Setup

Start by creating a new Next.js application with their CLI

```
npx create-next-app <YOUR APP NAME>
```

ReactSDK

Install the Storyblok React SDK. It helps us get the data from Storyblok, loads Storyblok Bridge for real-time visual updates, and provides us with a `storyblokEditable` function to link editable components to the Storyblok Visual Editor.

```
npm i @storyblok/react
```

Setting Up ReactSDK

Get the preview token from your Storyblok space settings. In the `_app.js` file -

```
import { storyblokInit, apiPlugin } from '@storyblok/react';
storyblokInit({
  accessToken: "your-preview-token",
  use: [apiPlugin],
  components: {}
});
```

The `apiPlugin` here helps us get the data. If you don't want to use `apiPlugin`, you can use your preferred method or function to fetch your data.

Fetching Data

To fetch data, we will utilize the `getStoryblokApi` function to retrieve and consume our content from the Storyblok API. In the `index.js` file -

```
import { getStoryblokApi } from '@storyblok/react';

export async function getStaticProps() {
  let slug = "home";
  let sbParams = {
    version: "draft", // or 'published'
  };
  const storyblokApi = getStoryblokApi();
  let { data } = await storyblokApi.get('cdn/stories/${slug}', sbParams);

  return {
    props: {
      story: data ? data.story : false,
      key: data ? data.story.id : false,
    },
    revalidate: 3600, // revalidate every hour
  };
}
```

Creating components with `storyblokEditable`

To make the components editable in Storyblok's Visual Editor, we add `storyblokEditable` function to the component.

```
import { storyblokEditable } from '@storyblok/react';

const Teaser = ({ blok }) => (
  <div className="" {...storyblokEditable(blok)}>
    {blok.name}
  </div>
);

export default Teaser;
```

Dynamic Component Rendering

In `_app.js`

```
... import { storyblokInit, apiPlugin } from '@storyblok/react';
import Page from '../components/Page';
import Teaser from '../components/Teaser';
const components = {
  teaser: Teaser,
  page: Page,
};
storyblokInit({
  accessToken: "your-preview-token",
  use: [apiPlugin],
  components,
});
...
```

We need to import the components and add those to storyblokinit. This will allow us dynamically render the component with the help of `StoryblokComponent`.

In `index.js`

```
...
import { getStoryblokApi, StoryblokComponent } from '@storyblok/react'
export default function Home(props) {
  const story = props.story
  return (
    <div>
      ...
      <StoryblokComponent blok={story.content} />
      ...
    </div>
  )
} ...
```

Live edits with `useStoryblokState`

To enable Storyblok's Visual Editor, we must connect the Storyblok Bridge. Use `useStoryblokState` hook to enable live updating for story content. What we are passing into `useStoryblokState` is the `story` prop in order to connect to the Visual Editor.
import { StoryblokComponent, useStoryblokState } from '@storyblok/react';
export default function Home({ story }) {
  story = useStoryblokState(story);
  return (
    <div>
      ...<StoryblokComponent blok={story.content} />
      ...
    </div>
  );
}

Rich Text Rendering

Rich Text Rendering is related to the field type, "Richtext". To render rich text you can use the storyblok-rich-text-react-renderer npm package.

```javascript
import { render } from 'storyblok-rich-text-react-renderer'
{render(blok.long_text)}
```

Adding Another Language

As all our routes are dynamic, adding another language is simple. There are two ways we can retrieve our content in different languages: Folder level and Field level.

**FIELD LEVEL**

Storyblok can store multi-language content on the field level. This means that you only need one content tree and you don’t have to create a stand-alone folder for each country.

In these screenshots, we have the (1) field. We activate the translation field by using the translation (2) toggle, and (3) modify the field.

```javascript
static async getInitialProps({ query }) {
  let language = query.language;
  let res = await StoryblokService.get(`cdn/stories/$lang/home`);
  return { res, }
}
```

**FOLDER LEVEL**

Create a new folder (1) under the pages directory for each language and query the desired language with the Storyblok API.

For adding translatable fields, in the following screenshots, we have the (1) field. We activate the translation field by using the translation (2) toggle, and (3) modify the field.

Directory Structure

For adding translatable fields, in the following screenshots, we have the (1) field. We activate the translation field by using the translation (2) toggle, and (3) modify the field.

Deployment on Vercel

Deploy your website by running the `vercel` command in your console.

```
npm i -g vercel
vercel
```