Content Management Systems of the Future: Headless, JAMstack, ADN and Functions at the Edge

Abhishek Tiwari

Citation: *A. Tiwari*, "Content Management Systems of the Future: Headless, JAMstack, ADN and Functions at the Edge", Abhishek Tiwari, 2018. doi:10.59350/n3ps2-yts66

Published on: November 03, 2018

Recently I was asked about content management systems (CMS) of the future - more specifically how they are evolving in the era of microservices, APIs, and serverless computing. For enterprise customers either undergoing or planning the digital transformation, this is an important question to ask. Enterprise customers spend a large chunk of their digital and marketing budget on CMS and associated modules such as digital asset management (DAM). Most of the CMS vendors dodge questions of evolution by talking about incremental innovation primarily focused on customer experience (CX) such as analytics and personalisation. Unfortunately, due to the lack of a good developer experience (DX) total cost of ownership of a traditional CMS and implementation failure rate remains all-time high. Strictly speaking, it cost companies more to implement and maintain a CMS than just CMS license fee. To sum it up, if anything CMS landscape now requires a big shift. Well, you don't have to wait, CMS of the future is already here - it is headless, static, distributed, cost-effective with equal focus on CX and DX.

Traditional CMS is a dead end

In the last 5 years, traditional CMS landscape is not evolved much - both CMS technologies as well market maturity remains unchanged. There is hardly any innovation from traditional CMS vendors. Although some vendors have added support for APIs and cloud services most have not even bothered to adapt with changing technology landscape. Case-in-point, most enterprise CMS vendors lack robust full-site content delivery network (CDN) integration. In fact, CDN support is often limited to static assets not realising that with the arrival of HTTP2 domain sharding is a thing of past.

When it comes to innovation, most of CMS solutions are constrained by their legacy architecture (read strong coupling between content management and content presentation) which makes it difficult to serve content to new types of emerging channels such as apps and devices. API support is critical for innovation but it needs to be a first-class citizen and not afterthought. In addition, traditional CMS solutions lack integration with modern software stack, cloud services, and software delivery pipelines. Possibility to apply continuous integration (CI) and continuous delivery (CD) concepts with a traditional CMS is mostly unheard-of. At the core, a traditional CMS is a monolith. Any organisation pursuing microservices strategy will find hard to fit a traditional CMS in their ecosystem.

Lastly, owning an enterprise CMS is a costly affair. Pure CMS licensing cost can be anywhere between 50,000 - 500,000 USD a year. You should expect one-time implementation cost (depending CMS and business requirements it can cost 200,000 USD to 3M USD) and yearly hosting infrastructure cost (proportional to load and traffic but typically 30,000 USD - 300,000 USD per year). Overall, the total cost of ownership for a traditional enterprise CMS solution is on the higher end. Cost is one of the key reasons why most government organisations, mid to large sized business, and publisher prefer open source CMS options such as WordPress and Drupal. Unfortunately, other than cost advantage Word-

press and Drupal have similar deficiencies i.e. legacy monolith architecture. In addition, open source CMS solutions also struggle with blotted plugin ecosystem.

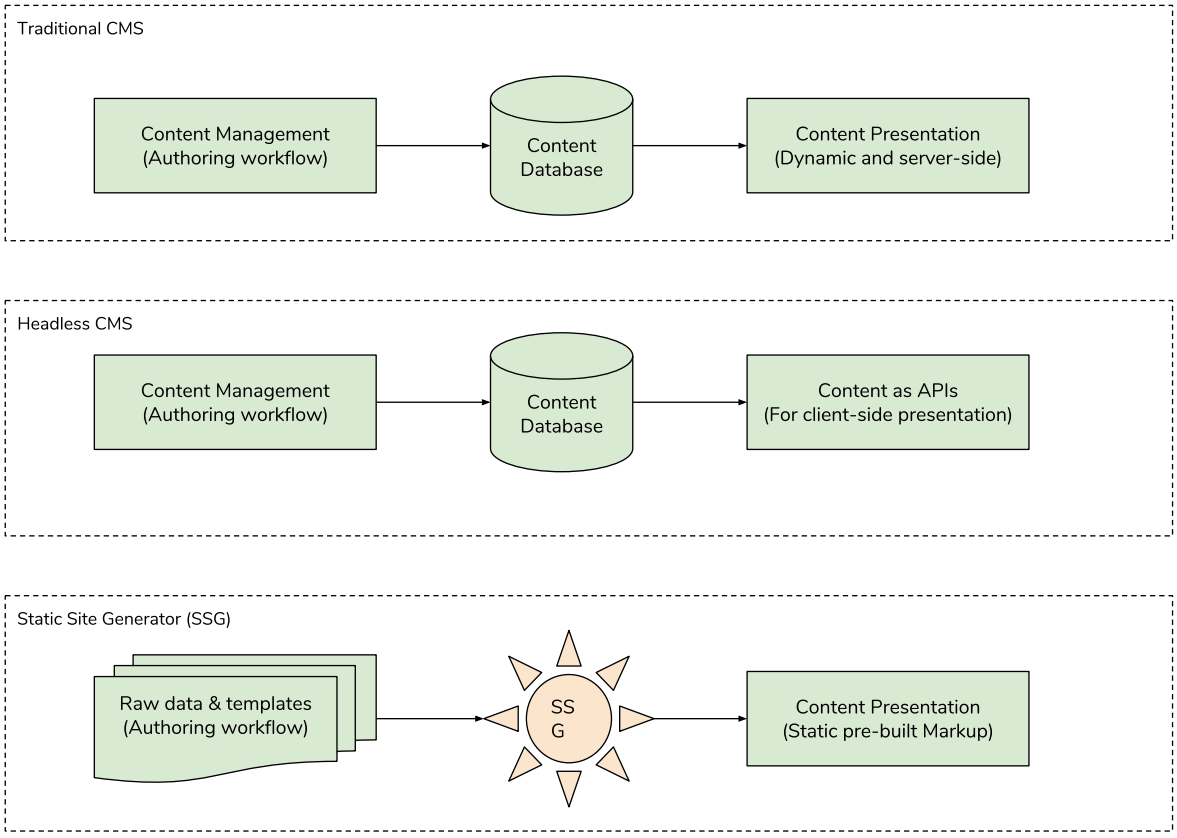

Figure 1: A ten thousand fit view of a traditional CMS, a headless CMS, and a static site generator.

A few months back, I was pulled into a scenario where a business has been working with a leading CMS vendor to roll-out a network of multi-regional websites. Ironically, the vendor was very hesitant to adopt or integrate their CMS solution with a CDN. Delivering the whole website via a CDN brings positive impact on website page response time (5-10 time times improvement in Time to first byte - TTFB) and overall user experience (2-4 times improvement in the first meaningful paint assuming HTTP2 is enabled). If you put your whole website on CDN, technically you don't need a large number of server infrastructure and CMS licenses. Using CDN for the whole website, you can offload most of the website traffic to your CDN which will handle not only large traffic spikes but also reduce the latency of content delivery.

Now all CMS vendors have their own reasons and financial incentive to not offer you the best solution. Unfortunately, to make it worse most of the CMS deals are managed by marketing and digital functions of the organisation, and in my own experience, these functions are often not equipped to

ask hard and right questions during the CMS selection process. They often get blindsided by vendor's pitch and end-up making decision based on some fancy demos (see my post from 2014 on Adobe AEM). In the CMS selection process, developer experience is not a factor, although successful implementation and ongoing maintenance require developer friendly tooling and support for modern software engineering practices. Not to mention, traditional CMS implementation cycle is generally waterfall or water-Scrum-fall at best.

Simply Static

A static site generator (SSG) is nothing but a digital printing press. Using raw content data (such as Markdown, YAML, JSON files) and templates an SSG engine (such as Jekyll, Hugo, Gatsby, etc.) can generate an HTML-only website without involving a CMS. Raw content data along with templates are version controlled using Git or similar versioning systems. An SSG offers a middle ground between a complex yet modular CMS solution and a simple yet involved hand-coded HTML site. Due to strong templating support, a website managed by SSG can be truly modular. For SSG, content is often written using the Markdown and configured using YAML/JSON data structures and files.

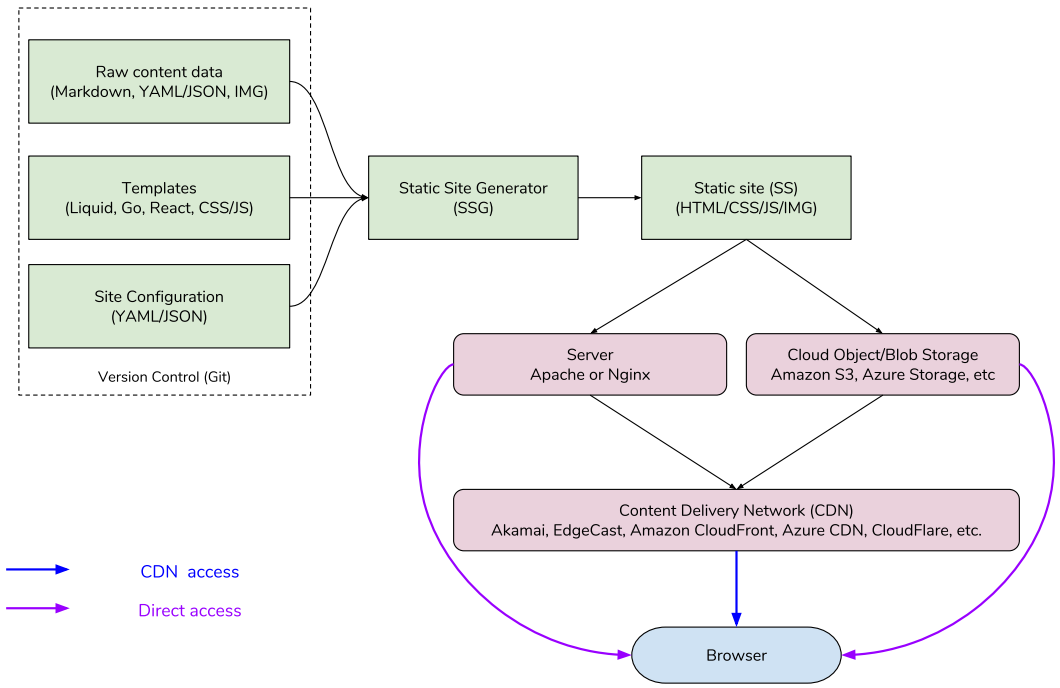

Figure 2: A static site generator (SSG) generates the static HTML site using raw content data, templates and site configuration.

The build output of an SSG is generally a directory containing the HTML pages, CSS, Images, JavaScript, etc. i.e. everything you need to render your site. This directory can be uploaded to a server and served using a web server such as Apache or Nginx. Alternatively, you can upload output directory to cloud object/blob storage such as Amazon S3 or Azure Blob Storage and serve your site from there. Most of cloud object/blob storage services have native support for static site hosting. When using either of these options, you can deploy a CDN such as Akamai, Amazon CloudFront, Cloudflare to accelerate the delivery of your static site.

SSGs existed for more than 10 years but initial use cases were limited to personal websites and as a tool for hackers and hobbyists. SSG became mainstream and got some real traction during the second Obama campaign. In 2011, the Obama campaign started using Jekyll extensively to power their fundraising platform. They used static websites and campaign micro-sites generated by Jekyll in conjunction with a CDN (Akamai) and consistently observed HTML transfer times around 20 milliseconds. I am yet to find a traditional CMS vendor who can deliver this level of performance on the scale (millions of page views per day).

JAMstack

JAMstack is a new way to build content-heavy websites and web apps. Using JAMstack delivers better performance, higher scalability with less cost, and overall a better developer experience as well as user experience. Your website architecture is JAMstack if it meets following three criteria: client-side JavaScript, reusable APIs powered by microservices, and prebuilt Markup. In many ways, JAMstack is obvious next evolution of SSG as it requires templated markup to be pre-built at deploy time usually using a site generator. However, it is important to note that JAMstack can serve both static as well as dynamic content. Using JAMstack you can add dynamic functionality such as user identity, HTML forms, and personalisation.

Circa 2014, I was working with a big Japanese automotive brand in Australia. They were using a very legacy CMS or more precisely a document management system which was painful to work with. Due to the high compliance requirements, their whole website was treated as a digital press. Just to give you a bit context, they have to ensure prices and model specifications displayed on their website are aligned with showrooms. Any pricing error can cost them hundreds of thousands of dollars if not millions. Not to mention, ACCC - consumer watchdog in Australia is quite vigilant about false advertising claims when it comes vehicle specifications, imagery, and disclaimers. To avoid this, every content change was editorially controlled, audited by the legal teams and then only a new version of the whole website was published. This made whole publishing process really slow and painful and CMS was part of growing pain. Lastly, the whole website was very slow to load - CDN caching was not effective as a large number of pages were personalised for pricing and availability depending on suburb and postcode.

Eventually, we decided to move them to Jekyll. It took about 6 months but the overall result was great. They had a new fully responsive yet static website. Page load improved 2X by moving the entire project on CDN which was previously ineffective. Raw data for each version of the website was version controlled using Git and Git Large File Storage (LFS). We used automated continuous integration(CI) jobs to create pre-built Markup using Jekyll. To avoid inconsistent state, rollback and deployments were atomic i.e. no changes can go live until all changed files have been staged. Once changed version was staged in a new directory, making things live was as simple as changing directory symlink to point to latest version. All personalisation including pricing, availability, and disclaimers were moved in APIs and assembled on top of static HTML using the client-side JavaScript. Without knowing we created a JAMstack for our client.

Headless CMS

A headless CMS such as Contentful, Zesty has no presentation layer - a key distinction between headless CMS and decoupled CMS given they both provide content as APIs. With a headless CMS, the task of the content presentation is performed by an external client consuming APIs exposed by headless CMS. Here are few examples of an external client utilising the APIs exposed by a headless CMS: static site generator (SSG), single page application (SPA) (client-side as well as server-side rendering), a mobile app, a WordPress site, or an IoT device.

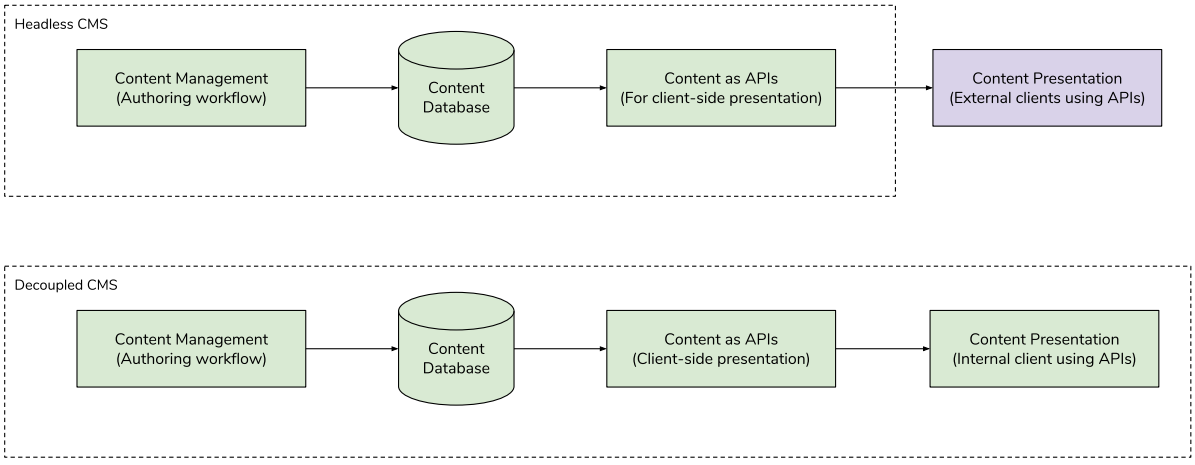

Figure 3: Decoupled CMS vs. headless CMS. Headless CMSs are a subset of decoupled CMSs. Both offer content APIs but headless lack internal content presentation layer.

Headless CMS or API-first pattern has been already adopted by many big publishers. As illustrated below content APIs exposed by headless CMS can be integrated with a variety of external clients for

content presentation including WordPress, SSG such as Jekyll, Hugo, Gatsby, or client-side UI frameworks such as React, or VueJS. WordPress is an interesting choice for publishers looking to syndicate their content across a network of websites. Using headless CMS content can be managed centrally which can be then syndicated via content APIs to a network of WordPress sites using the power of WordPress plugins. Using WordPress, publishers can still tweak the content presentation on a per-site basis like the selection of featured or most discussed content. Due to support for client-side as well server-side rendering React is another popular choice to use with headless CMS.

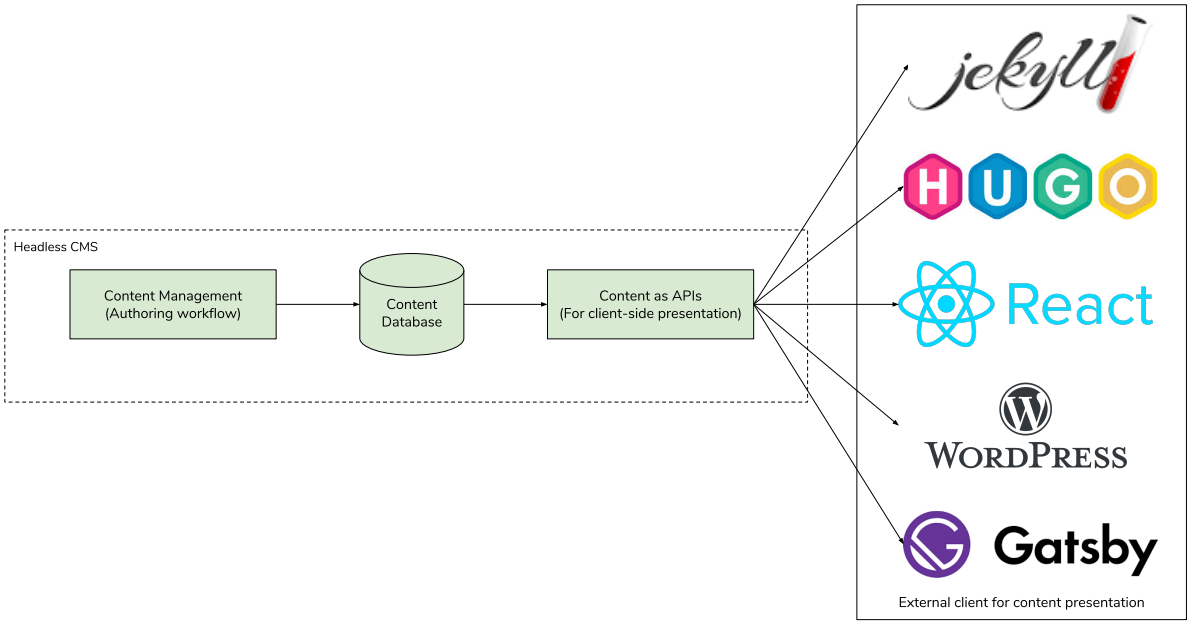

Figure 4: With a headless CMS you can use a variety of external clients for content presentation including Wordpress, SSG such as Jekyll, Hugo, Gatsby, or a client-side UI frameworks such as React, or Vue.

GraphQL

Some headless CMS solutions such as GraphCMS and Mozaik are taking content infrastructure to the next-level by offering GraphQL based content APIs. GraphQL - created and open sourced by Facebook - is a powerful query language for APIs and a more efficient alternative to REST. In a nutshell, GraphQL enables a client to specify exactly what data it needs from an API - nothing more and nothing less.

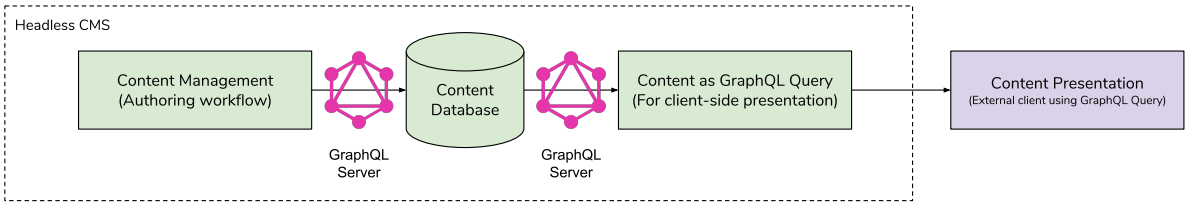

Figure 5: A headless CMS using GraphQL. GraphQL server provides a single endpoint which external clients can use to query the data and authoring workflow can use to write or update content data (mutations).

Application Delivery Network

A common theme across static sites, JAMstack, and Headless CMS is the extensive use of the content delivery network (CDN) to unlock the speed and performance. Caching API response of a headless CMS seems desirable but not essential - in the majority of cases, you are better served by caching the prebuilt Markup consuming the API response. Nonetheless, for static sites and JAMstack, CDN is essential.

Firstly, using a CDN has become a lot easier and cheaper. Now a day there are so many options - Amazon CloudFront, Cloudflare, Google CDN, Azure CDN, Edgecast, Fastly, and the list goes on. Gone the days when you required to have big fat-contract with Akamai. With most of CDN services, you can start small and pay as you go.

Secondly, having a CDN in front of origin (static site or APIs) reduces the global and regional latency. This is achieved by caching content (static HTML page, assets, APIs) at a large number of geographically distributed edge locations. In addition, CDNs scale really well - we are talking about 5-10k concurrent requests without any issues. To protect origin server from request overload some CDNs also support origin shield - an additional caching layer between edge and origin servers. Typically, this mid-tier caching is a designate edge or pop location closer to your origin server and other edge servers query origin shield rather than origin directly.

Lastly, several static sites hosting platforms such as Netlify have rolled out their own CDN. Netlify refers their CDN infrastructure as Application Delivery Network (ADN) which has no distinction between edge and origin servers. This is primarily to support the atomic deploy model and instant cache invalidation so that there is no risk of stale content or inconsistent state. With ADN, switching between multiple version of a static site is as easy as changing symlink to a directory. Atomic deploy model can be extended to advanced functionalities such as staging, instant rollbacks, phased rollouts, and A/B testing. Many conventional CDNs are unable to support some of these features due to restrictions and

limitations around the instant cache purge. For example, some CDNs rate limit the cache invalidation, while others charge for each cache invalidation request.

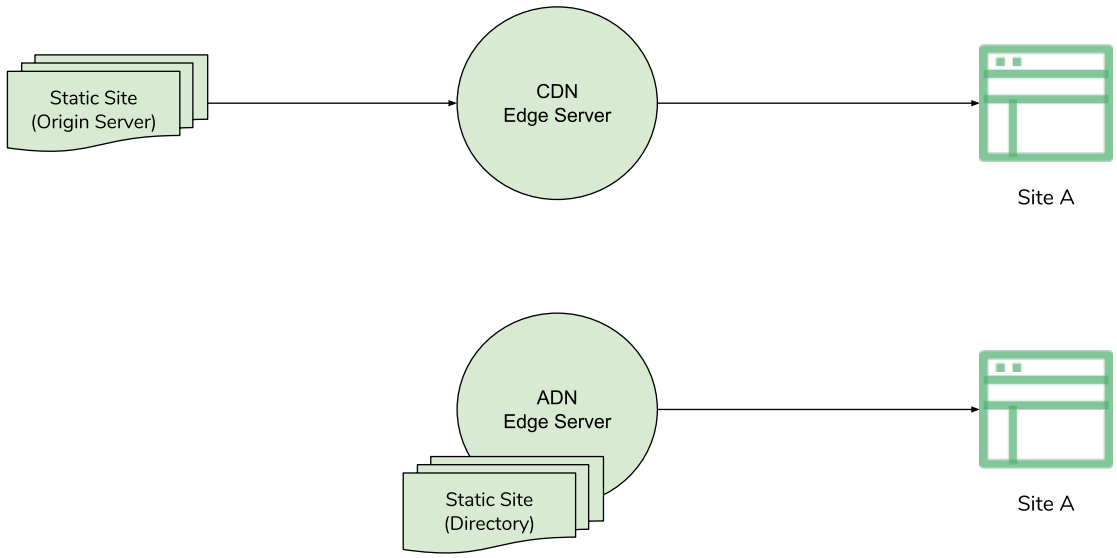

Figure 6: Content Delivery Network (CDN) vs. Application Delivery Network (ADN). With ADN your static site is replicated to all edge servers i.e. zero origin infrastructure.

Functions at the edge

Functions as a service (FaaS) is an emerging pattern to build APIs and microservices at scale. Most public cloud providers already offer runtime for the functions in respective hosting regions. Now it possible to execute functions at the edge - for instance Cloudflare Workers and Amazon Lambda@Edge offer ability to execute functions more than 100 edge locations globally. Functions at the edge can be used to not only to personalise the content of a static site but the also to handle dynamic functionality of the JAMstack. Using Lambda@Edge and Cloudflare Workers you can already perform A/B testing, authentication, and authorisation, intelligent routing, etc. For authentication and authorisation one can use a third-party identity service like Auth0 or Firebase. By all means, functions at the edge are immensely powerful compared to conventional Varnish Configuration Language (VCL) used by many of CDN providers. With functions at the edge, it is possible to deliver content with zero origin infrastructure.

Functions at the edge are also used for forms and submissions without having any server-side backend. In addition, they are also utilised for on-the-fly media optimisation - thumbnail generation, responsive images, etc - some common functionalities provided by a digital asset management (DAM) module.

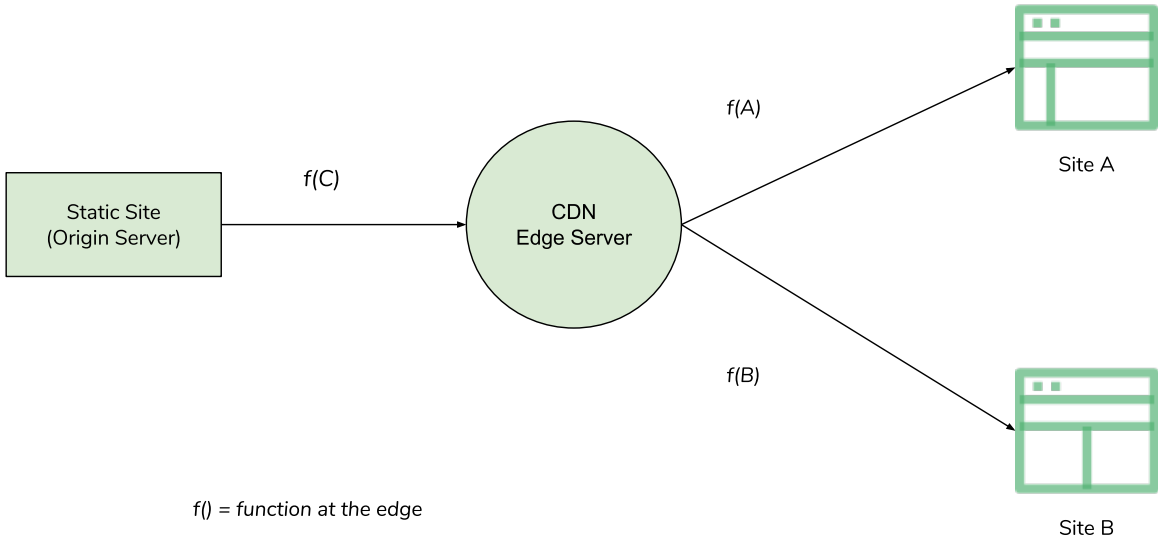

Figure 7: Functions at the edge is yet another way to make a static site more dynamic. Depending on requirements, functions can be executed on request/response of client or request/response of origin.

Developer Experience

With traditional CMS, developer experience (DX) has been a key pain point which boils down to lack of support for modern frameworks, microservices, versioning, CI/CD, and DevOps. In order to create more engaging user experiences, developers need more power and greater flexibility. This includes the ability to select content presentation frameworks suited to deliver a great user experience in a native way. As you decouple presentation layer from content management backend, you open the doors to create more content rich yet data heavy user interfaces.

Strictly speaking, if you are not a publisher like News Corp or Guardian then you don't have a pure CMS problem but what you have is a blended content and data problem - a sweet spot for headless CMS and microservices powered JAMstack. For instance, an enterprise bank will be able to produce a mobile app with content focused on home loans and supplement that content with the various type of tools such as home loan calculators, house price trends based on suburbs, etc. by utilising data-driven microservices. Unlike traditional CMS model where you generally require two different set of skills to support your blended requirements (CMS developers and application developers), by using a headless CMS and JAMstack model you only need full-stack Javascript developers.

Similarly, in this new ecosystem versioning, CI/CD, and DevOps are first class citizens. SSG, as well as JAMstack, provide robust versioning using modern version control systems like Git including raw

content data, templates, and configuration. They can also support branch based environments (such as Dev, UAT, Staging, and Live) and deployments. A static HTML site is generated from the version-controlled raw site using a CI/CD system such as Jenkins or CircleCI and deployed to your web servers automatically or manually. Actual CI build process can be triggered by a version control commit or headless CMS hook post content change. More importantly, platforms like Netlify and Aerobatic takes away the pain of building your own CI/CD process and web servers so you can stay focused on developing compelling user experiences.

Closing Thoughts

With the timely arrival of headless CMS, JAMstack, ADN and functions at the edge and their ability to integrate seamlessly with existing static site generators have created a perfect recipe for a much-needed disruption in CMS landscape. This new wave of CMS solutions offer enterprise-ready content infrastructure which is developer friendly, support business agility, and enables rapid build of content-centric apps.