Astro Handbook

Flavio Copes

Updated July 31, 2026

Contents

Introduction to Astro

Astro is a great tool to build websites.

I use it for a ton of stuff and it's always my default choice when I'm building a website nowadays.

This is an Astro site. My blog is an Astro site. I have a ton of other Astro sites around. That's to say, I'm a big fan.

Why is Astro so dear to me?

It's its focus on static sites, in particular **content sites**, and specifically sites that use Markdown to manage content. It has a lot of features for sites with a lot of content to manage.

With a unique DX (developer experience) that makes it super nice to build and maintain a website.

It's also the perfect introduction to more complex tools, because Astro has components that use a syntax similar to JSX (used by React), but also supports embedding any kind of frontend framework to add more interactivity to your pages.

Sites are very fast to build, and most importantly very fast to the user, since the end result is a static site.

And we can easily host an Astro site on any popular static site hosting like Netlify or Cloudflare Pages.

Those are just a few reasons.

You can also use Astro to build a SaaS or a site with login and authentication and a database.

Almost everything is possible.

With that said, let's build our first Astro site!

Your first Astro site

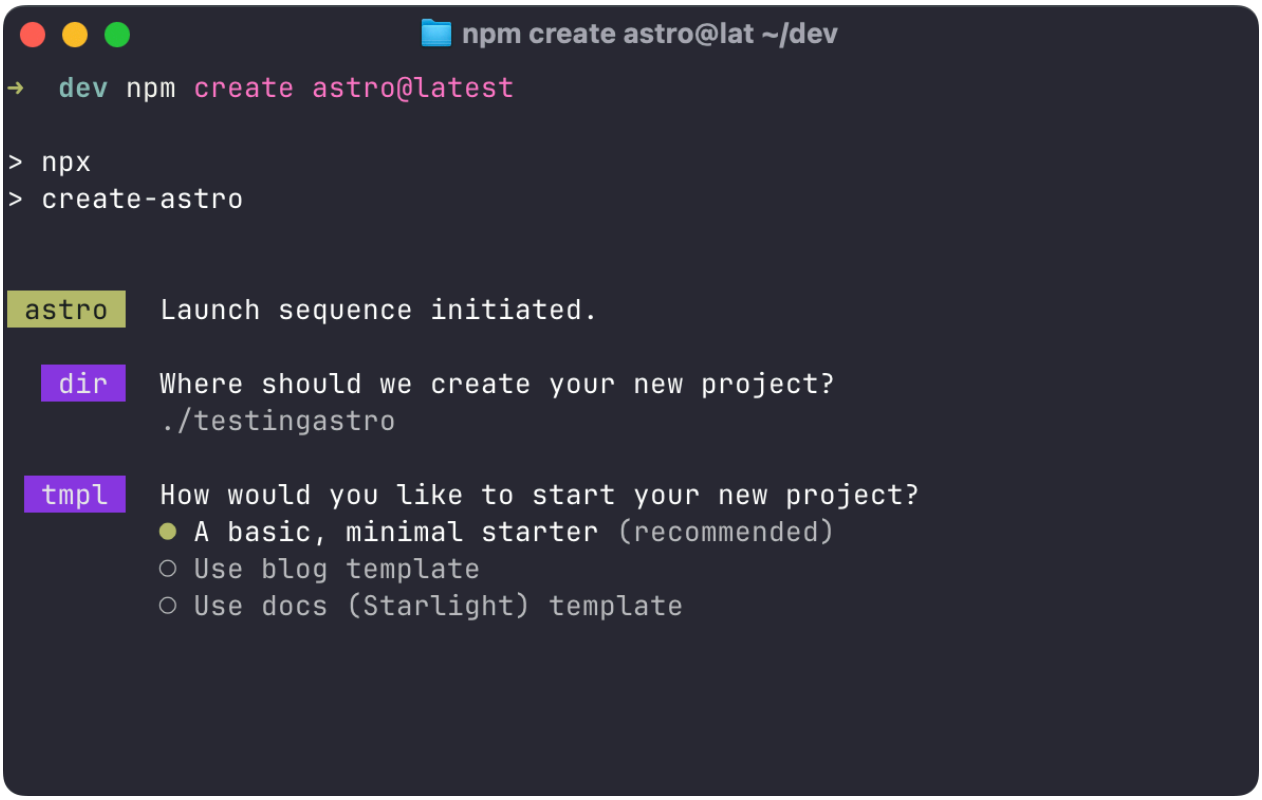
Go into a folder on your computer.

I assume you have Node.js installed, which provides `npm` and `npx`.

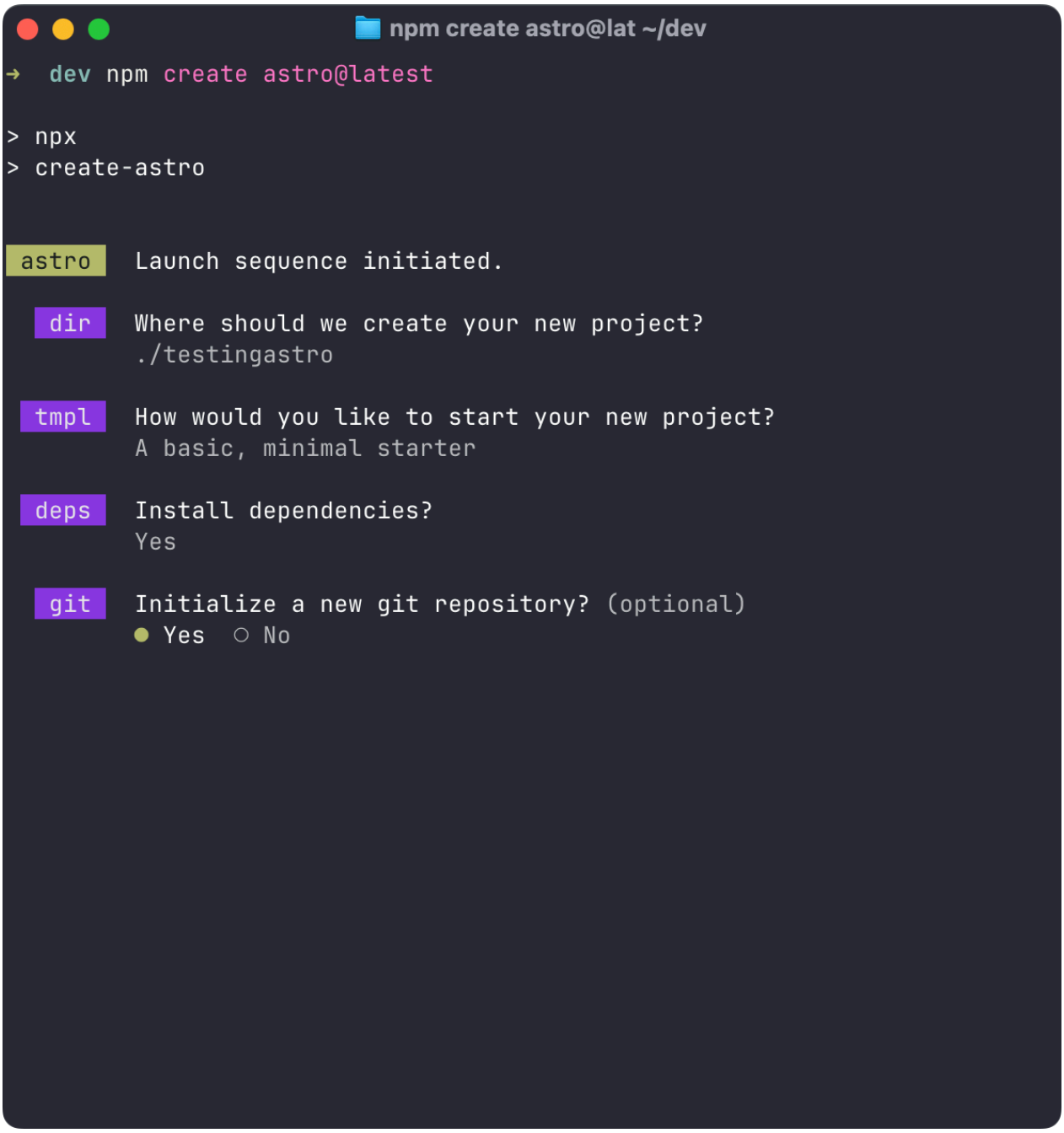
Run this command:

```
npm create astro@latest
```

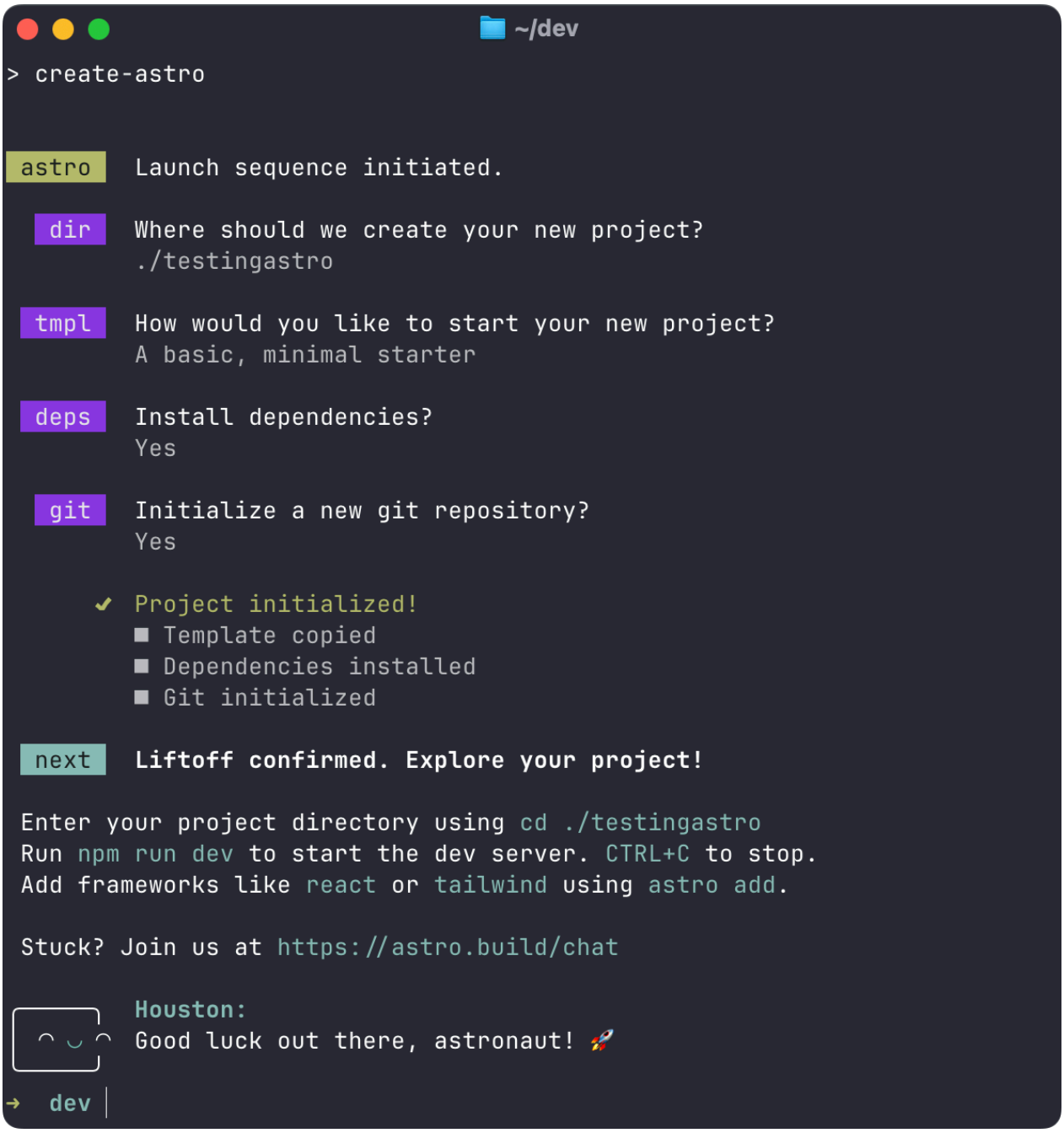
The Astro installer prompts you to pick a folder, I chose to name it `testingastro`



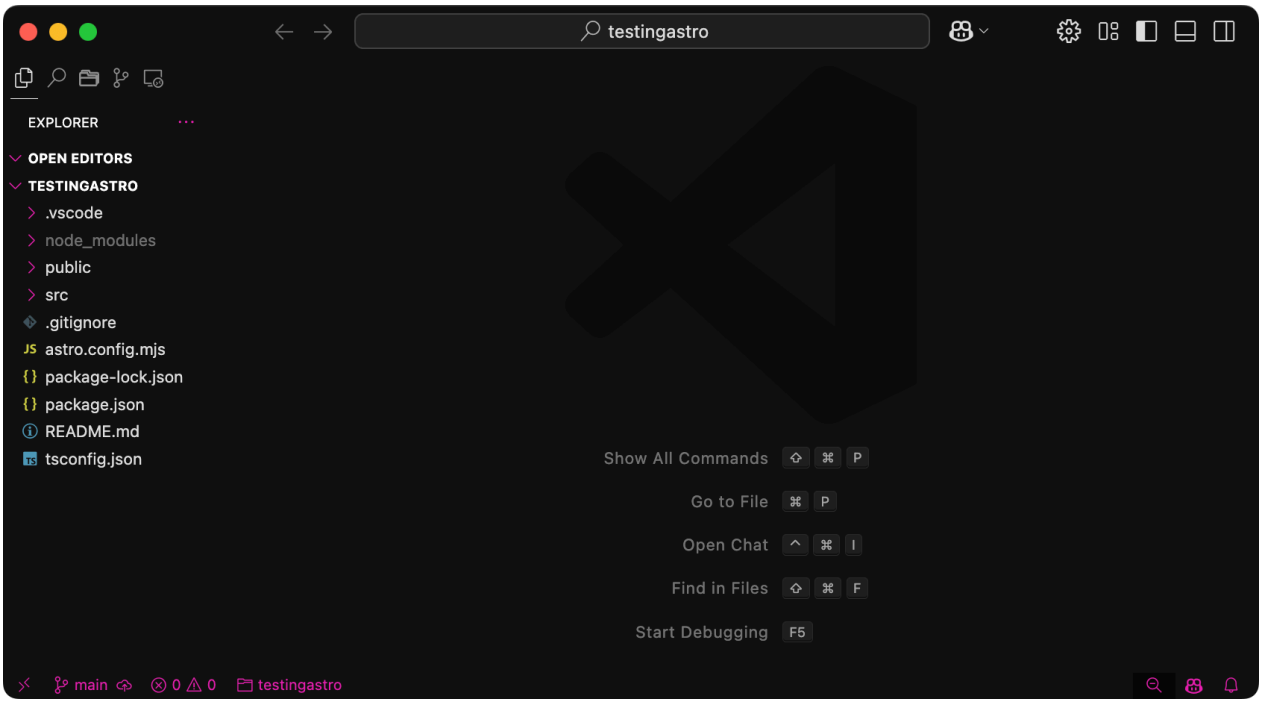
Now pick the “A basic, minimal starter” option, and say “Yes” to the “Install dependencies” and “Initialize a new git repository” questions:

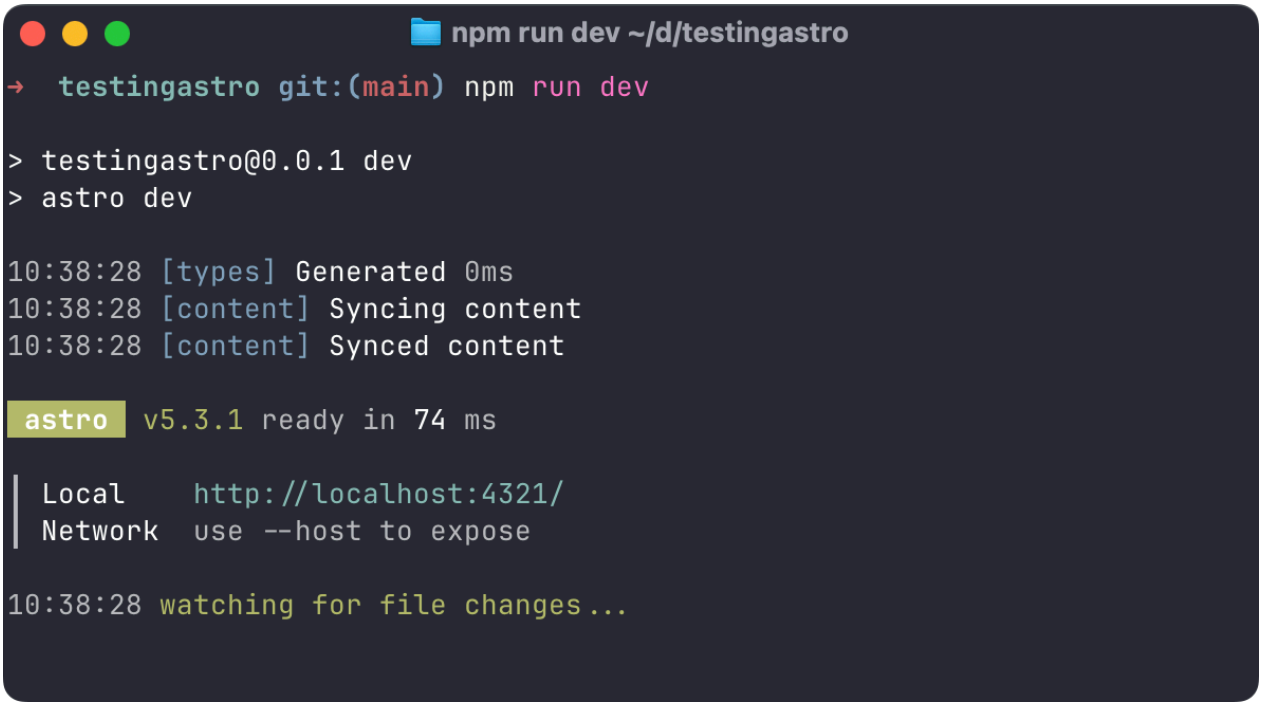


After a little while, things are set up!



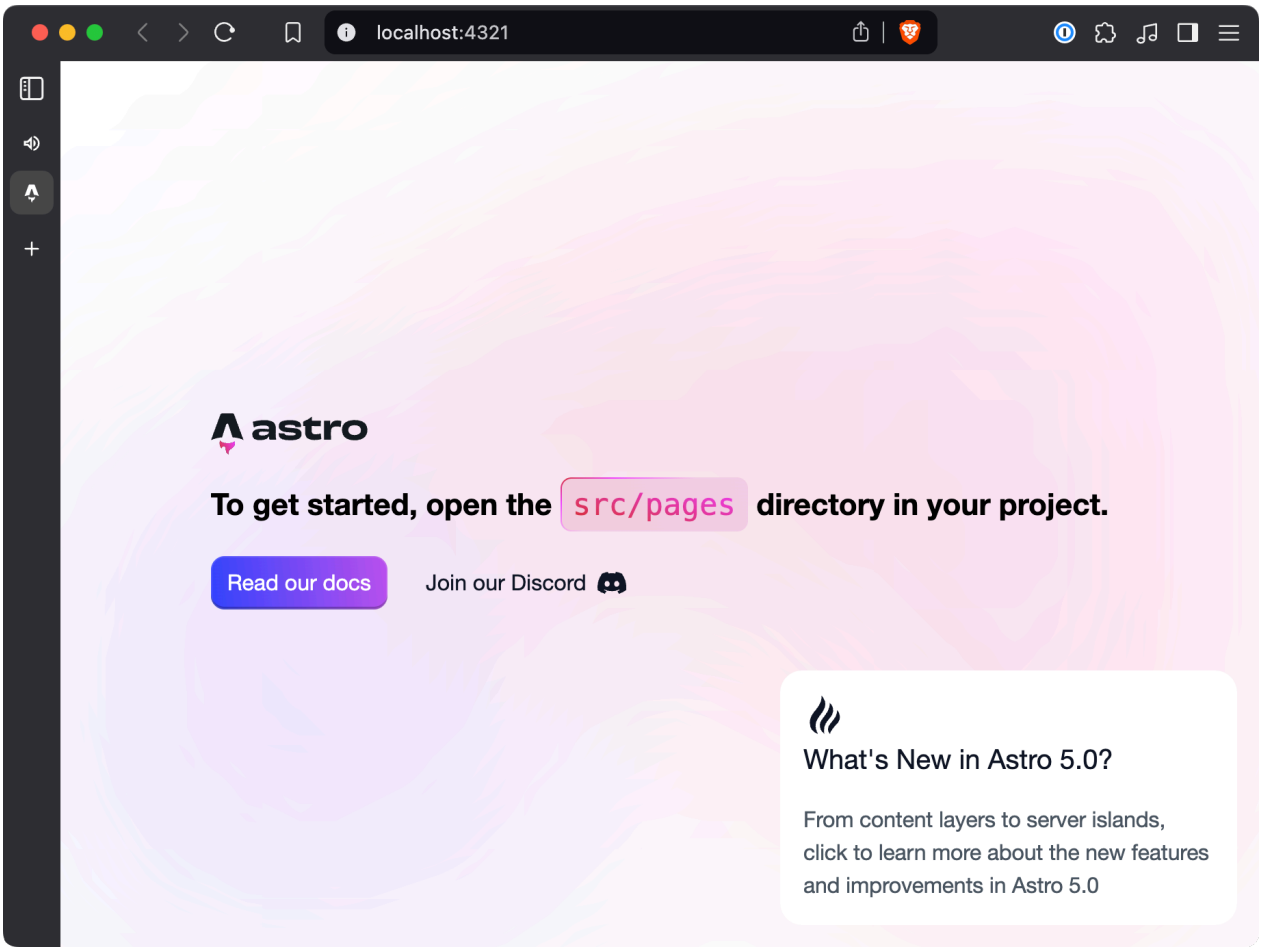
Now open the project in VS Code, or whatever your favorite editor is, and run the Astro project with `npm run dev`.





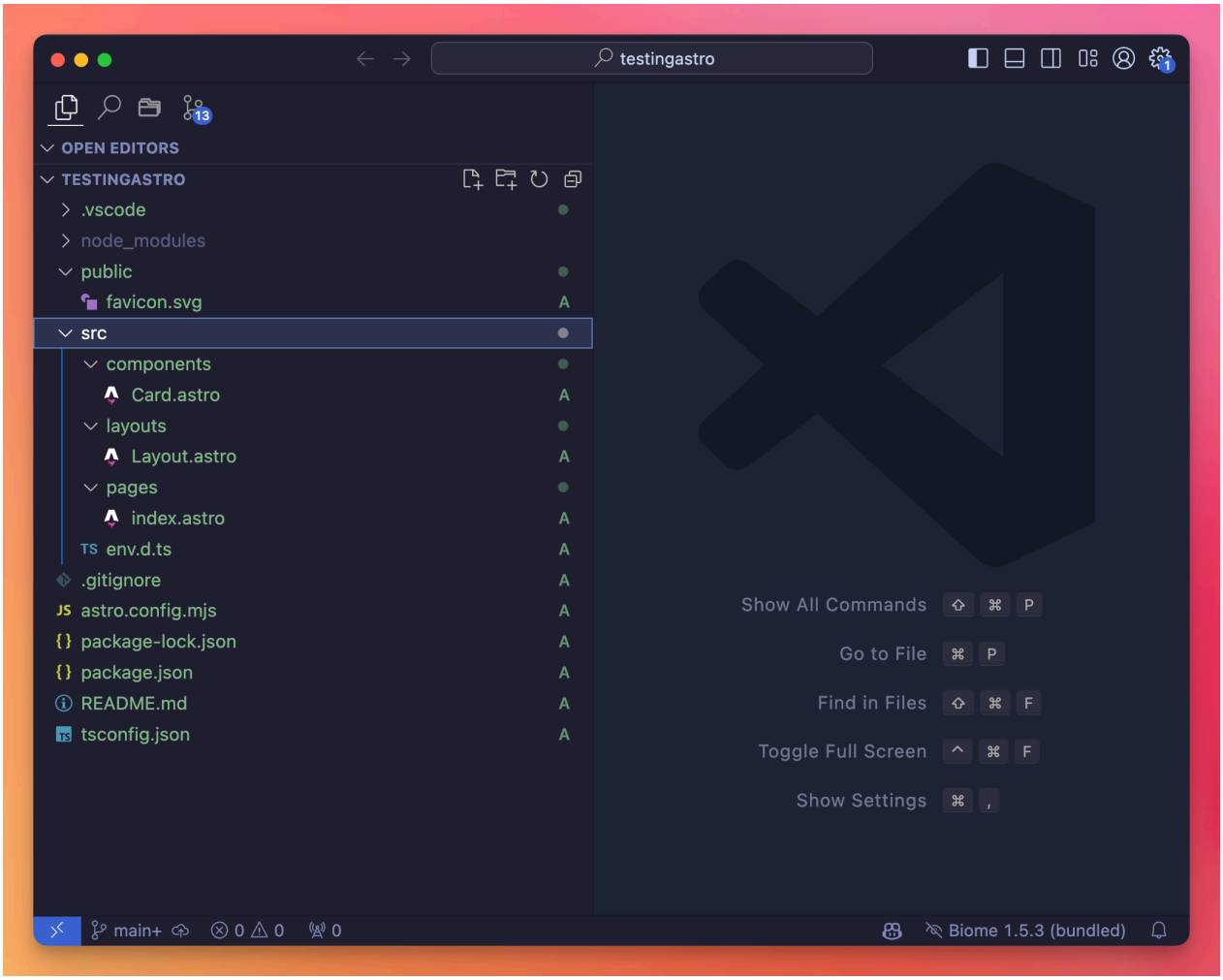
Astro will start on port `4321`, unless you have other things running on that part, in which case port could be `4322` or another one.

Now you can see this basic starter project in the browser:



The structure of an Astro site

Now that you've installed the basic Astro sample site, it's time to take a look around in the site structure.



See, we have 2 folders, except for the VS Code configuration `.vscode` and the Node packages in `node_modules`: `public` and `src`, and some configuration files.

`public` is (as in most frameworks) where you store assets like images that do not need any kind of processing and are served as-is. For example `public` contains the `favicon.svg` file, and you access it using the URL `http://localhost:4321/favicon.svg`.

Configuration files include `astro.config.mjs`, which (as we'll see later) you can extend to add more features (and configuration) to Astro.

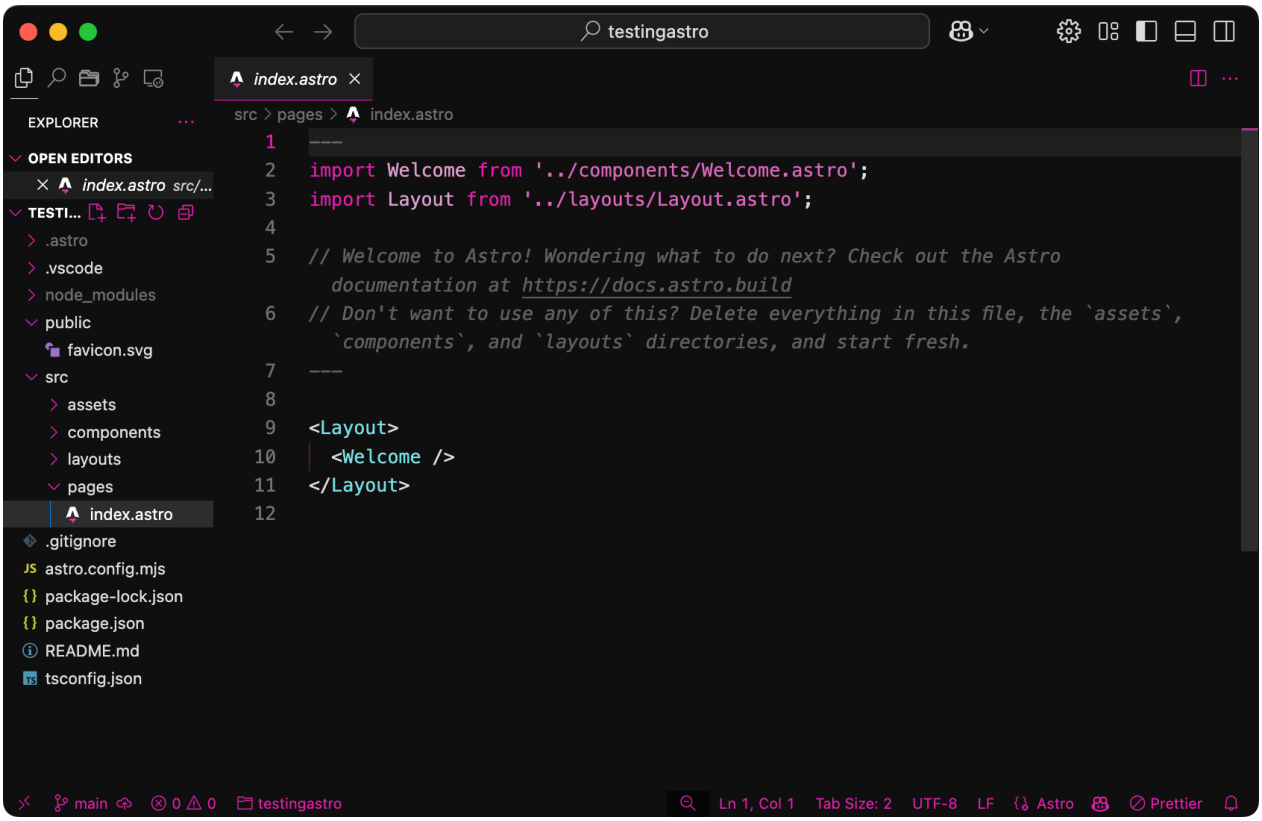
Everything else is under `src`.

The `src` folder now includes those subfolders:

- `src/assets`
- `src/components`
- `src/layouts`
- `src/pages`

`src/pages` contains the Astro page routes.

You now see `src/pages/index.astro`. This is the entry point for the `/` route, the one that serves the homepage.



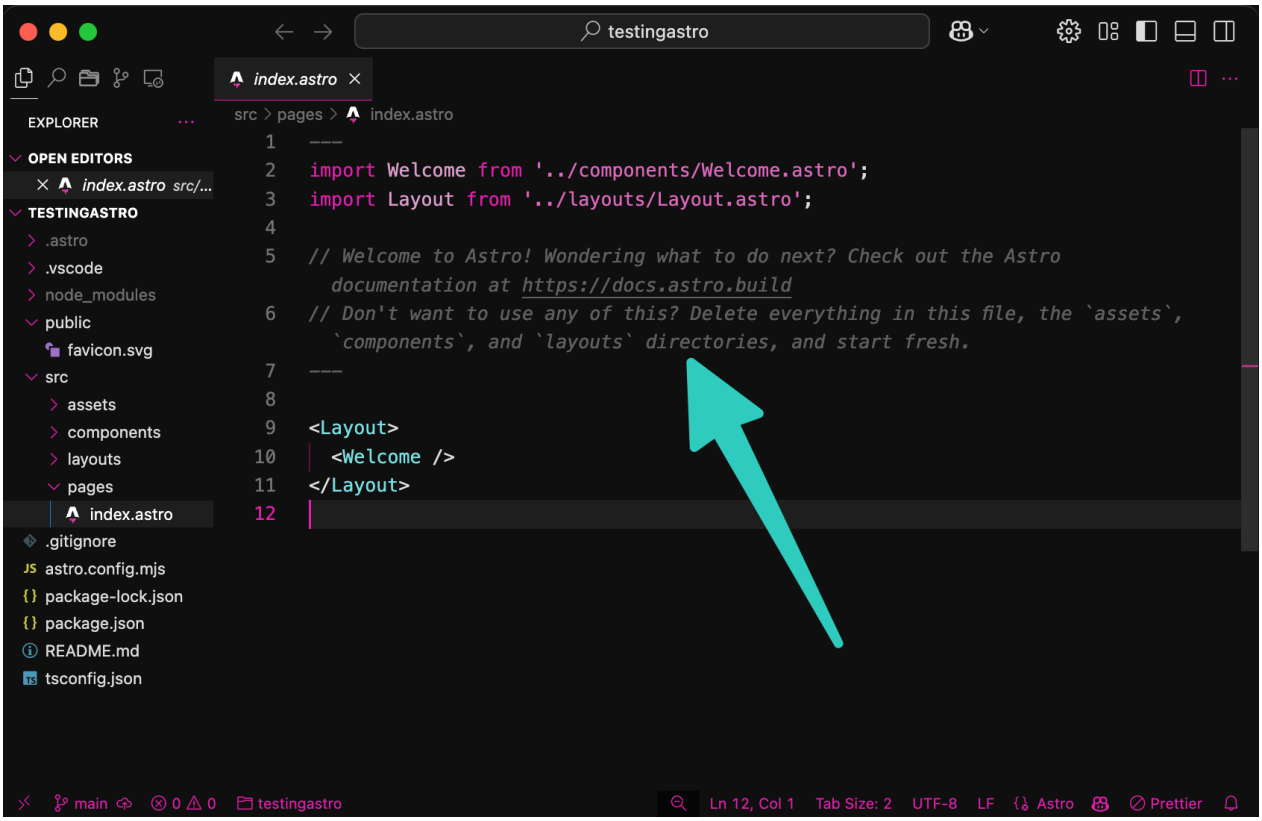
This is an Astro **component**, as the file ends with the `.astro` extension.

Being under `src/pages` makes it special because it's also a route, so Astro knows this component is what it will serve on that `/` route.

We'll see how to add more routes later.

But let's analyze this component. We can see 2 parts basically.

The first is the upper part, between the two `---` blocks, called frontmatter:



That's where we can write some JavaScript (TypeScript) code that runs when the page is built.

This is not client-side JavaScript. It's also not server-side JavaScript. It's *build-time JavaScript*.

Since the end result of an Astro site is a static site, it will not have a backend.

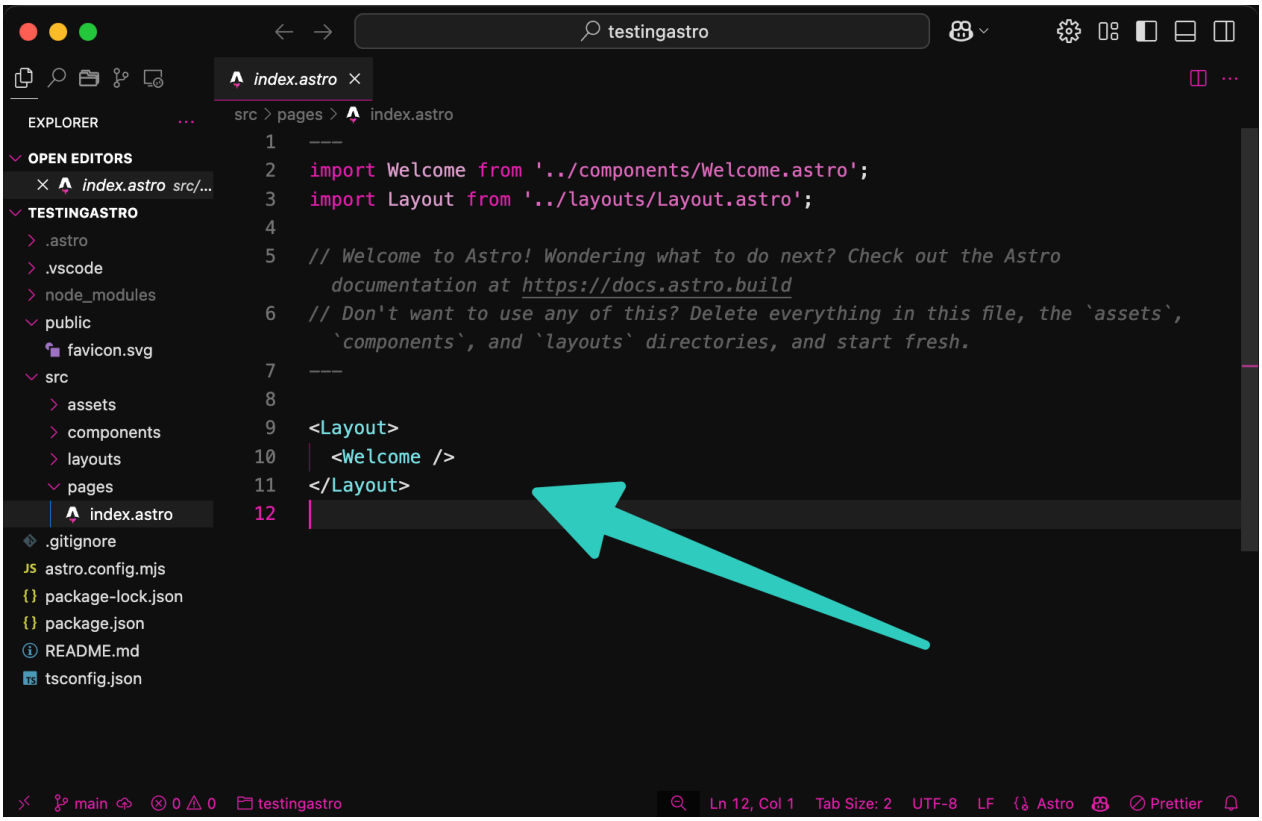
So here we can do various things, like fetching data across the network or look for data in the filesystem.

In this case we import a layout, and a component.

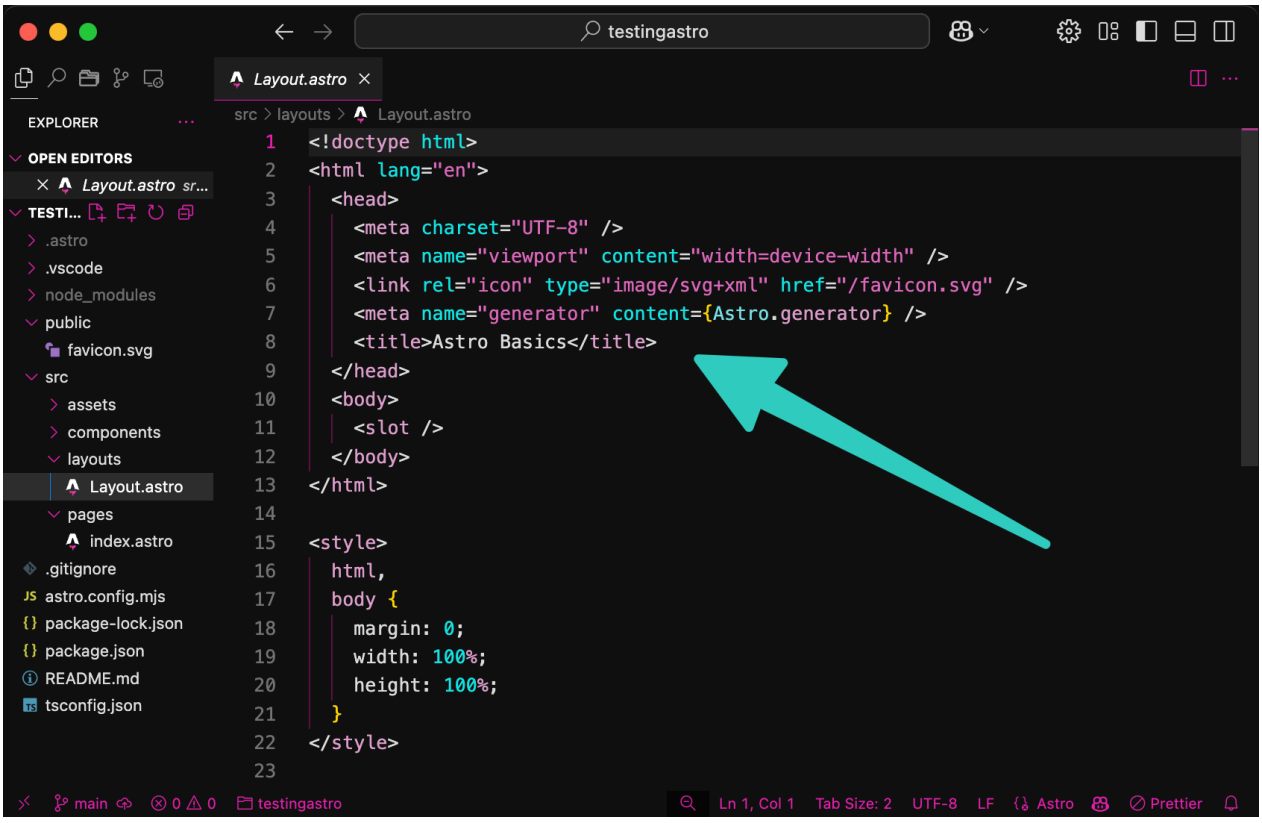
The layout is the "outer HTML" that this page will use.

We commonly use a layout, so you don't have to define the whole HTML structure for every page.

See, inside the "main" part of the component, we don't define the DOCTYPE, an `<html>` tag, `<body>` and so on.



That's defined in the layout:



The layout is the "container".

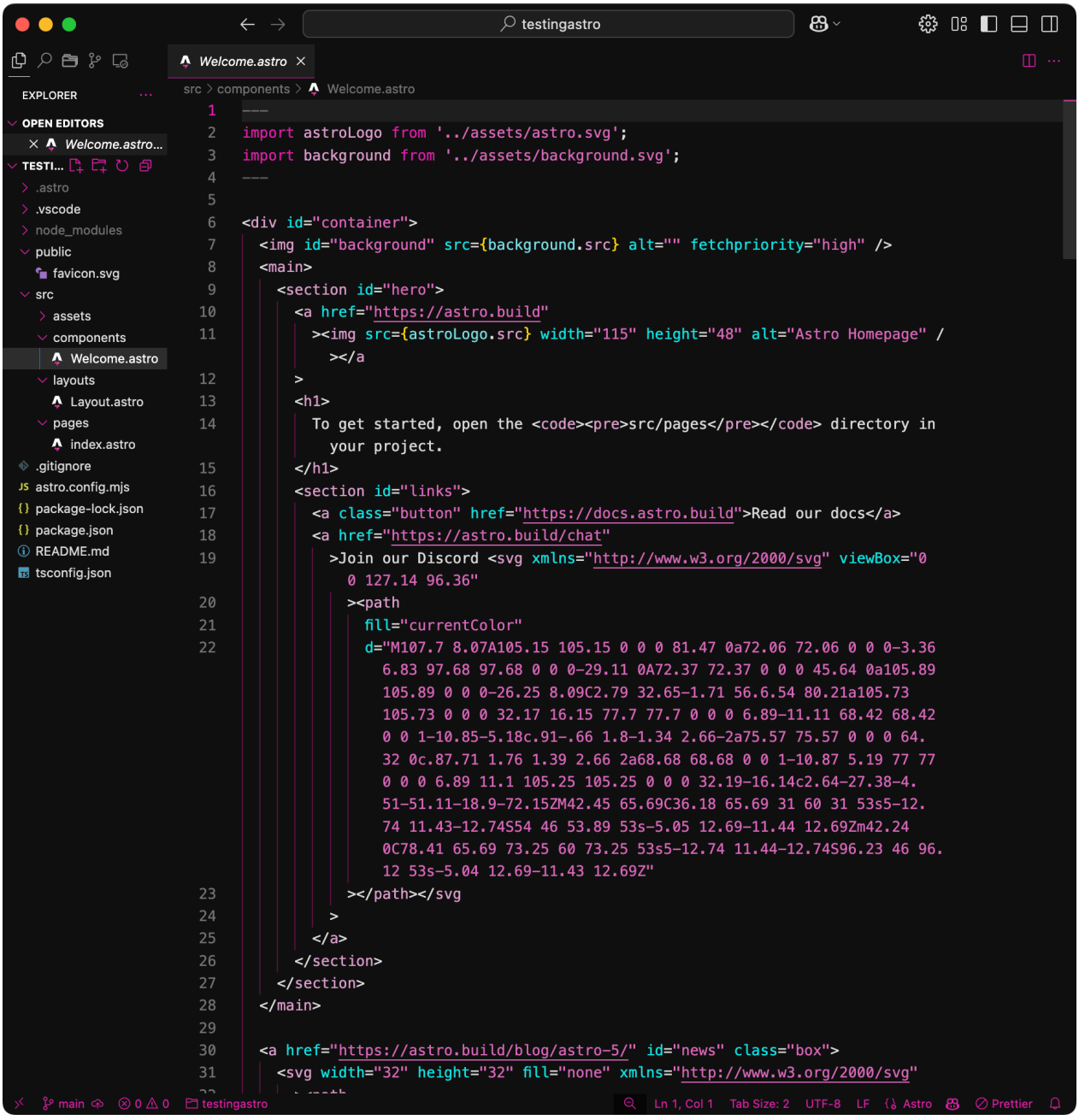
We do this so it can be reused by multiple pages, without duplicating the page structure that's common across the site.

The `<slot />` special element is where the “page” content will be inserted.

We’ll see more about layouts later.

So we’ve seen how Astro page components can have a frontmatter that executes JavaScript at build time and can be used to import other components.

Finally the last item in this basic starter kit is an example of separating some piece of UI to a separate component, not a page component, that is defined in `src/components` as a convention:

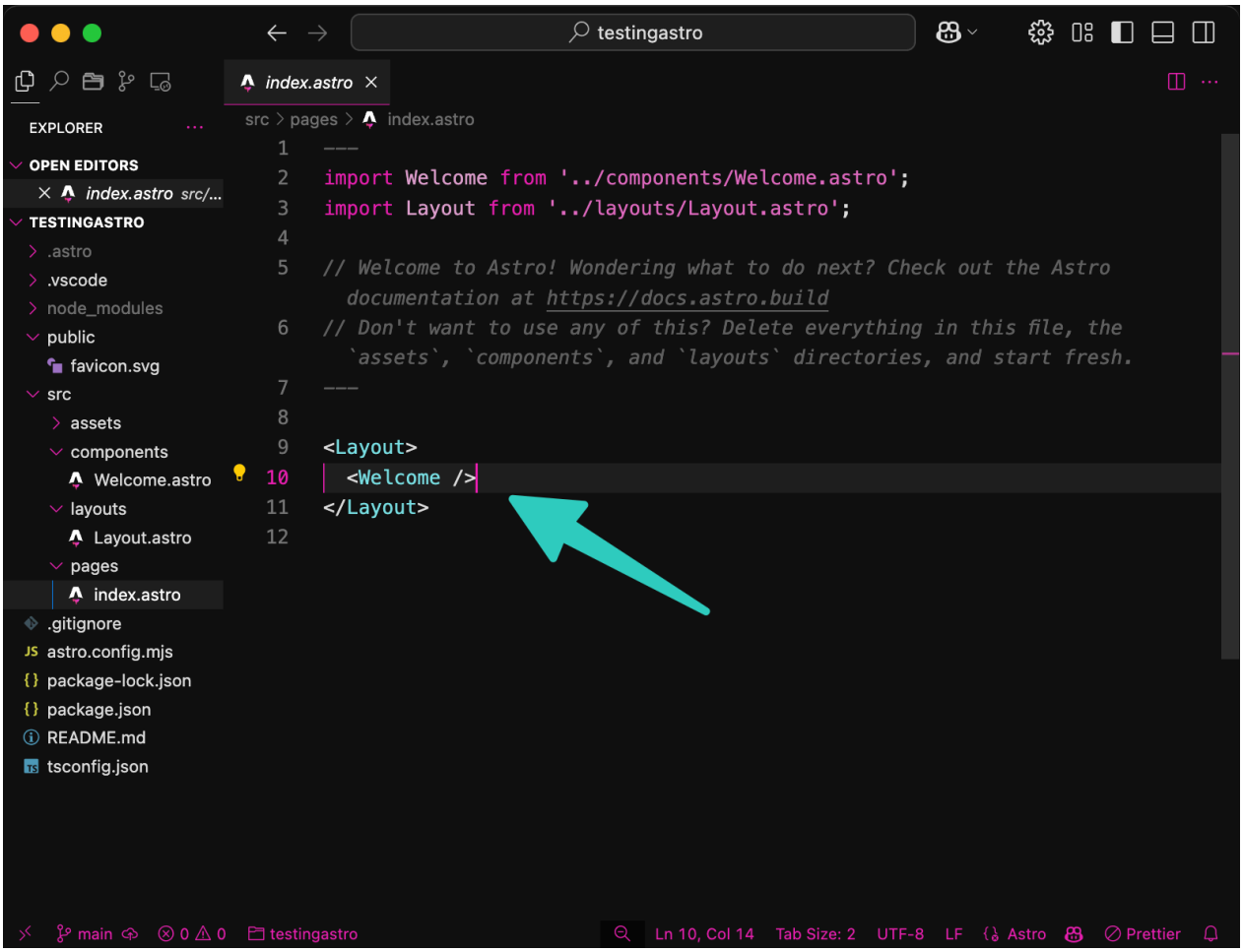


We can define components to create little units of code that we can reuse across the site.

In this example the Welcome component is used in the `src/pages/index.astro` component by first importing it in the component frontmatter:

```
import Welcome from '../components/Welcome.astro';
```

and then it’s used in the page component’s HTML template



Components are great because they avoid duplication.

We'll talk more about them next, and we'll see how cool they are.

Astro components

We've created our first Astro site and I introduced components.

Let's now talk more about them.

In their simplest form, components can just be some HTML tags, for example

```
<p>test</p>
```

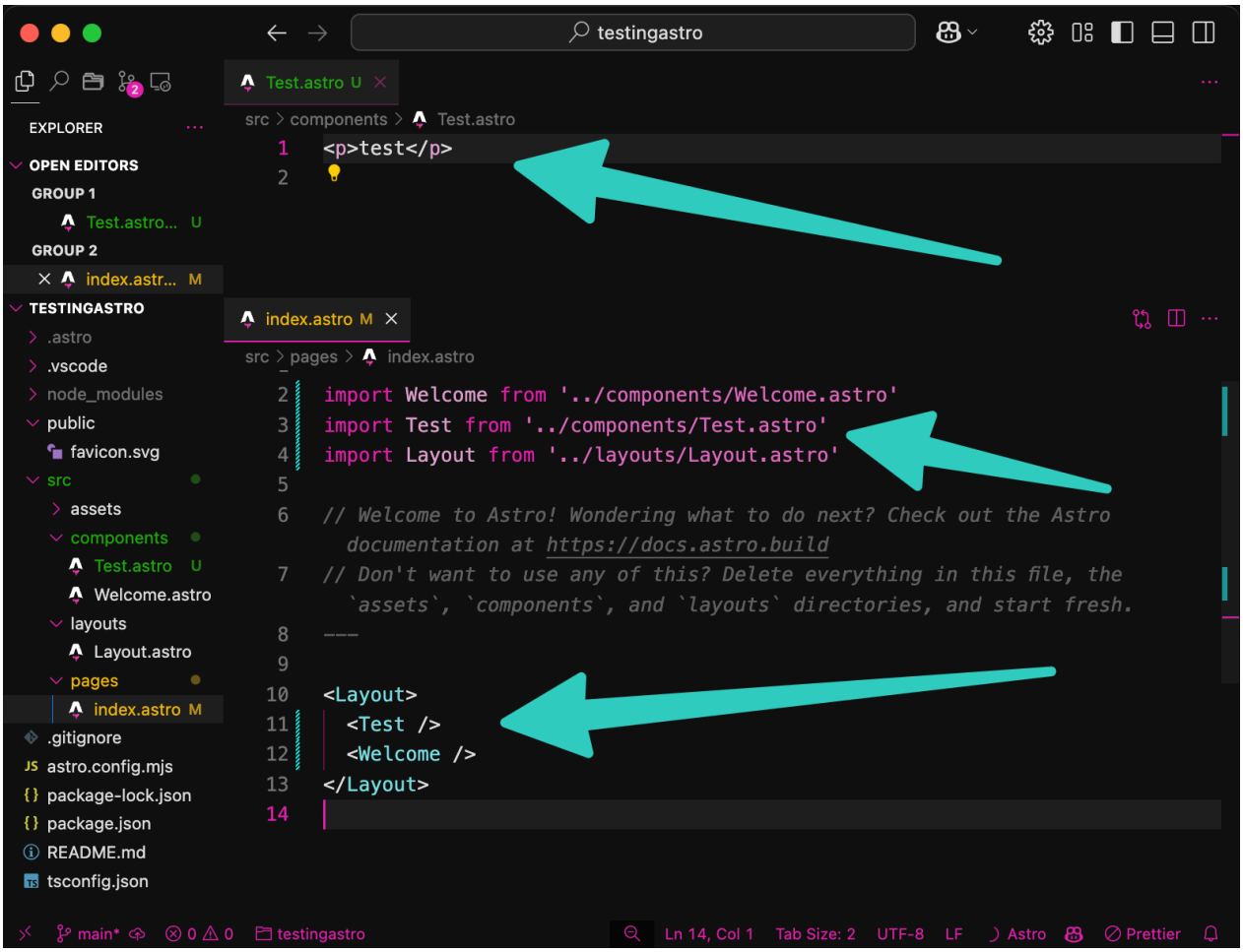
Create a `src/components/Test.astro` file with that content, and import that in a page component:

```
import Test from '../components/Test.astro'
```

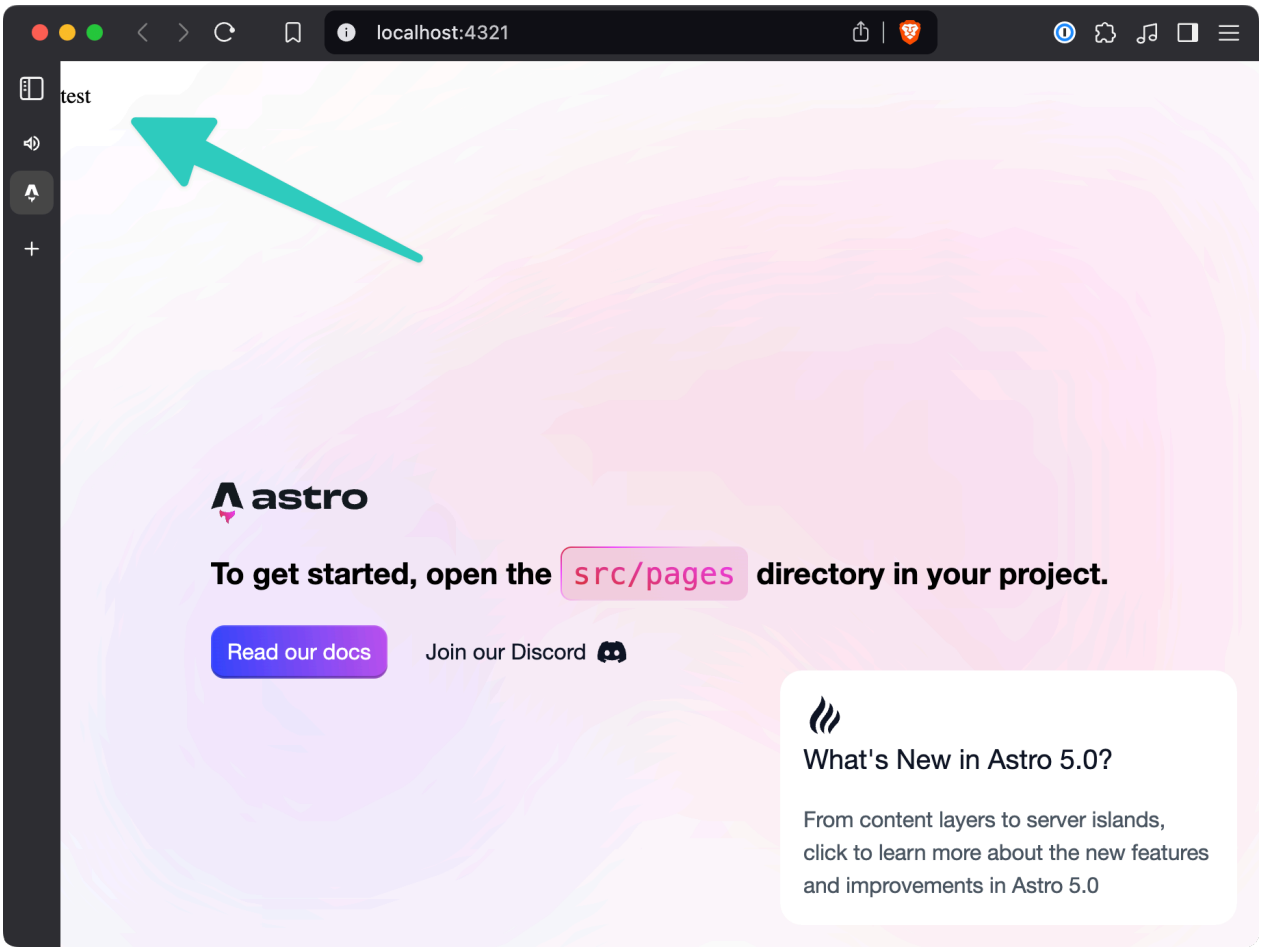
...then add it to the component markup as you would write an HTML tag, like this:

```
<Test />
```

> IMPORTANT: the component name's first letter must be uppercase



See, the markup now appears in the page:



The syntax:

```
<Test />
```

was borrowed from JSX, a "templating language" introduced by React.

> TIP: If you already know JSX, in Astro components the syntax is a little different, because for example we can have siblings in a tree (no need for fragment or <>) and no need to use className= to add HTML classes, just class=.

Components can add JavaScript variables in their "frontmatter", and you can use them in the HTML markup:

```
---
const test = 'hello'
---

<p>{test}</p>
```

Let's now talk about **props**.

Astro components accept props.

This means you can pass values to a component from a parent component.

For example we can pass a `name` attribute to the Test component like this:

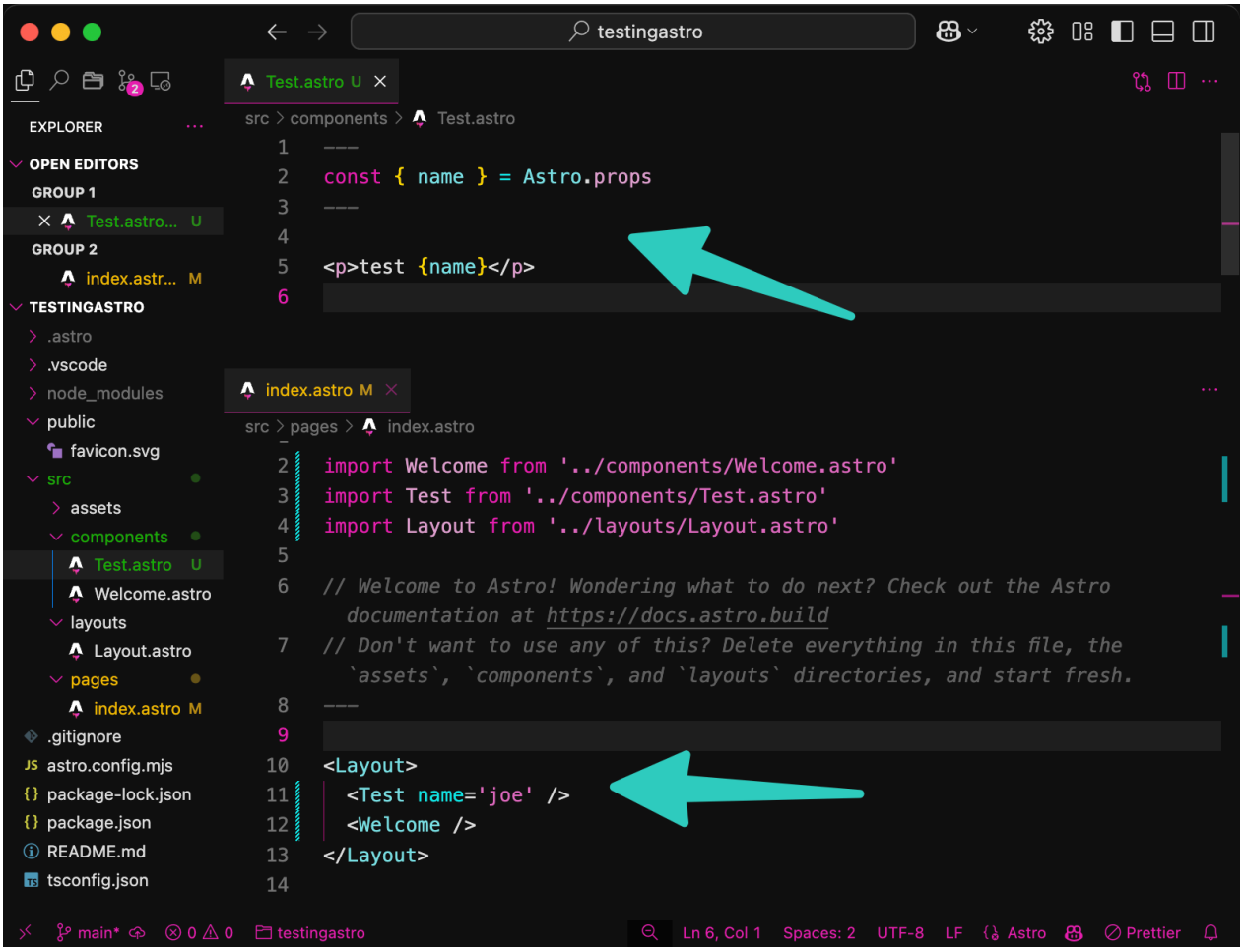
```
<Test name="joe" />
```

Inside the Test component we can use this value by adding this to the component frontmatter:

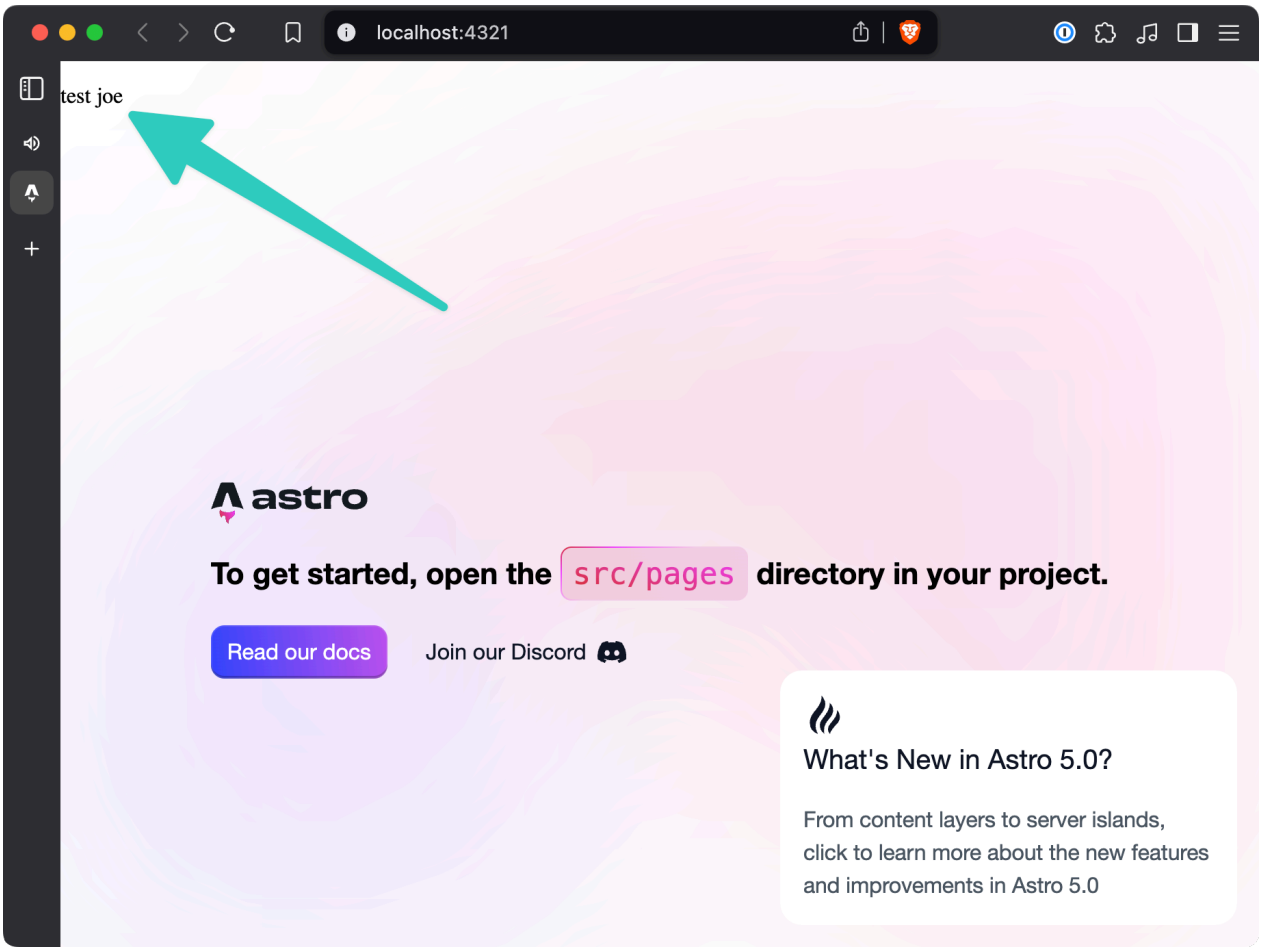
```
---
const { name } = Astro.props
---

<p>test</p>
```

And now we can use it in the markup, using a special JSX-like syntax that lets us embed JavaScript inside the HTML:

```
---
const { name } = Astro.props
---

<p>test {name}</p>
```



Here's the result:



And this is how we can embed data in Astro components.

If you had an array, you could print all items in the array using this syntax:

```
---
const list = [1, 2, 3, 4]
---

<p>Some numbers:</p>

<ul>
  {list.map(n => <li>{n}</li>)}
</ul>
```

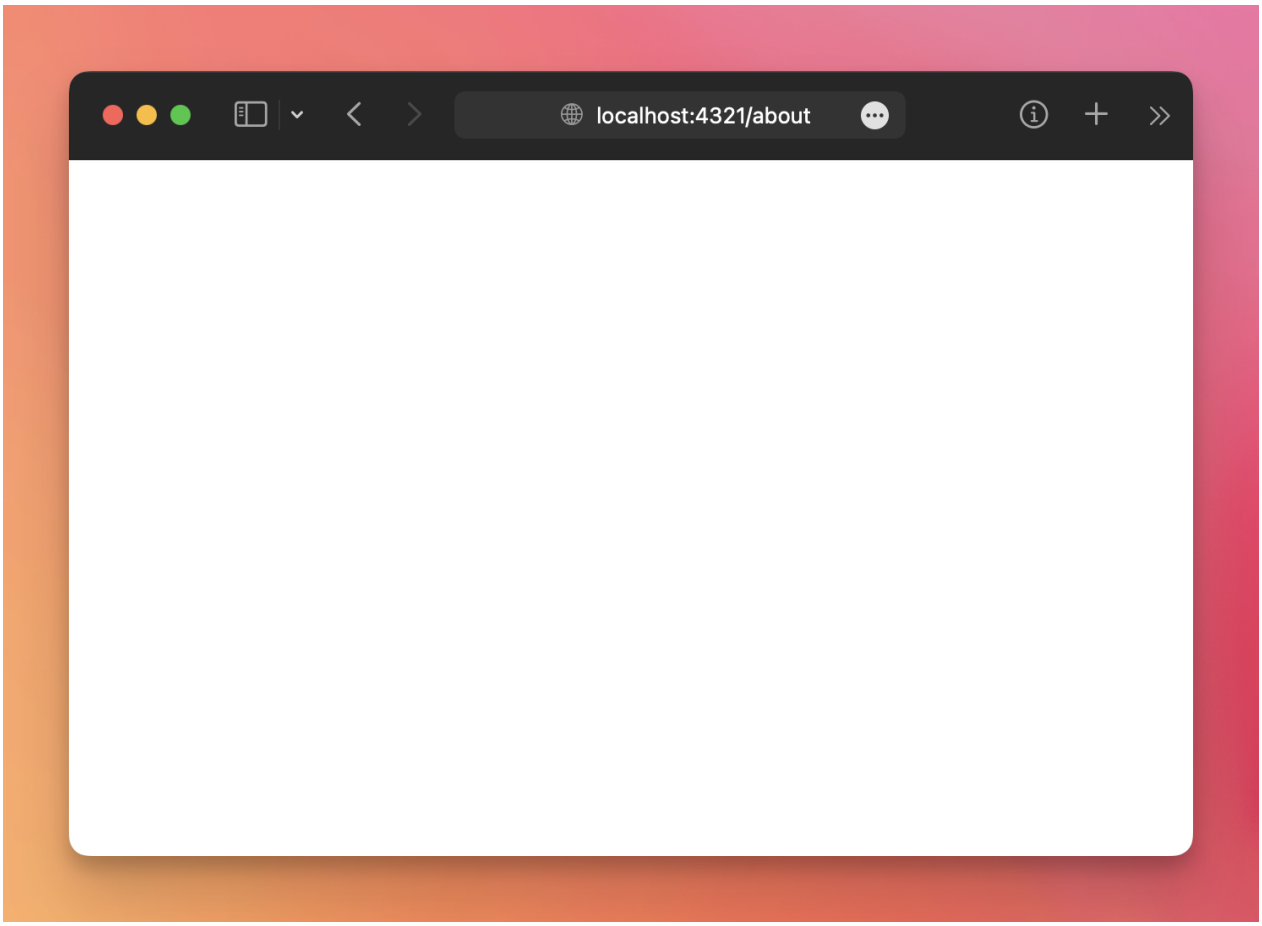
Adding more pages

Now let's try adding more pages to your Astro site.

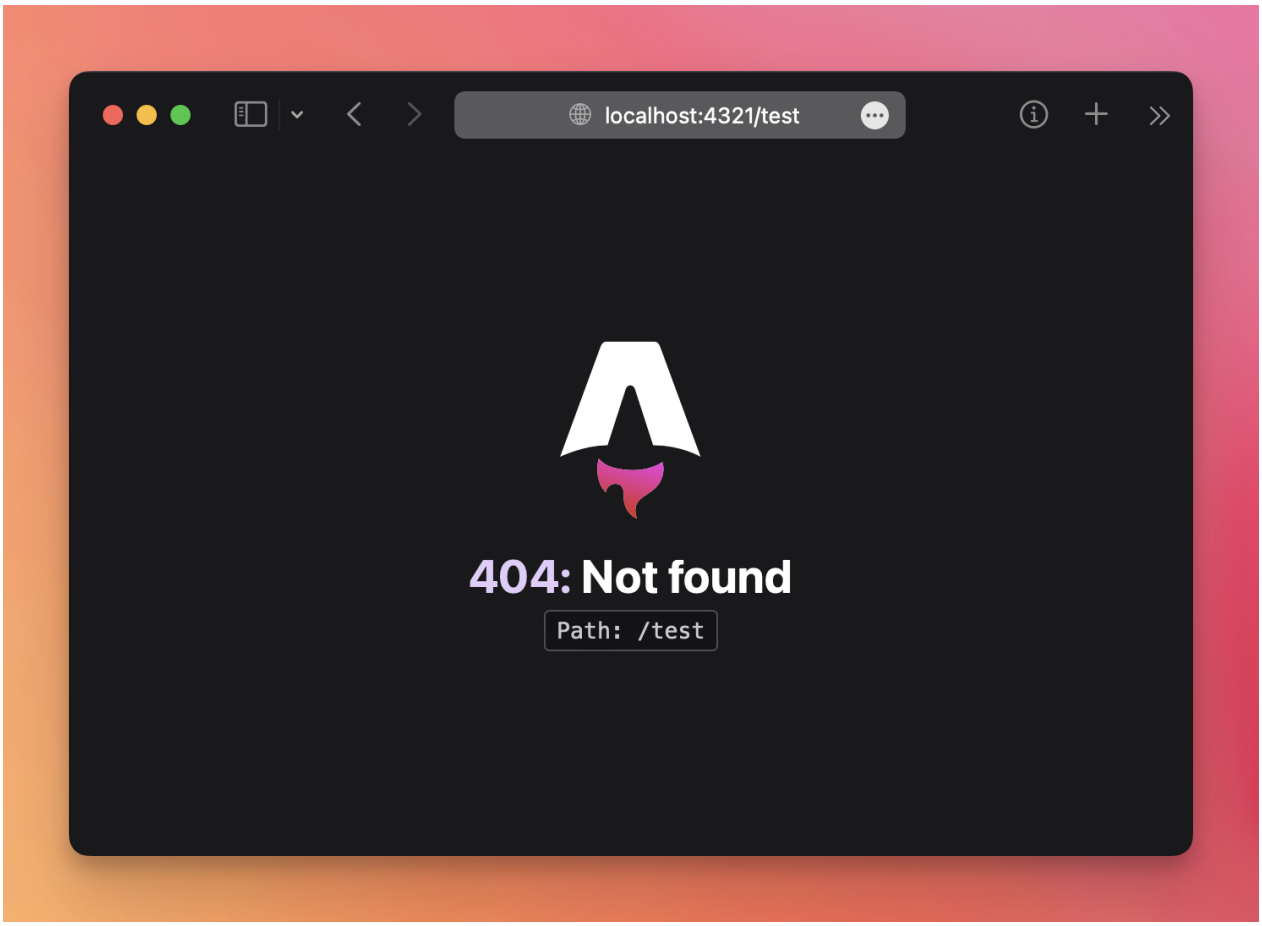
Create a `about.astro` file under `src/pages`

This file represents a new page that listens on the route `/about`

If you visit http://localhost:4321/about, it works:

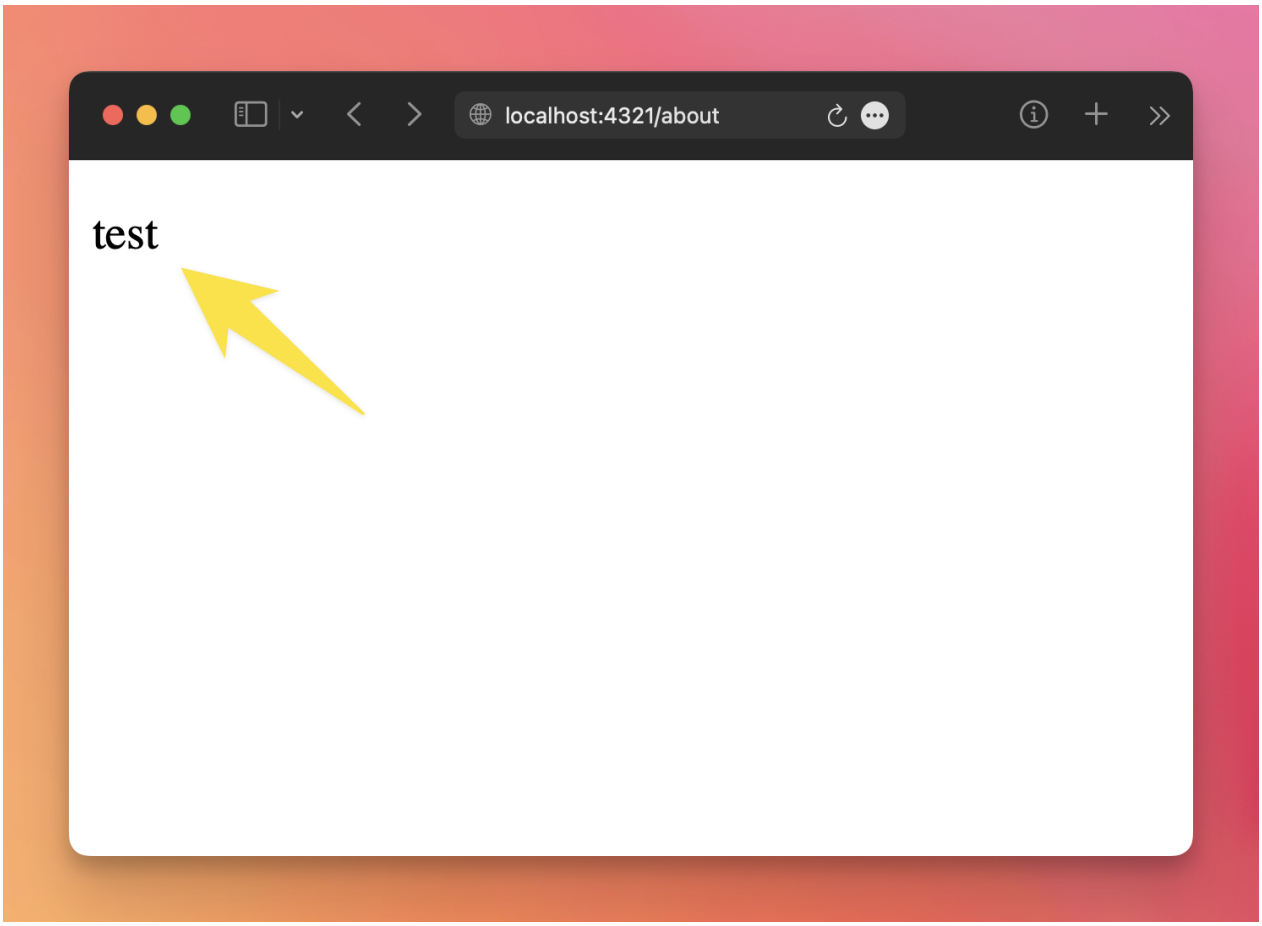


The page is a blank page, but no error like you'd get if you went to a route not defined, for example http://localhost:4321/test:

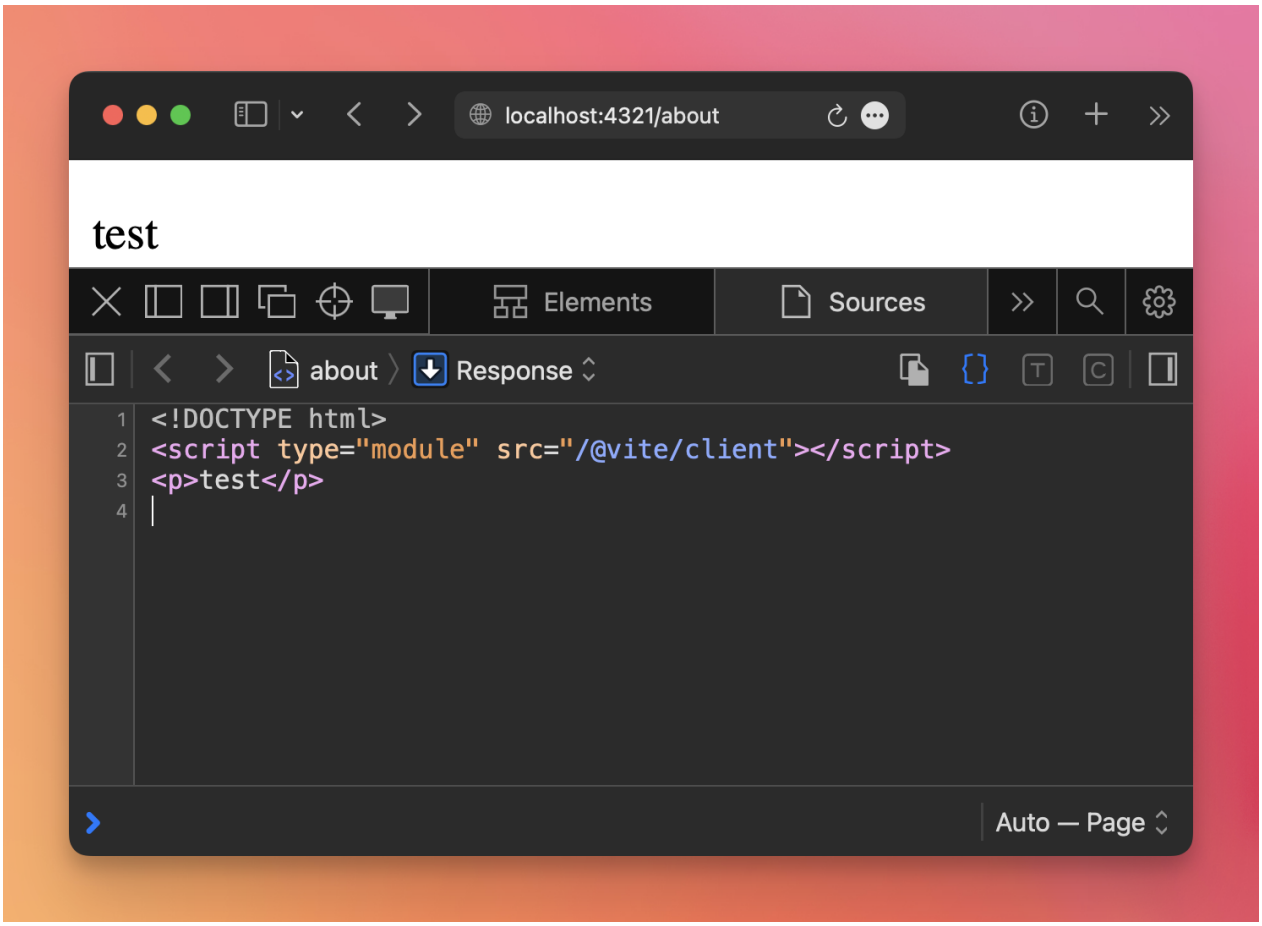


Now you could add a simple HTML tag to the page, like we did before with the Test component:

```
<p>test</p>
```

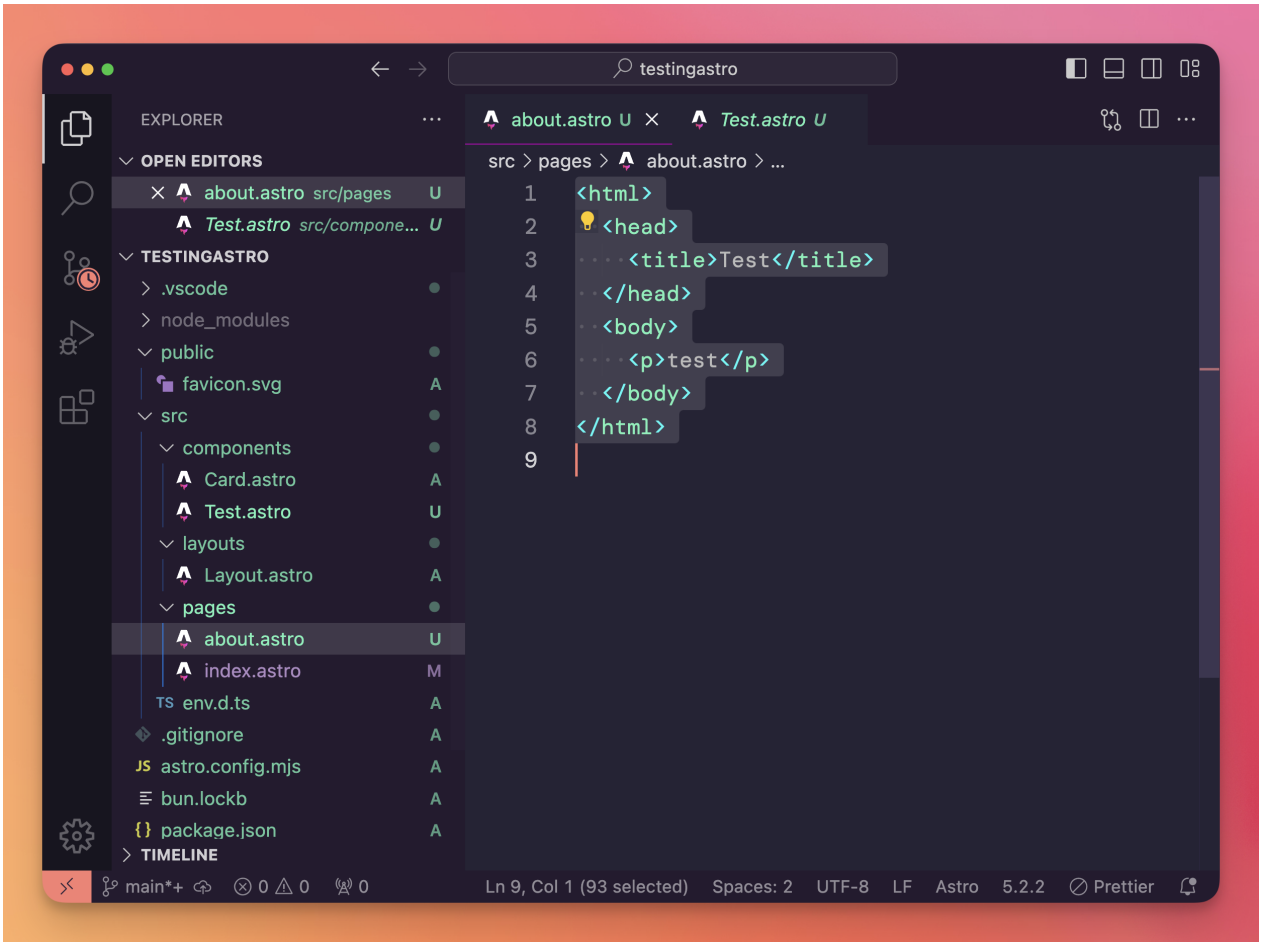


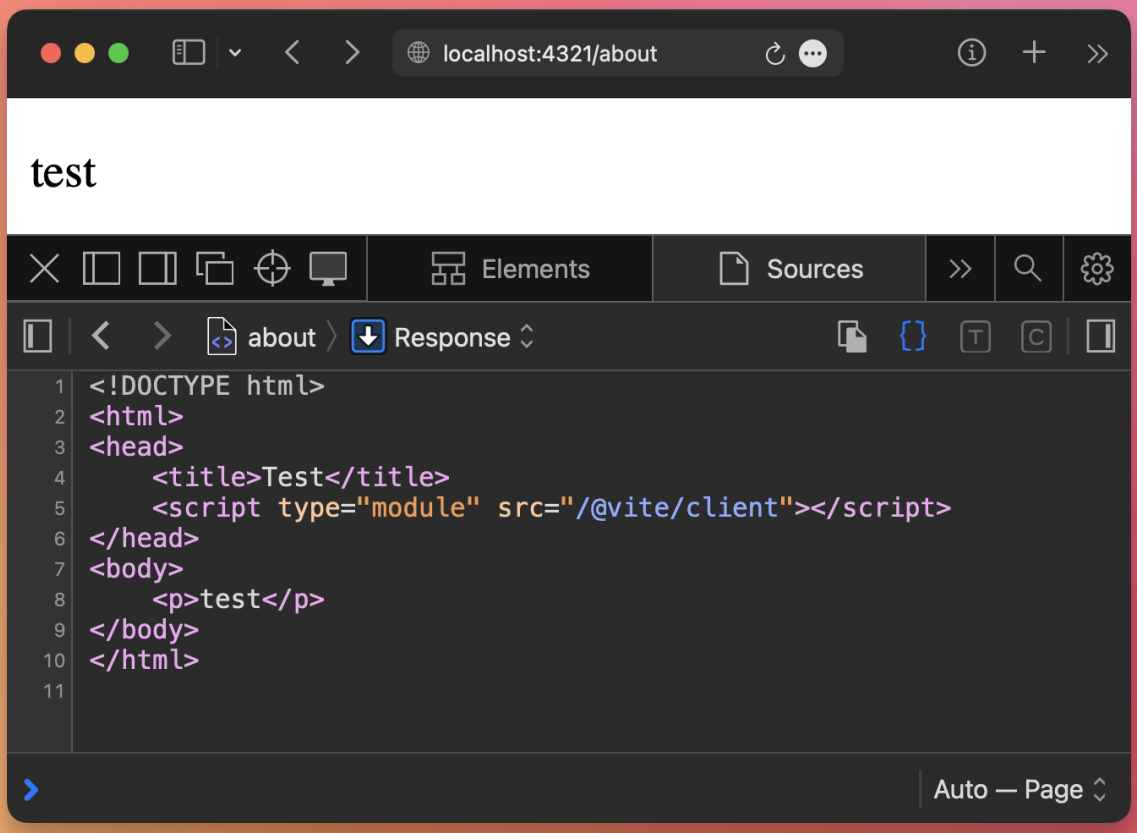
If you open the DevTools you'd see a simple HTML structure, without all the `<head>` and `<body>` tags and everything that makes an HTML page "correct":



(the `DOCTYPE` and the `script` tag were added by Astro automatically)

You could add a full HTML page structure, complete with `html`, `head` and `body` tags, directly in the component:


testingastro
EXPLORER
about.astro U
Test.astro U
src > pages > about.astro > ...
OPEN EDITORS
about.astro src/pages U
Test.astro src/compone... U
TESTINGASTRO
.vscode
node_modules
public
favicon.svg A
src
components
Card.astro A
Test.astro U
layouts
Layout.astro A
pages
about.astro U
index.astro M
env.d.ts A
.gitignore A
astro.config.mjs A
bun.lockb A
package.json A
TIMELINE
1 <html>
2 <head>
3 <title>Test</title>
4 </head>
5 <body>
6 test
7 </body>
8 </html>
9
main*+ 0 0 0
Ln 9, Col 1 (93 selected) Spaces: 2 UTF-8 LF Astro 5.2.2 Prettier


localhost:4321/about
test
Elements
Sources
about
Response
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Test</title>
5 <script type="module" src="/@vite/client"></script>
6 </head>
7 <body>
8 test
9 </body>
10 </html>
11
Auto — Page

But what you'd usually do is use a layout:

```
---
import Layout from '../layouts/Layout.astro'
---

<Layout>
  <p>test</p>
</Layout>
```

and the `src/layouts/Layout.astro` will provide the base markup, which is shared with `src/pages/index.astro` too.

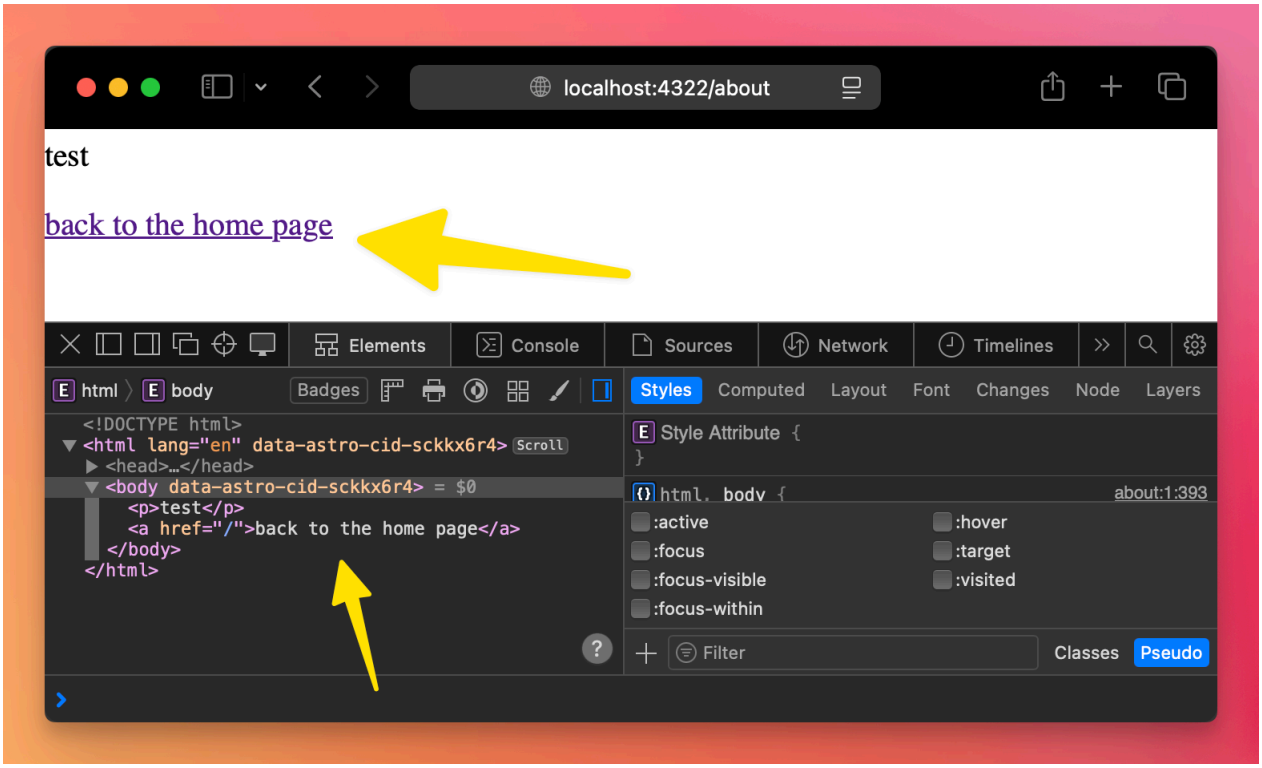
This is a common way to have a base layout.

You can have multiple layouts, for example if you want to have a layout for single pages, and a different one with a sidebar for blog posts. For example, I have 4 different layouts in my site.

To link pages with each other we use the HTML `<a>` element.

So we can link back to `/` using:

```
---
import Layout from '../layouts/Layout.astro'
---

<Layout>
  <p>test</p>
  <a href="/">back to the home page</a>
</Layout>
```



Astro does not provide client-side navigation by default, so all links do a full-page refresh, like it happens in plain HTML pages.

However you can easily add **view transitions** to add client-side navigation, as we'll see later.

Dynamic routing

We've seen how to create pages, that have a static route.

- `src/pages/index.astro` has the `/` route
- `src/pages/about.astro` has the `/about` route

Sometimes you have the need for dynamic routes.

Dynamic routes allow you to manage multiple different URLs with a single page.

Think about a blog, for example.

You have multiple blog posts, but all use the same structure.

We'll get to writing posts in markdown soon, but let's do a simple example now to explain dynamic routes.

A dynamic route is created by adding a file with square brackets under `src/pages`.

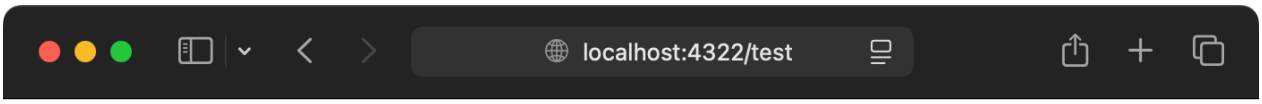
Create a file `src/pages/[post].astro`

`post` inside the square brackets is the variable that will be passed to the page and will contain the dynamic segment.

You can grab that from `Astro.params` in the page frontmatter.

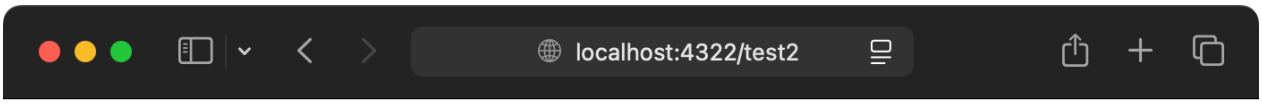
You must however also define and export a `getStaticPaths()` function, that returns an array of objects which contain the values allowed for the dynamic segment:

```
---
import Layout from '../layouts/Layout.astro'
const { post } = Astro.params

export async function getStaticPaths() {
  return [
    { params: { post: 'test' } },
    { params: { post: 'test2' } },
    { params: { post: 'test3' } },
  ]
}
---

<Layout title='Post'>
  <h1>{post}</h1>
</Layout>
```



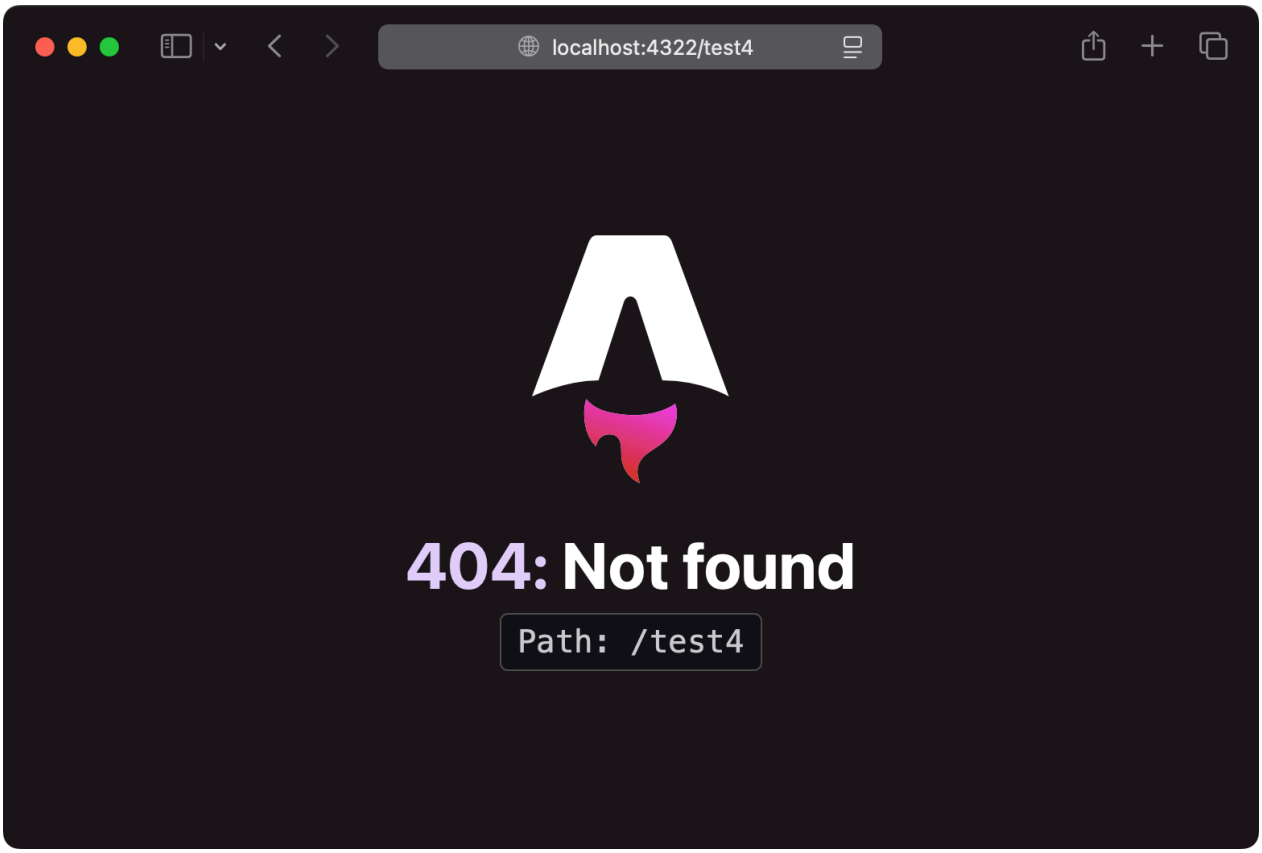
test

If you go to the `/test2` route, that's still a route that's taken care by this file.



test2

`/test4` would be not, and would generate a 404 page not found message.



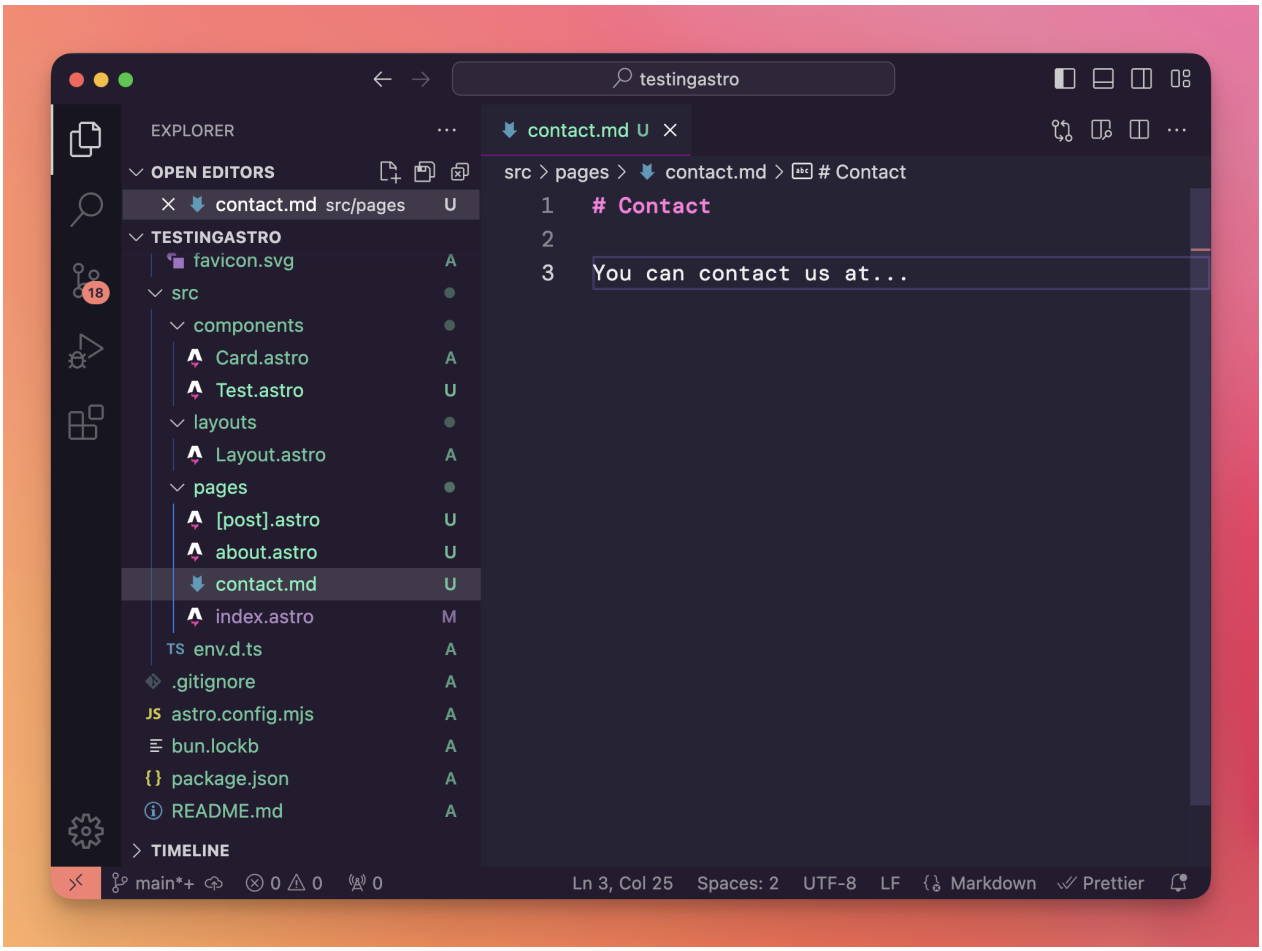
Markdown in Astro

You can write content directly in Astro components by writing HTML, but Astro comes with a very powerful Markdown and MDX engine.

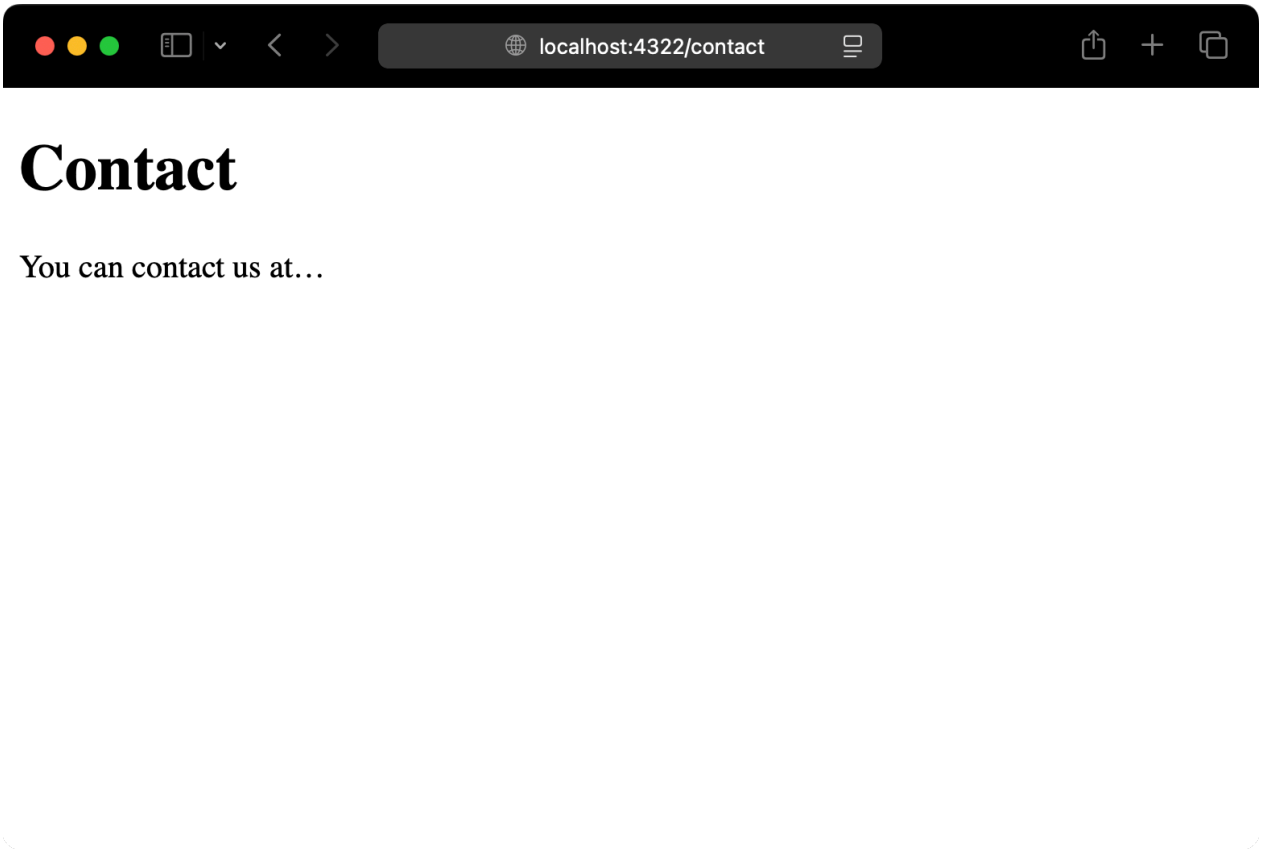
MDX is basically Markdown with superpowers. It allows you to import components and show them in the page.

You might not need it most of the time (I don't), but it's supported.

To create a markdown page, you can add it directly into `src/pages`:



Astro will render this:



You can now tell Astro to use a specific layout for this markdown file, and you can set it in the frontmatter of the markdown page:

```
---
layout: ../layouts/Layout.astro
---

# Contact

You can contact us at...
```

From within the layout you can access any frontmatter variable by importing the frontmatter from `Astro.props`:

```
---
const { frontmatter} = Astro.props
---
```

You usually have the title, the description, maybe a date.

Collections are a handy way to work with lots of markdown files. We'll see them later.

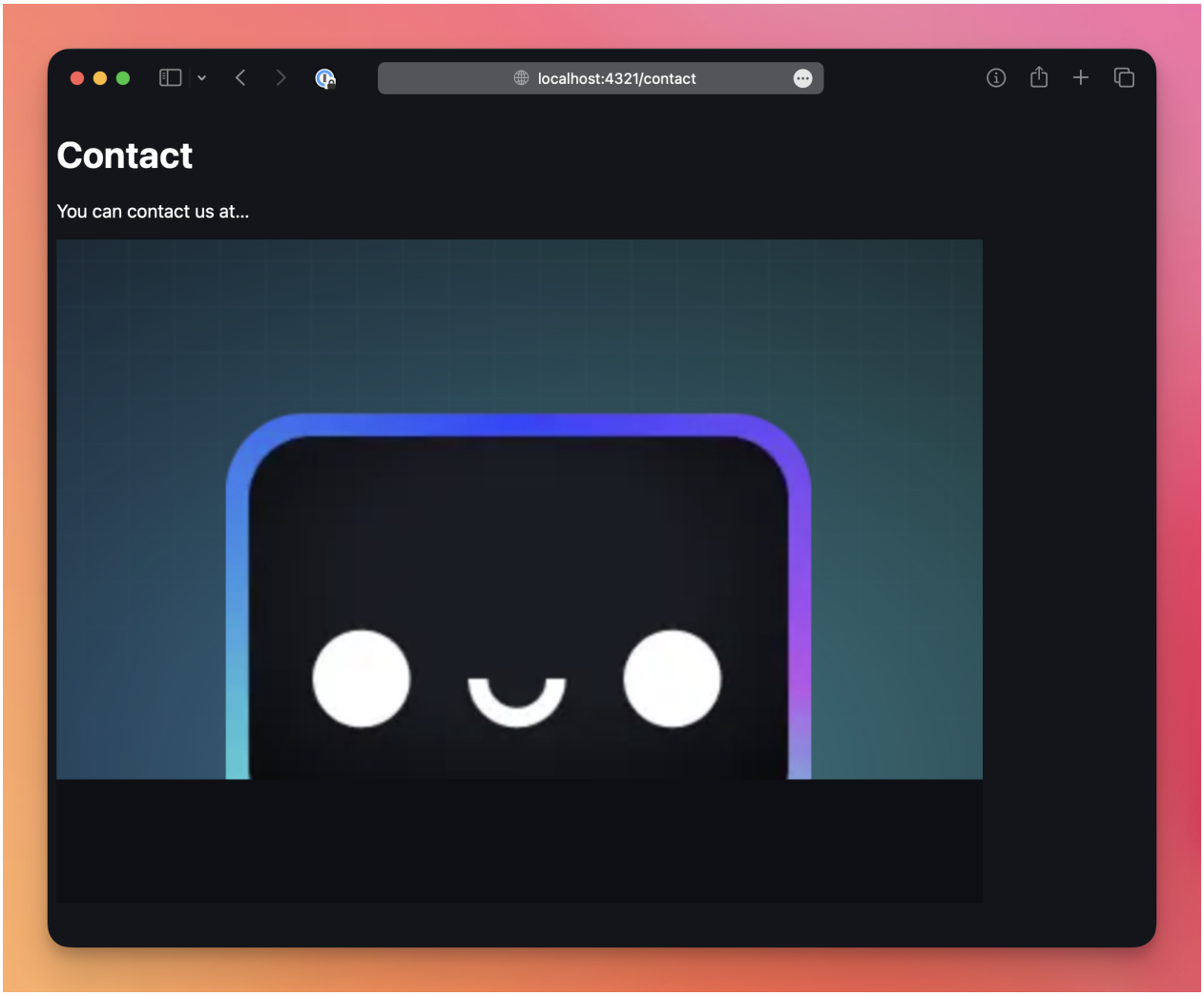
Images

You can add images to the `/public` folder in your Astro site and they will be accessible through the browser.

For example upload an image to `/public/test.png`, you'll be able to reach it at http://localhost:4321/test.png.

You can use those images in your components by using an `<img />` HTML tag or markdown files:

```
---
layout: ../layouts/Layout.astro
---

# Contact

You can contact us at...

![](/test.png)
```



Images stored in `/public` are served as-is.

However Astro can do a lot more.

But to unlock the functionality you have to store images inside `src`, for example in a new folder called `src/images`.

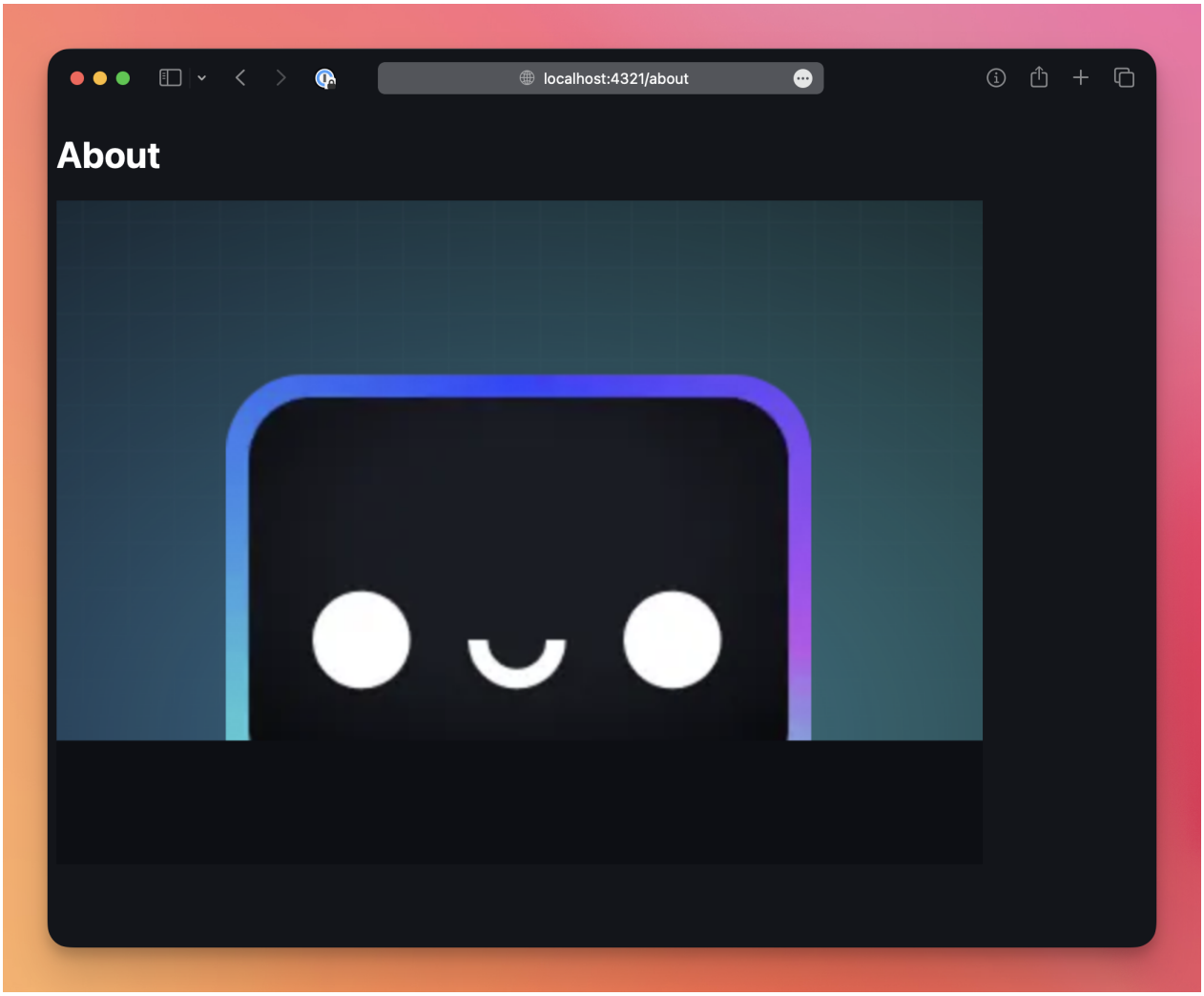
Now from a page component, for example `src/pages/about.astro`

```
---
import Layout from '../layouts/Layout.astro'
---

<Layout title='About'>
  <h1>About</h1>
</Layout>
```

we can add images using the HTML `<img />` tag in this way:

```
---
import Layout from '../layouts/Layout.astro'
import testImage from '../images/test.png'
---
```

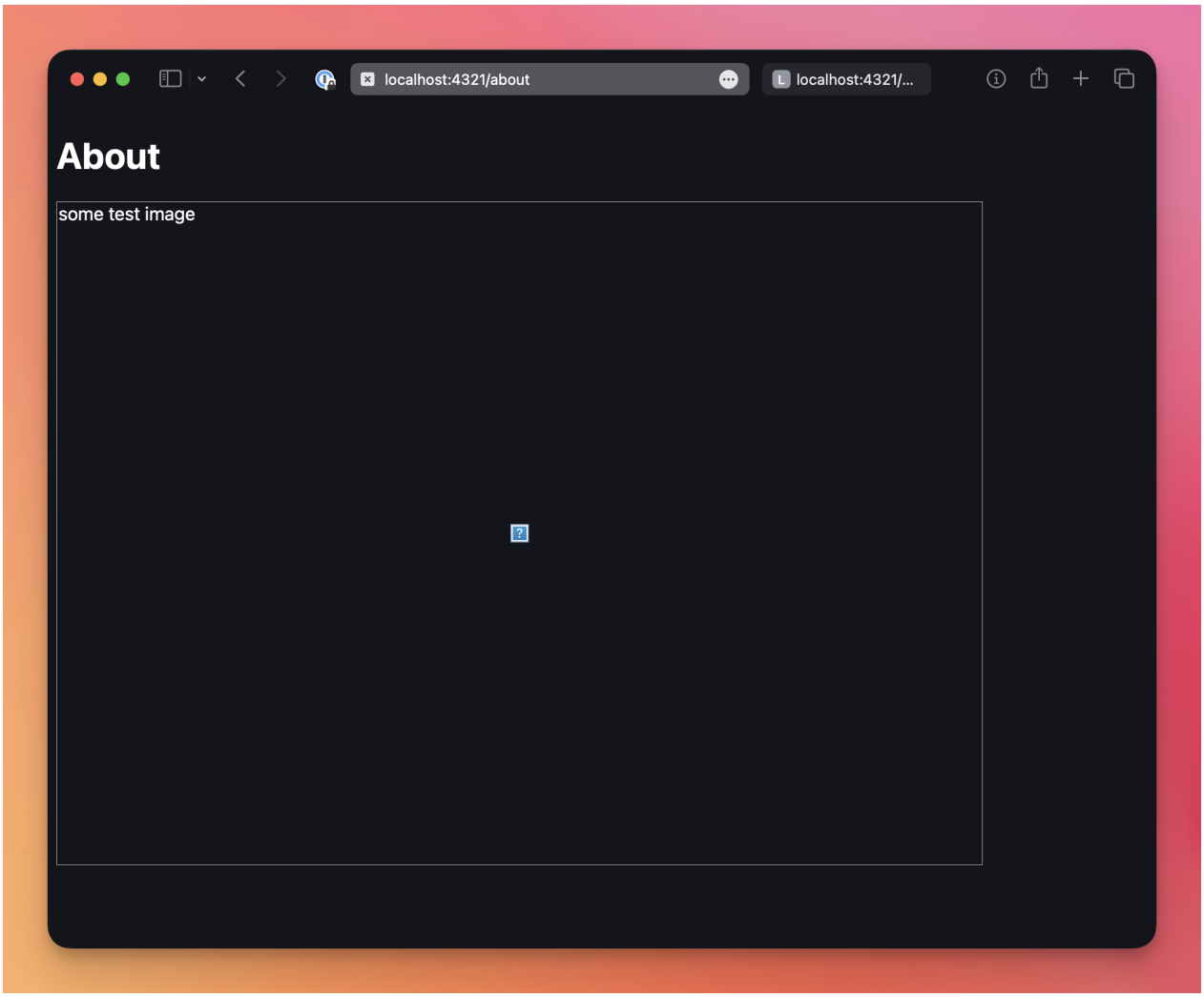
```
<Layout title='About'>
  <h1>About</h1>
  <img src={testImage.src} alt="some test image">
</Layout>
```



But we can also use the `<Image />` component provided by Astro:

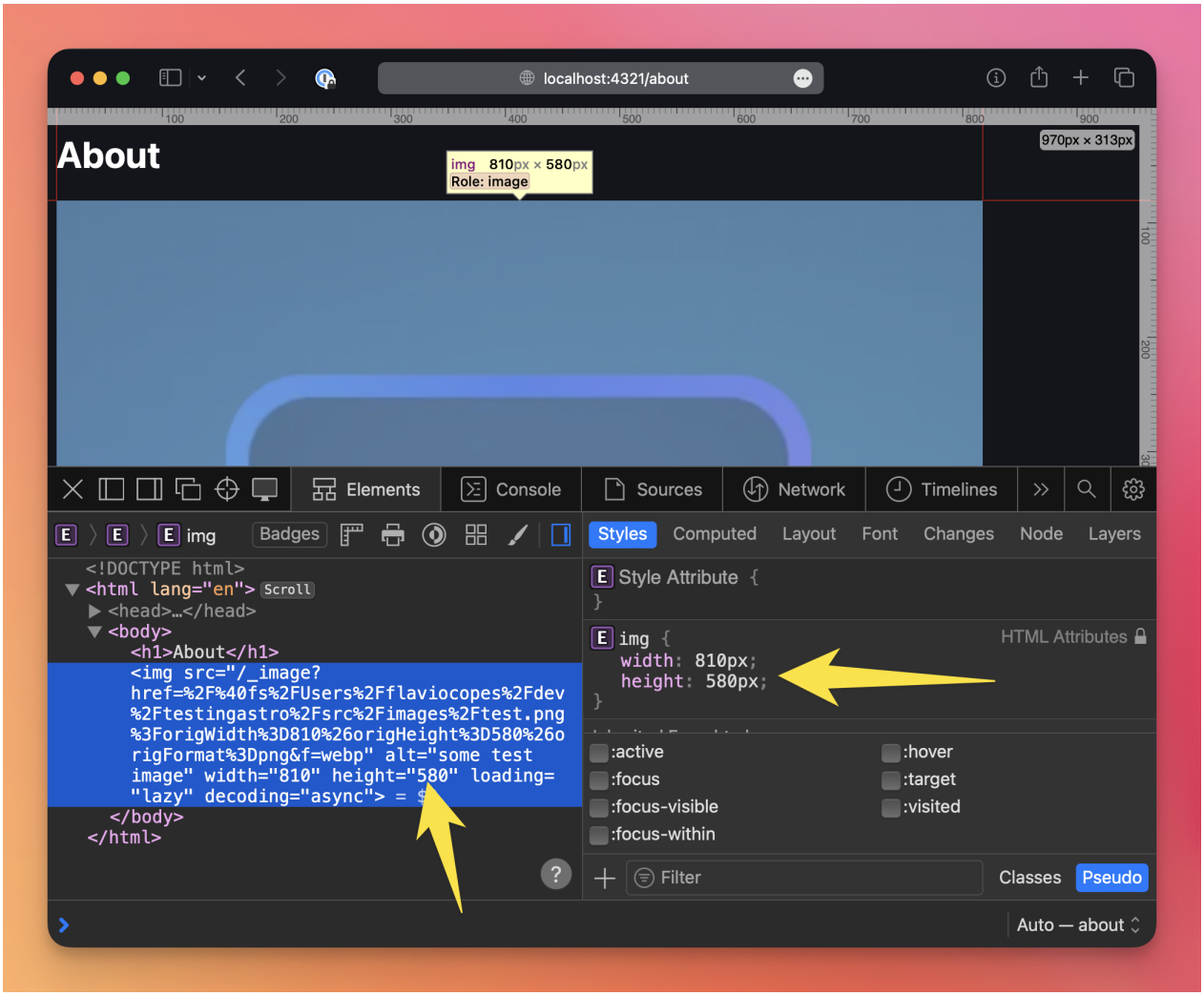
```
---
import { Image } from 'astro:assets';
import Layout from '../layouts/Layout.astro'
import testImage from '../images/test.png'
---

<Layout title='About'>
  <h1>About</h1>
  <Image src={testImage} alt="some test image" />
</Layout>
```

You might get an error, opening the image in a new tab will show a helpful error message saying "Server Error: MissingSharp: Could not find Sharp. Please install Sharp (`sharp`) manually into your project or migrate to another image service."



Go back to the terminal and run `npm install sharp`, then restart the Astro dev server with `npm run dev`

This is worth doing because now Astro will automatically add to the `img` tag the image width and height and makes it load lazily. This improves performance and avoids CLS (cumulative layout shift) - you know, when you refresh a page and there's a ton of layout shifts when images load.



This helps you build a better experience for your users.

In addition to `<Image />`, Astro also provides a `<Picture />` tag that you can use to generate a `<picture>` HTML tag which is handy to generate responsive images.

Content layer API

I talked about how to work with Markdown files in a basic way.

If you build a content-heavy site, Astro has a powerful feature you will want to know about: the **content layer API**.

One word of caution: Astro 2.0 introduced a feature "content collections", but that was deprecated in Astro v5 and considered legacy. If you find old documentation around content collections, just be aware it might be "the old way".

The content layer API lets you define content and render it in pages, in a generic way, so you can use any "content source" to define the content that will be served by your website.

Here I'll show you how to use it to work with markdown files.

For example we want to show a list of blog posts.

Create a folder `src/posts`.

Then create a few blog posts using markdown, for example

- src/posts/first.md
- src/posts/second.md
- src/posts/third.md

This could be the content of src/posts/first.md:

```
---
title: First
date: 2025-02-22
---

First post content
```

If you were tasked to build an e-learning site, you could have to manage lessons, so those would live in src/lessons for example. Or src/data/lessons. You can put them anywhere you want.

Then create a src/content.config.ts file.

By convention, here is where we define our collections.

In there, we define the posts collection like this:

```
import { z, defineCollection } from 'astro:content'
import { glob } from 'astro/loaders'

const posts = defineCollection({
  loader: glob({ pattern: '*.md', base: './src/posts' }),
  schema: z.object({
    title: z.string(),
    date: z.date(),
  }),
})

export const collections = {
  posts: posts,
}
```

> NOTE: the astro:content import might have a red line, because the types are not generated until you run npm run dev in the terminal, don't worry.

Each post will have a title, and a date.

We can add more configuration to add required fields for the frontmatter of each markdown file, so if you miss for example the tag on a post, Astro will complain. But this is a start.

When we used markdown files before, we created them in the src/pages folder, which automatically generated the route for us.

With content collections, we have to handle this ourselves

We create a dynamic route under src/pages. Remember how we made a dynamic route for some test data before?

Now we'll serve content from the posts collection.

Say you want to have a blog in `/blog`, and each post has the route `/blog/<slug>`, like `/blog/first` and `/blog/second`

Create a `src/pages/blog/[slug].astro`

Let's replicate what we had made before with dynamic routes, we got the dynamic parameter from `Astro.params` and we had a frontmatter with a `getStaticPaths()` function exported:

```
---
import Layout from '../../layouts/Layout.astro'

export async function getStaticPaths() {
  return [
    //...
  ]
}
---

<Layout title=''>

</Layout>
```

Here's how it works with the content layer API:

```
---
import { getCollection, render } from 'astro:content'

import Layout from '../../layouts/Layout.astro'

export async function getStaticPaths() {
  const posts = await getCollection('posts')

  return posts.map(post => ({
    params: { slug: post.id },
    props: { post },
  }))
}

const { post } = Astro.props
const { Content } = await render(post)
---

<Layout title={post.data.title}>
  <h1>{post.data.title}</h1>
  <Content />
</Layout>
```

> NOTE: the `id` property of the post is derived from the filename, without the file extension (`first.md` → `first`)

Now we import `getCollection` from Astro, and we use that to query for all the posts data using

`await getCollection('posts')`.

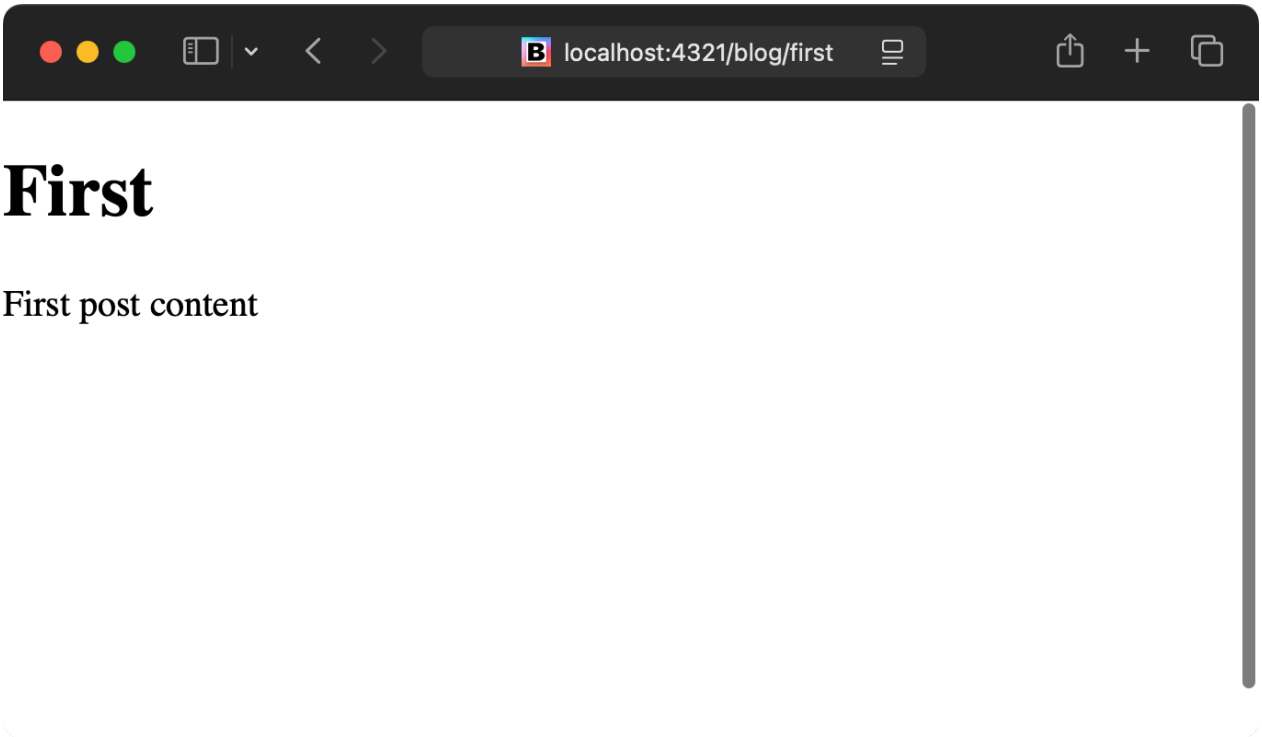
We use this data to populate the posts data returned from `getStaticPaths()`.

The component then when asked to render a single item gets the post data from `Astro.props`, and we use this to get a `<Content />` component, by calling the `render()` function provided by `astro:content`, that's responsible for displaying the content of the markdown file.

Finally, we return the markup.

Here's the result:

http://localhost:4321/blog/first



CSS in Astro

Let's now see how to include CSS in your Astro site.

We've seen that in `.astro` components we can add a `<style>` tag, and inside it we can write CSS that is scoped to the component.

```
<style>

...

</style>
```

If you want, you can make those styles global using

```
<style is:global>

...
```

```
</style>
```

You can import a `.css` file in the component frontmatter:

```
---
import '../styles.css'
---
```

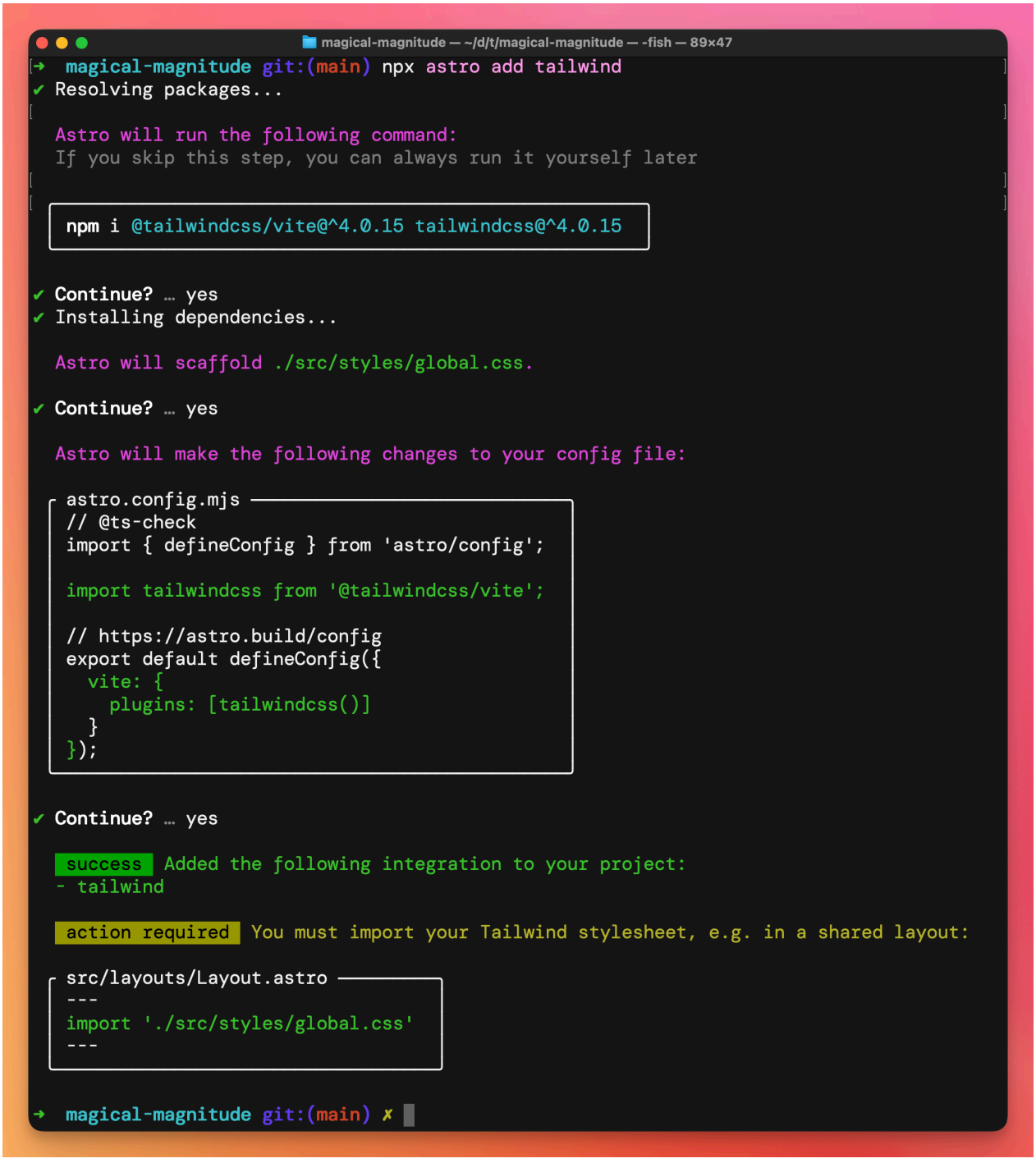
or by including them in your `<html>` page structure, in the layout for example:

```
<html>
  <head>
    <link rel="stylesheet" href="/styles.css" />
    ...
```

In this case the CSS file needs to be in the `public` folder.

To use Tailwind CSS, run the command

```
npx astro add tailwind
```



Astro will configure `astro.config.mjs` to include Tailwind CSS support in the Astro configuration:

```
// @ts-check
import { defineConfig } from 'astro/config';

**import tailwindcss from '@tailwindcss/vite';**

// https://astro.build/config
export default defineConfig({
  **vite: {
    plugins: [tailwindcss()]
  }**
```

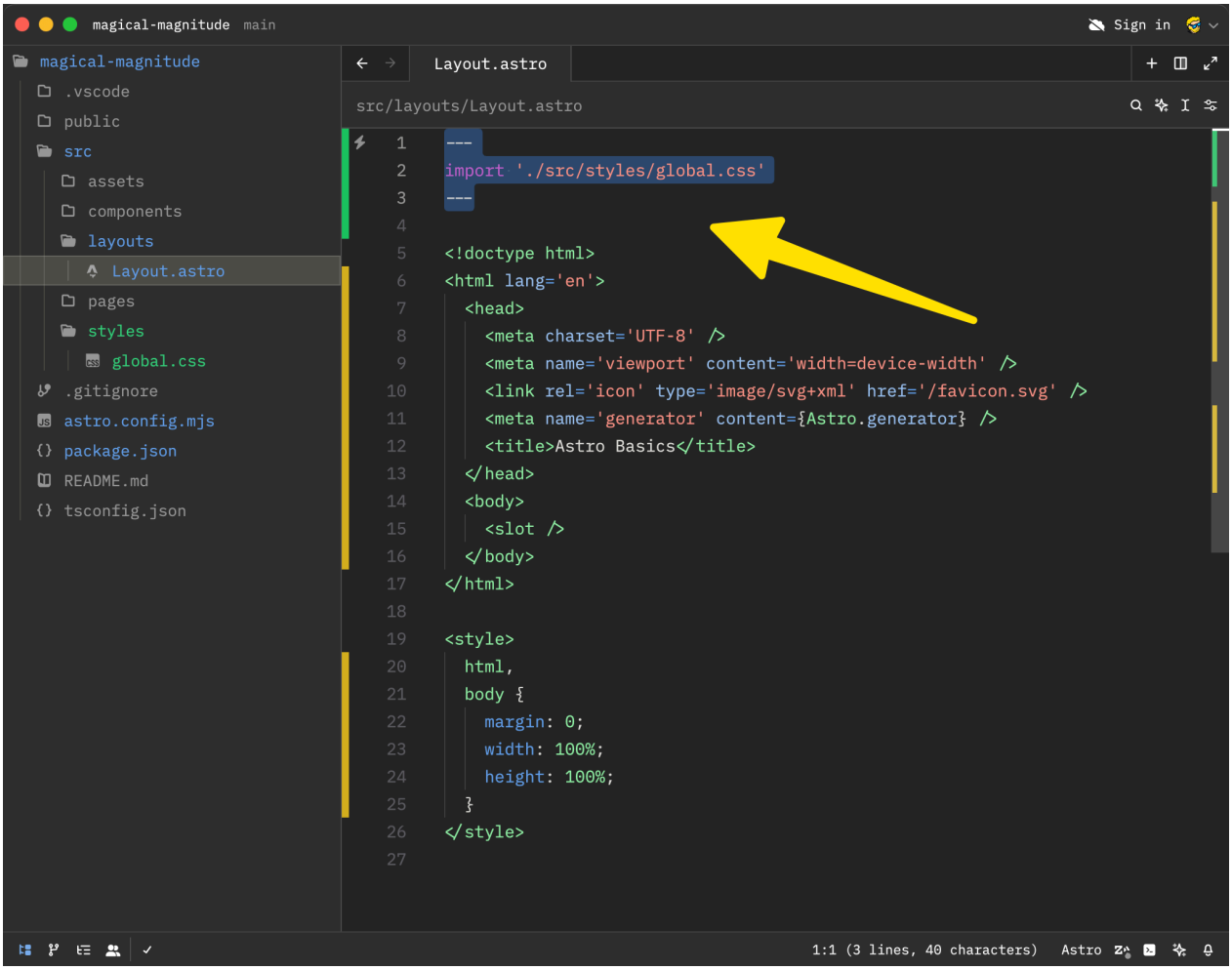
```
});
```

and will create a `src/styles/global.css` file that imports Tailwind, with a single line in it: `@import "tailwindcss";`.

That's it.

You can now use Tailwind classes in your pages and layouts, by importing that CSS file in the frontmatter:

```
---
import '../styles/global.css'
---
```



Running TypeScript code at Build Time

In Astro components we can write JavaScript/TypeScript in the component frontmatter:

```
---
const name = 'joe'
---

<p>test</p>
```

and we can use any variable we define inside the markup:

```
---
const name = 'joe'
---

<p>{name}</p>
```

This code is executed at **build time**.

It's not running in the browser and it's not ran every time a user requests the page.

This can be very useful in many scenarios. For example:

- Fetching data from an API during build
- Processing files or data
- Performing complex calculations

Here's a practical example where we fetch data during build time:

```
---
// This runs at build time only
const response = await fetch('https://api.example.com/data')
const data = await response.json()
---

<div>
  {data.map(item => (
    <p>{item.name}</p>
  ))}
</div>
```

Important considerations when running code at build time:

- The code only runs once during the build process
- You can use Node.js APIs and packages
- API calls are made once during build, reducing load on your servers
- The output is static HTML, improving performance

If you need dynamic data that changes frequently, you might want to consider client-side rendering or server-side rendering instead (we'll see more about this later)

View transitions

By default clicking a link to another page in Astro does a full page reload.

There's a full roundtrip to the browser, with page flickering as the browser has to reload all the HTML and assets.

There is no client-side navigation by default, and this is a good thing, because Astro does not ship a ton of JavaScript to handle it.

But if you need it, **view transitions** can help you.

View transitions are a Web standard that let us create smoother transitions between HTML pages.

In Astro you can enable view transitions in each individual page, or by adding them to your layout(s) they will be applied globally.

You import `ClientRouter` from Astro and include that component in the page `head` tag:

```
---
import { ClientRouter } from "astro:transitions";
---

<!doctype html>
<html lang="en">
    <head>
    <ClientRouter />
    </head>
    <body>
        <slot />
    </body>
</html>
```

Now when the user hovers a link (goes over it with the mouse), browsers that support view transitions are going to prefetch the destination page in the background, store it in their cache, and when the user clicks, they load that page from their cache.

This can make the site feel **very fast**.

And on top of that, the browser does a smooth transition between pages.

Running TypeScript code for each request

I mentioned previously how Astro creates a static site, and how you can run TypeScript code at build time very easily.

However, sometimes you want to run some code for each request.

We call this Server-Side Rendering (SSR).

You can enable it globally (for all the site) or just for some pages.

Perhaps you want to serve some dynamic data from the database. Or whatever.

And still want to use Astro instead of another framework.

In this case, you enable SSR with:

```
npx astro add node
```

This changes the `astro.config.mjs` file to:

```
import { defineConfig } from 'astro/config'

import node from '@astrojs/node'

// <https://astro.build/config>
export default defineConfig({
  output: 'server',
```

```
  adapter: node({
    mode: 'standalone',
  }),
})
```

Notice `output: 'server'`. This makes SSR the default mode, must be disabled on individual pages by setting:

```
export const prerender = true
```

in the frontmatter.

You can also use `output: 'hybrid'` and it makes SSR opt-in on individual pages that should be server-rendered using `export const prerender = false`.

Now you can do interesting things.

For example we can decide to only show a page if it's the weekend, otherwise the page is not shown.

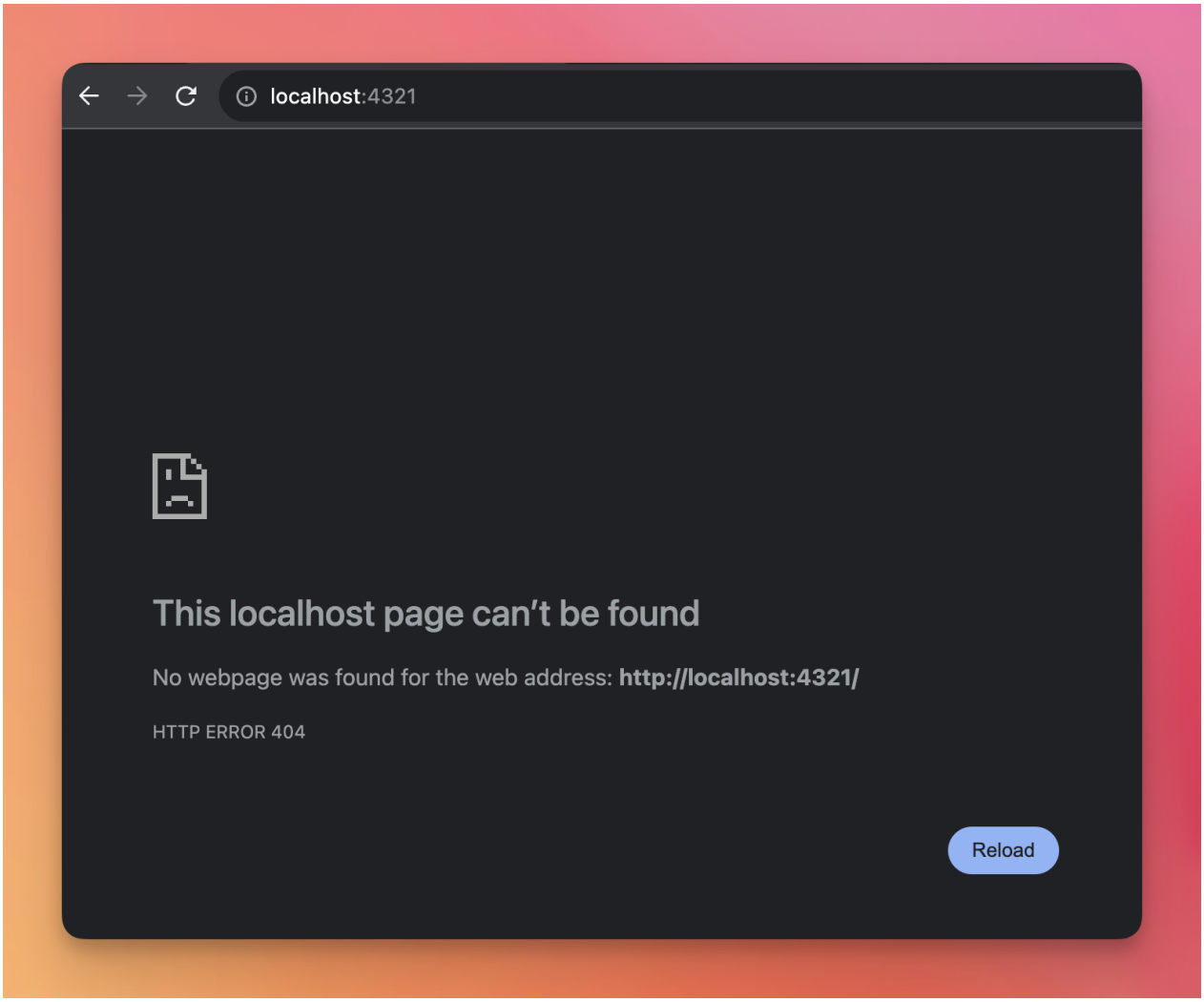
We create a function to detect if today is a weekend day:

```
function isWeekend() {
    const today = new Date();
    const dayOfWeek = today.getDay(); // 0 = Sunday, 6 = Saturday
    return dayOfWeek === 0 || dayOfWeek === 6;
}
```

and then we can call this function on each page request:

```
---
import Layout from '../layouts/Layout.astro'

function isWeekend() {
    const today = new Date();
    const dayOfWeek = today.getDay(); // 0 = Sunday, 6 = Saturday
    return dayOfWeek === 0 || dayOfWeek === 6;
}

if (!isWeekend()) {
  //not weekend!
  return new Response(null, {
    status: 404,
    statusText: 'Not found'
  })
}
---

<Layout title='Homepage'>
  <h1>Homepage</h1>
  <a href="/blog/first" class="underline">See blog post</a>
</Layout>
```

If the current day is not weekend, we return a 404 error:



The ability to run code server side for each request unlocks a lot of options, including processing forms, cookies adding authentication, or anything you'd do with a server-side framework.

API endpoints

One interesting feature we can use in Astro with SSR enabled is API endpoints.

To create an endpoint you add a file ending with `.json.ts` in the `src/pages` folder, and it will be an API route with GET, POST and the other HTTP methods.

For example create this file:

`src/pages/todos.json.ts`

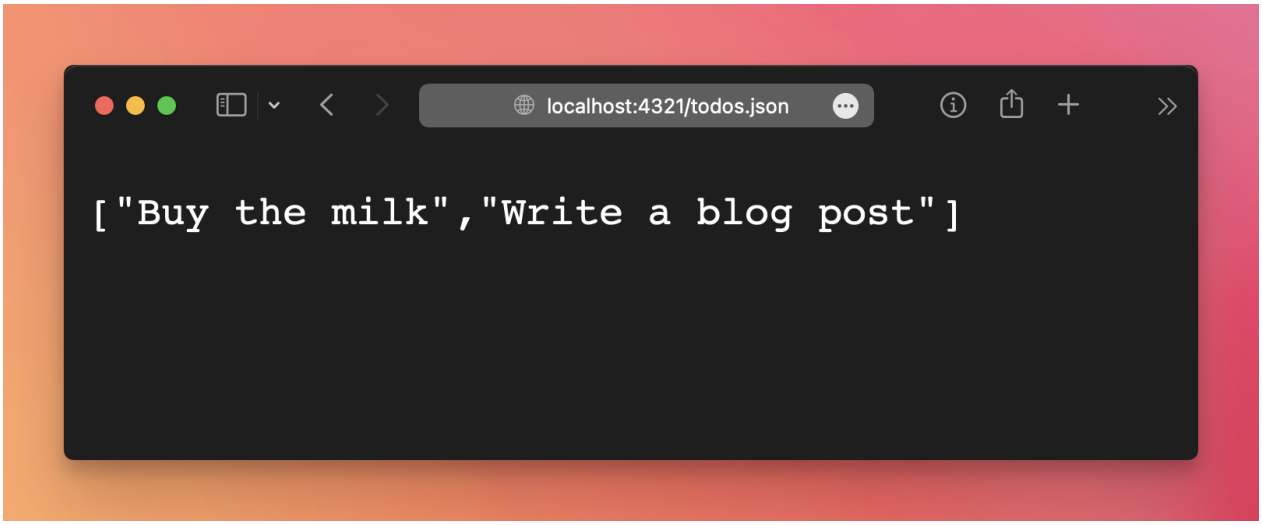
You can define a GET API endpoint by exporting a `GET` function:

```
import type { APIRoute } from 'astro';

export const GET: APIRoute = ({ params, request }) => {

}
```

You can return some JSON now by returning a new Response object with a JSON response:

```
import type { APIRoute } from 'astro'

export const GET: APIRoute = ({ params, request }) => {
  return new Response(JSON.stringify([
      'Buy the milk',
      'Write a blog post'
    ]))
}
```

Hit http://localhost:4321/todos.json to see the JSON returned from your API endpoint:



I returned a hardcoded set of data, but I could have hooked a database here.

You can inspect any data received in the URL using the `params` parameter (only in GET requests), and the request data in `request`.

For example you can inspect the body using `await request.json()` and header data using `request.headers`.

Just as we did a GET endpoint, we can add an HTTP POST endpoint using a `POST` function

```
export const POST: APIRoute = async ({ request }) => {

}
```

For example we can `console.log()` on the server any data coming in the request headers and the request body:

```
import type { APIRoute } from 'astro'

export const POST: APIRoute = async ({ request }) => {
  console.log(request.headers)
  console.log(await request.json())
  return new Response()
}
```

You could use this to add data to a database, handle form submissions, and what not.