Build Content Rich Progressive Web Apps with Gatsby and Contentful

Transcripts for Khaled Garbaya (https://egghead.io/instructors/khaled-garbaya) course on egghead.io (https://egghead.io/courses/build-content-rich-progressive-web-apps-with-gatsby-and-contentful).

Description

The JAMstack, short for "JavaScript, APIs, and Markup," has been making waves in the world of web development.

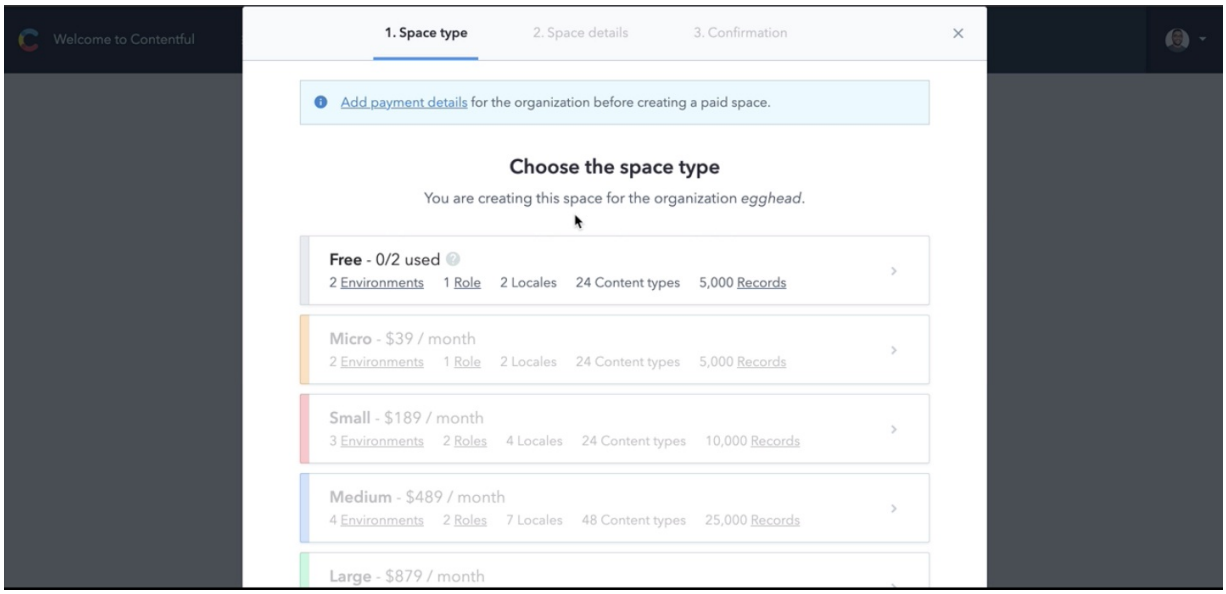
Building JAMstack applications removes the hassle of building out a backend from scratch, freeing you to focus on what really matters: your content.

In this course, you'll learn how to build and deploy your own static Gatsby site that pulls external data from Contentful and then deploys to the web with Netlify. After the course, you'll have all the

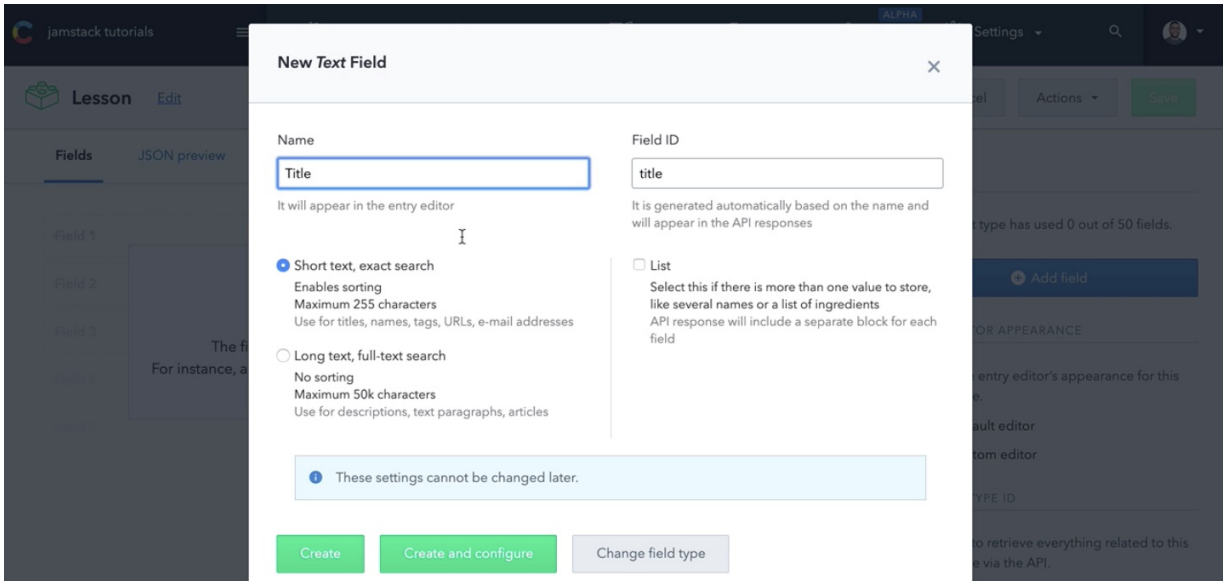
knowledge you to need to build a blog, marketing site, or portfolio with Gatsby. Just add content.

Model Content in the Contentful Web App

00:00 After we login to Contentful, first thing we need to do is create a space. This will hold all our content. We can pick one of the free spaces that we have, and we can name this `jamstack tutorials`, because we're building a website about JAMstack tutorials.



00:18 For our website, we need a lesson, and every lesson is assigned to an instructor. We can define the structure of that website simply by creating content types. The first one is `Lesson`, and for Lesson, we need a `Title`. This will be a text field.



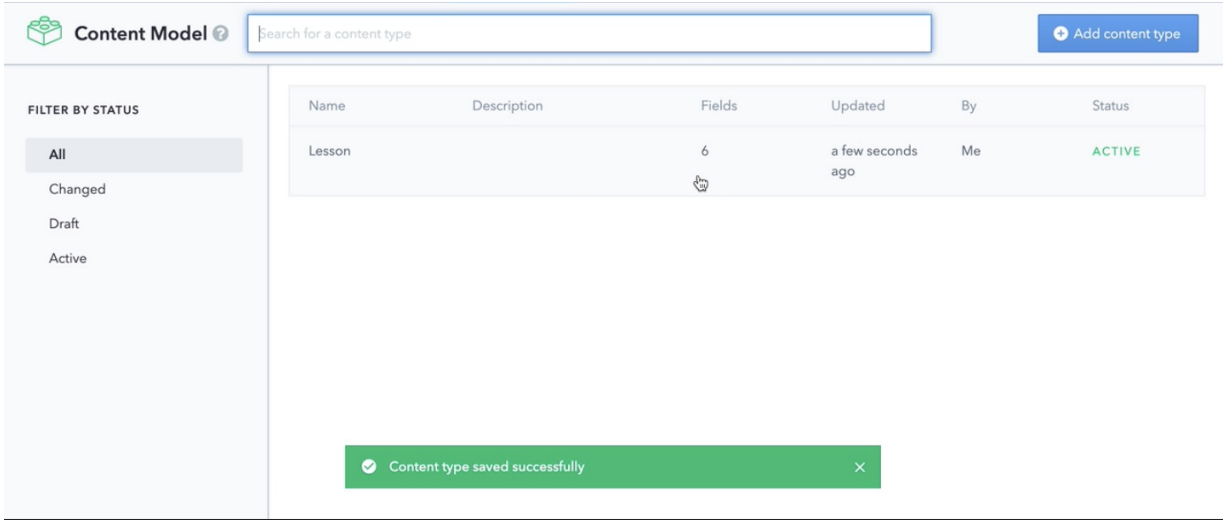
00:46 We'll need a `Slug` field. This will help us define human-readable URLs, and it will be autogenerated from the title automatically. We can configure it, and we tell Contentful, "Hey, this is a slug field." Now, we hit save.

01:04 After that, we need a rich text field. This will be the `Body` of the lesson, so this is the content of the lesson. We said we want to assign a lesson to an instructor. We need to create a reference field, and we call it `Instructor`.

01:25 This will link to a different content type that we will define later. A lesson also should have an image, so we can pick a media field, and we can call this `Image`. We only need one file, but we can list many files.

01:43 For some search engine optimizations, we can actually link another content type that will define just the metadata that we want to show to Google, meaning we will put them in the header of the HTML. Let's add that field. It will be another reference field, and we can call it simply `SEO`.

02:03 We hit create and save. Our first content type is done.



Now, we need to create the `Instructor` content type. Our instructors will have `Full Name`. It will be short text. Also, we want to have an instructor page, so we can use again a slug field to define human-readable URLs.

02:38 This will be called `Slug`, and we configure it to be a slug field. Instructor has a `Bio` field, so we know something about them. Also, the instructor can have a `website`. We can configure this to be a URL field. Also, for `Twitter`, we will do the same.

03:13 Why not link the `GitHub` account also? Finally, we want to see the face of our instructor, so we can actually have an `Avatar` for that. We save. We go back to the content type list, and we'll add our last content type. This will be called `SEO`.

03:44 For SEO, we can have the first field be `Title`. We can also have some `Description`, and lastly, `Keywords`, making a field for each one of those. They will all be of text fields. We save, we'll go back to our content type list. We need to change the lesson content type, to make sure our editors always link instructors and not any other content type.

04:23 For example, we can go to Settings and Validations, and we tell it only accept Instructor. We can do the same for SEO, and we only accept SEO content types. We hit save.

Model Content programmatically using the contentful-migration tool

00:00 Another way we can create content types in Contentful is using the Contentful migration tool. This will allow us to programmatically create them. Let's go ahead and code our instructor content type. First thing, we need to export the function.

Instructor.js

```
module.exports = function(migration) {

}
```

00:20 Inside of this function, we'll have access to the migration object that will allow us to do any sort of manipulations to our content type. First thing we need to do is `create the content type`, then we need to define its `fields`.

00:41 We can define the `appearances` for the slug field, for example.

```
module.exports = function(migration) {
// create the content type

//fields

//appearances
}
```

To create the content type, we can call the migration `.createContentTypes`, and we give it the id of `instructor`, the `name`, `description`, and `displayField`.

```
module.exports = function(migration) {
// create the content type
const instructor = migration
  .createContentType("intructor")
  .name("Instructor")
  .description("")
  .displayField("fullName")

//fields

//appearances
}
```

Now, we have an instance of our created content type. We can create the fields by calling createField on the instance.

01:08 The field will have the id of `instructor`, the `name`, and `type`.

```
//fields
instructor
  .createField("fullName")
  .name("Full Name")
  .type("Symbol")
```

Let's define the rest of the fields.

```
// fields
instructor
  .createField("fullName")
  .name("Full Name")
  .type("Symbol")
instructor
  .createField("slug")
  .name("Slug")
  .type("Symbol")
instructor
  .createField("bio")
  .name("Bio")
  .type("Symbol")
instructor
  .createField("website")
  .name("website")
  .type("Symbol")
instructor
  .createField("twitter")
  .name("Twitter")
  .type("Symbol")
instructor
  .createField("github")
  .name("Github")
  .type("Symbol")

instructor
  .createField("avatar")
  .name("Avatar")
  .type("Link")
  .linkType("Asset")
```

Now, this is done, we need to tell Contentful how to deal with the slug field. Remember, the previous lesson, we set it to a type `slug`. We can do the same using the migration tool.

01:38 Here, we call the `.changeEditorInterface` on the content type, and we give it the field ID and then the ID of the widget. We can do the same for the website field and other similar fields.

```
// appearances
  instructor.changeEditorInterface("slug",
"slugEditor", {})
  instructor.changeEditorInterface("website",
"urlEditor", {})
  instructor.changeEditorInterface("twitter",
"urlEditor", {})
  instructor.changeEditorInterface("github",
"urlEditor", {})
}
```

01:58 Let's save our file, and now, it's time to run our migration.

I already went ahead and installed the `contentful-cli`.

Terminal

```
npm install -g contentful-cli
```

Then after that's done, you need to call `contentful login`.

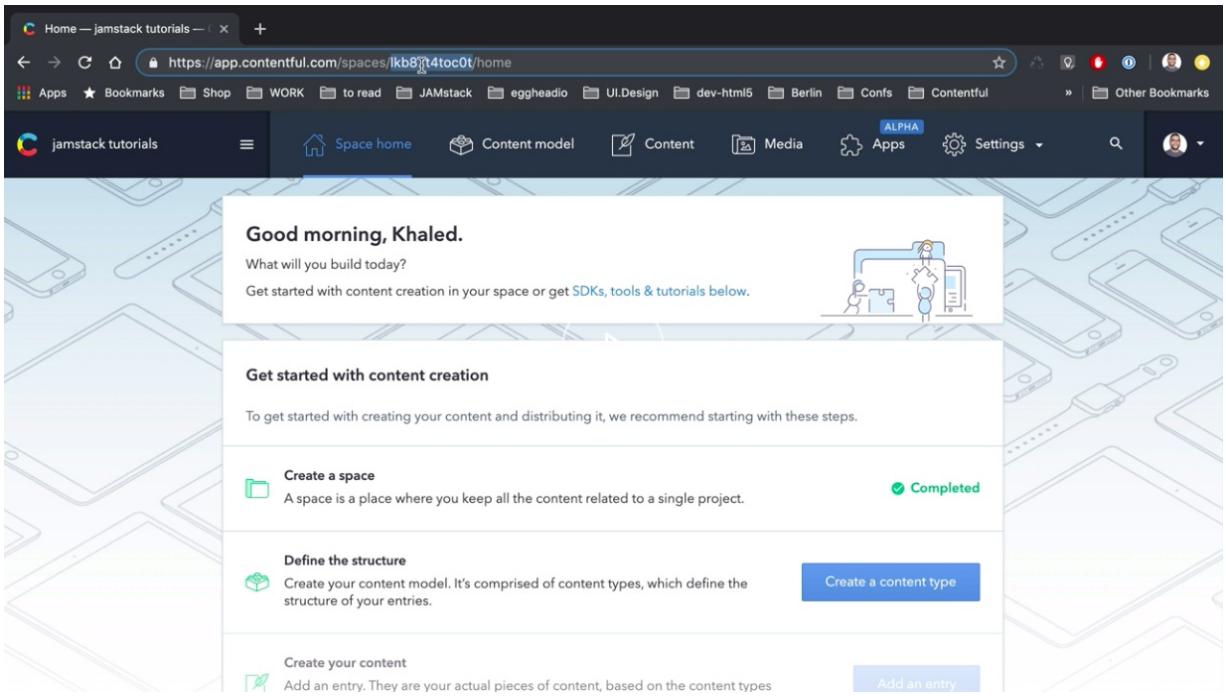
```
contentful login
```

This will open up a browser window and authenticate you to Contentful.

02:21 What we need to do here is to call the migration on the space that we have.

```
contentful space migration --space-
id=lkb87t4toc0t
```

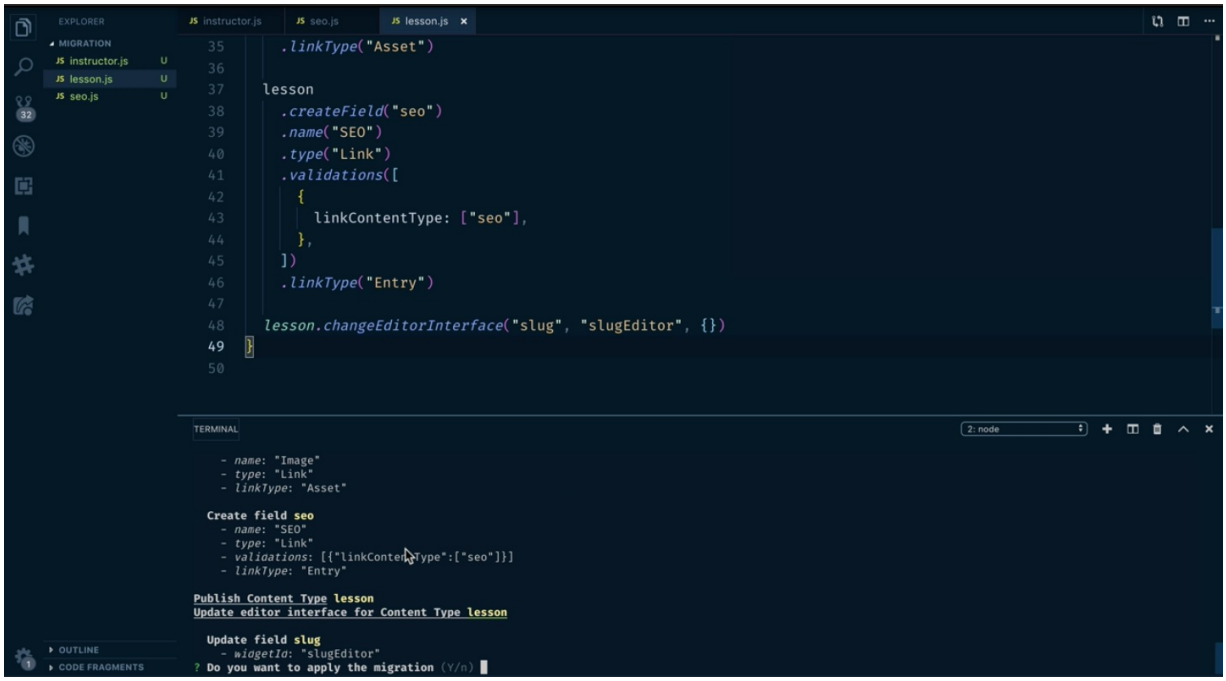
The `space-id`, we can get from Contentful. It's basically the URL.



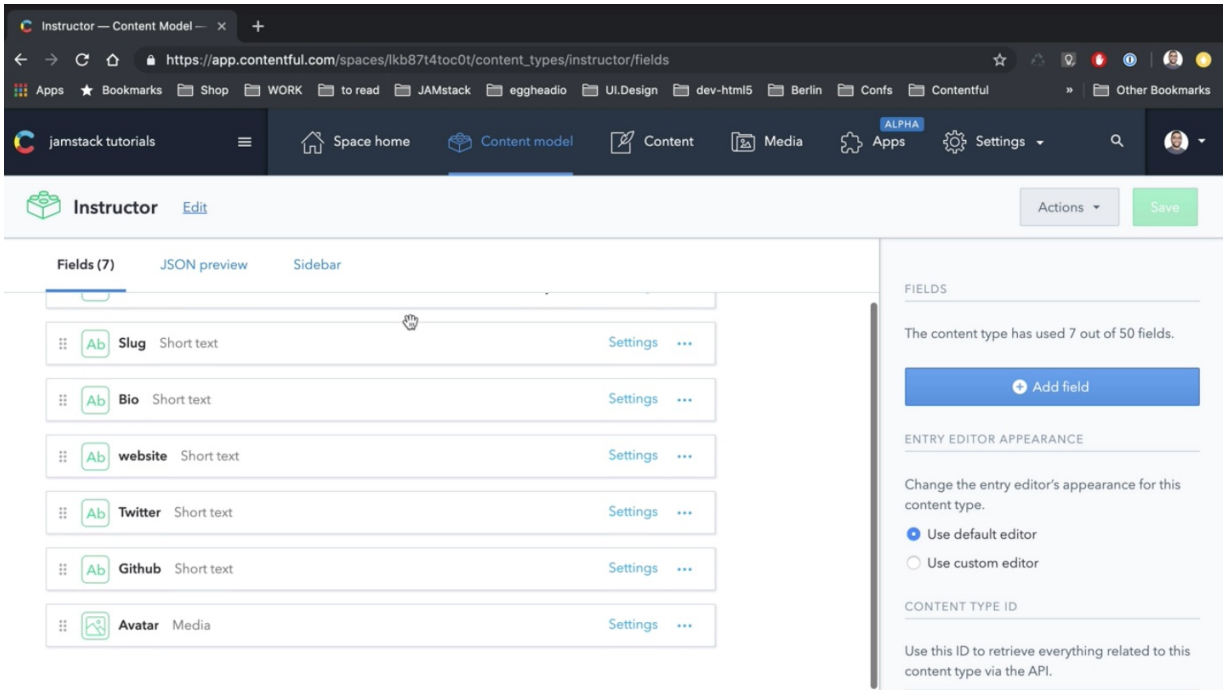
02:52 Here, we pass it the file that we want to run, basically our migration code.

```
contentful space migration --space-
id=lkb87t4toc0t instructor.js
```

We hit enter. You can see here, the tool gives us the summary of all the migration, and it's waiting for us to say yes. We say yes, and now it's creating our content type.



03:23 That's done. Let's check. You can see here, we have our instructor content type, with all the fields.



Let's do the same for our SEO content type. First thing, in `seo.js`, exporting the function. Then we create the content type. Lastly, we define the fields.

seo.js

```
module.exports = function(migration) {
  const seo = migration
    .createContentType("seo")
    .name("SEO")
    .description("")
    .displayField("title")
  seo
    .createField("title")
    .name("Title")
    .type("Symbol")
  seo
    .createField("description")
    .name("Description")
    .type("Symbol")
  seo
    .createField("keywords")
    .name("Keywords")
    .type("Symbol")
}
```

04:16 We save, and let's run our migration. Here, instead of passing the instructor, we will pass the `seo.js` file.

```
contentful space migration --space-
id=lkb87t4toc0t seo.js
```

Again, we work through the summary. Everything looks OK, and we accept our migration.

Lastly, we can define the Lesson content type that will link to both these content types.

04:56 In `lesson.js`, we create the content type again. We define the fields. We have the `title`, the `slug`, the `body` with type `RichText`. This is the `instructor`, and the syntax is a bit different. It's `linkContentType`, and in the `validations` we tell to only link to `instructor` field. For the image, we add our `SEO`, and we tell it also to only link to the `seo` content type.

```
module.exports = function(migration) {
  const lesson = migration
    .createContentType("lesson")
    .name("Lesson")
    .description("")
    .displayField("title")
  lesson
    .createField("title")
    .name("Title")
    .type("Symbol")
  lesson
    .createField("slug")
    .name("Slug")
    .type("Symbol")
  lesson
    .createField("body")
    .name("Body")
    .type("RichText")

  lesson
    .createField("instructor")
    .name("Instructor")
    .type("Link")
    .validations([
      {
        linkContentType: ["instructor"],
      },
    ])
    .linkType("Entry")
```

```
lesson
  .createField("image")
  .name("Image")
  .type("Link")
  .linkType("Asset")

lesson
  .createField("seo")
  .name("SEO")
  .type("Link")
  .validations([
    {
      linkContentType: ["seo"],
    },
  ])
  .linkType("Entry")
```

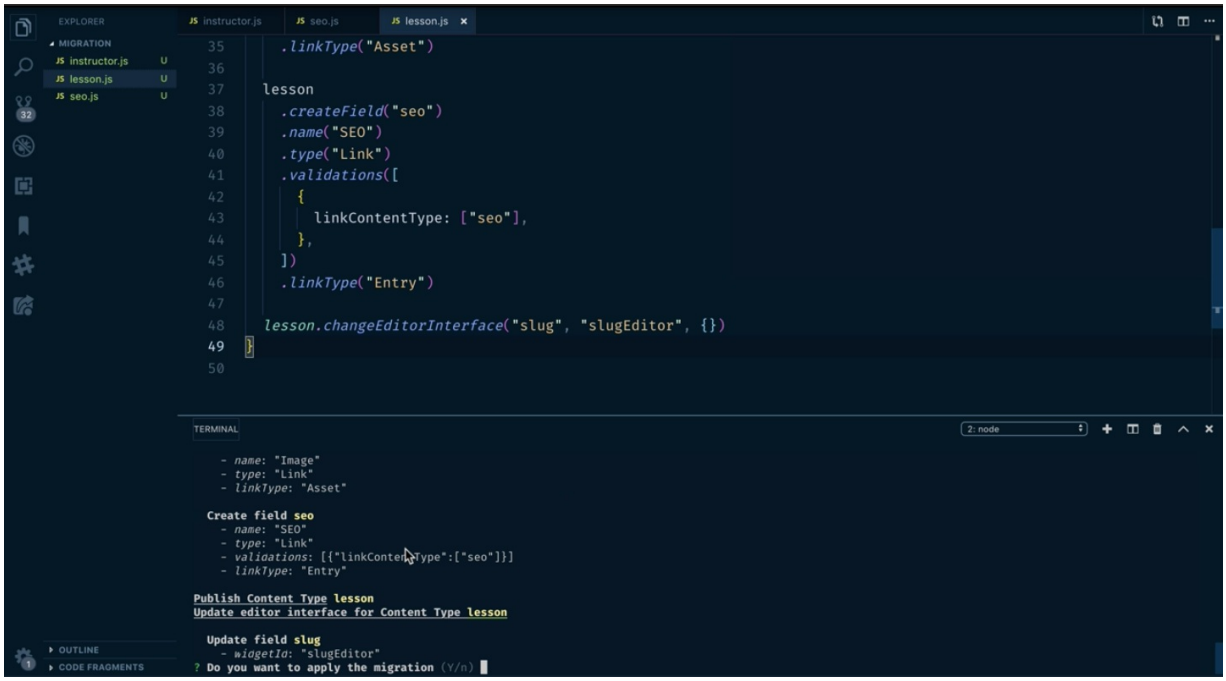
05:41 The last thing is to change the appearance of the `slug` field.

```
lesson.changeEditorInterface("slug",
"slugEditor", {})
```

We run our migration again for the lesson.

```
contentful space migration --space-
id=lkb87t4toc0t lesson.js
```

Summary looks good, and we hit enter. Now, if we go to Contentful and refresh this page, we should see all three content types.



06:18 That's how you can create them programmatically. This code should live next to your website in the repository, so another developer can bootstrap a separate space. For example, for testing.

Add Contentful as a data source for Gatsby

00:00 First, run `npx gatsby new` and give it the name of the website.

```
npx gatsby new jamstacktutorials
```

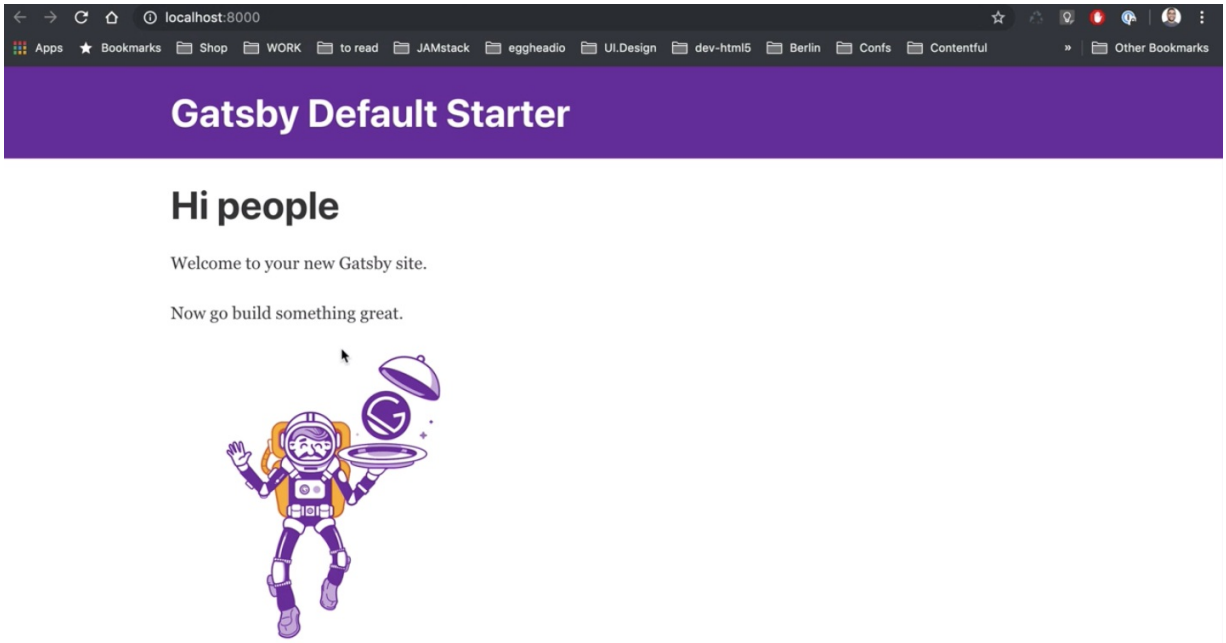
Now let's `cd` to this directory.

```
cd jamstacktutorials
```

Let's run `npm run develop`.

```
npm run develop
```

If we click on this link, you can see this is the "Hello, World" example provided by Gatsby.



00:31 Now let's add some Contentful dependency to it.

We'll stop our server, and we'll install the `gatsby-source-contentful` plugin.

```
npm i gatsby-source-contentful
```

Now let's open our project in a text editor. We need to go to the `gatsby-config.js` and add our new plugin in there.

01:08 We will need to provide two options -- the `spaceId` and the `accessToken`. To get this, we need to go to Contentful and Settings, API Keys. We will click on the first entry here, copy the `spaceId` and copy the Delivery `accessToken`.

01:45 This is a read-only Delivery `accessToken`. We paste it in here.

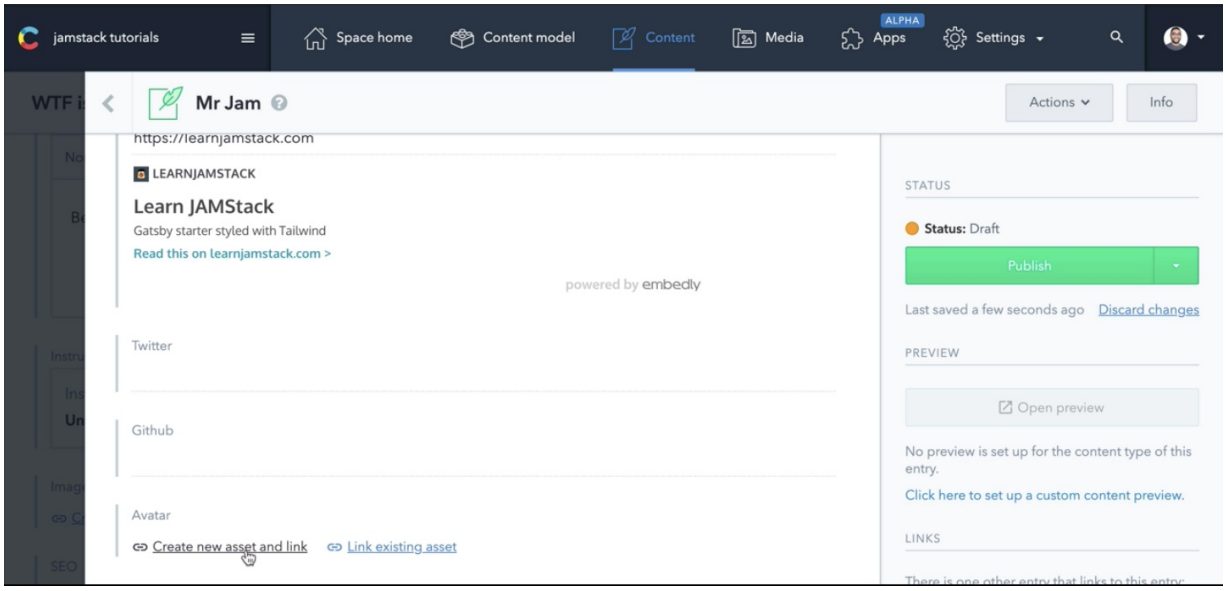
```
        resolve: `gatsby-source-contentful`,
        options: {
          spaceId: `u2hjug1nowzr`,
          accessToken:
`sJJaBCxUdA4BFqtfR_f5y4m9lvmyOHa3siR8iEETKEc`,
        }
```

01:55 This is done from the Gatsby side. We need to add some content to the website. To do that, we can go to Content, click on `Add entry`.

02:05 We'll add our first lesson. It will be called `"WTF is JAMstack."` I would paste some text in the body.

02:26 Now let's create an instructor and link it to this lesson. Our instructor name is `Mr Jam`. The website will be `https://learnjamstack.com`.

02:48 For the avatar, we need to create an asset and link it.



Now we can publish the asset and publish our lesson. Also, we can add an image to our lesson. We publish the asset.

03:43 Finally, we can create a new reference for SEO and link it. The title will be the same. We add in some description and some keywords. We publish that, and finally, we publish our entire lesson.

04:13 Let's go back to our command line. Let's run `npm run develop`.

```
npm run develop
```

To test if this works, we can go to the GraphQL server that's provided by Gatsby.

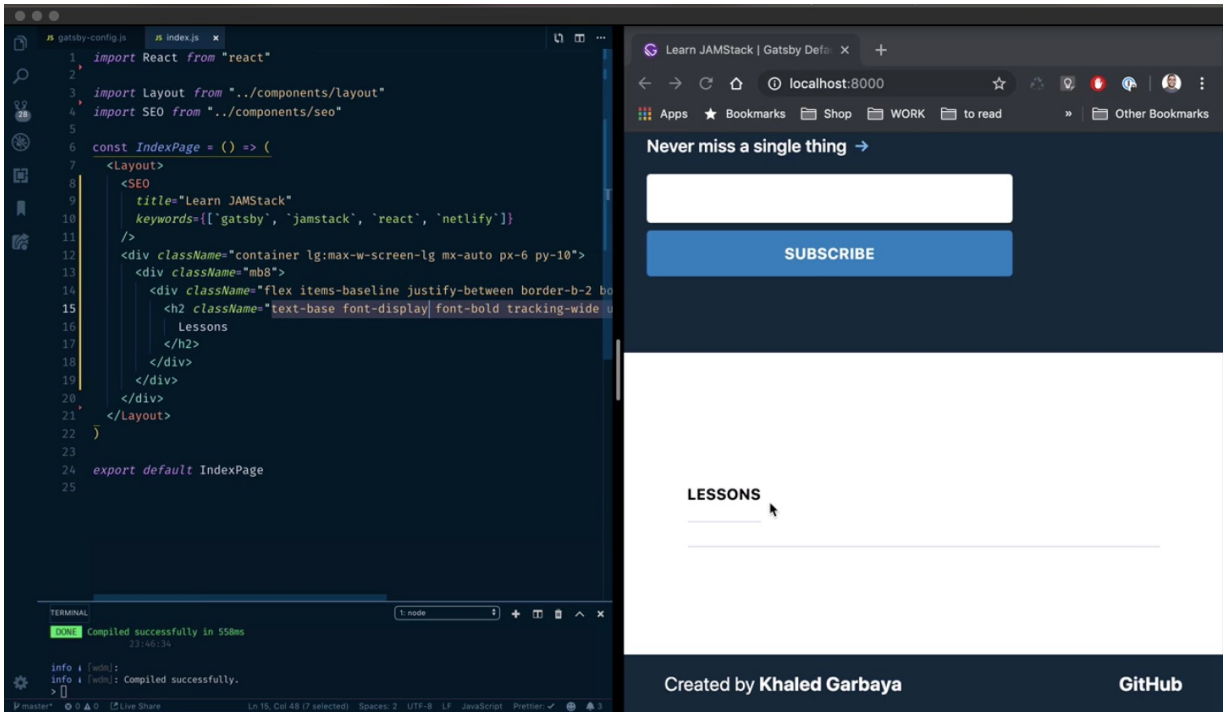
04:28 Then here, we go to docs, query. We look for some Contentful-related nodes. As you can see here, we have all Contentful content types and Contentful lessons.

04:44 Here we can simply query for all lessons. Inside of the node, we can get all the fields that we have, so let's get titles.

As you can see here, this is the title of our lesson.

List data entries from Contentful in Gatsby

00:00 Here, I have a Gatsby website running. I am using tailwind for styling. We would like to list all the lessons that we stored in Contentful in our website.

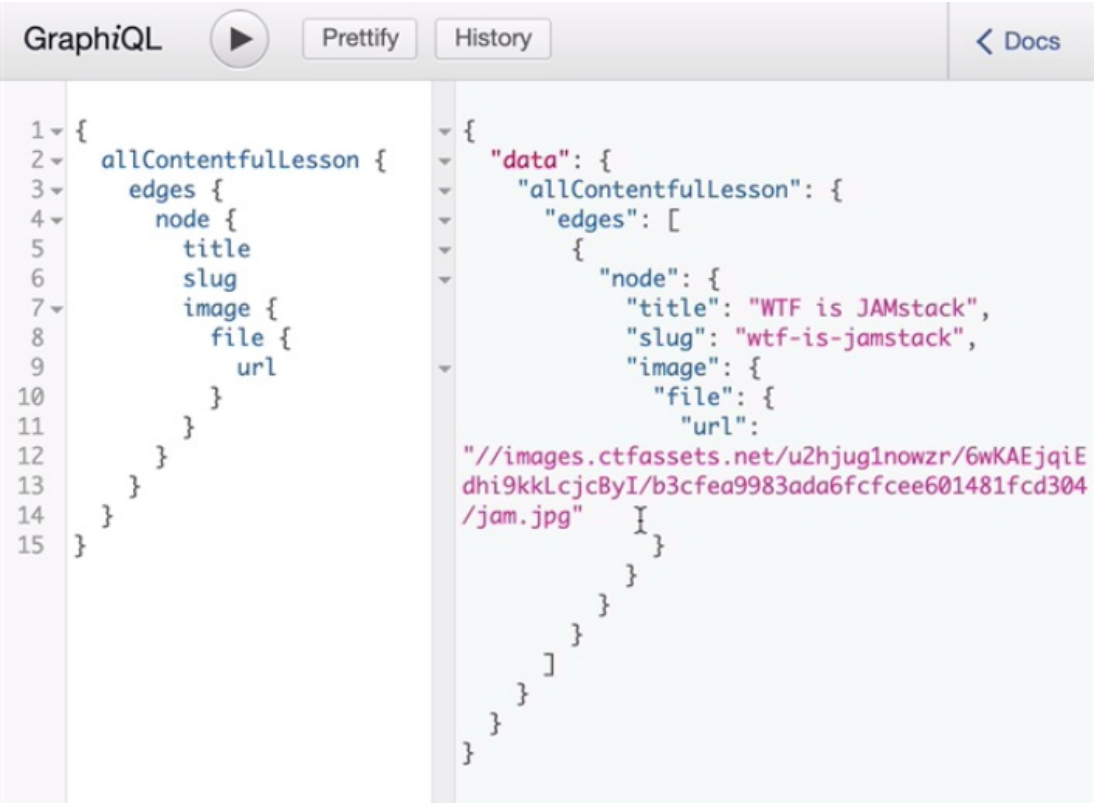


00:11 First, we need to define the GraphQL query that gets all the lessons. To test that, we can use GraphiQL before we add the query to our code.

00:27 In here, we can type `allContentfulLesson`. Inside of that, we'll have `edges`. Inside of edges, we have `node`. Inside the node, this is where we get the fields that we need.

00:46 For the list, we will need the `title`. We will need the `slug` to link to the detailed page of this lesson. We will need the `image`. Inside of the image, we will get the `file` property which contains the `URL` property.

01:07 Let's test this.



You can see here we indeed are getting the data that we want. Let's copy our query and go to the `index.js` page. Here we need to export the `const`. We'll call it query. This will be GraphQL. Inside of that, we will paste the query that we already tested.

```
export const query = graphql`
  {
    allContentfulLesson {
      edges {
        node {
          title
          slug
          image {
            file {
              url
            }
          }
        }
      }
    }
  }
```

01:37 Let's `import { graphql } from "gatsby"`.

```
import { graphql } from "gatsby"
```

Once the query is done, Gatsby will pass in the data in the props. Here, we can extract data. Inside of `data`, we will have `allContentfulLesson`.

```
const IndexPage = ({ data: { allContentfulLesson
} })
```

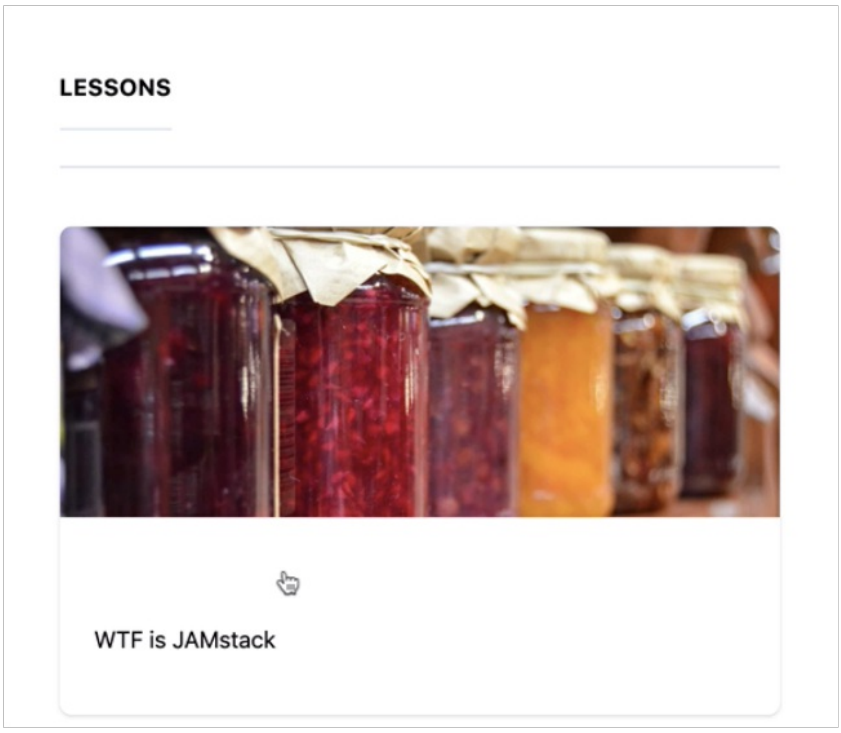
01:58 We have the data ready. Let's render it. We will have a `div` with our class names. We will look through all the `edges` inside of `allContentfulLesson` and render this card component.

```
<div className="flex flex-wrap -mx-3">
  {allContentfulLesson.edges.map(({ node }) => (
    <Card
      node={{ ...node, slug:
`/lessons/${node.slug}` }}
      key={node.id}
    />
```

02:16 Of course, here, let's import our `Card` component.

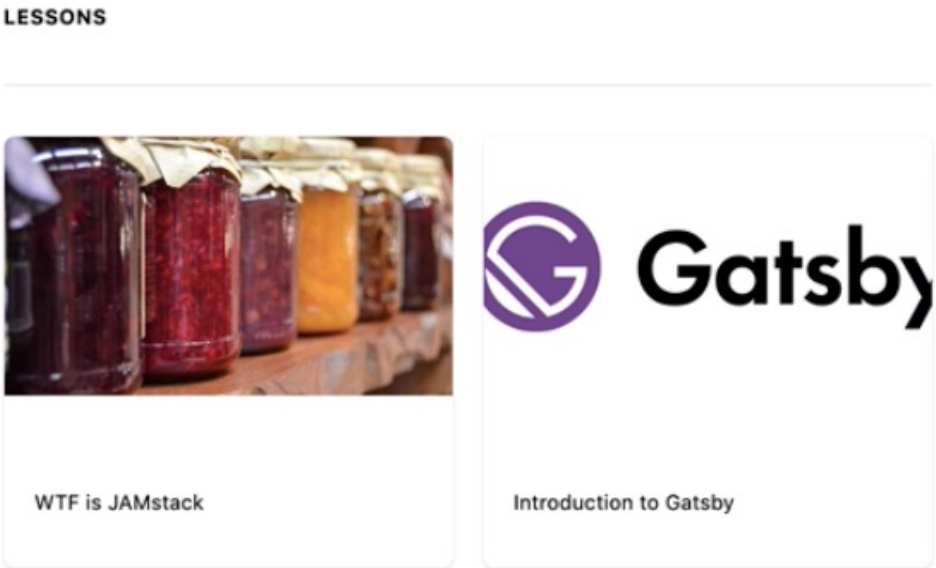
```
import Card from "../components/card"
```

If we save and go back to our website and scroll down, you can see here we are listing all the lessons.



Programmatically create Gatsby pages from Contentful data

00:00 Here, I have my Gatsby website, and it's listing all the lessons that are coming from Contentful.



If we click in one of the lessons, you can see here, Gatsby is giving us a 404 page. That's because we need to create a page for every lesson to show its details.

00:20 To do that, we need to go to the `gatsby-node.js` and write some code here to create the pages. First, we need to export a `createPages` function. We can then extract the `createPage` from the actions object. To be able to create a page, we need a path to a template, which is basically a React component.

```
exports.createPages = ({ graphql, actions }) =>
{
  const { createPage } = actions
  const lessonTemplate =
path.resolve(`src/templates/lesson.js`)
```

00:50 We will create the `lesson.js` later.

Let's import path in here. Now, we can use GraphQL to query the Gatsby data for all the Contentful lessons.

```
const path = require(`path`)

exports.createPages = ({ graphql, actions }) =>
{
  const { createPage } = actions
  const lessonTemplate =
path.resolve(`src/templates/lesson.js`)
  const instructorTemplate =
path.resolve(`src/templates/instructor.js`)
  return graphql(`
    {
      allContentfulLesson {
        edges {
          node {
            slug
          }
        }
      }
      allContentfulInstructor {
        edges {
          node {
            slug
          }
        }
      }
    }
```

01:07 The GraphQL function returns a promise that will contain our `result`, and inside of the result, we need to check for `errors`. If so, we `throw result.error`.

```
`).then(result => {
  if (result.errors) {
    throw result.errors
  }
```

Otherwise, we can create our pages. To do that, we need to loop through all the edges and create a page for every lesson.

01:33 The `createPage` will accept the `path` the `component`, and a `context` for additional data.

```
result.data.allContentfulLesson.edges.forEach(edge => {
  createPage({
    path: `/lessons/${edge.node.slug}`,
    component: lessonTemplate,
    context: {
      slug: edge.node.slug,
    },
  })
})
```

Let's save this and create our lesson template. Let's go to `src` and create a new folder. Call it `templates`. Inside of `templates`, we need to create a file called `lesson.js`.

01:53 Let's do some imports first. We need `React`, we need `graphql` from `gatsby`, we need the `Layout` component, and the `SEO` component.

```
import React from "react"
import { graphql } from "gatsby"
import Layout from "../components/layout"
import SEO from "../components/seo"
```

Before we render anything, we need to get some additional data for the lesson detail.

02:09 We need to export a GraphQL query. This will get the lesson query and use the slug that we passed in the component context when you create the page, and get all the data that we need.

```
export const query = graphql`
  query lessonQuery($slug: String!) {
    contentfulLesson(slug: { eq: $slug }) {
      title
      body {
        json
      }
      seo {
        title
        description
      }
    }
  }
`
```

Once the query is successful, we will get this `data` object that has all the data, so we can fill in our components.

```
function Lesson({ data }) {
  return (
    <Layout>
      <SEO
        title={data.contentfulLesson.seo.title}
        description=
{data.contentfulLesson.seo.description}
      />
      <div className="lesson__details">
        <h2 className="text-4xl">
{data.contentfulLesson.title}</h2>

{documentToReactComponents(data.contentfulLesson
.body.json, {
          renderNode: {
            [BLOCKS.HEADING_2]: (node, children)
=> (
              <h2 className="text-4xl">
{children}</h2>
            ),
            [BLOCKS.EMBEDDED_ASSET]: (node,
children) => (
              <img src=
{node.data.target.fields.file["en-US"].url} />
            ),
          },
        })}
      </div>
    </Layout>
  )
}
```

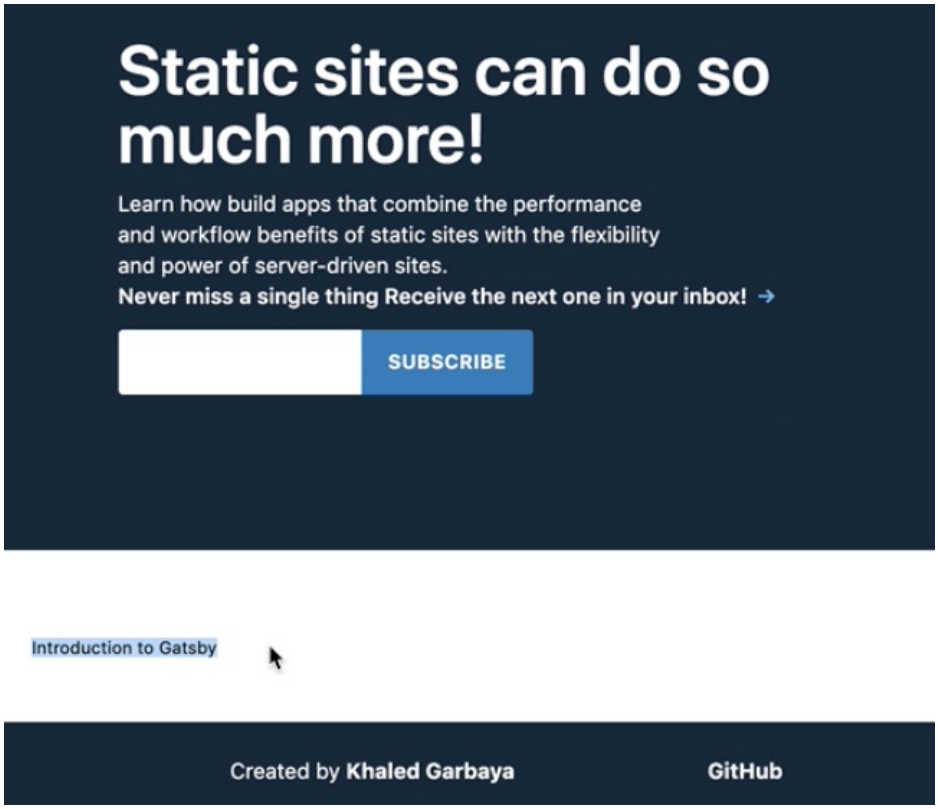
02:41 Finally, let's export the `Lesson` component.

```
export default Lesson
```

Let's hit save, and we need to restart our server.

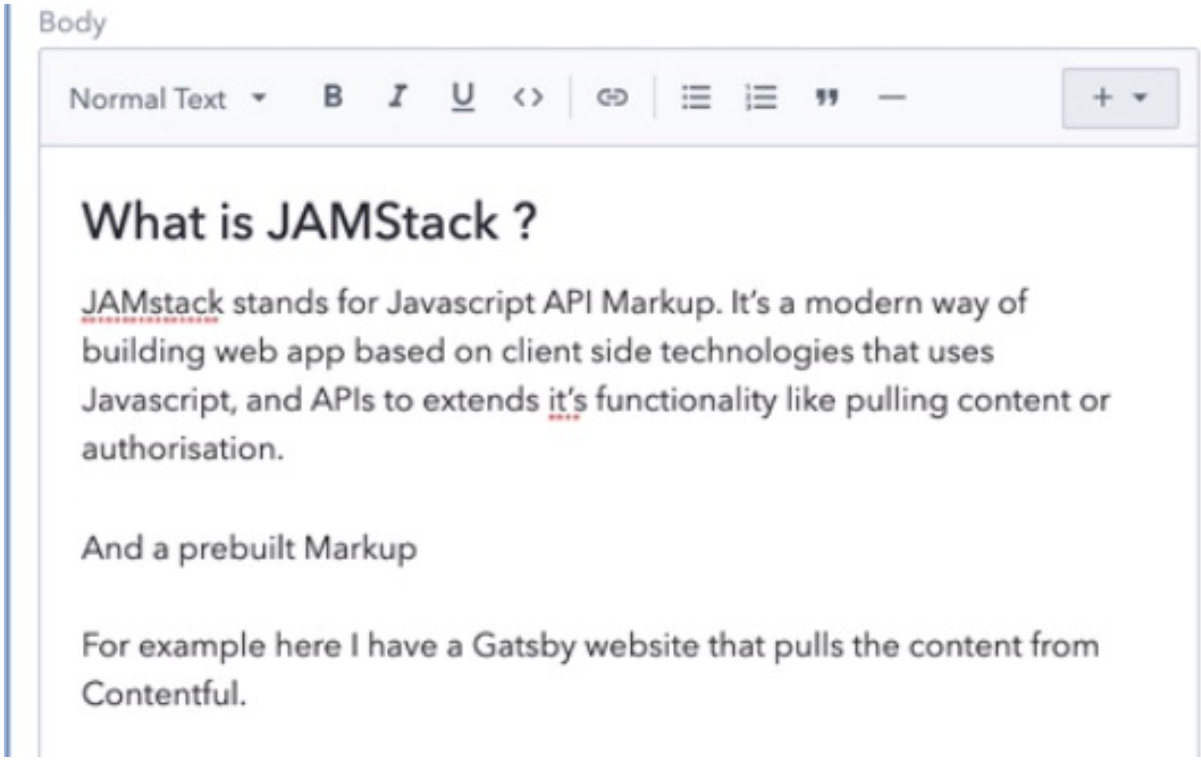
```
npm run develop
```

Now, if we refresh, go to the one of the lessons, and scroll down, you can see here the title of our lesson. If we check the second one, indeed, we can see its title.

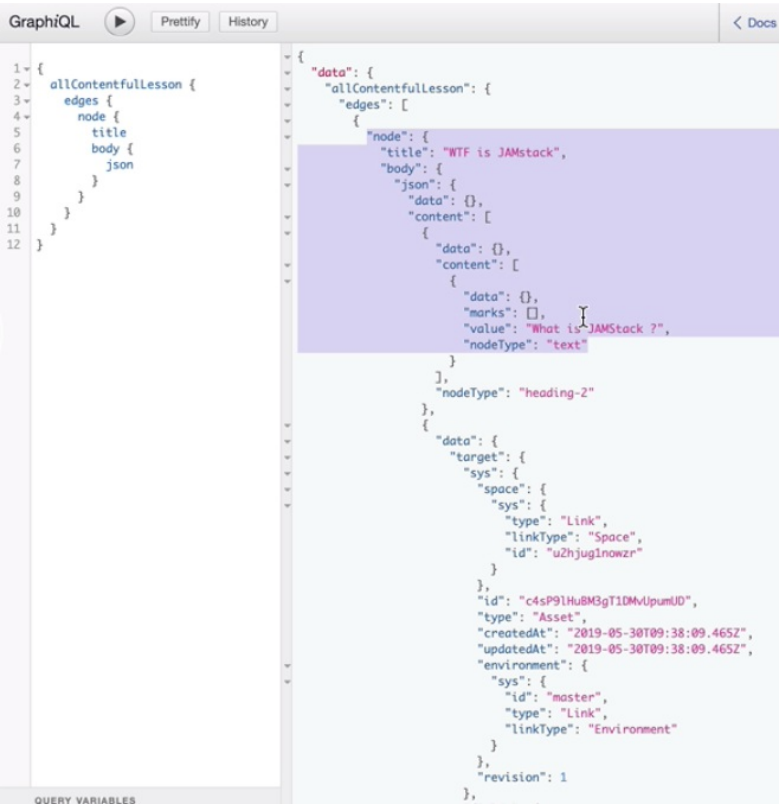


Render Contentful rich text in Gatsby

00:00 We have our Gatsby website, listing lessons coming from Contentful. If we click in one of the lessons, we can see the detail, but here, we're only showing the title. We would like to show more text from this body, which is a rich text field.



00:19 Let's take a look first at how this data is sent to us. If we go to the GraphiQL, and then request the body, inside of the body, we can see JSON. In the result here, we have all our nodes, and you can see the type and all the content.



00:41 We need a way to parse this JSON to React components. To do that, we can use the rich text React renderer. First, let's stop our server, and then:

```
npm i @contentful/rich-text-react-renderer
@contentful/rich-text-types
```

01:06 In the lesson.js here, we need to add the `body` to the `query`. We require the `json` data from the body.

```
export const query = graphql`
  query lessonQuery($slug: String!) {
    contentfulLesson(slug: { eq: $slug }) {
      title
      body {
        json
      }
      seo {
        title
        description
      }
    }
  }
```

We need to import the `documentToReactComponents` function from the rich text React renderer.

```
import { documentToReactComponents } from
"@contentful/rich-text-react-renderer"
```

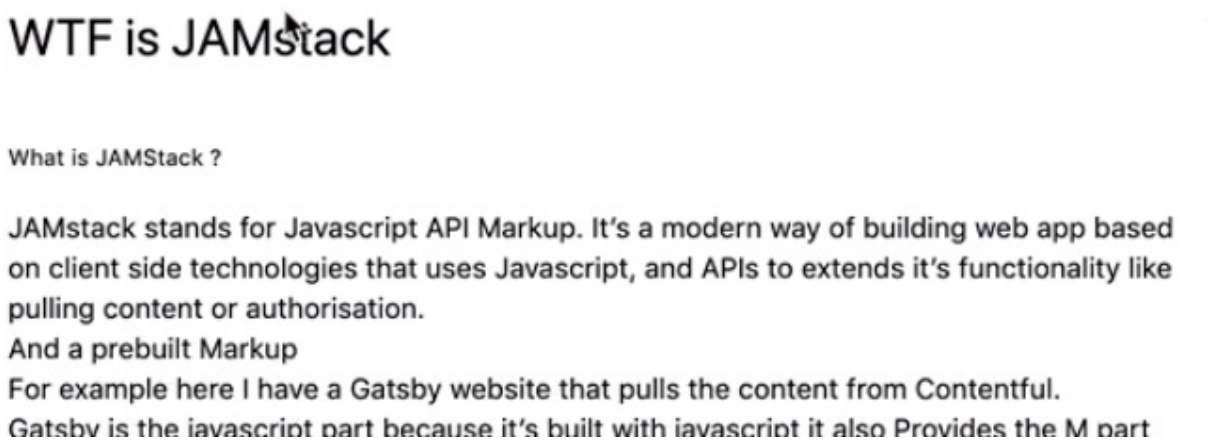
We take this function, and in the markup here, we give it the data from the body. It will be `data.contentfulLesson.body.json`.

```
<div className="lesson__details">
  <h2 className="text-4xl">
{data.contentfulLesson.title}</h2>

{documentToReactComponents(data.contentfulLesson
.body.json)}
```

01:46 If we save now and run our server again, and once we refresh, we can see indeed here we have the content.

```
npm run develop
```



There is a little problem here. If we inspect this element, for example, it's an h2. We want to apply to it the same class name as the title in here. To do that, the `documentToReactComponent` accepts a second argument to add additional styling and markup to every node type.

02:25 Let's first import `BLOCKS` from `@contentful/rich-text-types`.

```
import { BLOCKS } from "@contentful/rich-text-types"
```

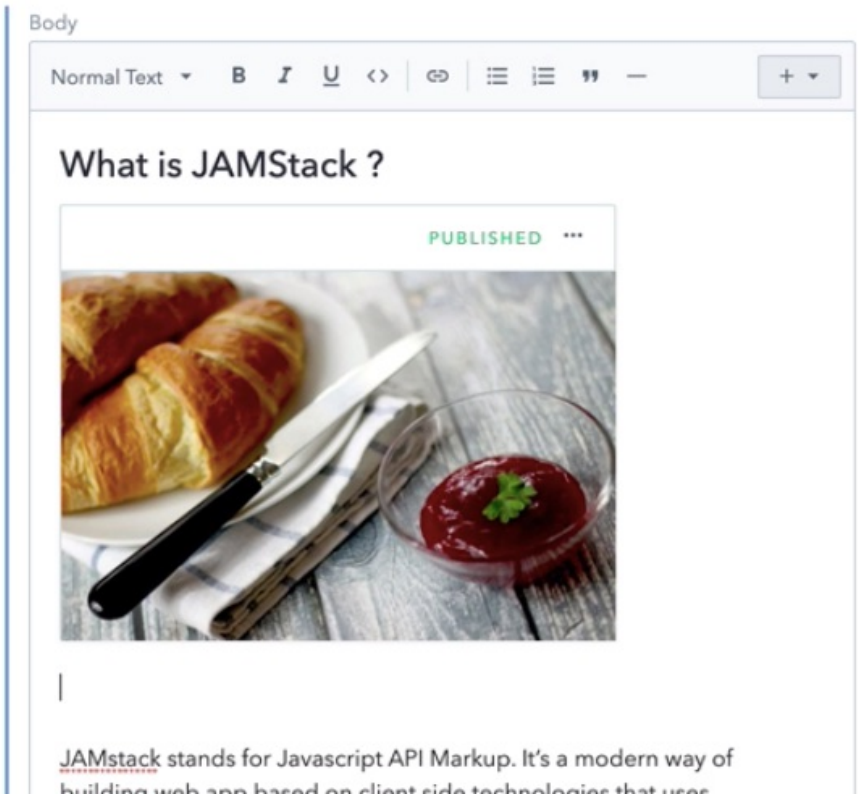
And we add here a second argument, an object configuration. For every h2, we will receive this callback, and then we can return how it will show up.

```
<h2 className="text-4xl">
{data.contentfulLesson.title}</h2>

{documentToReactComponents(data.contentfulLesson
.body.json, {
    renderNode: {
      [BLOCKS.HEADING_2]: (node, children) => (
        <h2 className="text-4xl">{children}</h2>
      )
```

02:42 What we'll take here is the same content, but we will wrap it into an h2 with a class name `"text-4xl"`. Let's save. You can see here our headings are fixed.

Let's go ahead and add an image in here. It will be an embedded asset. We'll pick this one I already uploaded.



03:06 Let's publish. Let's stop our server here, and then run it again. You can see here there is a problem. We're not seeing our image. That's because the renderer does not display images by default, mainly because it's an asset, and an asset can be something else different than an image.

03:33 We can do the same, like we did with headings, and then render the correct element. Let's go to the options here, and then add the `BLOCKS.EMBEDDED_ASSET`, and return an image with the correct URL. Let's save this.

```
 [BLOCKS.EMBEDDED_ASSET]: (node, children) => (
  <img src={node.data.target.fields.file["en-US"].url} />
),
```

03:54 We have an error. That's probably because of cache problems. Let's remove the cache and public folder, and then run again.

```
rm -rf .cache public
```

```
npm run develop
```

You can see here that we are indeed seeing our image, and all the headings are rendered correctly.

What is JAMStack ?

JAMstack stands for Javascript API Markup. It's a modern way of building web app based on client side technologies that uses Javascript, and APIs to extends it's functionality like pulling content or authorisation.
And a prebuilt Markup
For example here I have a Gatsby website that pulls the content from Contentful.
Gatsby is the javascript part because it's built with javascript it also Provides the M part

04:15 That's how you render the rich text data from Contentful into Gatsby.

Use Graphql backreference to avoid circular dependencies between Content model

00:00 I've gone ahead and created pages for each instructor that we have in Contentful. I did the same process as the lessons. Here, we query all the instructors, and then look through the result and create a page for every instructor.

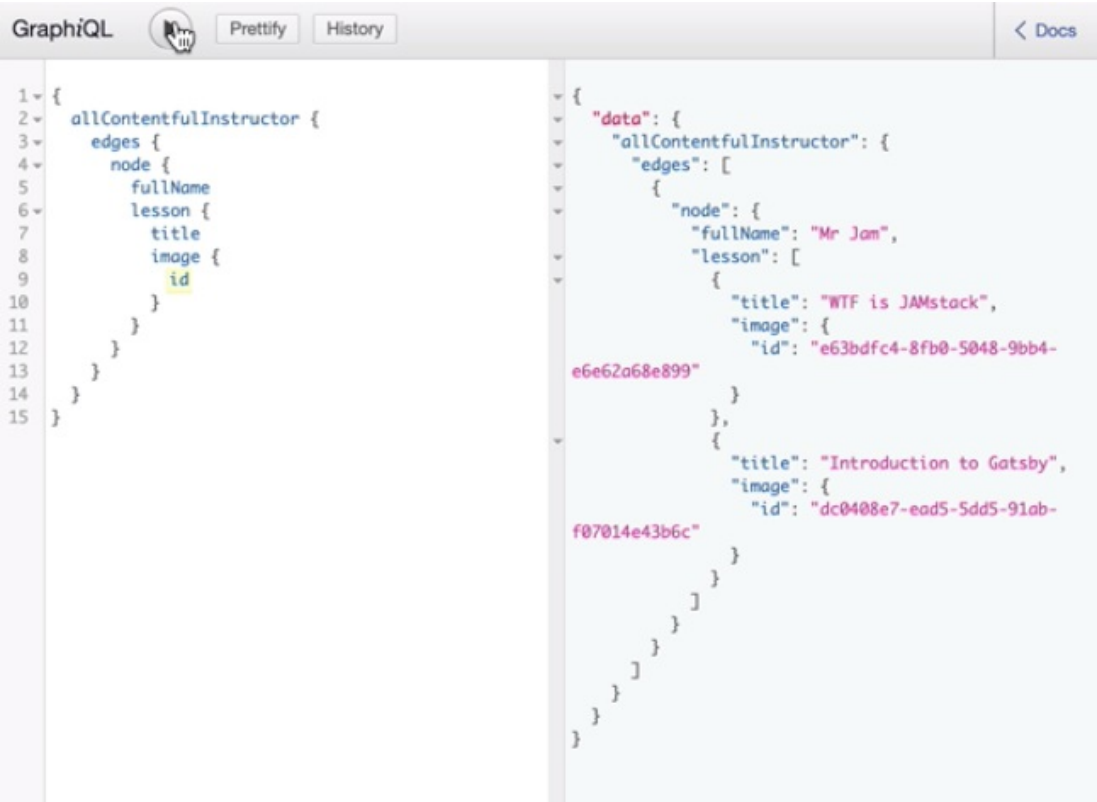
00:20 This will be based on this React component, which is the template. I'm simply now displaying the `image`, the `name`, and the `description`. It would be nice if we can display the lessons that are created by this instructor.

00:36 Let's check our content modules first. If we go to the lesson, you can see here that every lesson is linking to an instructor, but if we go to an instructor, we have no way of knowing which lesson is made by this instructor.

00:55 We could create however a new reference in here and link to a lesson, but this will create a circular dependency, and it's hard to maintain. A better way of doing this is using back references that are created by Gatsby in the GraphQL node.

01:12 To check that out, let's go to GraphiQL and query for all the instructors. In here, we can get the `edges`, and then the `node`. We can get things like `fullName`. This is the direct field that belongs to the instructor.

01:39 If we type `lesson`, you can see here that we have access to the lesson, even if it's not linked directly to the instructor in Contentful. Here, we can grab stuff like the `title`, the `image`, and so on. Let's do the same in our code and render a list of lessons.



02:02 First thing we need to do is to update this query that we already exported. After `website`, let's add the `lesson`, get the `id`, the `title`, the `slug`, and the `image`.

```
      bio
      website
      lesson {
        id
        title
        slug
        image {
          file {
            url
          }
        }
      }
    }
  }
```

Now that's available for us, let's render it. For that, we need to check first if the lesson is not `null`. Otherwise, we look through that lesson array and we render it. We will use the same `Card` component that we use in our index page.

```
{data.contentfulInstructor.lesson && (
  <div className="border-t my-6">
    <h2 className="text-4xl text-grey-dark my-
6">Lessons</h2>
    {data.contentfulInstructor.lesson.map(node
=> (
      <Card
        node={{ ...node, slug:
`/lessons/${node.slug}` }}
        key={node.id}
      />
    ))}
  </div>
```

02:40 Now that's done, let's save and go back again to the instructor page. In here, you can see lessons, and these are all the lessons that are attached to this instructor. We can of course click in here, and this will take us to the lesson detail page that we have.

Deploy a Gatsby website on Netlify

00:00 In GitHub, let's create a new repository. Here, we will name it `jamstacktutorials`. We can create the repository. Now, let's follow these instructions. Mainly, we'll add this remote.

00:30 Go to the command line. Here, we will paste this. Hit enter.

```
git remote add origin git@github.com:
Khaledgarbaya/jamstacktutorials.git
```

Now that we have the remote added, let's add everything and commit.

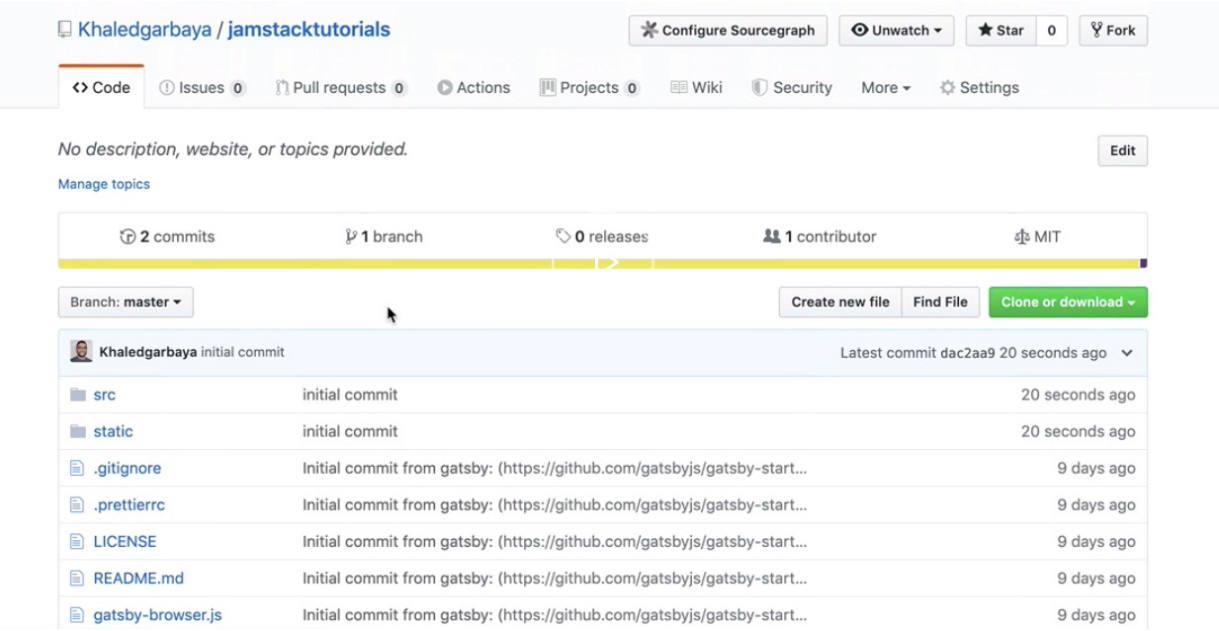
```
git add
```

```
git commit -m 'initial commit'
```

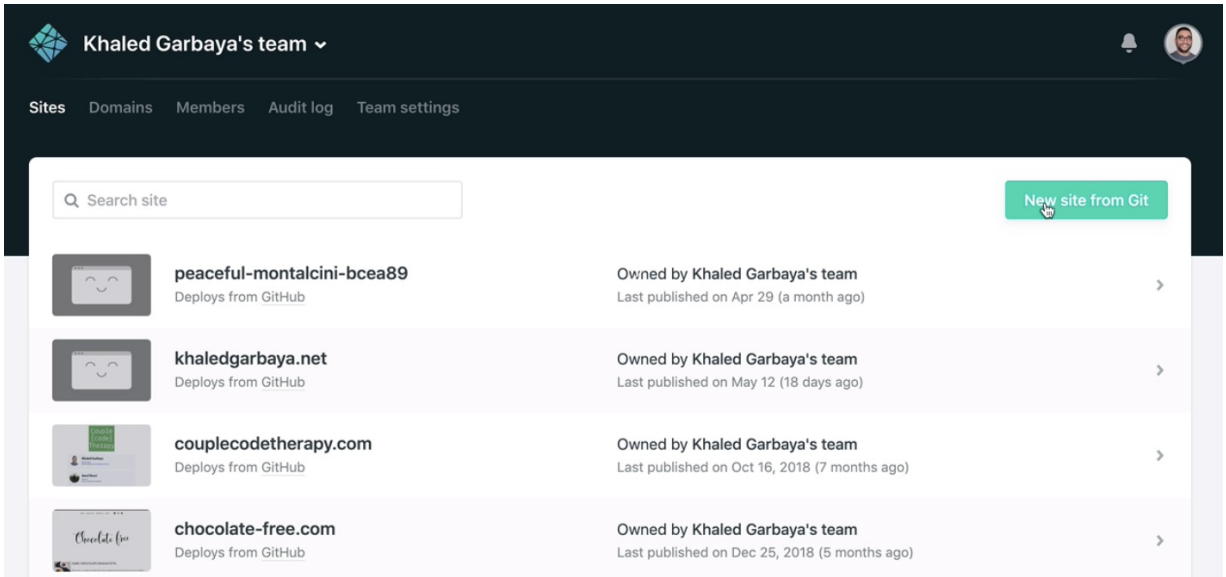
Now, let's push.

```
git push -u origin master
```

00:51 Let's go back to our repository. Refresh. Now, we have all the code in here committed.

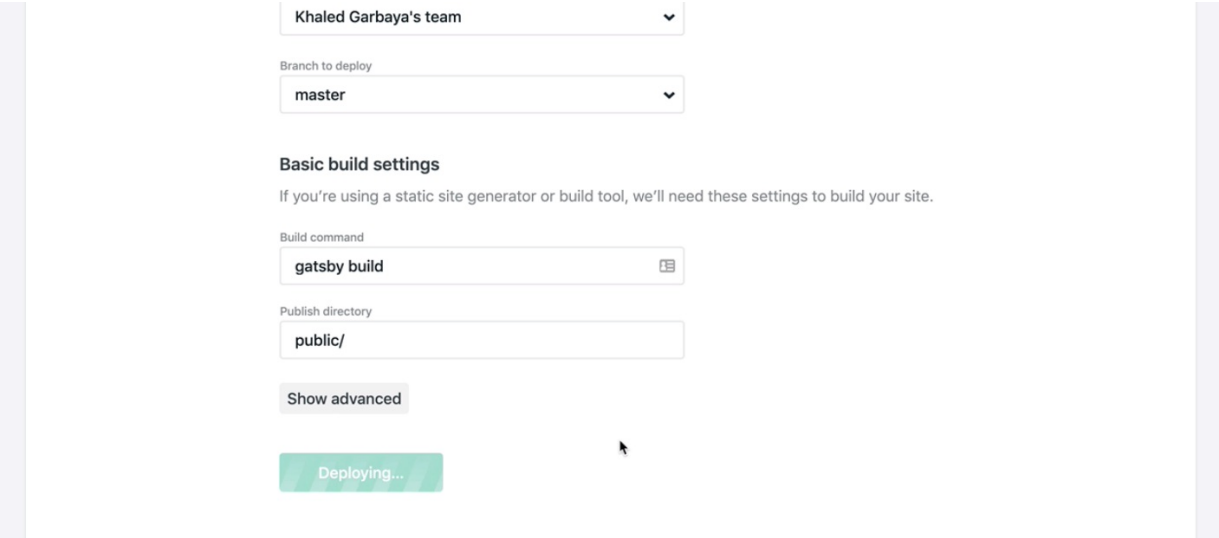


Let's go to Netlify. After you log in, this is your main dashboard. You click new site from Git. Here we click GitHub.

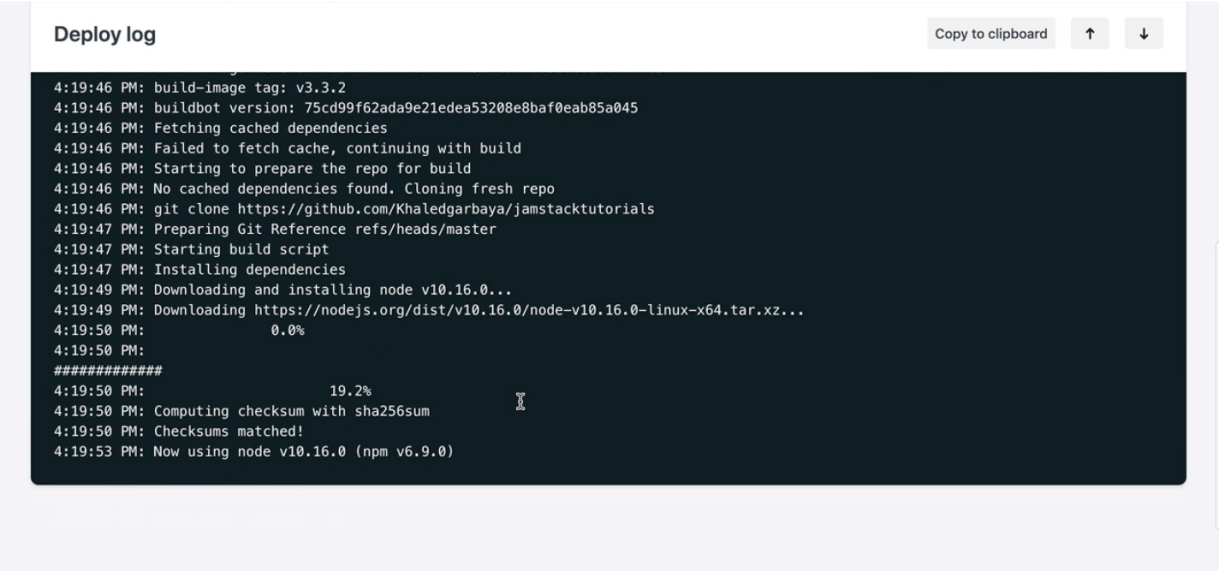


01:15 After I am authorized, I can look for the repository that they have. If we type just JAMstack, it will be more than enough.

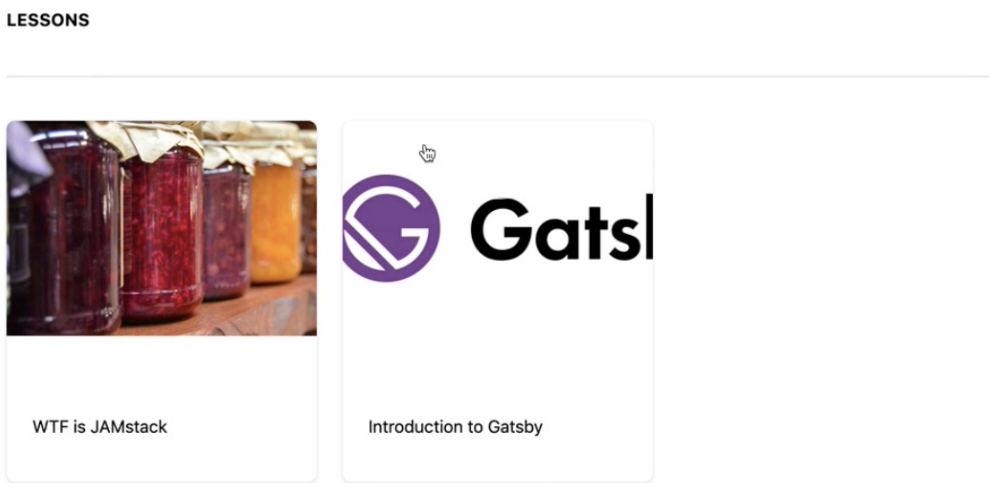
01:29 This is our repository. You can see here that it's a Gatsby project. It will run this Gatsby build command from my server. Let's deploy the site.



You can see the logs. This is our website being built.



01:55 Now, our site is live. We can click on the preview button. You can see here this is our website.

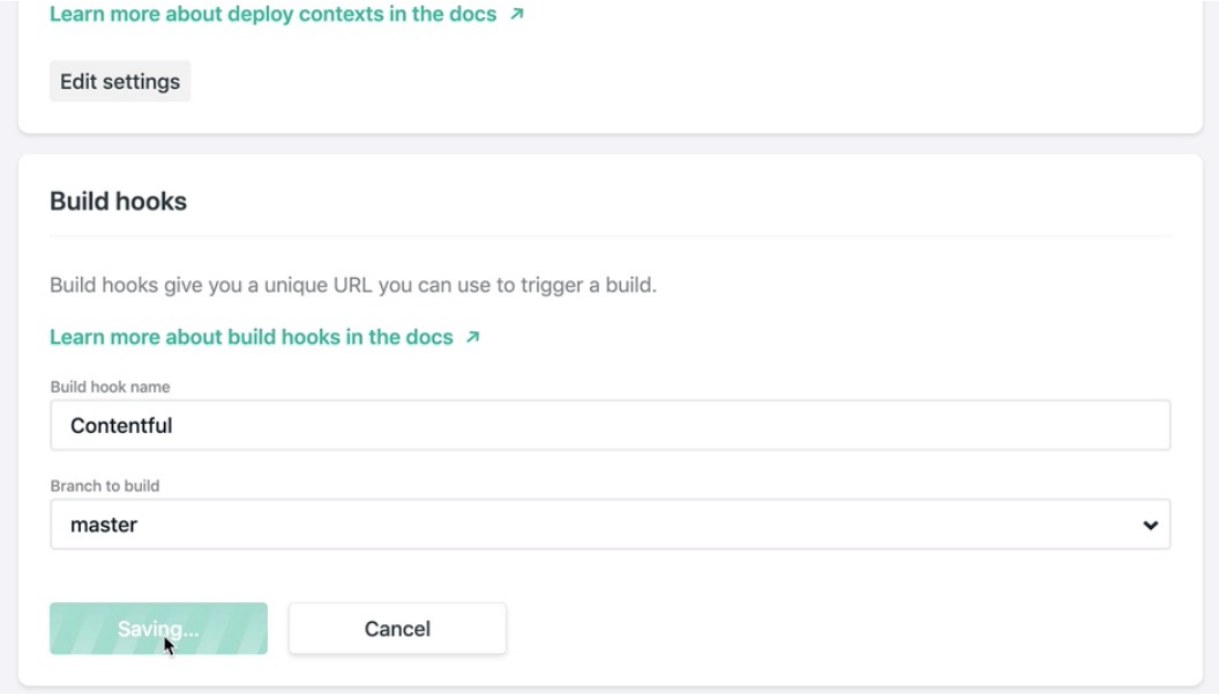


We can navigate and go to the lessons and read everything. That's how you deploy your Gatsby website to Netlify.

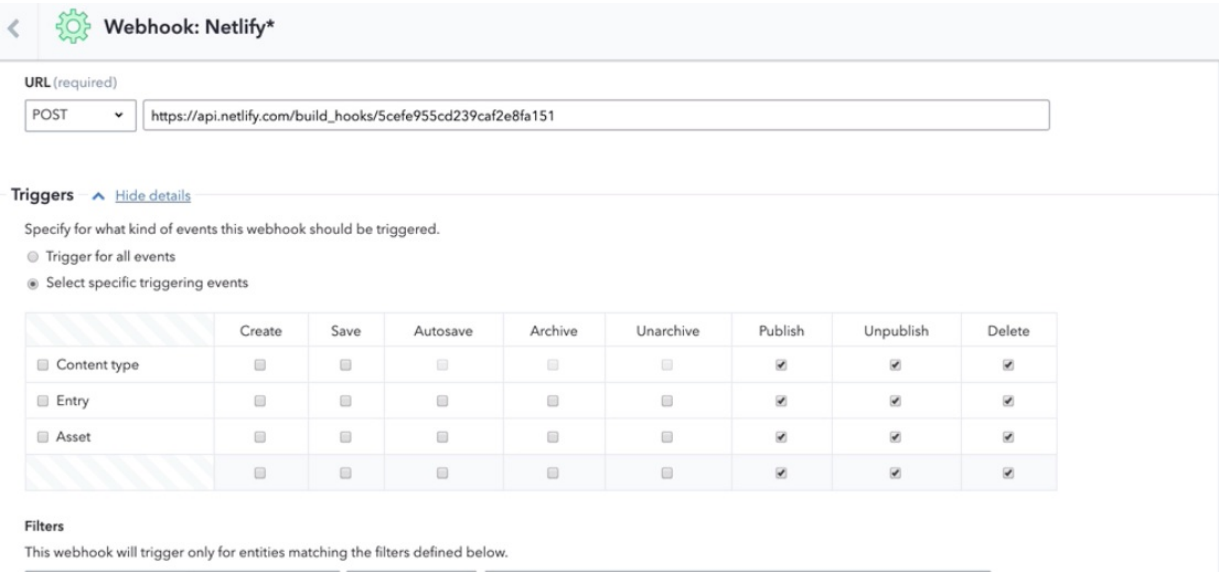
Trigger Netlify Builds when content changes in Contentful

00:00 Netlify will rebuild our website whenever we push new code, but we also want to trigger the rebuild whenever we do content changes. We can do that using what's called webhooks. Let's go to

deploy settings, and in build hooks, you can click add build hook. Let's call this `Contentful` and hit save.



00:28 We grab this URL and go to Contentful. In there, we go into settings, webhooks, and we add webhook. This one, we'll call it `Netlify`. We will paste our URL in here, and we select specific events that will trigger this rebuild.



00:53 We want that whenever we publish a new `Entry`, a `Content Type`, or an `Asset`. We can go to `Publish` and check these. Also, for `Unpublish` and `Delete`. Let's hit save, and to test our

webhook, we can go to the content. Go to this lesson.

01:23 Let's change something. Let's remove this question mark, and we hit publish.
Now, when we go back to Netlify, and go to deploys, you can see that here, it's triggered by Contentful. Now, it's rebuilding our website.

01:50 When we preview this, we should see the new content reflected in the website.