Search Lullabot.com

Decoupled Drupal: A Comprehensive Guide

Table of contents:

What does decoupled Drupal mean?

A decoupled CMS separates how content is stored and managed from how content is presented to users. In other words, the back end is “decoupled” from the front end. In a decoupled scenario, a website is just one consumer among many others of the CMS’s data. For example, a mobile app and an interactive JavaScript application also pull from the same content using an API.

Compare this to a traditional CMS, or *monolithic* CMS, where one platform controls everything published to the web. The website, in this case, is expected to be the primary way people consume the content.

Traditionally, Drupal has been a monolithic CMS where you enter content into Drupal, and then Drupal renders that content to the user. If you had to create other services and integrations, such as analytics and newsletters, you integrated them with Drupal.

willing to invest enough resources, but the inclusion of JSON:API made fully decoupled Drupal more accessible. There are also other APIs you can enable with Drupal, like **GraphQL**.

Decoupled vs. headless

Headless is another term used to describe decoupled architecture. The term comes from the idea of cutting off the CMS's head (the front end). Headless Drupal and decoupled Drupal mean the same thing. In the Drupal community, *decoupled Drupal* has become the preferred term. It has nice alliteration, its origins feel less violent, and *decoupled* is the more precise term.

Microservices architecture

Microservice is a term thrown around when talking about decoupled applications. A microservices architecture refers to a front end (like a website or a mobile application) pulling content from other lightweight

these to form a larger application. Each microservice is so small that one can be swapped out with another with little risk of downtime and minimal expense.

That's the theory, anyway. **And there are a lot of successes to back up that theory**. Some of the world's largest, most successful companies migrated to microservices before it was a term everyone used. Amazon AWS came about because of Amazon's desire to decouple their services from one another. In 2017, Netflix made use of over 700 microservices. **Uber faced scaling issues and decided to move toward microservices in 2015**. **Etsy moved to microservices** to help with performance issues and support new mobile application features. These successes are a reason why microservice architectures are even discussed, but there are also negatives. **A previous GitHub CTO called going "full" microservices the biggest architectural mistake of the past decade**. **Even former proponents of microservices have walked back some of their initial enthusiasm**.

Just know that you don't have to decouple Drupal to use microservices. Monolithic Drupal has allowed integrations like this for years, and there's probably already a module for most services you want to use. If not, Drupal's architecture is extensible enough for experts to write one.

A microservices architecture also requires more expertise and larger teams. However, a more traditional architecture works fine if you have a small team or need a small application.

Why structured content is important for decoupled Drupal

Many content-as-a-service providers, like Contentstack and **Contentful**, warn against so-called hybrid CMSes. The front ends of hybrid CMSes still act like traditional CMSes but offer multiple APIs for other front ends and services to access the content. Although, content in a CMS is often unstructured and useless to external services, even if easily accessed, or the service must write a lot of processing around the data it consumes, leading to fragility and maintenance headaches.

Content-as-a-Service providers are correct to point out how a lack of content modeling in many traditional CMSes hinders integrations and makes omnichannel publishing difficult. For example, some Drupal sites

column list" is supposed to be a pricing table. Consumers of an API for this content also have no idea what to expect because they could be getting anything.

The good news is that Drupal has first-in-class content modeling capabilities right out of the box. You don't need to move away from Drupal to model your content, and you'll be hard-pressed to find another CMS with better content-modeling tools, especially in the open-source world. While Drupal is a fantastic hybrid CMS, you have to use the tools intelligently for them to be useful. It takes research and discipline to **properly structure your content for the needs of your users and organization**. We have had a lot of success with domain modeling coupled with **sketching** and **paper prototyping** to inform how best to structure content.

While structured content is required for decoupling your Drupal website, it has benefits even if you don't decouple. It makes your content:

- more findable,
- sortable,
- look consistent across devices

- and much more.

If you have a **design system**, structured content helps your team determine how best to implement it. Additionally, structure adds context and meaning to your content, which you should strive for even if you aren't decoupling your Drupal website.

See how structured content gives the State of Georgia the power to customize landing pages consistently and keep things up-to-date.

JavaScript front ends to use with decoupled Drupal

No JavaScript front end has a monopoly on pairing with decoupled Drupal. However, there have been some favorites. **React** is a popular option, as well as front-end frameworks built with React, like **Next.js** and **Gatsby.js**.

Vue.js claims to be an approachable Javascript framework that is easier to pick up than React. It's also versatile and progressive. You can use it to

suitable for progressively decoupling a website. More on that below.

But Decoupled Drupal is agnostic regarding the framework you choose to consume its content. That's the whole point of going decoupled. The back end doesn't have to know any details about the front end, so choose the framework your team has the most experience with.

Should you decouple your Drupal website?

You must first understand the advantages and disadvantages of decoupling. With decoupling, you get the following benefits:

1. **Clean APIs.** Mobile and JavaScript applications, or other websites, consume your content. Your API becomes a primary initiative and main touchpoint for getting your content everywhere, so anyone depending on it knows they aren't a second-tier user. This API can be documented and provided to the public. More applications can be created without a deep knowledge of the back end.

end, and **the back end can be changed without touching any front ends** (as long as the API remains consistent).

3. **Reduction of individual responsibility and promotion of content reuse.** Back-end developers and content creators don't need to worry about the vagaries of the front end. Front-end developers can focus on styles and layouts without wrestling with back-end assumptions.

There are disadvantages and **hidden costs** that come with decoupling, too. While a decoupled architecture provides flexibility, it comes at the expense of simplicity. Your minimum **development team** size grows, leading to cost increases. You also lose a lot of the built-in functionality included in Drupal's ecosystem of modules, which is easy to take for granted. For example, you need to create a preview system for your editors. Finally, maintaining the benefits of decoupling requires that the whole organization be disciplined to avoid falling back into bad habits. Editors, stakeholders, and developers must work together to avoid the siren song of short-term fixes.

It's also important to pay attention to how your content is currently organized. Are your editors cramming too much into the body field? Does your content have any structure to it? If you want to decouple, you're

Hard problems to solve when decoupling Drupal

Many of Drupal's systems assume that Drupal has control of the back and front ends, so when you move from a monolithic architecture to a decoupled architecture, you run into some gnarly problems. **The good news is that these problems can be solved**.

Image Styles

Image styles allow site builders to define image derivatives that display in various contexts. However, image styles are tied to the needs of the front end, and the back end shouldn't know about those needs. This means that using Drupal's image styles out of the box violates one of the main principles of decoupling. You need to let consumers of your API send what they need in their request. **See how we solved the problem of using image styles**.

Routing

Based on a URL, a Drupal website routes the request and returns the appropriate content. URLs also have redirects and aliases. With decoupled Drupal, you can't assume your content will appear on a web page, and if you make this assumption, adding future API consumers become painful.

Routing in decoupled applications becomes challenging because of three factors:

1. There is no single point of truth for a route. Each front end has its own route for the content it constructs, and then the back end keeps a separate route for when editors need to change something.

2. Each front end has different routing patterns.

3. You lose the one-to-one correspondence with an entity. A front end has curated several entities into one page; by default, the back end has no idea. How do editors know where to go to make changes?

These routing challenges can be overcome, but each solution adds constraints to your applications.

advantage of that system for the back end, but what about the front end? Each front end must develop its own solution if an authentication system is required. Everything is unified with single-sign-on (SSO), which adds costs and complexity.

Decoupled authentication often avoids session cookies, which Drupal core uses by default. Browsers have excellent cookie management, but other front-end stacks don't. However, you can expose Drupal's user authentication system using authentication providers. An example is **Simple OAuth**, which implements the OAuth2 protocol and best practices using Drupal.

Much of the work for decoupled authentication on the back end is solved. The front end, however, still has to deal with things like:

- Storing tokens securely.
- Refreshing tokens that expire.
- Building user interfaces that trigger the log-in and log-out workflows.

Translation

translations with JSON:API lists the current limitations of translating content:

- You can fetch existing translations correctly.
- You can update existing translations correctly.
- You cannot delete an existing translation.
- You cannot create a new translation.

A ***JSON:API Translation* module in Drupal core** may fix these one day. Until then, analyze your translation requirements before going down the rabbit hole.

Progressively decoupling Drupal

You don't have to decouple all at once nor aim for a fully decoupled architecture to realize some of the benefits. For example, **you can create widgets designed to be embedded and configured by Drupal editors.**

React. **Once they are created, they can be embedded in other places, not just your Drupal website**.

Exposing a single content type with an API for other front ends to consume is also a good way to take advantage of decoupling. For example, a restaurant wants a mobile application but doesn't want to double-enter its menu items. They could expose the website's menu items for the mobile app to consume, but the content type must be structured appropriately. However, architecting a single content type is much easier than architecting an entire website. Moreover, the discipline required to structure that content type properly is good practice for the future.

Decoupled Drupal examples

Edutopia.org was a decoupled Drupal website using React as the front end. The system allowed editors to build customized landing pages and control their layouts. It also included a custom-built preview system so editors could experiment with different components before publishing live.

interactivity to Drupal, making it easier to port business logic to other platforms.

Universal Kids has a React front end that pulls content from Drupal. We helped them swap out the back end, which was a combo of Drupal 7 and JavaScript, for a single Drupal 8 instance. Since the APIs remained the same, we didn't have to change the front end. Having a swappable back end is one of the promised benefits of decoupling, and Universal Kids took advantage of it successfully.

As one of the first Drupal agencies, we have the expertise to make your project successful.

LET'S CONNECT

CAREERS

ACCESSIBILITY STATEMENT

TERMS OF SERVICE

PRIVACY POLICY

LULLABOT NEWSLETTER

Stay connected with our latest news, articles, webinars, and more.

SIGN UP