A How-to Guide to Headless and Decoupled CMS

Choosing the right architecture to support Javascripts frameworks

Table of Contents

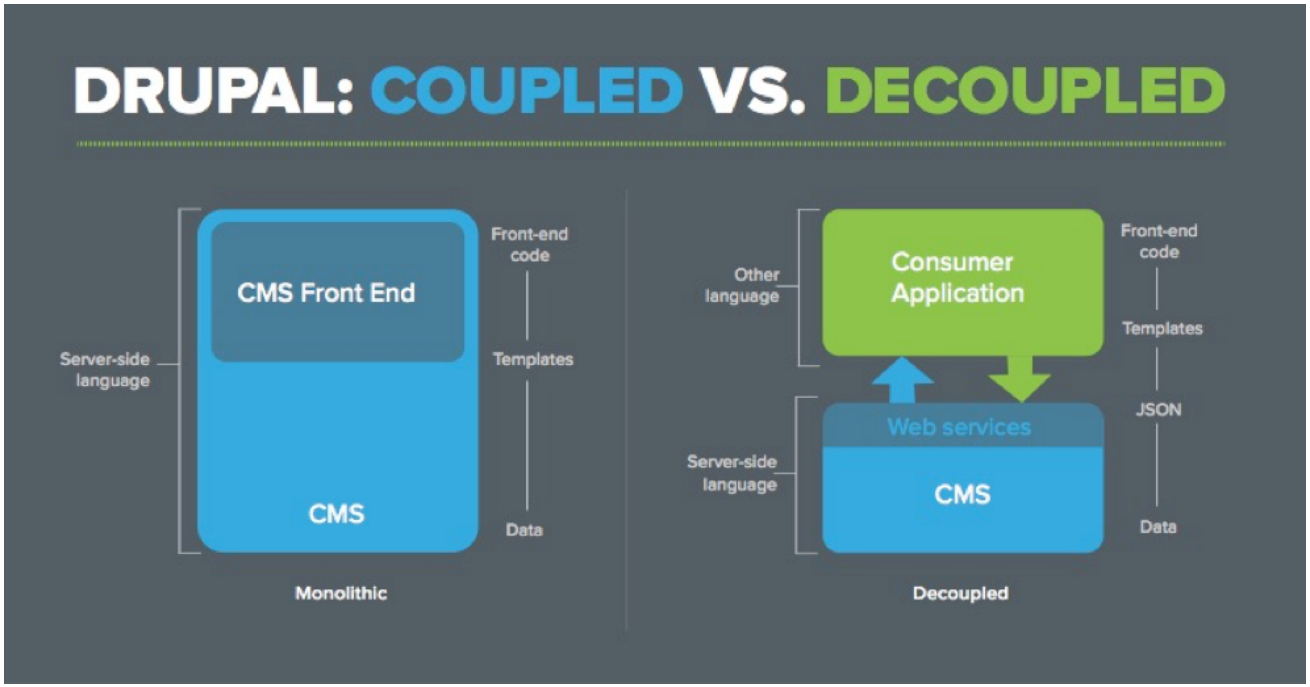
Introduction

Until recently, most websites were built the same way going back to the dawn of the Web in the mid 1990s - with a monolithic architecture, where the Content Management System (CMS) includes both the front-end and the back-end. There are many reasons why this architecture has proven so resilient… having the tools to author a front-end web experience that conforms to a custom built back-end is straightforward and puts power in the hands of developers. But the rapid evolution of diverse end-user clients and applications has given rise to a dizzying array of digital channels to support. These digital options result in a variety of new presentation standards that have resulted in renewed discussion of decoupling front-end presentation layers from the backend content management system. The concept led technology research firms Forrester[1] and Gartner[2] to highlighted so-called headless and decoupled architecture as a core requirement of future digital experience platforms in early 2016. This shows the increasing importance of developing digital applications that share a common platform as we move further in to the 21st century.

"...the rapid evolution of diverse end-user clients and applications has given rise to a dizzying array of digital channels to support."

Websites in the past were built from monolithic architectures utilizing web content management (WCM) solutions that deliver content through a templating solution tightly 'coupled' with the CMS on the back-end. Agile organizations crave flexibility, and strive to manage structured content across different presentation layers consistently in a way that's scalable. Accomplishing this efficiently requires that teams have flexibility in the front-end frameworks that dominate the modern digital landscape. That's why decoupled and headless CMS is taking off. That's why you're here. But now you need the right technology to support the next phase of the web and beyond.

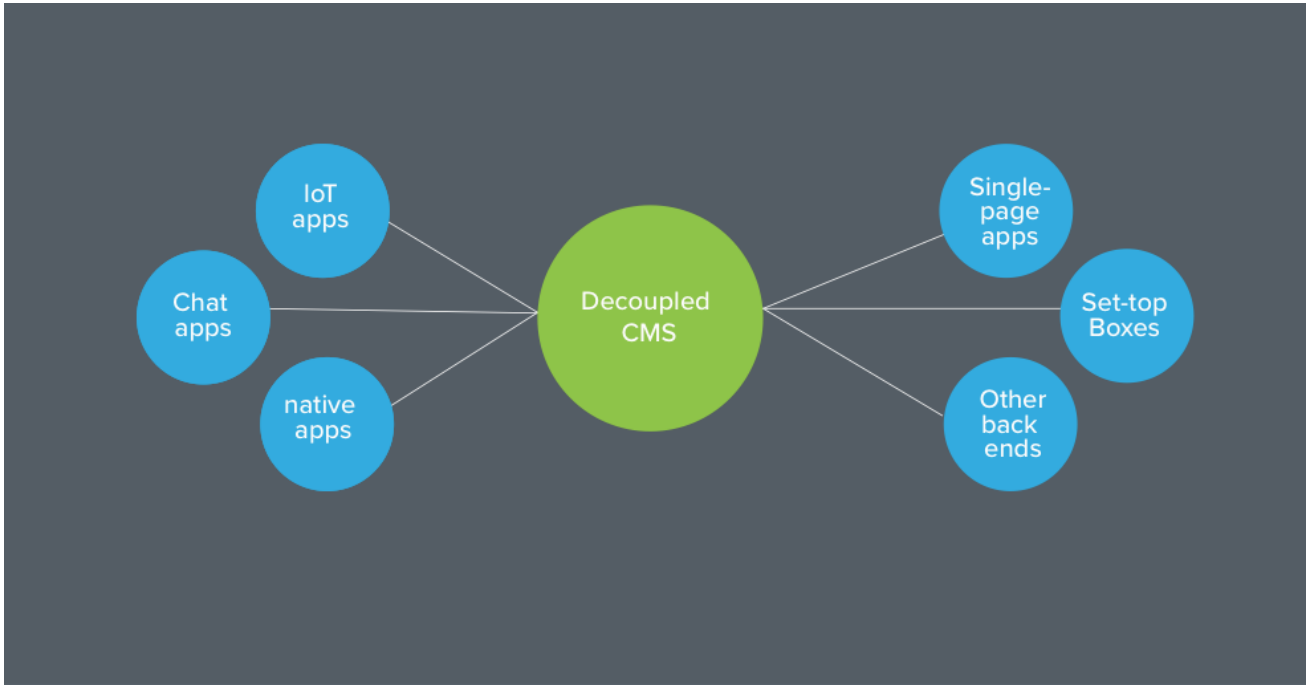
1 "The Rise Of The Headless Content Management System", Forrester Research, March 17, 2016
2 "Digital Experience Platforms Need to Feature Headless Content and Smart Client Capabilities", Gartner, December 6, 2016



Defining Decoupled CMS

So first thing's first: what is the difference between a traditional CMS, headless CMS, and a Decoupled CMS?·

- **Traditional CMS:** Users create content through an editor and store it in a database (the back end). That content is then served into a front-end rendering layer that is tightly coupled to the back end.
- **Headless CMS:** Users create content through an editor and store it in a standalone database fronted by APIs. The content is retrieved by an entirely separate front-end rendering layer via those APIs.
- **Decoupled CMS:** Blends the traditional and headless CMS. Users create content through an editor and store it in a database. The content can be served flexibly either through the existing front-end rendering layer or retrieved by an entirely separate front-end rendering layer via APIs.



In all cases, front-end templates define how content is displayed in the digital application. What is changing is the landscape of languages and frameworks available to the task. In order to meet business (and consumer) demand for a consistent experience across channels, teams need to have expertise across multiple frontend frameworks, and in the last several years that list has changed. Popular languages for traditional website development include PhP, .NET, and Java. However, these languages have been eclipsed. According to a 2016 survey by Stack Overflow, JavaScript is now the dominant programming language for frontend web development.

When considering a traditional CMS architecture versus headless or decoupled architecture, it is important to understand the use cases best suited to the architecture. For example, the majority of projects that use JavaScript front-end frameworks on Node.js web servers typically lean on the inherent strength of Node.js: real-time or asynchronous functionality. Use cases usually revolve around the ability to maintain single threading in a non-blocking way. A common example would be SPA (single-page application) website which would use a combination of multiple views or API endpoints without the need for a page refresh.

Other common examples of this functionality would include:

- Dynamic API Factory
- Real-time or data streaming
- Chatbots / Chat clients
- Messaging over WebSockets

If your organization is evaluating a decoupled or headless architecture, and you have any of the following requirements, a decoupled architecture may **not** be a great fit:

- Full-fledged editorial experience for your content team
- Frequent needs to manipulate display and layout
- End-to-end preview of the application prior to launch
- Minimal resources needed to maintain your digital application infrastructure

Decoupled Drupal: Drupal 8 is an open source, API-First CMS. API-First is what really enables decoupled to happen because the application “phones home” to the CMS to retrieve content. Content can sit in one place and then be distributed outward. Instead of having stack in the silo for a website, you have a hub-and-spoke model. The Decoupled CMS is the hub and the spokes are single-page apps, set-top boxes, and even other back-ends for other apps.

Platform Considerations

Digital platform teams require a system that will help them support web, mobile web, mobile apps, chatbots, voice activated applications and more. Options to support these channels are split among the technology approaches above. Traditional CMSes, like Adobe, Sitecore, Episerver or Joomla are best suited to monolithic architecture. There are a number of young, API-only (headless) CMS systems gaining traction like Built.io, Contentful and Prismic.io. And finally, there are API-first options that are able to deliver on the promise of decoupled, like Hippo and Drupal. The challenge is determining the right option for not only your current use cases, but for your organization’s strategy roadmap.

The decision is driven by the changing digital landscape. Chatbots are poised to become very popular. The Alexa App was among the top 5 apps downloaded on both the Apple App Store and the Google Play store on December 25, 2017, according to Social Media Week. The question is, how this will impact the way your organization addresses your participation in the growing digital landscape. Yes, you have companies developing applications with specific mobile teams that are separate from the web team. Traditionally, organizations viewed native app and website development as different: different developer skill sets, different infrastructure requirements, different use cases, different audiences, different expectations. Decoupled enables you to build differently, so that the silos of web and mobile are broken down and you can support a unified digital apps team. Once this is accomplished, the team can organize around content structure versus presentation, instead of web versus mobile app. This can quickly result in freeing front-end developers from back-end obstacles.

With a new organization in place and bottlenecks addressed, teams can now employ pipeline development methodology. Pipeline development automates build and integration, test and deployment stages of application development. This allows development teams to deliver new applications and features to users faster and more efficiently. In a decoupled website or application development, this approach gives both front-end and back-end teams the independence to develop structured content models and modern presentations that best meet the objectives of the project.

In order to reap the advantages of the architecture, you need a platform that is designed to support these use cases. With pipeline development, teams benefit from a common development tool set to aid in communication. Support for application hosting for front-end and back-end on a single platform will aid you through the consolidation of infrastructure suppliers and a single support structure and SLA.

In order to deliver a complete omnichannel digital experience, your platform must either easily integrate with, or include:

- Front-end framework (e.g. Node.js)
- CMS(e.g. Drupal)
- Personalized content delivery
- Customer journey orchestration
- System integration (eCommerce, marketing automation)
- A robust API-first approach

- Support for application and infrastructure from a world-class team

The benefits of doing decoupled with JavaScript on a single platform are common development tools and a common UI for your teams to use. If you're using some sort of pipeline automation, they can both be using the same tools. Teams are able to work more efficiently by sharing unified management consoles and developer tools.

Decoupled Flexibility: Progressive vs. Fully Decoupled

For many projects, you may have a full-stack, or monolithic, Drupal website that is performing for your company, and your development team is familiar with its core strengths. However, there is interest in more modern technologies for your site, or the additional digital channels available. A progressive decoupling solution uses APIs to display interactive elements within Drupal, but maintains this content and additional workflow within the Drupal stack.

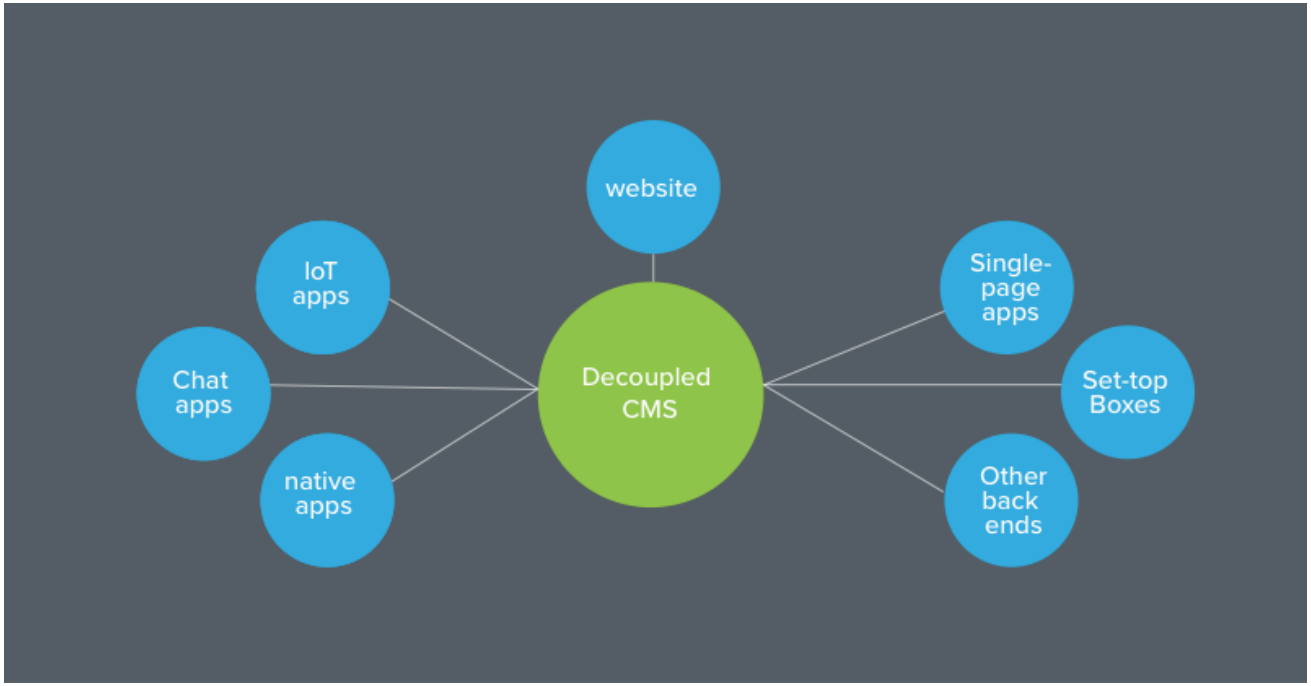
Progressively decoupled applications mitigate many of limitations of fully decoupled architecture by leveraging Drupal's presentation layer, but inserting decoupled elements. This enables the teams in the scenario above to mitigate the concerns of maintaining additional workflows.

Progressively Decoupled

Progressive decoupling combines the benefits of traditional Drupal applications with the benefits of front-end frameworks using JavaScript by enabling teams to:

- Rely on Drupal to render and deliver the Skeleton boilerplate and dynamic sections of the page
- Decouple the elements of the page which need to be dynamic or uncached
- Leverage the BigPipe Drupal module to speed up page loads while decoupled and dynamic elements populate asynchronously
- Offload some front end rendering to lower server resource usage to allow pages to load faster

- Optimize decoupled requests to arrive at smaller, lightweight requests



Practical Decoupled Use Cases

Before selecting a decoupled architecture, it is important to understand what you are asking of your application. This architecture provides benefits in front-end compatibility with new channels, real-time data updates and time-to-market. However, you are trading off some of the key benefits of a robust CMS like end-to-end preview, design templates, and the reduction of complexity to manage multiple technologies. Even with the tradeoffs, there are some key use cases where decoupled is growing in popularity because the advantages outweigh the disadvantages.

Decoupled Commerce

Let's look at commerce as an everyday example of a use case for decoupled CMS. To have a functioning commerce practice, you need more than just an online store. Commerce today means IoT apps, native apps, chat applications for customer support, and integrations with Alexa, Siri and Google Home. There needs to be a connected bridge between your commerce application, and your content -- product info, testimonials / reviews, and so on.

For example: A customer - we'll call her "Olive" - goes to Amazon.com on her desktop and fills a shopping cart but then needs to catch a train so she packs up and heads out. On the train she logs into her phone and sees his items, still in cart. It's a seamless experience. The next action about be for Olive to confirm his order when she gets home by asking Alexa for the estimated ship date.

Digital Signage

Another great use case for decoupled CMS is digital signage. Digital signage is becoming more prevalent, from the menu boards in your local coffee shop to the train information at your subway station. In many cases, this signage requires real-time, or near real-time updates of information. It could be for train schedules, or campus alerts, or the latest specials in a restaurant.

The goal for any organization connecting the digital dots is to deliver a seamless integrated experience. Remember the film Minority Report, when Tom Cruise is walking through the mall and through facial recognition, the billboards were personalized to him? That's the kind of experience we're on the cusp of and decoupled makes it more readily available.

Why Javascript Matters:

Decoupled brings together the best of the old web (pre-smartphone) and the best of what the current web has to offer now and in the future. But all these shiny new things need to be developed in a way that lets them connect. Looking at how to build decoupled applications one thing has become clear: you need JavaScript. JavaScript, and its frameworks like Angular, React, and Vue have been ranked by StackOverflow as the most popular scripting language since 2013 in a survey of over 50,000 developers..

Choosing the Right JavaScript Framework

There are many JavaScript frameworks to choose from. There's React - it's one of the most popular frameworks that's being added to Drupal core. There's Express.js. There's Vue. There is Ember. There's Angular. The challenge with JavaScript from a developer's standpoint is it's kind of a wild, wild west. The choice of framework rests on use case and developer preference. For example, Angular.js has a strong CLI that allows for integration of module loaders and advanced build systems for complex web applications.

For customers that have already made the choice to go with the decoupled application where they're running their front-end on a completely different stack, in this case a JavaScript stack which includes Node.js as the runtime, those people have to set up their stack that supports the CMS which is essentially the LAMP stack like what Acquia does today, then they go somewhere else like a Heroku or Amazon Web Services and they set up this front-end runtime to run over there.

When Acquia launched our Node.js support back in September of 2017, we took a lot of the complexity out for our customers because now they've got one UI to run both the front and back-end stack. We designed our support for Node.js around delivering the optimal stack configuration for developing decoupled applications. Rather than building a complete Node.js stack and a LAMP stack, decoupled applications on Acquia Cloud are supported by a full LAMP stack and the Node.js runtime - all on a single platform.

Best Practices with the decoupled API: Acquia Engage

In October 2017, just as Acquia was releasing support for Node.js on Acquia Cloud, we acquired a new customer: ourselves.

Acquia was facing the same situation that many of their customers face on a regular basis: a requirement for a digital experience tied to an event. In Acquia's case, it was building a decoupled application for our annual conference, Acquia Engage.

The first step in a successful decoupled project is aligning on requirements. In the case of Acquia Engage, they were as follows:

- Provide real-time updates with presentations and speaker information
- Showcase information about Award Finalists by category
- Showcase other types of content relating to the conference
- Integrate content from standard APIs such as JSON into the application
- Work responsively on multiple devices and screen sizes

Once all the requirements were set, the next step was to understand the decoupled Drupal workflow.
To do this effectively, Acquia:

- Used development to help contrast assumptions around level of effort based on tasks
- Standardize workflow along with deployment requirements when building two applications that work in parity.
- Streamline communications around potential issues, understand how to troubleshoot issues with build based on technical level of the audience.

The Acquia Engage application was designed and built in JavaScript to execute on a Node.js runtime environment in Acquia Cloud. t was important to test any limitations or technical opinions of Acquia Cloud's Node.js hosting before getting

started. This included documenting implications of how a Node platform built for decoupled Drupal is different than a stand-alone Node hosting offering. Acquia also had to adjust the application based on requirements of automating a deployment cycle based on Acquia pipelines.

Once limitations and technical options were established, a content workflow was enabled for non-technical content authors, who would be working with and updating the content within the Acquia Engage app. To maximize efficiency, the marketing team needed the ability to manage the content with a standard CMS-managed workflow and provide real-time updates without the need for code deployment. In addition, the marketing team needed authorization to upload media assets based on type of content. To meet these requirements, Acquia needed to allow for customizations integrated with Drupal that carried over to the JavaScript application.

Results

- Successfully supported ~250 users per day during conference
- Cut development time by 30%
- Node.js processes allowed for real-time data to be presented to Acquia Engage attendees through the Engage App
- Editor workflow increased by 35% during conference updates

Conclusion

Decoupled architectures are foundational to the direction digital teams are taking applications into the 21st century. In fact, Forrester stated “the microservices approach is the future of digital experience architectures”[1]. Despite that, moving to this pattern requires thoughtful decision.

Understand your use cases or digital applications and make sure you are aligning to the best architecture to meet your needs. If your priorities include a heavy reliance on delivery of content over API, real-time data, or omnichannel support,

[1] “The Rise Of The Headless Content Management System”, Forrester Research, March 17, 2016

decoupled Drupal should be seriously considered. If your objective is a blazing fast experience for the web or other digital application, leveraging Drupal as a service to a Node.js runtime makes it easy to deliver an amazing frontend experience.

Despite the seemingly endless benefits, ensure you weigh the tradeoffs of managing a decoupled stack. Ensure your partners and/or internal teams can deliver the support needed to keep your applications running optimally. Make sure your team is ready to build applications differently. Decoupled can introduce new development tools, languages and approaches. Make sure your teams are prepared for the coming changes. Lastly, make sure you evaluate your platform approach based on the support you need for both your front-end and back-end teams.