Chapter 1

Day One with Sanity Studio

`Sanity Studio` provides content creators with tailored editing interfaces that match the unique ways content drives your business. Built as open-source, the Studio acts as a central hub for content creation and operations for your composable business.

A fully customizable content workspace that mirrors your business and unleashes velocity and creativity. The most flexible content workspace.

In this course, you'll set up a Sanity Studio from scratch. You'll create content types for events, venues, and artists. Improve the authoring experience, query content with GROQ, and render it to a front end.

1.1 Lesson 1 - Prerequisites

To complete this course you will need the following:

- A free Sanity account to create new projects and initialize a new Sanity Studio. If you do not have yet have an account, running the command below will prompt you to create one.
- Some familiarity with running commands from the terminal. Wes Bos' Command Line Power User video course is free and can get you up to speed with the basics.
- Node and npm installed (or an npm-compatible JavaScript runtime) to install and run the Sanity Studio development server locally.
- Some familiarity with JavaScript. The code examples in this course can all be copied and pasted and are written in TypeScript, but you will not need advanced knowledge of TypeScript to take advantage.

If you're stuck or have feedback on the lessons here on Sanity Learn join the Community Slack or use the Feedback form at the bottom of every lesson.

1.1.1 Create a new project

```
npm create sanity@latest – –template clean –create-project ”Day one with Sanity” –dataset production –typescript –output-path day-one-with-sanity
```

This command creates a new project and dataset, and sets up a new project folder with all the files and dependencies you need to get started.

- A dataset is a collection of content within a project that's hosted in the Sanity Content Lake.
- A project can have many datasets and is also where you'd configure other project-level settings like members, webhooks, and API tokens.

Project-level settings are configured in sanity.io/manage.

The command above prepares code for a Sanity Studio that connects to the new project and dataset. Even though the configuration for the Studio is in your local development environment, all the content will be automatically synced to your hosted dataset.

In the following lessons, you'll configure the Studio locally by modifying its code and, when ready, deploy it for other authors. In this course, you will deploy the Studio to Sanity's provided hosting, but know that you can host it with most web hosting providers.

1.2 Lesson 2 - Getting started

With a new Sanity project created and a new Studio project installed, you're ready to get started with local development!

With the Studio files and dependencies installed in the last lesson, you are now ready to start the local development server for Sanity Studio. It allows you to open the Studio in a browser and instantly see the changes reflected when you update its configuration and when you customize it later in this course.

From the command line and inside the folder where your Studio was installed, start the development server by running this command:

```
npm run dev
```

Sanity Studio uses Vite as its default development and build tooling to provide rapid updates as you make configuration changes. Also consider that the Studio is "just" a React dependency and can be used with most modern development tooling and even be embedded in most modern web frameworks.

1.2.1 Log in to the Studio

Open the Studio running locally in your preferred browser on http://localhost:3333.

You should now see a screen like the one below prompting you to log in to the Studio. Use the same service (Google, GitHub, or email) that you used to access sanity.io/manage when you authenticated with the CLI.

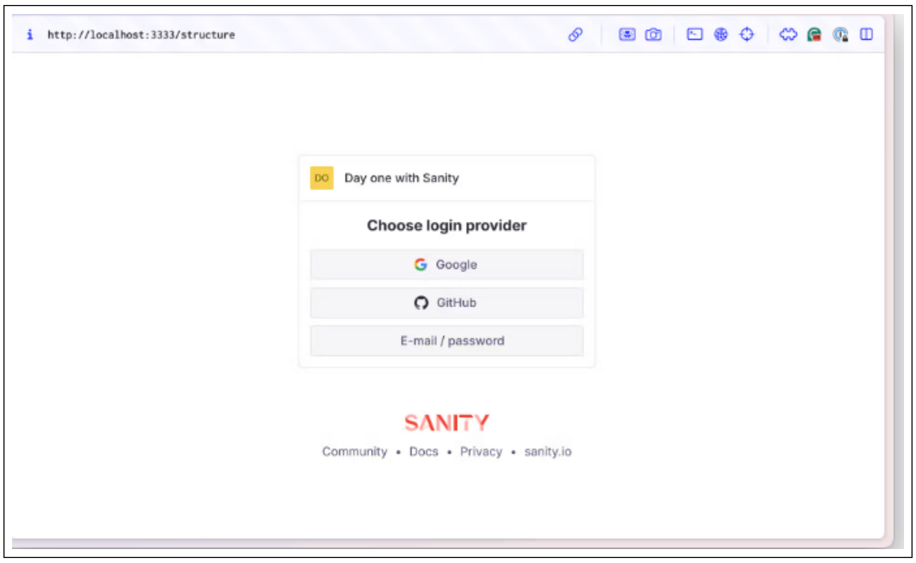


Figure 1.1: Sanity login

Why is it empty? The central point of all your Studio's configuration takes place in the file sanity.config.ts, including declaring all your schema types.

Take a look at your configuration file. It exports a helper function defineConfig by default and contains a single workspace.

`sanity.config.ts`

```
// ...all other imports
import schemaTypes from './schemaTypes'
```

```
export default defineConfig({
  // ...all other config
  schema: {
    types: schemaTypes,
  },
})
```

In this file, you can see the schema types for this config are imported from another file schemaTypes/index.ts, which currently contains an empty array.

`schemaTypes/index.ts`

```
export const schemaTypes = []
```

So when you create new schema types, they must be added to this array, which is then loaded into the Studio configuration. Let's do that!

1.3 Lesson 3 - Creating a schema

Learn how to configure a schema for Sanity Studio that defines your content model and builds out an editorial interface. Let's start with some foundational knowledge of how your Sanity Studio and Content Lake are integrated and how to think about the "schema." You can skip right to the code part and return to this later if you prefer to be hands-on first.

1.3.1 What is a schema?

The schema for a Sanity Studio workspace defines what document types and fields can be accessed by a content team. In the schema configuration you define much of the editorial experience for these documents and fields, like field descriptions, validation, initial value, and so on.

If you have used other CMSes, the "schema" will be similar to what is commonly referred to as "content model," "fields and entities," "custom types," "advanced custom fields," etc.

It's important to note that the schema is confined to a Studio workspace, not to the Sanity Content Lake dataset, which is considered "schemaless." That means that you can store pretty much any JSON document in it, as long as it has a value for the _type property.

As with everything, this has advantages and trade-offs. An advantage is that you can store more content in your dataset without being constrained to a specific Studio schema. A trade-off is that if you update content with APIs, you must recreate whatever data validation you have for documents and fields in the Studio. A large part of configuring the schema is configuring the content types that one can create and edit in the Studio. This is also where you shape how and what content you can query in applications.

1.3.2 Planning your schema types

In a production project, you should first consult with your wider team of designers, content creators, and others to work with them to design a content model that best represents your business and your goals. In the following lessons, you'll be building the content model from the Hello, Structured Content course. Configuring schema types to represent a live music production company.

1.3.3 Create a new document type

Create and open a new file in your Studio's schemaTypes folder called eventType.ts. Copy-paste the following code into it:

`schemaTypes/eventType.ts`

```
import {defineField, defineType} from 'sanity'

export const eventType = defineType({
  name: 'event',
  title: 'Event',
  type: 'document',
  fields: [
    defineField({
      name: 'name',
      type: 'string',
    }),
  ],
})
```

The *defineField* and *defineType* helper functions in the code above are not required, but they provide autocomplete suggestions and can catch errors in your configuration in code editors with TypeScript tooling.

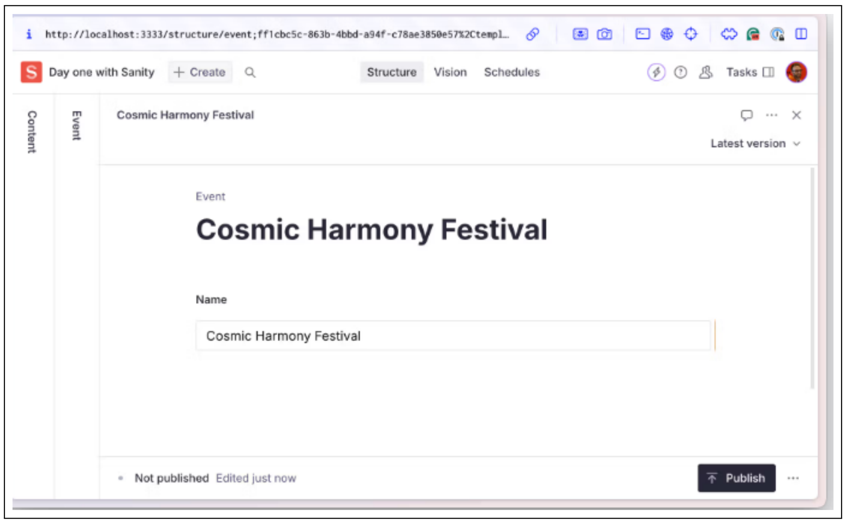
Now you can import this document type into the schemaTypes array in the index.ts file in the same folder.

`schemaTypes/index.ts`

```
import {eventType} from './eventType'

export const schemaTypes = [eventType]
```

When you save these two files, your Studio should automatically reload and show your first document type. You can and should create a new "event" document.


http://localhost:3333/structure/event;ff1cbc5c-863b-4bbd-a94f-c78ae3850e57%2Ctempl…
Day one with Sanity
+ Create
Structure
Vision
Schedules
Tasks
Content
Event
Cosmic Harmony Festival
Latest version
Not published Edited just now
Publish

Event
Cosmic Harmony Festival
Name
Cosmic Harmony Festival

Figure 1.2: New event

When you add content in the Name field, all your changes are automatically synced to your project's dataset in the Content Lake. Now, let's add some more document types with fields in them. Same procedure as with the event type: add new files, copy-paste the code into them, and import and add them to the schemaType array in index.ts.

`schemaTypes/artistType.ts`

```
import {defineField, defineType} from 'sanity'

export const artistType = defineType({
  name: 'artist',
  title: 'Artist',
  type: 'document',
  fields: [
    defineField({
        name: 'name',
        type: 'string',
      }),
    ],
  })
```

`schemaTypes/venueType.ts`

```
import defineField, defineType from 'sanity'

export const venueType = defineType({
  name: 'venue',
  title: 'Venue',
  type: 'document',
  fields: [
    defineField({
      name: 'name',
      type: 'string',
    }),
  ],
})
```

Notice how all these document types use singular names and titles. This is because the singular form makes sense in most contexts where these values are used. Later in this course, you will learn how to customize document lists to use plural names.

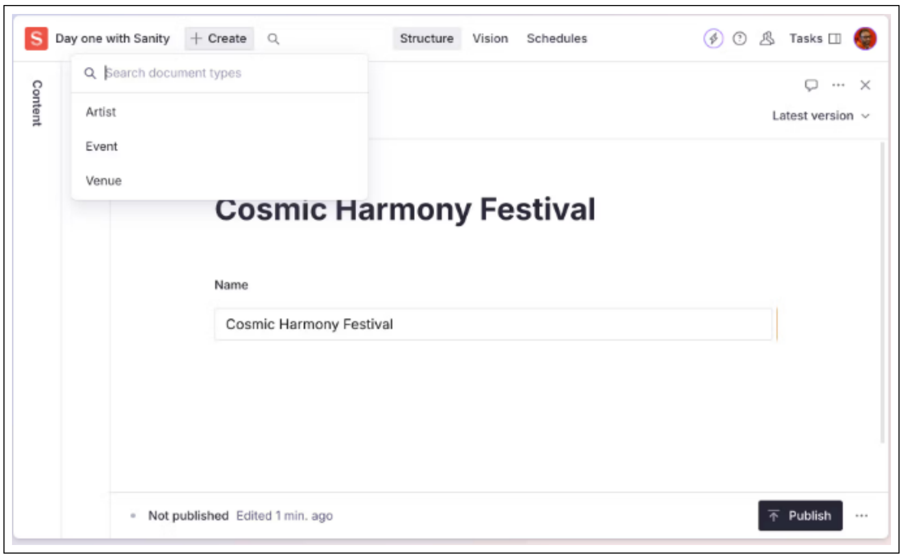
`schemaTypes/index.ts`

```
import {eventType} from './eventType'
import {artistType} from './artistType'
import {venueType} from './venueType'

export const schemaTypes = [artistType, eventType, venueType]
```

All these document types only have one field; you'll need to add more.

Before we go further, confirm in your Sanity Studio that you can create new Artist, Event and Venue type documents and that they all have a single field – name.



Cosmic Harmony Festival

Name

Cosmic Harmony Festival

Figure 1.3: New events

1.3.4 Adding familiar field types

Sanity Studio has the field types you'd expect for storing content in a JSON format. For example string, number, boolean, array, object, and more. In a typical project, the document types you create and the fields you add within them should be informed by conversations you've had with designers and content creators.

Using the Sanity schema docs as a guide, complete the fields we need for our project. See: *Schema / Field Types.*

Add the following fields to your event schema type. You will extend the configuration later to make their purpose clearer:

- slug: a slug type field
- eventType: a string type field
- date: a datetime type field
- doorsOpen: a number type field
- venue: a reference type field to the venue document type
- headline: a reference type field to the artist document type
- image: an image type field
- details: an array of block type fields
- tickets: a url field

Once complete, your eventType file should look like this:

`schemaTypes/eventType.ts`

```
import {defineField, defineType} from 'sanity'

export const eventType = defineType({
  name: 'event',
  title: 'Event',
  type: 'document',
  fields: [
    defineField({
      name: 'name',
      type: 'string',
    }),
    defineField({
      name: 'slug',
      type: 'slug',
    }),
    defineField({
      name: 'eventType',
      type: 'string',
    }),
    defineField({
      name: 'date',
      type: 'datetime',
    }),
    defineField({
      name: 'doorsOpen',
      type: 'number',
    }),
    defineField({
      name: 'venue',
      type: 'reference',
      to: [{type: 'venue'}],
    }),
    defineField({
      name: 'headline',
      type: 'reference',
      to: [{type: 'artist'}],
    }),
    defineField({
      name: 'image',
      type: 'image',
    }),
    defineField({
      name: 'details',
      type: 'array',
      of: [{type: 'block'}],
    }),
    defineField({
      name: 'tickets',
      type: 'url',
    }),
  ],
})
```

You can now compose and publish documents with multiple fields of varying data types, including a "reference" field that can relate one document with another. You could deploy this to content creators in its current state. It's a fully-functioning content management system!

`The details field as 'block content'`

You might notice that the details field appears as a block content (or "rich text") editor in the Studio. Any array type field that includes a block type will automatically change the UI for the field to this editor.

This is how Sanity Studio is designed for authoring and storing block content. Instead of saving block content and rich text in formats like Markdown or HTML as a string, Sanity Studio stores it in the open-source specification called Portable Text. This unlocks powerful querying and filtering capabilities in your projects and makes integrating across most platforms and frameworks easier.

1.3.5 Create some content

Lastly, create and publish at least three documents:

- One event document for an upcoming show
- One venue document for an imagined location
- One artist type document using that "really great band name" you thought of once

Composing content is now possible, but the editorial experience can be improved. You will work on that in the next lesson.

1.4 Lesson 4 - Improving the editorial experience

Elevate the basic editorial experience with field titles, descriptions, validation, conditional fields, field groups, and document list previews.

1.4.1 Consistency

Helping content creators create content consistently is simpler when they're presented with fewer options and helpful guardrails. Make slug creation simpler by adding a source for generating the slug field value.

`schemaTypes/eventType`

```
// Replace ”slug” in the array of fields:
defineField({
name: ’slug’,
type: ’slug’,
options: {source: ’name’},
}),
```

Text input fields can contain any string. Avoid accidental duplicates or misspellings by providing preset options.

Limit the eventType field to a few options by providing a list of values:

`schemaTypes/eventType`

```
// Replace ”eventType” in the array of fields:
defineField(
name: ’eventType’,
type: ’string’,
options:
list: [’in-person’, ’virtual’],
layout: ’radio’,
,
),
```

If new documents could benefit from sensible default values which are often correct, an initial value can be set at the field level.

Note that this won’t affect existing documents. But every new document will now start with this value.

Set an initial value for the doorsOpen field to 60.

`schemaTypes/eventType`

```
// Replace ”doorsOpen” in the array of fields:
defineField(
name: ’doorsOpen’,
type: ’number’,
initialValue: 60,
),
```

1.4.2 Adding context

Adding more context and intentionality to fields can be very helpful for content teams.

With only its name to describe it, this field fails to give the author context for its use. Adding descriptions to fields helps clarify their intention and guide the content creation experience.

Add a description to the doorsOpen field

1.5 Validation

Structured content is more trustworthy when it is validated. Validation rules can also give content creators more confidence to press Publish. See *Validation* in the documentation.

Make the slug field required.

`schemaTypes/eventType`

```
// Replace ”slug” in the array of fields: defineField( name: ’slug’, type: ’slug’, options: source: ’name’, validation: (rule) =¿ rule .required() .error(‘Required to generate a page on the website‘), ),
```

Note that the error message can be customized so that authors better understand the context of why a warning or error is being displayed.

Custom rules have access to the entire document so that fields may be validated against one another. They should return a string in case of an error or true once the field is considered valid.

Using a custom validation rule, make sure that virtual events do not have a venue.

`schemaTypes/eventType`

// Replace "venue" in the array of fields: defineField(name: 'venue', type: 'reference', to: [type: 'venue'], validation: (rule) =¿ rule.custom((value, context) =¿ if (value && context?.document?.eventType === 'virtual') return 'Only in-person events can have a venue'

return true),
),

1.6 Conditionally hidden and read-only

It can often be useful to hide less common or more complex fields until they are required. While hidden and readOnly can be set to true or false – they can also accept a function to apply some logic.

See *Conditional fields* in the documentation.

Hide the slug field if the name field is empty

`schemaTypes/eventType`

// Replace "slug" in the array of fields: defineField(name: 'slug', type: 'slug', options: source: 'name', validation: (rule) =¿ rule.required().error('Required to generate a page on the website'), hidden: (document) =¿ !document?.name,),

Set venue to readOnly if the field does not have a value and the event type is virtual

`schemaTypes/eventType`

// Replace "venue" in the array of fields: defineField(name: 'venue', type: 'reference', to: [type: 'venue'], readOnly: (value, document) =¿ !value && document?.eventType === 'virtual', validation: (rule) =¿ rule.custom((value, context) =¿ if (value && context?.document?.eventType === 'virtual') return 'Only in-person events can have a venue'

return true),),
Note that "hidden" only affects the Studio interface. Fields will still retain their values whether they are visible in the Studio or not.

1.7 Group fields together

The document form is still one long column of fields. Let's introduce a set of field groups (think of them like tabs) to tidy up the form. See *Field Groups* in the documentation.

Create two Groups in the event document schema: details and editorial.

`schemaTypes/eventType`

// Above the "fields" array
groups: [
name: 'details', title: 'Details',
name: 'editorial', title: 'Editorial',
],
Assign each field to one or more of these Groups.

`schemaTypes/eventType`

// Assign each field to one group
group: 'details',
// or several!
group: ['details', 'editorial'],
The image and details fields make the most sense in "editorial" and the rest in "details." You could customize further by adding an icon to each group and setting one active by default.

1.8 Document list previews

By default, documents are indicated with a plain icon and a "best guess" at the document's title. The first and simplest thing we can do is give documents of a certain type a unique icon so that every document list and search result will use them by default.

You may need to add the package @sanity/icons as a dependency for the following code snippet.

Import the Calendar icon and make it the default on all event-type documents

`schemaTypes/eventType`

```
import CalendarIcon from '@sanity/icons'

export const eventType = defineType( name: 'event', title: 'Event', icon: CalendarIcon, // ...all other settings
)
```

Let's go further! So that our documents are even more discoverable, you can update the preview property for our document type.

In preview, values are "selected" from the document and then fed into the document preview's title, subtitle, and media slots.

See *List Previews* in the documentation.

- Show the event name in title
- Show the artist's name in subtitle
- Show the image in the media

`schemaTypes/eventType`

```
// After the "fields" array preview: select: title: "name", subtitle: "headline.name",
media: "image",
,
,
```

Take a look at your document list now. It's much easier to discern the document types at a glance. There are other values in the document that would be useful in list previews. With the prepare function, you can modify values before returning them to the preview.

Update your event type's preview configuration to the below:

`schemaTypes/eventType`

```
// Update the preview key in the schema
preview:
select:
name: 'name',
venue: 'venue.name',
artist: 'headline.name',
date: 'date',
image: 'image',
,
prepare(name, venue, artist, date, image)
const nameFormatted = name ||'Untitled event'
const dateFormatted = date
? new Date(date).toLocaleDateString(undefined,
month: 'short',
day: 'numeric',
year: 'numeric',
hour: 'numeric',
minute: 'numeric',
)
: 'No date'
return
title: artist ? `$nameFormatted ($artist)` : nameFormatted,
subtitle: venue ? `$dateFormatted at $venue` : dateFormatted,
media: image ||CalendarIcon,
,
,
```

1.9 Review

Take a step back and compare the editorial experience with the simple form you had initially. Content creators can discern the different types of documents and more easily create trustworthy content.

You could go further by adding icons to the artist and venue-type documents. This is a much better experience.

Up to now, you have mostly configured the Studio using the built-in options. But you can take it even further and customize the Studio. That is what you will learn in the next lesson!