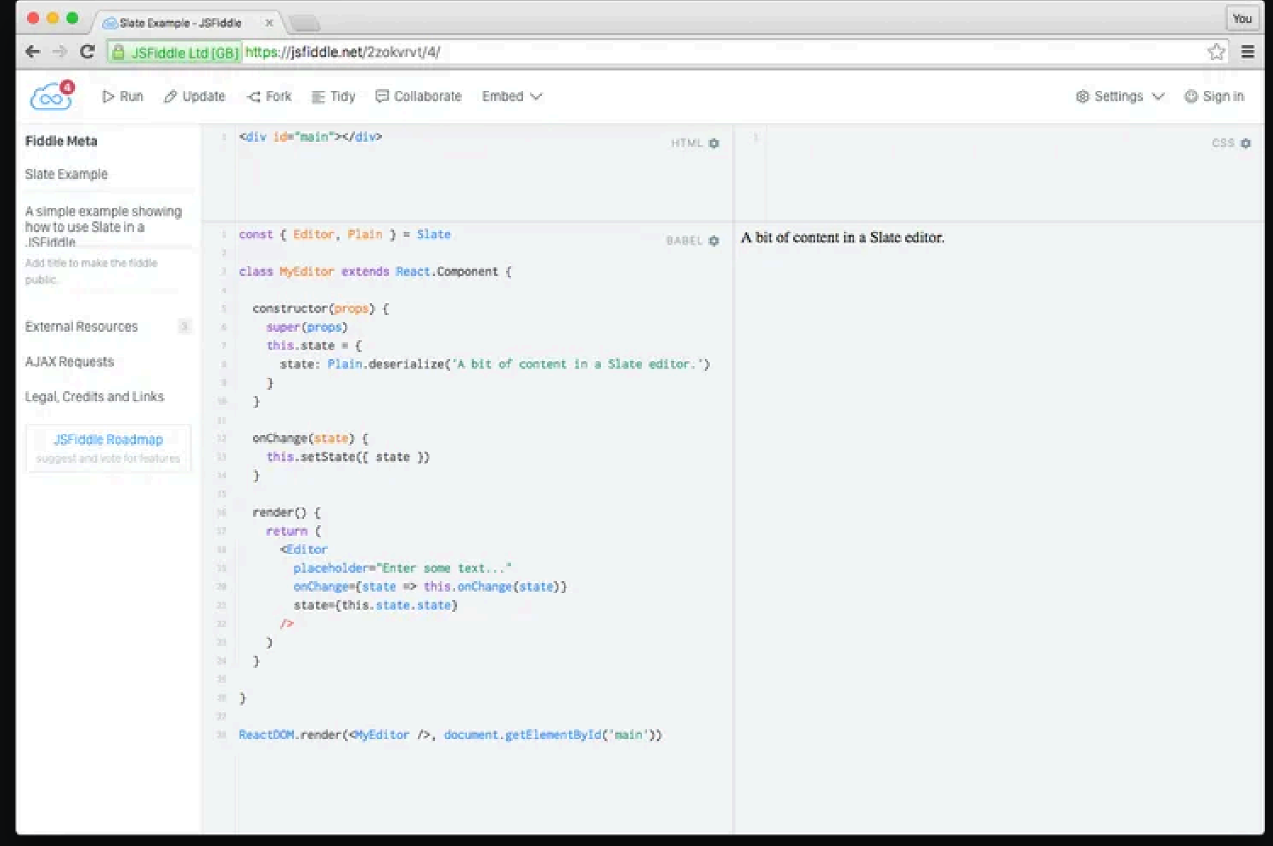
If you run into any weird behavior while using Slate, feel free to open a new issue in this repository! Please run a **search before opening** a new issue, to make sure that someone else hasn't already reported or solved the bug you've found.

Any issue you open must include:

- A JSFiddle that reproduces the bug with a minimal setup.
- A GIF showing the issue in action. (Using something like RecordIt.)
- A clear explanation of what the issue is.

Here's a JSFiddle template for Slate to get you started:

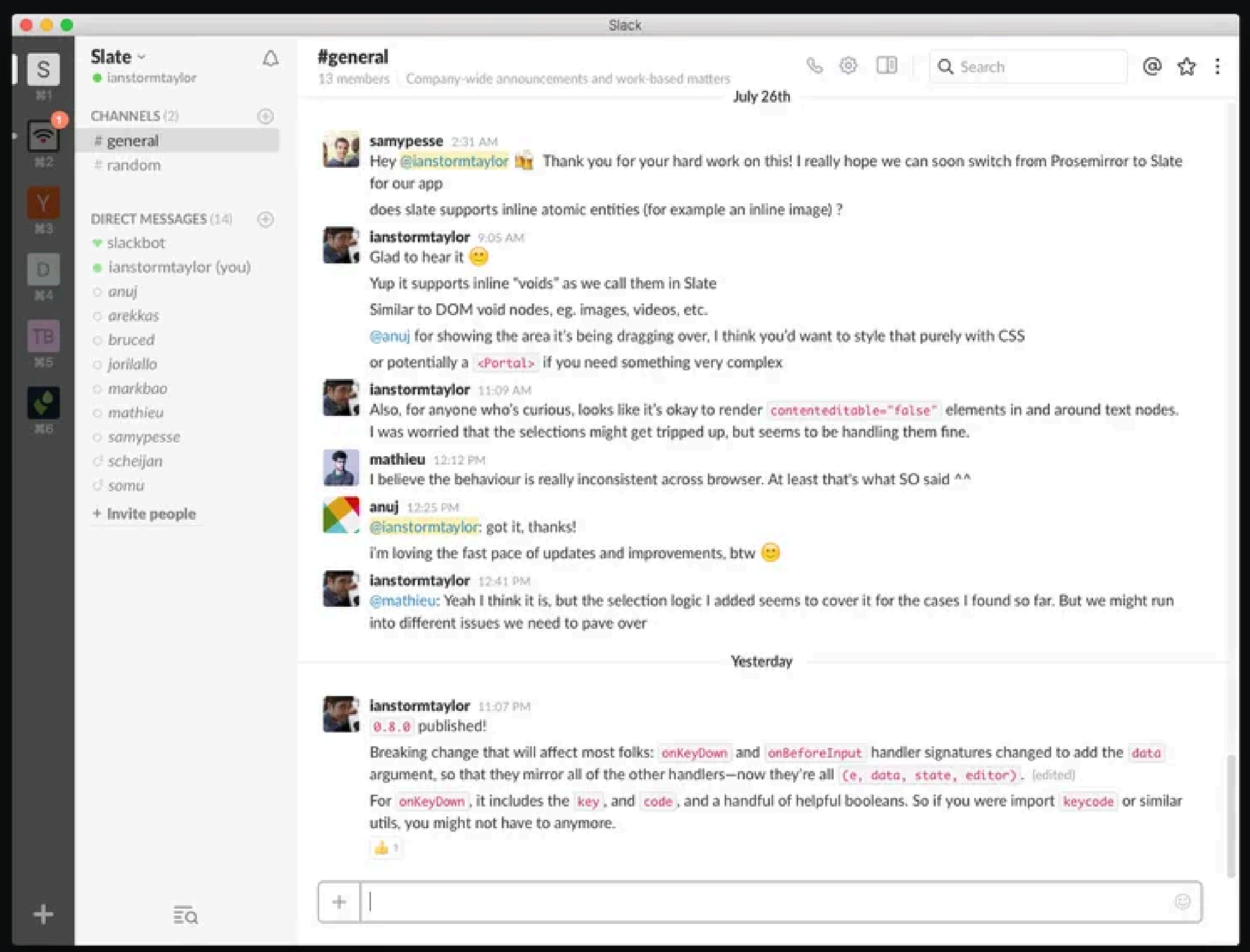
https://jsfiddle.net/01pLxfzu/



Asking Questions

We've also got a [Slate Slack team](https://slate-slack.herokuapp.com) where you can ask questions and get answers from other people using Slate:

https://slate-slack.herokuapp.com



Please use the Slack instead of asking questions in issues, since we want to reserve issues for keeping track of bugs and features. We close questions in issues so that maintaining the project isn't overwhelming.

Submitting Pull Requests

All pull requests are super welcomed and greatly appreciated! Issues in need of a solution are marked with a `♥ help` label if you're looking for somewhere to start.

Please include tests and docs with every pull request!

Repository Setup

The slate repository is a monorepo that is managed with [lerna](). Unlike more traditional repositories, this means that the repository must be built in order for tests, linting, or other common development activities to function as expected.

To run the build, you need to have the Slate repository cloned to your computer. After that, you need to `cd` into the directory where you cloned it, and install the dependencies with `yarn` and build the monorepo:

```
yarn install
yarn build
```

Running Examples

To run the examples, start by building the monorepo as described in the Repository Setup section.

Then you can start the examples server with:

```
yarn start
```

Running Tests

To run the tests, start by building the monorepo as described in the Repository Setup section.

Then you can rerun the tests with:

```
yarn test
```

If you need to debug something, you can add a `debugger` line to the source, and then run `yarn test:inspect`.

If you only want to run a specific test or tests, you can run `yarn test --fgrep="slate-react rendering"` flag which will filter the tests being run by grepping for the string in each test. (This is a Mocha flag that gets passed through.)

In addition to tests you should also run the linter:

```
yarn lint
```

This will catch TypeScript, Prettier, and Eslint errors.

```
yarn fix
```

This will fix Prettier and Eslint errors.

Running integration tests

To run integrations with cypress, first run `yarn start` to run the examples website, then run `yarn cypress:open` in a separate session to open the cypress GUI.

Testing Input Methods

Here's a helpful page detailing how to test various input scenarios on Windows, Mac and Linux.

Publishing Releases

Important: When creating releases using Lerna with the instructions below, you will be given choices around how to increase version numbers. You should always use a `major`, `minor` or `patch` release and must never use a `prerelease`. If a prerelease is used, the root package will not link to the packages in the `packages` directory creating hard to diagnose issues.

Publishing Normal `@latest` Release

Since we use Lerna to manage the Slate packages this is fairly easy, just run:

```
yarn release:latest
```

And follow the prompts Lerna gives you.

Note that this will automatically run the prelease script first that will build, test and lint before attempting to publish.

Publishing `@next` Release

If we are unsure as to the stability of a release because there are significant changes and/or particularly complex changes, release with the `@next` tag.

```
yarn release:next
```

And follow the prompts Lerna gives you.

Publishing `@experimental` Release

If you need to create an experimental release to see how a published package will behave during an actual publish, release with the `@experimental` tag. End users should have no expectation that an `@experimental` release will be usable.

```
yarn release:experimental
```

Running Prerelease Script

If we want to make sure that Slate code follows the preparations for a release but without actually publishing, run:

```
yarn prerelease
```

Which will build, test and lint Slate code.

6.3. Changelog

Couldn't convert your doc.

6.4. FAQ

A series of common questions people have about Slate:

- Why is content pasted as plain text?
- What browsers and devices does Slate support?

Why is content pasted as plain text?

One of Slate's core principles is that, unlike most other editors, it does **not** prescribe a specific "schema" to the content you are editing. This means that Slate's core has no concept of "block quotes" or "bold formatting".

For the most part, this leads to increased flexbility without many downsides, but there are certain cases where you have to do a bit more work. Pasting is one of those cases.

Since Slate knows nothing about your domain, it can't know how to parse pasted HTML content (or other content). So, by default whenever a user pastes content into a Slate editor, it will parse it as plain text. If you want it to be smarter about pasted content, you need to override the `insert_data` command and deserialize the `DataTransfer` object's `text/html` data as you wish.

What browsers and devices does Slate support?

Slate's goal is to support all the modern browsers on both desktop and mobile devices.

Slate is in beta and is community-driven and so its support is not as robust as it could be.

On the desktop, it's currently tested against the latest few versions of Chrome, Edge, Firefox and Safari on desktops. And it does not work in Internet Explorer.

On mobile, iOS devices are supported but not regularly tested. Chrome on Android was until recently unsupported except for in older versions of Slate (0.47) but has recently been added. For clarity, due to the differences in Android's support of the `beforeInput` event, Android input uses compositions and mutations which is different from other browsers. This means that Android support progresses separately from other browsers and due to it being new, may have more bugs.

If you want to add or improve browser or device support, we'd love for you to submit a pull request! Or in the case of incompatible browsers, build a plugin.

For older browsers, such as IE11, a lot of the now standard native APIs aren't available. Slate's position on this is that it is up to the user to bring polyfills (like https://polyfill.io) when needed for things like `el.closest`, etc. Otherwise we'd have to bundle and maintain lots of polyfills that others may not even need in the first place. For clarity, Slate makes no guarantees that it will work with older browsers, even with polyfills and at present, there are still unresolved issues with IE11.