Drupal and Astro

Build a fully decoupled application with Drupal and Astro

Vincenzo Gambino

Drupal Developer from London, UK.

14 years experience with Drupal

Speaker at DrupalCamp London, DrupalCamp Munich, Drupal Camp Poland, Drupal Camp Switzerland, Drupaljam Utrecht and Drupal Dev Days Vienna.

Co-author of the Book: Jumpstart Jamstack Development: Build and deploy modern websites and web apps using Gatsby, Netlify, and Sanity.

Concentrix

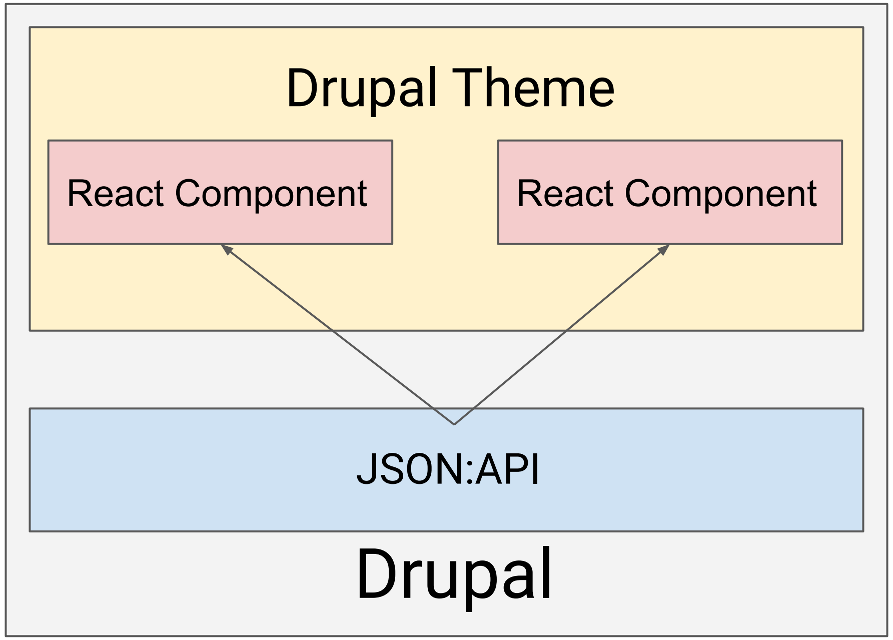
Concentrix is a customer engagement and business performance consulting firm based in the United States. Concentrix has been a subsidiary of SYNNEX Corporation (NYSE: SNX) since 2006 and went public in 2020. Concentrix's headquarters are located in Fremont, California.

Building an employee ideation platform with Drupal.

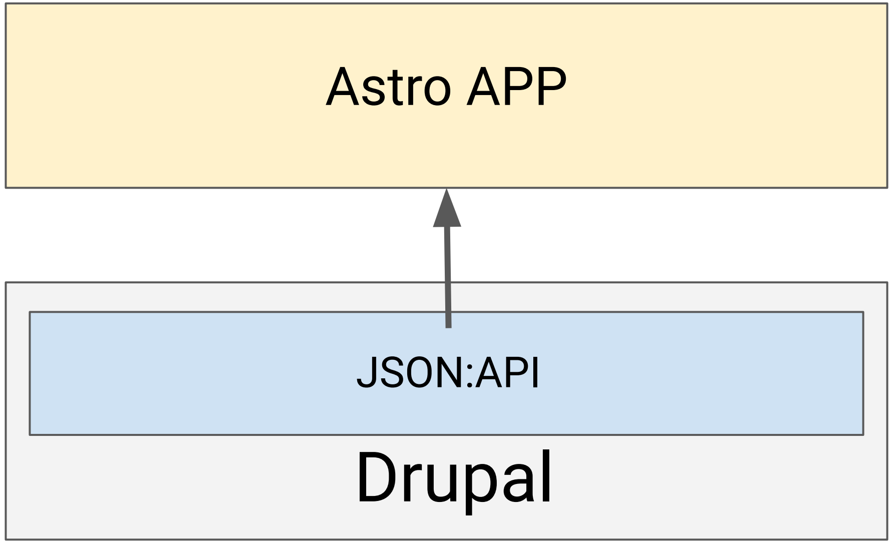
Agenda

- Progressive decoupled vs fully decoupled vs Headless
- Astro Concepts
- Drupal JSON:API
- Drupal+Astro

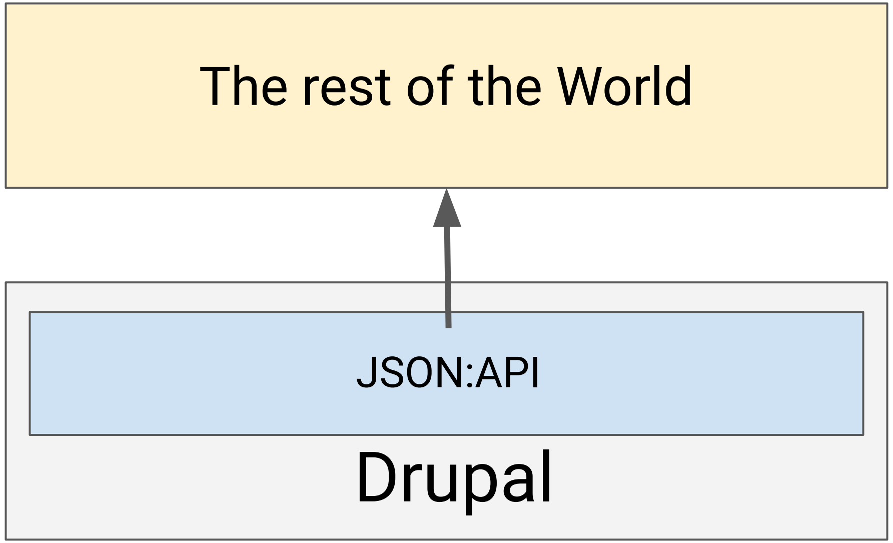
Progressive Decoupled



Decoupled



Headless



Astro

Astro is the web framework for building **content-driven websites** like blogs, marketing, and e-commerce.

Astro is best-known for pioneering a new frontend architecture to reduce JavaScript overhead and complexity compared to other frameworks, providing fast loading and great SEO.

The most important thing to know about Astro components is that they **don’t render on the client**.

They render to HTML either at build-time or on-demand using server-side rendering (SSR).

Features

- **Islands:** A component-based web architecture optimized for content-driven websites.
- **UI-agnostic:** Supports React, Preact, Svelte, Vue, Solid, Lit, HTMX, web components, and more.
- **Server-first:** Moves expensive rendering off of your visitors' devices.
- **Zero JS, by default:** Less client-side JavaScript to slow your site down.
- **Content collections:** Organize, validate, and provide TypeScript type-safety for your Markdown content.
- **Customizable:** Tailwind, MDX, and hundreds of integrations to choose from.

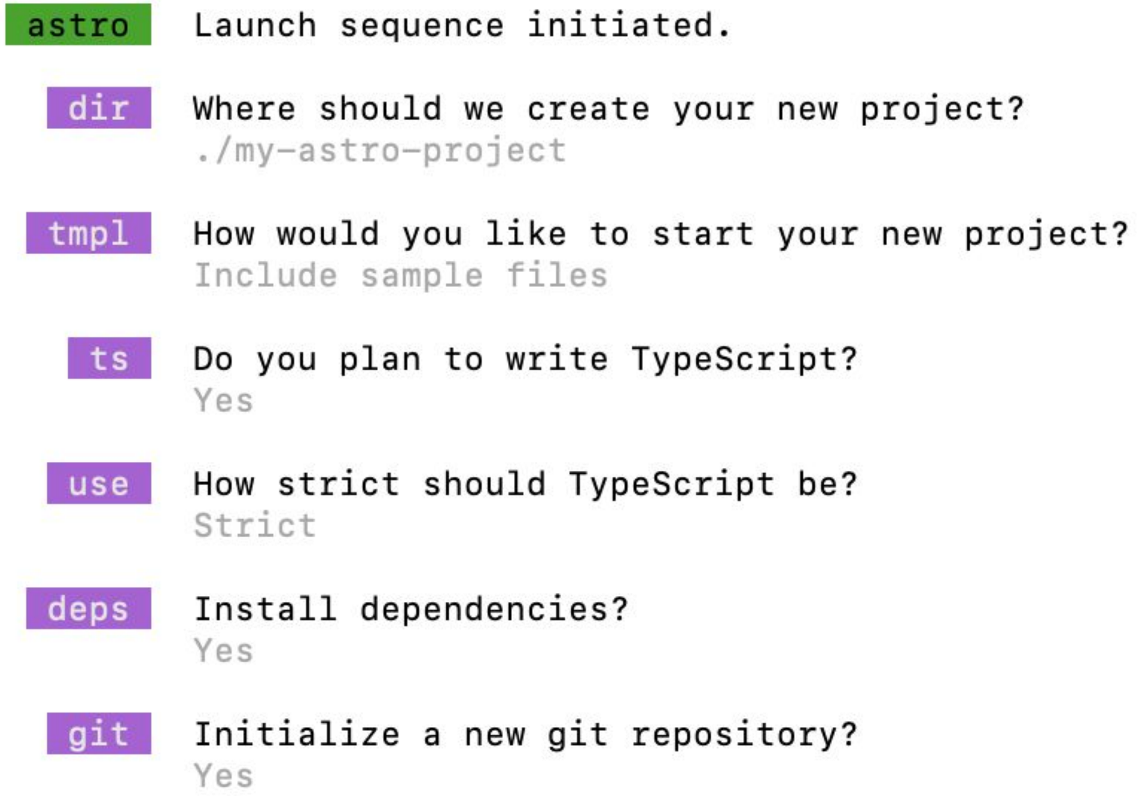
Installation

Prerequisites

- **Node.js** - v18.14.1 or higher.
- **Text editor** - Official Astro extension for VS Code and JetBrains.
- **Terminal** - Astro is accessed through its command-line interface (CLI).

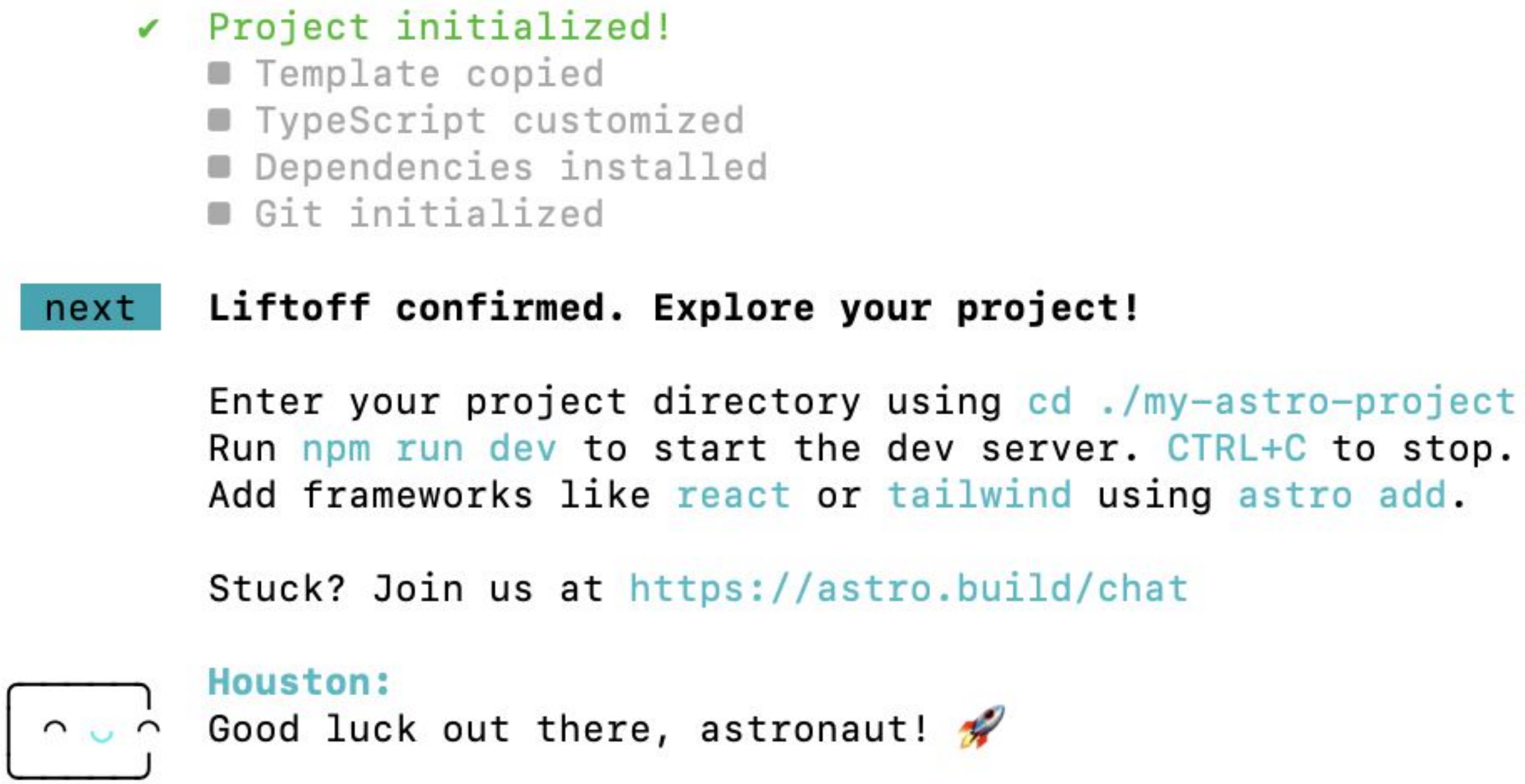
```
# create a new project with npm
npm create astro@latest
```

Installation process



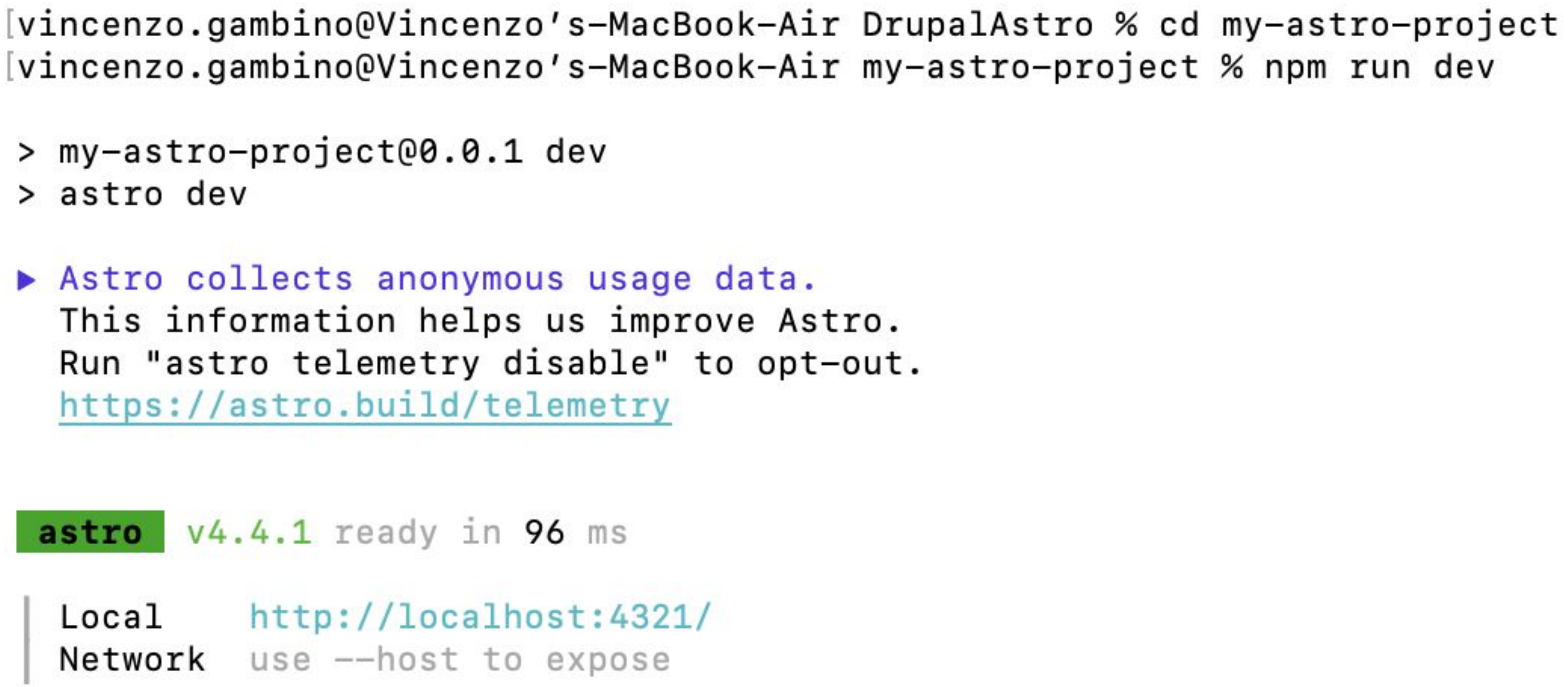
```
astro   Launch sequence initiated.

  dir   Where should we create your new project?
        ./my-astro-project

 tmpl   How would you like to start your new project?
        Include sample files

   ts   Do you plan to write TypeScript?
        Yes

  use   How strict should TypeScript be?
        Strict

 deps   Install dependencies?
        Yes

  git   Initialize a new git repository?
        Yes
```

Installation completed



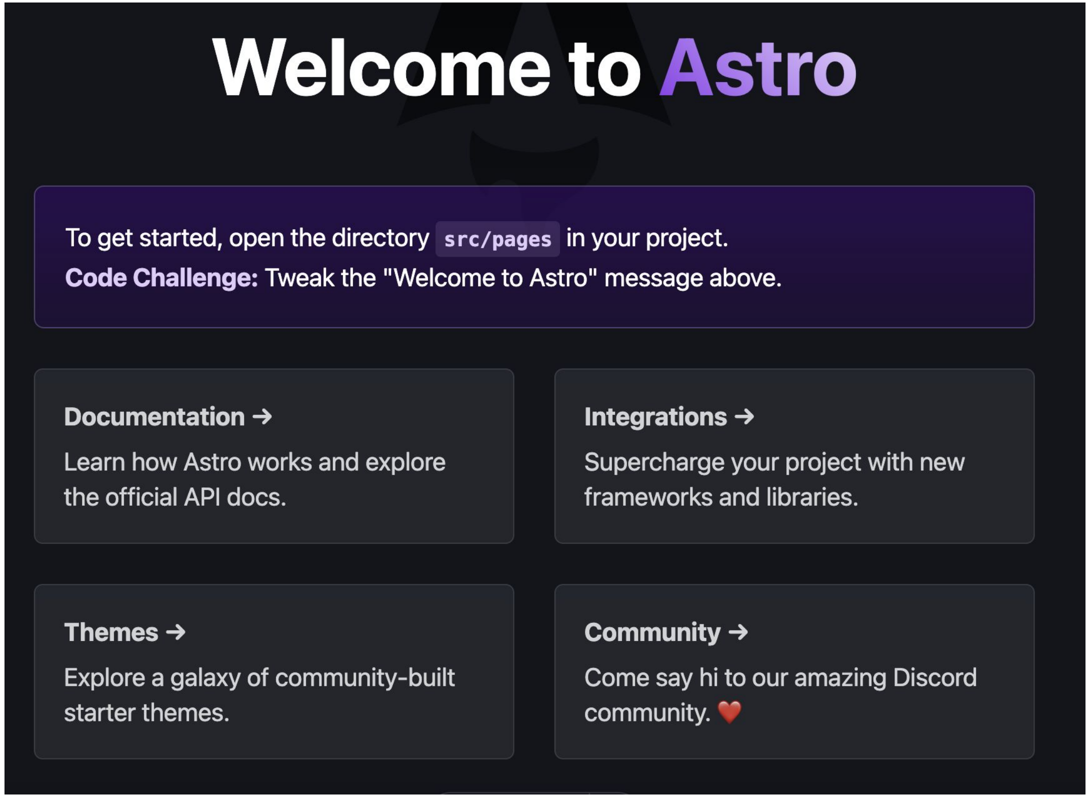
```
✔  Project initialized!
   ■ Template copied
   ■ TypeScript customized
   ■ Dependencies installed
   ■ Git initialized

 next   Liftoff confirmed. Explore your project!

        Enter your project directory using cd ./my-astro-project
        Run npm run dev to start the dev server. CTRL+C to stop.
        Add frameworks like react or tailwind using astro add.

        Stuck? Join us at https://astro.build/chat

╭─────╮  Houston:
│ ◠ ◡ ◠  Good luck out there, astronaut! 🚀
╰─────╯
```

Start Astro



```
[vincenzo.gambino@Vincenzo's-MacBook-Air DrupalAstro % cd my-astro-project
[vincenzo.gambino@Vincenzo's-MacBook-Air my-astro-project % npm run dev

 > my-astro-project@0.0.1 dev
 > astro dev

 ▶ Astro collects anonymous usage data.
   This information helps us improve Astro.
   Run "astro telemetry disable" to opt-out.
   https://astro.build/telemetry


 astro  v4.4.1 ready in 96 ms

 ┃ Local    http://localhost:4321/
 ┃ Network  use --host to expose
```

Astro homepage



Welcome to Astro

To get started, open the directory `src/pages` in your project.
Code Challenge: Tweak the "Welcome to Astro" message above.

Documentation →
Learn how Astro works and explore the official API docs.

Integrations →
Supercharge your project with new frameworks and libraries.

Themes →
Explore a galaxy of community-built starter themes.

Community →
Come say hi to our amazing Discord community. ❤️

Project Structure

Astro leverages an opinionated folder layout for your project. Every Astro project root should include the following directories and files:

- **src/*** - Your project source code (components, pages, styles, etc.)
- **public/*** - Your non-code, unprocessed assets (fonts, icons, etc.)
- **package.json** - A project manifest.
- **astro.config.mjs** - An Astro configuration file. (recommended)
- **tsconfig.json** - A TypeScript configuration file. (recommended)

Astro Components

Astro components are the basic building blocks of any Astro project. They are HTML-only templating components with no client-side runtime. You can spot an Astro component by its file extension: .astro.

They render to HTML either at build-time or on-demand using server-side rendering (SSR).

You can include JavaScript code inside of your component frontmatter, and all of it will be stripped from the final page sent to your users' browsers.

The result is a faster site, with zero JavaScript footprint added by default.

Component Structure

File: src/component/HelloComponent.astro

```
---
// Frontmatter: Component Script (JavaScript)
const name = 'Drupal';
---
<!-- Component Template (HTML + JS Expressions) -->
<p>Hello { name }!</p>
<!-- Usage: <HelloComponent />  -->
```

Component Example with Props

File: src/components/GreetingHeadline.astro

```
---

const { greeting, name } = Astro.props;

---

<h2>{greeting}, {name}!</h2>
```

Component Example with Props

File: src/components/GreetingCard.astro

```
---
import GreetingHeadline from './GreetingHeadline.astro';
const name = 'Drupal';
---
<h1>Greeting Card</h1>
<GreetingHeadline greeting="Hi" name={name} />
<p>I hope you have a wonderful day!</p>
```

Add custom style to a page/component

File: src/components/GreetingHeadline.astro

```
---

const { greeting, name } = Astro.props;

---
<h2>{greeting}, {name}!</h2>

<style>
   h2 {
       font-size: 18px;
   }
</style>
```

Add custom global style to a page/component

File: src/components/GreetingHeadline.astro

```
---

const { greeting, name } = Astro.props;

---
<h2>{greeting}, {name}!</h2>

<style is:global>
   h2 {
       font-size: 18px;
   }
</style>
```

Combining classes with class:list

File: src/components/GreetingHeadline.astro

```
---
const { isRed } = Astro.props;
---

<!-- If `isRed` is truthy, class will be "box red". →
<!-- If `isRed` is falsy, class will be "box". -->

<div class:list={['box', { red: isRed }]}><slot /></div>

<style>
   .box { border: 1px solid blue; }
   .red { border-color: red; }
</style>
```

CSS Variables

```
---

const foregroundColor = "rgb(221 243 228)";
const backgroundColor = "rgb(24 121 78)";

---

<style define:vars={{ foregroundColor, backgroundColor }}>
   h1 {
       background-color: var(--backgroundColor);
       color: var(--foregroundColor);
   }
</style>

<h1>Hello</h1>
```

Tailwind integration

Tailwind lets you use utility classes instead of writing CSS.

These utility classes are mostly one-to-one with a certain CSS property setting: for example, adding the text-lg to an element is equivalent to setting font-size: 1.125rem in CSS.

Installation:

```
# add tailwind integration with npm
npm astro add tailwind
```

astro.config.mjs

```
import { defineConfig } from 'astro/config';
import tailwind from '@astrojs/tailwind';

export default defineConfig({
   // ...
   integrations: [tailwind()],
});
```

tailwind.config.mjs

```
/** @type {import('tailwindcss').Config} */

export default {
   content: ['./src/**/*.{astro,html,js,jsx,md,mdx,svelte,ts,tsx,vue}'],
   theme: {
       extend: {},
   },
   plugins: [],
};
```

Tailwind Example

File: src/components/GreetingHeadline.astro

```
---

const { greeting, name } = Astro.props;

---

<h2 class="text-4xl font-extrabold">{greeting}, {name}!</h2>
```

Add JavaScript to the page

File: src/components/GreetingHeadline.astro

```
---
const { greeting, name } = Astro.props;
---
<h2>{greeting}, {name}!</h2>
<script define-vars={{ name }}>
   alert(`Hello{name}`);
</script>
```

Import Local Scripts

File: src/components/GreetingHeadline.astro

```
---
const { greeting, name } = Astro.props;
---
<h2>{greeting}, {name}!</h2>

<!-- relative path to script at `src/scripts/local.js` -->
<script src="../scripts/local.js"></script>
<!-- also works for local TypeScript files -->
<script src="./script-with-types.ts"></script>
```

Import External Scripts

File: src/components/GreetingHeadline.astro

```
---
const { greeting, name } = Astro.props;
---
<h2>{greeting}, {name}!</h2>

<!-- absolute path to a script at `public/my-script.js` -->
<script is:inline src="/my-script.js"></script>
<!-- full URL to a script on a remote server -->
<script is:inline src="https://my-analytics.com/script.js"></script>
```

Alert Button component

File: src/components/AlertButton.astro

```
---

---

<button class="alert">Click me!</button>
<script>
   // Find all buttons with the `alert` class on the page.
   const buttons = document.querySelectorAll('button.alert');

   // Handle clicks on each button.
   buttons.forEach((button) => {
       button.addEventListener('click', () => {
           alert('Button was clicked!');
       });
   });
</script>

// Usage <AlertButton />
```

Astro Island

In Astro, an “island” refers to any interactive UI component on the page.

The most obvious benefit of building with Astro Islands is performance: the majority of your website is converted to fast, static HTML and JavaScript is only loaded for the individual components that need it.

You can tell Astro exactly how and when to render each component attaching a special client directive that tells Astro to only load the island when it becomes visible on the page. If the user never sees it, it never loads

Install React

Installation:

```
# add react integration with npm
npm astro add react
```

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import tailwind from '@astrojs/tailwind';
import react from '@astrojs/react';

export default defineConfig({
   // ...
   integrations: [tailwind(), react()],
});
```

Counter.jsx Component

```
import { useState } from 'react';

export default function Counter() {

   const [count, setCount] = useState(0);

   function handleClick() {
       setCount(count + 1);
   }

   return (
       <button onClick={handleClick}>
           You pressed me { count} times
       </button>
   );
}
```

Example

File: src/components/CounterCard.astro

```
---
import Counter from './Counter.jsx';
const name = 'Drupal';
---
<h1>Counter Card</h1>
<Counter client:visible />
<p>I hope you have a wonderful day!</p>
```

Client Directives

`client:load` - Load and hydrate the component JavaScript immediately on page load

`client:idle` - Load and hydrate the component JavaScript once the page is done with its initial load and the requestIdleCallback event has fired.

`client:visible` - Load and hydrate the component JavaScript once the component has entered the user’s viewport.

`client:visible={{rootMargin}}` When rootMargin is specified, the component JavaScript will hydrate when a specified margin (in pixels) around the component enters the viewport, rather than the component itself.

Pages

Pages are files that live in the src/pages/ subdirectory of your Astro project. They are responsible for handling routing, data loading, and overall page layout for every page in your website.

Astro supports the following file types in the src/pages/ directory:

- `.astro`
- `.md`
- `.mdx` (with the MDX Integration installed)
- `.html`
- `.js/.ts` (as endpoints)

Routing

Astro leverages a routing strategy called file-based routing. Each file in your src/pages/ directory becomes an endpoint on your site based on its file path.

/src/pages/about-us.astro *is reachable at www.yoursite.com/about-us*

A single file can also generate multiple pages using dynamic routing. This allows you to create pages even if your content lives outside of the special /pages/ directory, such as in a content collection or a CMS.

Routing Structure

- src
  - pages
    - index.astro
    - about-us.astro
    - recipes/
      - index.astro
      - [slug].astro
    - articles/
      - [page].astro
      - [id].astro
    - [...slug].astro

Page example

File: src/pages/index.astro (Homepage)

```
---
const pageTitle = 'Umami Demo';
---

<html lang="en">
  <head>
   <title>{pageTitle}</title>
  </head>
  <body>
   <h1>Welcome to {pageTitle}!</h1>
  </body>
</html>
```

Page example

File: src/pages/about-us.astro (/about-us)

```
---
const pageTitle = 'About US | Umami Demo';
---

<html lang="en">
  <head>
   <title>{pageTitle}</title>
  </head>
  <body>
   <h1>Welcome to {pageTitle}!</h1>
  </body>
</html>
```

Slots

File: src/Layouts/Layout.astro

```
---
const {pageTitle} = Astro.props;
---

<html lang="en">
  <head>
   <title>{pageTitle}</title>
  </head>
  <body>
   <slot />
  </body>
</html>
```

Slot

File: src/pages/index.astro

```
---
import Layout from '../layouts/Layout.astro';
---
<Layout pageTitle='Umami Demo'>
   <h1>Welcome to Umami demo!</h1>
<Layout />
```

Data fetching

```
---

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

const response = await fetch('https://randomuser.me/api/' );

const data = await response.json();

const randomUser = data.results[0];

---

<Layout title='Umami Demo' >

   <h1>User</h1>

   <h2>{randomUser.name.first} {randomUser.name.last}</h2>

<Layout />
```

Dynamic Route

An Astro page file can specify dynamic route parameters in its filename to generate multiple, matching pages.

For example, **src/pages/recipes/[recipe].astro** generates a page for every recipe on your app and ***recipe*** becomes a parameter that you can access from inside the page.

In Astro’s default static output mode, these pages are generated at build time, and so you must predetermine the list of recipe that get a corresponding file. In SSR mode, a page will be generated on request for any route that matches.

Dynamic Route SSG

Because all routes must be determined at build time, a dynamic route must export a **getStaticPaths()** that returns an array of objects with a params property. Each of these objects will generate a corresponding route.

[id].astro defines the dynamic **id** parameter in its filename, so the objects returned by **getStaticPaths()** must include **id** in their params. The page can then access this parameter using Astro.params.

Dynamic Route Example

File: src/pages/[id].astro

```
---
export function getStaticPaths() {
   return [
       {params: {id: 'cliff'}, props: {title: 'Cliff'}}, //Path: /cliff
       {params: {id: 'rover'}, props: {title: 'Rover'}}, //Path: /rover
       {params: {id: 'spot'}, props: {title: 'Sport'}} //Path:/spot
   ];
}
const {id} = Astro.params;
const {title} = Astro.props;
---

<div>Your ID is, {id} and your title is {title}!</div>
```

Recipes pages:

File: src/pages/[recipe].astro

```
---
export async function getStaticPaths() {
   const recipes = await getRecipes();
   return recipes.map((recipe) => {
   return {
       params: { recipe: recipe.path.alias },
       props: { recipe: recipe}
       }
   });
}

const {recipe} = Astro.props;
---

<h1>{ recipe.title }</h1>
...
```

Drupal REST and JSON:API

REST API vs JSON:API

Choose REST if you have non-entity data you want to expose. In all other cases, choose JSON:API.

Core's REST module allows for anything (any format, any logic, any HTTP method) and extreme configurability. Powerful but complex and hence relatively brittle.

JSON:API focuses on exposing Drupal's biggest strength (entities/data modeling) in a coherent manner. Simple yet sufficiently powerful for most use cases.

JSON:API

JSON:API is designed to minimize both the number of requests and the amount of data transmitted between clients and servers.

This efficiency is achieved without compromising readability, flexibility, or discoverability.

Defines how a client should request that resources be fetched or modified, and how a server should respond to those requests.

HTTP Methods

GET - Retrieve data, can be a collection of resources or an individual resource

POST - Create a new resource

PATCH - Update an existing resource

DELETE - Remove an existing resource

URL Structure

GET, POST

/jsonapi/{entity_type_id}/{bundle_id}

/jsonapi/node/article (list of articles)

/jsonapi/{entity_type_id}/{bundle_id}/{entity_uuid}

/jsonapi/node/article/2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e (a specific article)

PATCH, DELETE

/jsonapi/{entity_type_id}/{bundle_id}/{entity_uuid}

/jsonapi/node/article/2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e

Basic Response

```
{
   data: {
       type: "node--article",
       id: "2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e",
       attributes: {
           title: "Testing JSON:API",
           created: "2024-02-16T18:11:47+00:00",
           changed: "2024-02-16T18:11:46+00:00",
           ...
       },
       relationships: {...},
   }
}
```

Relationships user

```
{
    data: {
        type: "node--article",
        id: "2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e",
        attributes: {...},
        relationships: {
            uid: {
                data: {
                    type: "user–user",
                    id: "d3c1fdb2-b937-41b3-83e9-ea70944a89cc"
                }
            }
        }
    }
}
```

Relationships Taxonomy

```
{
    data: {
        type: "node--article",
        id: "2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e",
        attributes: {...},
        relationships: {
            field tags: {
                data: {
                    type: "taxonomy-term--tags",
                    id: "d3c1fdb2-b937-41b3-83e9-ea70944a89cc"
                }
            }
        }
    }
}
```

Retrieving certain fields

Retrieve only certain field by adding the Query String ***field*** to the request.

GET: /jsonapi/{entity_type_id}/{bundle_id}?**field[entity_type]=field_list**

Examples:

/jsonapi/node/article?**fields[node--article]=title,created**

/jsonapi/node/article/2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e?**fields[node--article]=title,created,body**

Response with fields

```
{
   data: {
       type: "node--article",
       id: "2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e"
       attributes: {
           title: "Testing JSON:API",
           created: "2024-02-16T18:11:47+00:00",
           body: {
               value: "<p>This is my testing node</p>",
               format: "basic_html",
               processed: "<p>This is my testing node</p>",
               summary: ""
           }
       }
   }
}
```

Filtering

Add a filter to your request by adding the ***filter*** Query String.

The simplest, most common filter is a key-value filter:
?filter[field_name]=value&filter[field_other]=value

Examples:

*?**filter[title]=Testing JSON:API&filter[status]=1***

*?fields[node--article]=title&**filter[title]=Testing JSON:API***

Filtering Options

```
\Drupal\jsonapi\Query\EntityCondition::$allowedOperators = [
    '=', '<>',
    '>', '>=', '<', '<=',
    'STARTS_WITH', 'CONTAINS', 'ENDS_WITH',
    'IN', 'NOT IN',
    'BETWEEN', 'NOT BETWEEN',
    'IS NULL', 'IS NOT NULL',
];
```

Filtering Examples

Simple Filtering

filter[filter-name][condition][path]=my_field

filter[filter-name][condition][operator]=allowedOperators

Filter[filter-name][condition][value] =value

Array of values

filter[username-filter][condition][path]=uid.name

filter[username-filter][condition][operator]=IN

filter[username-filter][condition][value][1]=admin

filter[username-filter][condition][value][2]=john

Sorting

Add a sorting to your request by adding the **sort** Query String.

The default sorting is ascending.

Sort a collection by its "created" timestamp ASC

?filter[status]=1&**sort=created**

Sort a collection by “created” timestamp, in descending order.

?filter[status]=1&**sort=-created**

Pagination

Add a filter to your request by adding the **page[limit]** Query String.

/jsonapi/{entity_type_id}/{bundle_id}?field[entity_type]=field_list&**page[limit]=limit**

Example:

/jsonapi/node/article?fields[node--article]=title,created&**page[limit]=10**

Result with pagination

```
{
   "data": [
       {"type": "node-article", "id":
"2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e" },
       {"type": "node-article", "id":
"2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e" },
       {"type": "node-article", "id":
"2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e" }
   ],
   "links": {
       "self": "<collection_url>?page[offset]=3&page[limit]=3" ,
       "next": "<collection_url>?page[offset]=6&page[limit]=3" ,
       "prev": "<collection_url>?page[offset]=0&page[limit]=3"
   }
}
```

Includes

Include all related entities using the **include** query string.

*/jsonapi/{entity_type_id}/{bundle_id}?field[entity_type]=field_list&**include=field_name***

Example

/jsonapi/node/article?fields[node--article]=title**&include=uid**

Response Example

```
{
   data: {
       type: "node--article",
       id: "2ee9f0ef-1b25-4bbe-a00f-8649c68b1f7e",
       attributes: {
           title: "Testing JSON:API",
       }
   },
   included: {
       type: "user--user",
       id: "23219f0ef-1b25-4bbe-a00f-8649c68b1f67",
       attributes: {
           name: "John",
           ...
       }
   }
}
```

Response with Tags

Request:
jsonapi/node/article?fields[node--article]=title&include=field_tags&fields[taxonomy_term--tags]=name

```
{
   data: {...},
   included: [{
       type: "taxonomy term--tags",
       id: "2ac25ebd-27a1-4898-92dc-2ac8ebd43088",
       attributes: {
           name: "This is the tag name"
       }
   }]
}
```

Drupal Modules

List of modules

Subrequests

Simple Oauth

JSON:API Extras

JSON:API Menu Items

JSON:API Image Style

JSON:API Include

JSON:API Views

And more…

Astro+Drupal

Umami Demo

Creator: Vincenzo Gambino

Collaborator: Francesco Battaglia

Reviewer: Christopher Pecoraro

Live Demo: https://astrojs-drupal-umami.netlify.app/

Contribute: https://github.com/VincenzoGambino/astrojs-drupal-umami

DrupalJSONAPI Params

This module provides a helper Class to create the required query. While doing so, it also tries to optimise the query by using the short form, whenever possible.

Installation:

```
# add drupal-jsonapi-params integration with npm
npm i drupal-jsonapi-params
```

From *`jsonapi/node/article?fields[‘node–article’]=title&filter[title]={title}`*

DrupalJSONAPI Params Example

```
const params = new DrupalJsonApiParams();

params.addFields("node--article", ["title"])

    .addFilter("title", title)

const path = params.getQueryString();  //fields[‘node–article’]=title&filter[title]=TestingJSONAPI
```

Usage: `` `jsonapi/article?{path}` ``

DrupalJSONAPI Params Example

```
const params =  new DrupalJsonApiParams();
params.addFields("node--article", ["title", "path", "field_media_image"])
    .addInclude(["field_media_image.field_media_image" ])
    .addFields("media--image", ["field_media_image"])
    .addFields("file--file", ["uri", "image_style_uri", "resourceIdObjMeta"])
    .addFilter("status", "1")
    .addSort("created", "DESC");
const path = params.getQueryString();
```

Jsona

This package helps formatting the JSON:API response.

It removed unnecessary nestings and add the includes under the field objects.

Installation

```
# add asona integration with npm
npm i jsona
```

Before Jsona

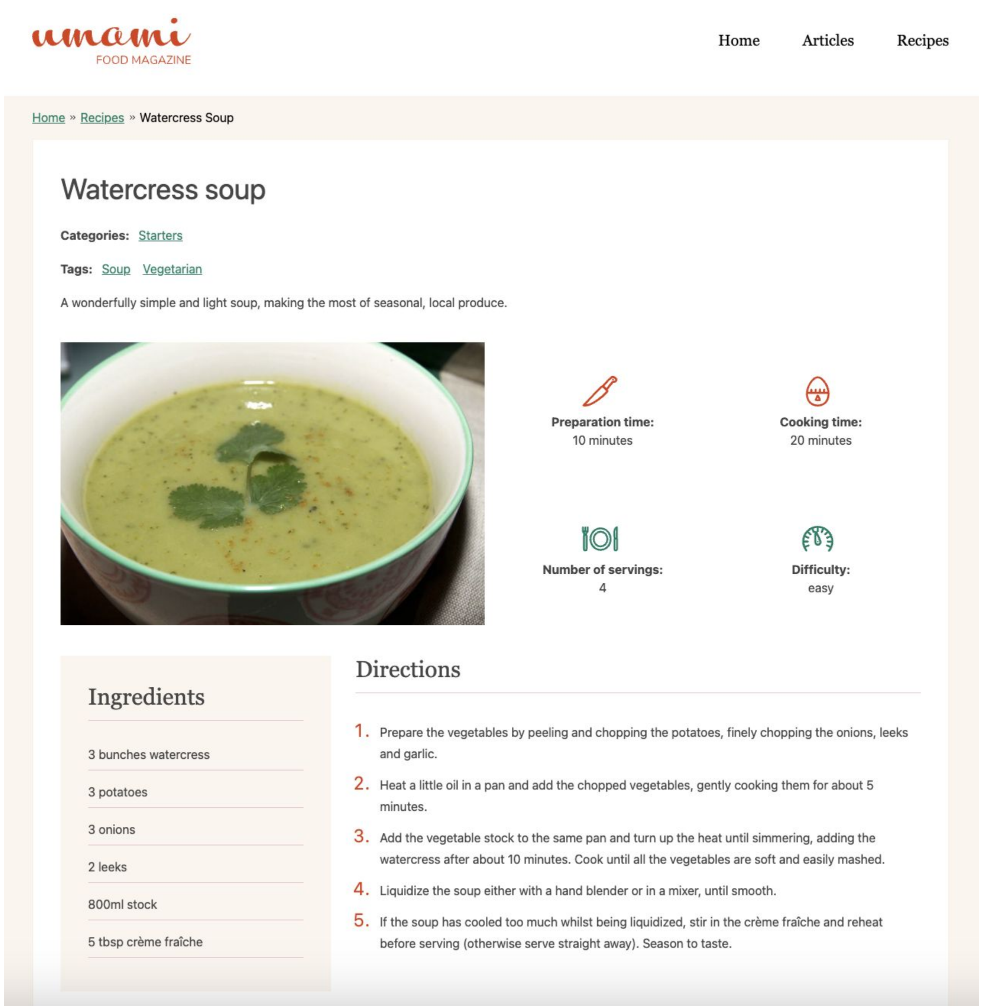
```
const json = {
   data: {
       attributes: {...},
       relationships: {
           uid: {
               data: {
                   type: "user--user",
                   id: "53bb14cc-544a-4cf2-88e8-e9cdd0b6948f"
               }
           }
       }
   },
   included: [{
       type: "user--user",
       id: "23219f0ef-1b25-4bbe-a00f-8649c68b1f67",
       attributes: {
           name: "John",
       }
   }]
}
```

After Jsona

```
const dataFormatter: Jsona = new Jsona();

const data = dataFormatter.deserialize(json);
```

// Output of data

```
{
    title: "Drupal JSON:API",
    uid: {
        type: "user--user",
        id: "53bb14cc-544a-4cf2-88e8-e9cdd0b6948f"
        name: "John",
    },
    relationshipNames: ['uid']
}
```



umami
FOOD MAGAZINE

Home Articles Recipes

Home » Recipes » Watercress Soup

Watercress soup

Categories: Starters

Tags: Soup Vegetarian

A wonderfully simple and light soup, making the most of seasonal, local produce.

Preparation time: 10 minutes	**Cooking time:** 20 minutes
Number of servings: 4	**Difficulty:** easy

Ingredients

- 3 bunches watercress
- 3 potatoes
- 3 onions
- 2 leeks
- 800ml stock
- 5 tbsp crème fraîche

Directions

1. Prepare the vegetables by peeling and chopping the potatoes, finely chopping the onions, leeks and garlic.
2. Heat a little oil in a pan and add the chopped vegetables, gently cooking them for about 5 minutes.
3. Add the vegetable stock to the same pan and turn up the heat until simmering, adding the watercress after about 10 minutes. Cook until all the vegetables are soft and easily mashed.
4. Liquidize the soup either with a hand blender or in a mixer, until smooth.
5. If the soup has cooled too much whilst being liquidized, stir in the crème fraîche and reheat before serving (otherwise serve straight away). Season to taste.

Recipe Pages

Route: **/recipes/url-alias**

Create a file under src/pages/recipes and name it **[recipe].astro**

[recipe] will be the url alias.

In the frontmatter, inside the **getStaticPath()** function, we will get the collection of all recipes, we will loop through the collection, map the path alias to recipe and then pass the content to the props.

We can get a collection of recipes through the function getRecipes().

getRecipes()

```
export const getRecipes = async (): Promise<DrupalNode[]> => {
   const params: DrupalJsonApiParams = new DrupalJsonApiParams();
   params.addFields("node--recipe", [
       "title", "status", "path", "field recipe category", "field cooking time",
       "field difficulty", "field ingredients", "field number of servings",
       "field preparation time","field recipe_instruction", "field_summary",
       "field_tags", "field_media_image",
   ])
       .addInclude([
           "field media image.field_media_image",
           "field recipe_category",
           "field_tags",
       ])
       .addFields("media--image", ["field media image"])
       .addFields("file--file", ["uri", "resourceIdObjMeta"])
       .addFields("taxonomy term--recipe category", ["name", "path"])
       .addFields("taxonomy term--tags", ["name", "path"])
       .addFilter("status", "1");
   const path: string = params.getQueryString();
   return await fetchUrl(baseUrl + '/jsonapi/node/recipe?' + path);
}
```

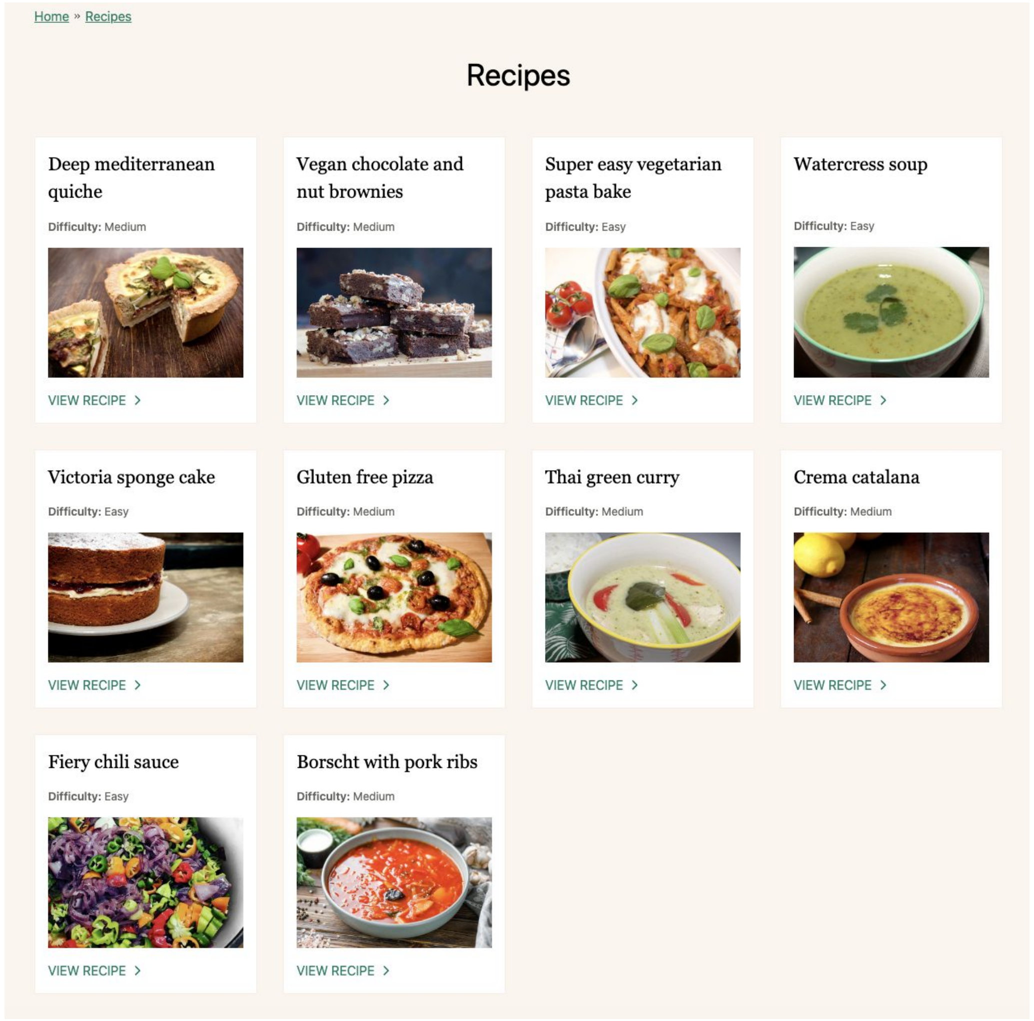
fetch()

```
export const fetchUrl = async (url: string): Promise<any> => {
    const request: Response = await fetch(url);
    const json: string | TJsonApiBody = await request.json();
    const dataFormatter: Jsona = new Jsona();
    return dataFormatter.deserialize(json);
}
```

Recipes Page

File: src/pages/recipes/[recipe].astro

```
---
export async function getStaticPaths() {
   const recipes = await getRecipes();
    return recipes.map((recipe: DrupalNode) => {
       return {
           params: {
               recipe: recipe.path.alias.split('/')[2]
           },
           props: {
               recipe: recipe,
           }
       }
   });
}
const {recipe} = Astro.props;
---

<h1>{recipe.title}</h1>
```



Home » Recipes

Recipes

Deep mediterranean quiche
Difficulty: Medium
VIEW RECIPE >

Vegan chocolate and nut brownies
Difficulty: Medium
VIEW RECIPE >

Super easy vegetarian pasta bake
Difficulty: Easy
VIEW RECIPE >

Watercress soup
Difficulty: Easy
VIEW RECIPE >

Victoria sponge cake
Difficulty: Easy
VIEW RECIPE >

Gluten free pizza
Difficulty: Medium
VIEW RECIPE >

Thai green curry
Difficulty: Medium
VIEW RECIPE >

Crema catalana
Difficulty: Medium
VIEW RECIPE >

Fiery chili sauce
Difficulty: Easy
VIEW RECIPE >

Borscht with pork ribs
Difficulty: Medium
VIEW RECIPE >

Recipe Listing page

Route: **/recipes**

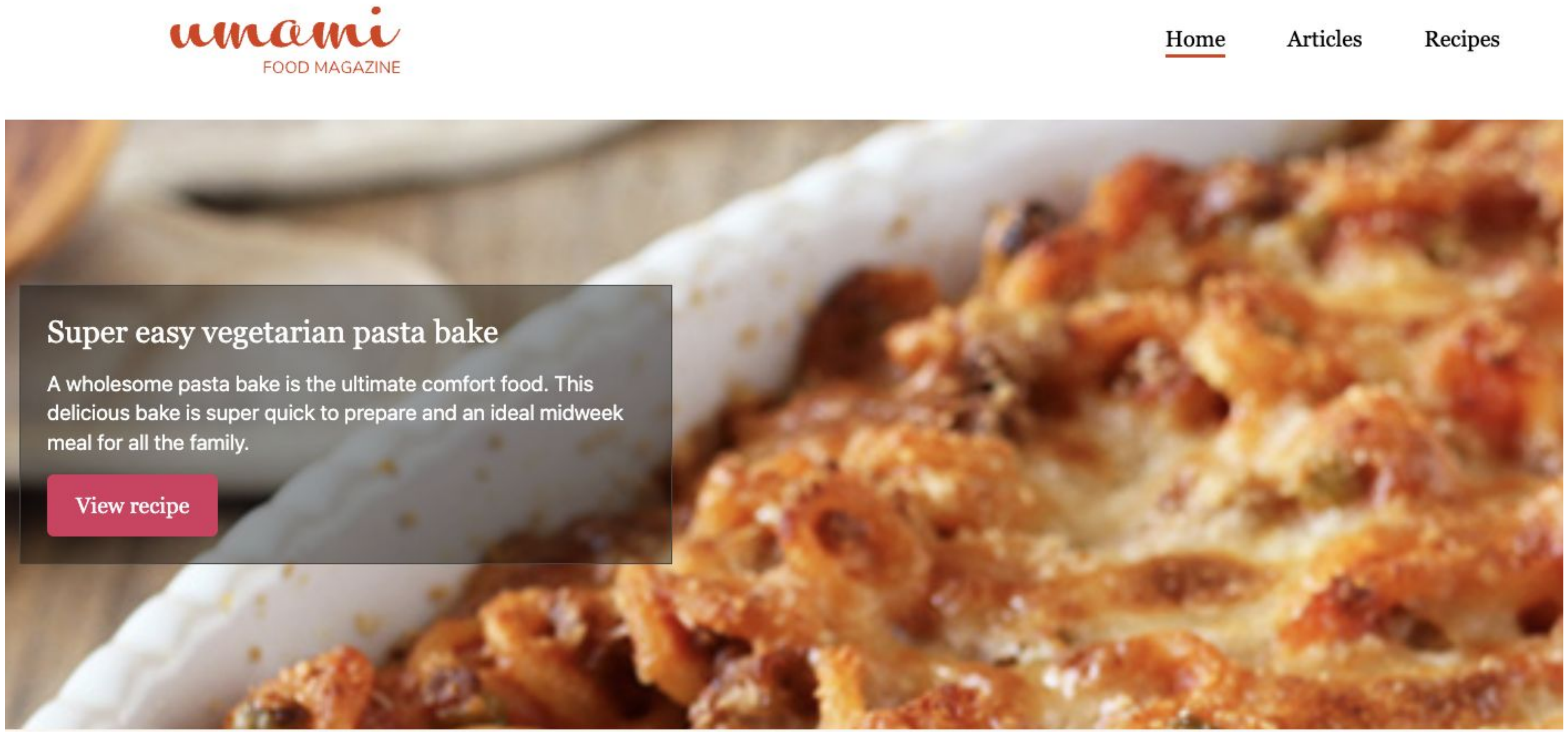
Create a file under src/pages/recipes and name it **[page].astro**

[page] will handle the pagination

In the frontmatter, inside the **getStaticPath()** function, we will get the collection of all recipes, we will loop through the collection, map the path alias to recipe and then pass the conten to the props.

Recipes listing page

```
---
export const getStaticPaths =  (async ({paginate}) => {
   const recipes = await getRecipes();
   return paginate(recipes, {
       pageSize: 10,
   })
}) satisfies GetStaticPaths;
const {page} = Astro.props;

---

<div class="...">
   {page.data.map((recipe: DrupalNode) => (<RecipeCardTeaser recipe={recipe}/>))}
</div>
{page.url.prev ? <a href={page.url.prev}>Previous</a> : null}
{page.url.next ? <a href={page.url.next}>Next</a> : null}
```



Super easy vegetarian pasta bake

A wholesome pasta bake is the ultimate comfort food. This delicious bake is super quick to prepare and an ideal midweek meal for all the family.

Homepage Block Banner

Create a new component under src/components and name it **HomePageBanner.astro**

In the frontmatter, call the funciton **getHomepageBanner()** to retrieve the homepage banner drupal block.

getHomepageBanner()

```
export const getHomepageBanner = async (): Promise<DrupalBlock[]> => {
   const params: DrupalJsonApiParams = new DrupalJsonApiParams();
   params.addFields("block_content--banner_block", [
       "field_title",
       "field_summary",
       "field_content_link",
       "field_media_image",
   ])
       .addInclude(["field_media_image.field_media_image"])
       .addFields("media--image", ["field_media_image"])
       .addFields("file--file", ["uri", "image_style_uri", "resourceIdObjMeta"])
       .addFilter("info", "Umami Home Banner")
       .addPageLimit(1);
   const path: string = params.getQueryString();
   return await fetchUrl(baseUrl + '/jsonapi/block_content/banner_block?'+ path);
}
```

HomepageBanner component

```
---
import { getHomepageBanner, baseUrl } from "../api/drupal";
const data = await getHomepageBanner();
const url = baseUrl + data[0].field_media_image.field_media_image. uri.url;
---

<div class="...">
 <div class="..."><img src={url}/></div>
 <div class="...">
   <div class="...">
     <p class="...">{data[0].field_title}</ p>
     <p class="...">{data[0].field_summary}</ p>
     <a class="..."
href={data[0].field_content_link. uri.replace("internal:",
"")}>{data[0].field_content_link. title}</a>
   </div>
 </div>
</div>
```

Recipe Collections

Alcohol free	Baked	Baking	Breakfast
Cake	Carrots	Chocolate	Cocktail party
Dairy-free	Dessert	Dinner party	Drinks
Egg	Grow your own	Healthy	Herbs

Recipes collection component

Create a new component under src/components and name it **RecipeCollection.astro**

In the frontmatter, call the function **getRecipeCollection()** to retrieve the the recipe tags taxonomy term.

getRecipeCollection ()

```
export const getRecipesCollection = async (): Promise<DrupalTaxonomyTerm[]>
=> {
   const params: DrupalJsonApiParams = new DrupalJsonApiParams();
   params.addFields("taxonomy_term--tags", ["name", "path"])
       .addPageLimit(16);
   const path: string = params.getQueryString();
   return await fetchUrl(baseUrl + '/jsonapi/taxonomy_term/tags?' + path);
}
```

RecipeCollection Component

```
---

import {getRecipesCollection} from "../api/drupal";
const tags = await getRecipesCollection();

---

<section class="...”>
 <div class="...">
   <h2 class="...">
       Recipe Collections
   </h2>
   <div class="...">
       {tags.map((tag) => <a class="..." href={tag.path.alias}>{tag.name}</a>)}
   </div>
 </div>
</section>
```

Resources

Drupal JSON:API: https://www.drupal.org/docs/core-modules-and-themes/core-modules/jsonapi-module

Astro: https://docs.astro.build/

Umami Demo: https://astrojs-drupal-umami.netlify.app/

Drupal JSON-API Params: https://www.npmjs.com/package/drupal-jsonapi-params

Jsona: https://www.npmjs.com/package/jsona

Drupal JSON:API modules: https://www.drupal.org/project/jsonapi/ecosystem

Q&A