What if you want custom UI interactions embedded in your Markdown?

By using `rehype-react` with the `htmlAst` field, you can write custom React components and then reference them from your Markdown files, or map generic HTML elements like `<ul>` or `<h2>` to your own components.

Note: this functionality was added in version 1.7.31 of gatsby-transformer-remark

Writing a component

Write the component the way you normally would. For example, here's a simple "Counter" component:

```
import React from "react"

const counterStyle = {
  /* styles skipped for brevity */
}

export default class Counter extends React.Component {
  static defaultProps = {
    initialvalue: 0,
  }

  state = {
    value: Number(this.props.initialvalue),
  }

  handleIncrement = () => {
    this.setState(state => {
      return {
        value: state.value + 1,
      }
    })
  }

  handleDecrement = () => {
    this.setState(state => {
      return {
        value: state.value - 1,
      }
    })
  }

  render() {
    return (
      <span style={counterStyle}>
        <strong style={{ flex: `1 1` }}>{this.state.value}</strong>
        <button onClick={this.handleDecrement}>-1</button>
        <button onClick={this.handleIncrement}>+1</button>
      </span>
    )
  }
}
```

Enabling the component

In order to display this component within a Markdown file, you'll need to add a reference to the component in the template that renders your Markdown content. There are five parts to this:

1. Install `rehype-react` and `unified` as a dependency

   ```
   npm install rehype-react unified
   ```

2. Import `rehype-react` & `unified` and whichever components you wish to use

   ```
   import { createElement } from "react"
   import rehypeReact from "rehype-react"
   import unified from "unified"
   import Counter from "../components/Counter"
   ```

3. Create a render function with references to your custom components

   ```
   const processor = unified().use(rehypeReact, {
     createElement,
     components: {
       "interactive-counter": Counter,
     },
   })

   export const renderAst = ast => {
     return processor.stringify(ast)
   }
   ```

   If you use TypeScript:

   ```
   const processor = unified().use(rehypeReact, {
     createElement,
     components: {
       "interactive-counter": Counter,
     },
   })

   export const renderAst = (ast: any): JSX.Element => {
     return processor.stringify(ast) as unknown as JSX.Element
   }
   ```

   I prefer to use hyphenated names to make it clear that it's a custom component.

4. Render your content using `htmlAst` instead of `html`

   This will look different depending on how you were previously referring to the post object retrieved from GraphQL, but in general you'll want to replace this:

```
<div dangerouslySetInnerHTML={{ __html: post.html }} />
```

with this:

```
{
  renderAst(post.htmlAst)
}
```

5. Change `html` to `htmlAst` in your `pageQuery`

```
# ...
markdownRemark(fields: { slug: { eq: $slug } }) {
  htmlAst # previously `html`

  # other fields...
}
# ...
```

Using the component

Now, you can directly use your custom component within your Markdown files! For instance, if you include the tag:

```
<interactive-counter></interactive-counter>
```

You'll end up with an interactive component:

In addition, you can also pass attributes, which can be used as props in your component:

```
<interactive-counter initialvalue="10"></interactive-counter>
```

Mapping from generic HTML elements

You can also map a generic HTML element to one of your own components. This can be particularly useful if you are using something like styled-components as it allows you to share these primitives between markdown content and the rest of your site. It also means the author of the Markdown doesn't need to use any custom markup - just standard markdown.

For example if you have a series of header components:

```
const PrimaryTitle = styled.h1`…`
const SecondaryTitle = styled.h2`…`
const TertiaryTitle = styled.h3`…`
```

You can map headers defined in markdown to these components:

```
const renderAst = new rehypeReact({
  createElement: React.createElement,
```

```
  components: {
    h1: PrimaryTitle,
    h2: SecondaryTitle,
    h3: TertiaryTitle,
  },
}).Compiler
```

And headers defined in markdown will be rendered as your components instead of generic HTML elements:

```
# This will be rendered as a PrimaryTitle component

## This will be rendered as a SecondaryTitle component

### This will be rendered as a TertiaryTitle component
```

Caveats

Although it looks like we're now using React components in our Markdown files, that's not *entirely* true: we're adding custom HTML elements which are then replaced by React components. This means there are a few things to watch out for.

Always add closing tags

JSX allows us to write self-closing tags, such as `<my-component />`. However, the HTML parser would incorrectly interpret this as an opening tag, and be unable to find a closing tag. For this reason, tags written in Markdown files always need to be explicitly closed, even when empty:

```
<my-component></my-component>
```

Attribute names are always lowercased

HTML attribute names are not case-sensitive. `gatsby-transformer-remark` handles this by lowercasing all attribute names; this means that any props on an exposed component must also be all-lowercase.

> *The prop in the `Counter` example above was named `initialvalue` rather than `initialValue` for exactly this reason; if we tried to access `this.props.initialValue` we'd have found it to be `undefined`.*

Attributes are always strings

Any prop that gets its value from an attribute will always receive a string value. Your component is responsible for properly deserializing passed values.

- Numbers are always passed as strings, even if the attribute is written without quotes: `<my-component value=37></my-component>` will still receive the string `"37"` as its value instead of the number `37`.
- React lets you pass a prop without a value, and will interpret it to mean `true`; for example, if you write `<MyComponent isEnabled />` then the props would be `{ isEnabled: true }`. However, in your Markdown, an attribute without a value will not be interpreted as `true`; instead, it will be passed as the empty string `""`. Similarly, passing `somevalue=true` and `othervalue=false` will result in the string values `"true"` and `"false"`, respectively.

- You can pass object values if you use `JSON.parse()` in your component to get the value out; just remember to enclose the value in single quotes to ensure it is parsed correctly (e.g. `<my-thing objectvalue='{"foo": "bar"}'></my-thing>`).

> *Notice in the `Counter` example how the initial `value` has been cast using the `Number()` function.*

Possibilities

Custom components embedded in Markdown enables many features that weren't possible before; here are some ideas, starting simple and getting complex:

- Write a live countdown clock for an event such as Christmas, the Super Bowl, or someone's birthday. Suggested markup: `<countdown-clock> 2019-01-02T05:00:00.000Z </countdown-clock>`
- Write a component that displays as a link with an informative hovercard. For example, you might want to write `<hover-card subject="ostrich"> ostriches </hover-card>` to show a link that lets you hover to get information on ostriches.
- If your Gatsby site is for vacation photos, you might write a component that allows you to show a carousel of pictures, and perhaps a map that shows where each photo was taken.
- Write a component that lets you add live code demos in your Markdown, using component-playground or something similar.
- Write a component that wraps a GFM table and displays the data from the table in an interactive graph.