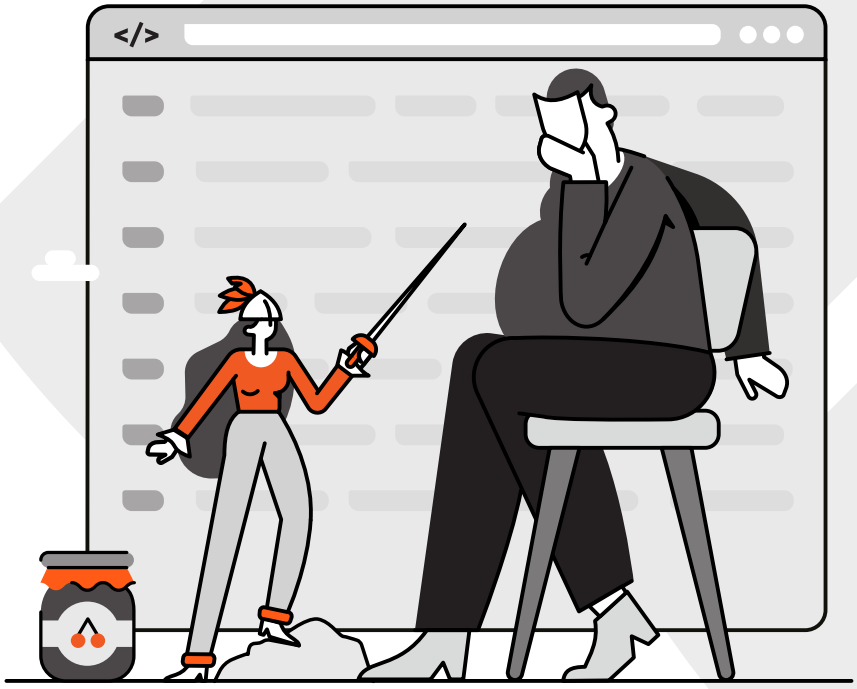
Kontent.
by Kentico



Serious Jamstack.

Headless CMS and Jamstack for enterprise projects

By Ondrej Polesny

Introduction.

Believe it or not, the term Jamstack has been with us for over six years now. But it's only the last year or two when developers really started to take Jamstack as their serious choice when building websites.

In this book, I don't want to explain the Jamstack basics and why we, developers, like to build pre-generated websites. If you're reading these lines, you probably already know the Jamstack benefits, and you're considering it for your next project. Here, I want to focus on the obstacles we always need to work around when building anything beyond a simple presentation website. I aim to show you that Jamstack is ready for enterprise projects.

Ondrej Polesny.
Developer Evangelist, Kentico Kontent

Let's start by outlining the advanced scenarios and their potential problems. The chapters in this book correspond to the items in this list:

1 Performance.

One of the key attributes why you choose Jamstack is the performance. But as time goes by and your project grows, you'll encounter two aspects of Jamstack sites that require your attention:

- Build and deploy time
- Hosting

It's not a question of "if," but "when" both of these become a bottleneck.

Build and deploy time.

Many developers don't see build time as an important metric. When a change is published, it doesn't matter if it takes a minute or two before it's visible on the front end. While that is true in most cases, the problem increases as the content grows.

According to a benchmark published on CSS Tricks that only worked with markdown files, it takes Gatsby over 70s to build a site with 8k pages. That includes only plain text with no images and does not include the time required to fetch data from a data source and convert them to GraphQL nodes.

Internally, we have experienced builds taking way over 15 minutes for about 3k content items. It highly depends on the site's implementation, data complexity, images processing, etc.

The same behavior occurs for other frameworks like Next.js, Nuxt, Jekyll, and so on.

So why is it a problem?

Content previews

If your site is truly static and does not allow editors to see server-rendered previews, you need to build it every time an editor wants to see their content in the website frame. Something that traditional websites let them see instantly. Obviously, they hate to wait a few minutes for it.

Scheduled publishing

Once editors are happy with their changes, they don't just hit "Publish" and walk away. They always expect to know if and when it gets published. They check the page again to see if they haven't made some horrible mistake like forgetting to publish some related content. It becomes very frustrating for them to wait a long (or worse, unpredictable) length of time to see it published.

Long and stacking builds

The previews are sometimes solved by hooking a headless CMS webhook into a build platform that rebuilds the site or its part on every content change. The problem is, there can be a lot of changes. Even if you rebuild only after a content item is moved to a special workflow step or is published, when there are multiple editors working on a single project, the build server will be busy building the site 24/7. That's not something developers, editors, or the project owner's wallet like to see.

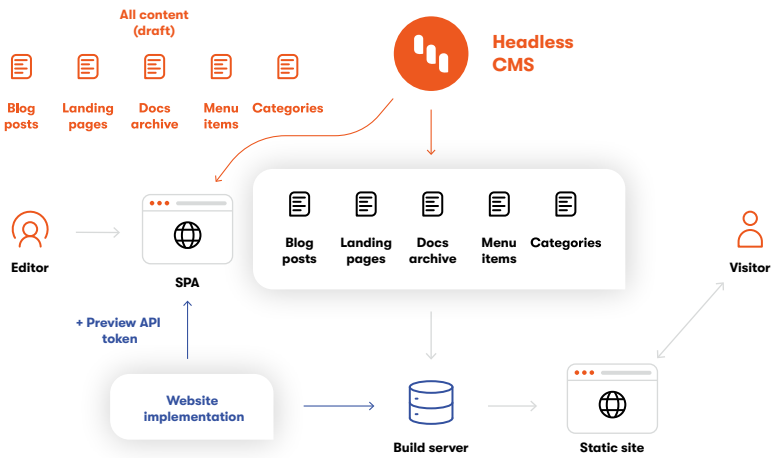
Solutions

Depending on your chosen framework, there are ways to make builds behave even for projects with large amounts of content.

Server-rendered content previews

Advanced static site generators like Next.js and Nuxt allow you to render a specific page on demand.

Next.js achieves this by using serverless functions and the logic is integrated into the platform. If you choose a web host that fully supports Next.js, there are no extra steps for you, as the functions are extracted and deployed automatically during `next build`.



NuxtJS lets you switch the server-side pre-rendering to classic SPA mode and build the preview site on the client. You'll probably end up deploying two sites—production and preview—to protect your preview API keys, but the rest is automatic.

Use native platform

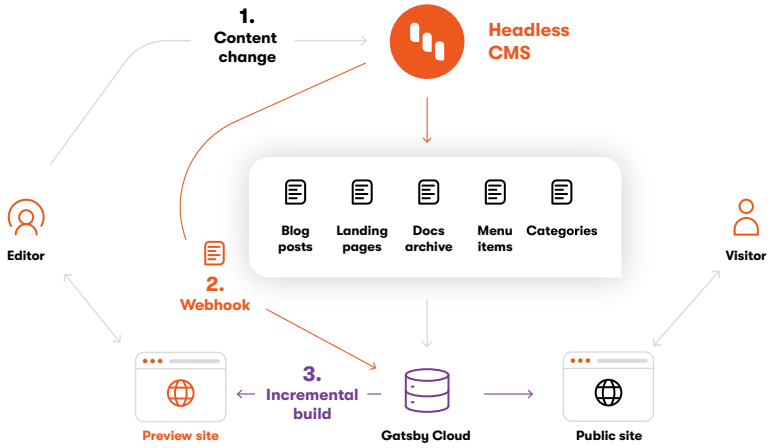
If your project does not need advanced features available in JS-based frameworks, like front-end bundles with client-side functionalities, you can leverage the benefits of compiled programming languages with static site generators like Hugo (Go) or Jekyll (Ruby). According to the same benchmark mentioned earlier, Hugo is about 15-30x faster than JS-based frameworks.

Don't build everything

But the best solution is to simply not build everything every time. Here the solution is highly dependent on the used framework.

Gatsby

If you're using Gatsby and have it hosted on Gatsby Cloud, you can use incremental builds. Provided your headless CMS features a first-class integration with Gatsby, you can hook the content editing notifications to Gatsby Cloud.



Whenever a content editor changes a piece of content, Gatsby Cloud receives the information and rebuilds only affected pages. The incremental build takes only a few seconds and works for both preview and production builds.

NuxtJS

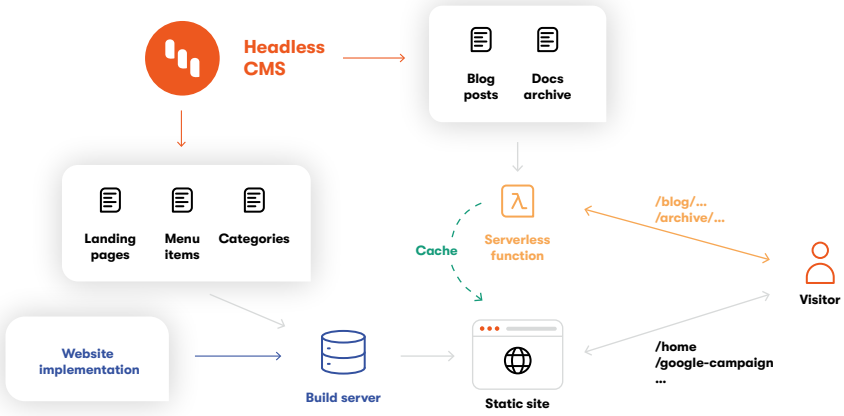
NuxtJS allows you to separate code and content builds. First, it builds the site and then crawls it starting from the homepage to find all internal links. This way, it incrementally pre-builds all pages during a single deployment. If you only change content, you don't need to touch the built website (i.e., the bundle), only recrawl it and regenerate the static pages. That's what NuxtJS does for you automatically.

Next.js

With Next.js, you don't have to build all the pages during the initial build. For each page, you can choose how it should be handled:

- **Static generation (SSG)**
  Page will be generated at build time.
- **Client-side rendering (CSR)**
  Page will be rendered during the initial load on the client.
- **Server-side rendering (SSR)**
  Page will be handled by Node server or serverless function for each request.
- **Incremental static regeneration (ISR)**
  Request to a page in this mode will return an already generated and cached page. If it doesn't exist or has already become stale, Next.js will rebuild it and cache it.

So if your website has 20k pages, you can pre-build (SSG) the most visited 1k. Pages that show live data can be set to server-side rendering (SSR), and the rest will be prepared and cached on demand (ISR). The concept is very similar to lazy loaded images on long pages where your browser downloads only the images in your viewport and keeps lazy loading the rest as you scroll.



Other platforms and Netlify's DPR

If you're using other platforms, you can still use on-demand-built pages with Netlify's distributed persistent rendering. The concept is pretty much like simplified incremental static regeneration of Next.js. You pre-build only the critical part of your website and leave the majority of pages to be rendered by on-demand builders. Those are essentially just serverless functions that do the same magic as your regular build, only on a page level and when needed.

This is still a new initiative, and as of August 2021, they support Next.js via the Essential Next.js plugin and 11ty, but the community will likely add support for more SSGs soon.

Watch for client-side JS bundle size

When using modern JS frameworks, your visitors download the first page and asynchronously fetch the client-side JS bundle that, once fetched, rehydrates the page. While that makes every subsequent page load incredibly fast and can even enable your site to work offline, it has the potential to hurt your Google Web Vitals results and thus SEO. Try to optimize the bundle size by including only necessary packages, keep the core framework packages updated, and follow its best practices.

Hosting.

People often have their preferred host that they use for all Jamstack sites they create, or they default to one based on the used platform. If your site uses Gatsby, you might host it on Gatsby Cloud, if it's a Next.js site, you might use Vercel, and so on. However, even with all of the above configured identically, different hosting providers may provide performance benefits over others.

- **Build time**
  Many hosts offer to handle the complete deployment process of your site including build. However, the build time on each host highly depends on the used architecture. When I tested a single Next.js site on multiple hosts, the build times were ranging from 40 seconds to 3.5 minutes. If your site is large and you need to decide which pages will be pre-built and which pages will be server-rendered, the build time becomes a very important metric.

- **CDN locations**
  Every Jamstack provider advertises that they provide a CDN. Jamstack sites are fast by default. Even hosting them on a standard server halfway around the world produces an acceptable visitor experience, but with larger sites, and especially e-commerce, the CDN size and performance start to matter.

  For example, Netlify states that their Edge network features 6 points of presence. They allow you to switch to a high-performance edge with 27 points for a custom price. Vercel claims to use 15 regions, Layer0 offers 31 edge locations and 85+ for custom plans, and Cloudflare states that they use 194 data centers across the globe.

  Also, pay attention to the resources that are not served from the deployed site. Depending on the site's implementation, it can be images, videos, external scripts, and so on.

- **Pricing**
  Every host offers a limited free tier that is always capable of hosting small to medium websites. For serious commercial sites that require guaranteed uptime and are maintained by teams rather than a single person, you will need to switch

to a paid plan. They start as low as $10/month and go up quickly. The typical differentiators are:

- Team members
- Bandwidth allowance
- Support
- SLA
- CDN network quality
- Build performance (prioritization)

Check out pricing matrixes of the most used providers:

- AWS
- Azure
- Gatsby Cloud
- Layer0
- Netlify
- Vercel

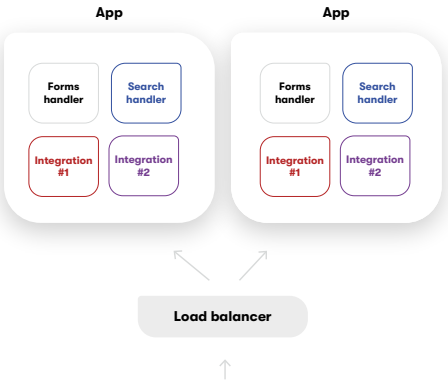
- **Support**
  Setting up Jamstack sites is usually very simple and you likely won't need to interact with support. However, with larger sites, you'll likely want an SLA and a guaranteed response time. This aspect is very easy to check—just ask a specific question, see what responses you get, and how quickly you get them.

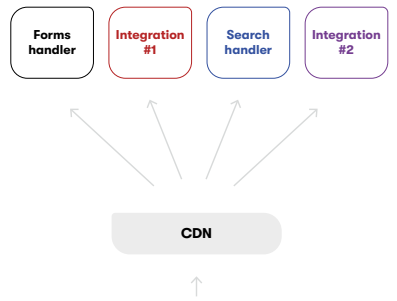
- **Reliability**
  All providers host the Jamstack sites on a CDN, so even if one node fails, the traffic gets rerouted to a different node, and the site remains operational. That alone greatly improves the reliability of site hosting. To verify each provider's reliability, you can check their status pages:

  - AWS
  - Azure
  - Cloudflare
  - Gatsby Cloud
  - Layer0
  - Netlify
  - Vercel

2 Server, serverless, security.

Any large Jamstack site needs a server. There's always something that needs dynamic server-side processing, like form submissions, personalization, search, and so on. The big difference is that, in the past, we hosted the website as a bundle on two or more servers behind a load balancer. With Jamstack, every piece of dynamic functionality is single-purpose, self-sustainable, and therefore can be hosted and scaled separately.





Server	Serverless
+ Easy logging and debugging	+ Has very limited access to only a single resource
– Has access to all valuable resources like databases, CRMs, payment gateways, and other systems that are necessary for the company processes	+ Access keys stored outside
– Access keys stored within the server	+ Scalable separately
– Scalable as a whole (scale-up, scale-out)	+ Deployed separately
– Deployed as a whole	– More complicated logging across multiple functions
– More attractive to hackers as it covers a larger surface area and represents a single entry point into company data	– Harder debugging

With servers, we were used to looking at resources specifications and possible scaling options. With serverless, the scaling is handled automatically, but there are other aspects we need to keep an eye on:

- **Cold starts**
  When a serverless function doesn't process any request for some time, it gets suspended. It's a way to save resources. Depending on the provider, it takes about 5–30 minutes before a function "goes to sleep."

What's more important is how long it takes the function to wake up. According to the conducted tests in the linked article, it's <1s for AWS, 0.5-2s for GCP, and unimpressive 5s for Azure provided you're running the functions on Linux.

Note: Cloudflare claims their workers are always on and don't suffer from cold starts.
Note: Netlify and Vercel use AWS for serverless functions.

- **Maximum execution time**
  Because serverless functions are single-purpose blocks of code, you should not experience problems with overtime. But if you're using them to solve other tasks like data transformations or integrations with other systems, their typical limit of 10-20s per run can become a bottleneck.

- **Memory limit**
  Similar to the previous point, serverless functions are limited in their usage of server memory. Typically it's around 1GB of memory and it should be sufficient for the vast majority of use cases.

- **Pricing**
  You typically pay for the number of executions, but higher tiers may give you a more generous execution time or memory limit.

- **Used platform**
  Most often, functions are deployed separately. On Azure, you need to deploy them in bulk as a web application, which means you're also scaling them that way.

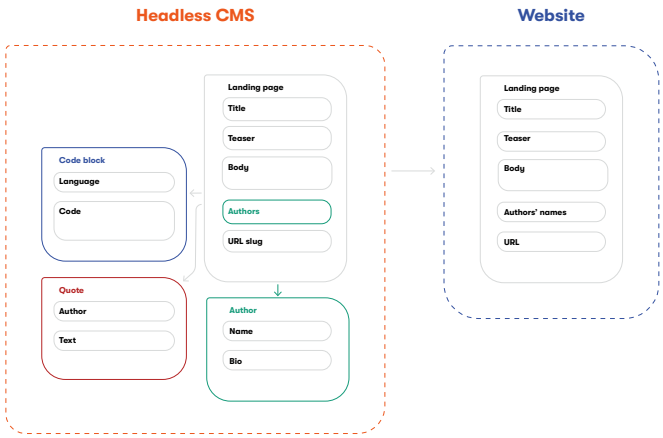
- **Caching**
  Most providers offer some form of caching the response of a function under specific circumstances—no auth, you provide the right caching headers, etc. This is useful for general data queries from external systems, like getting data of products in a specific catalog category.

- **Location**
  Functions typically run on your provider's server in a single specific data center. Some providers including Cloudflare and Netlify also offer functions that are executed on the edge, that is, on each server of their CDN and much closer to your visitors. This is useful for any time-critical processing like A/B testing and personalization (see the dedicated chapter in this ebook).

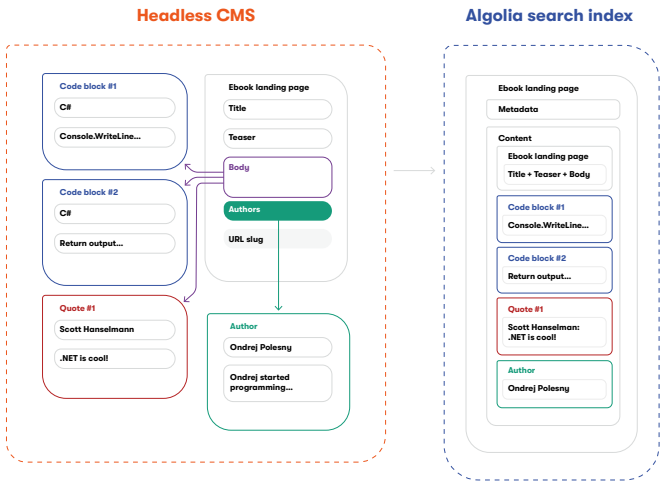
3 Search.

Depending on where you're sourcing data from, some headless CMSs like Contentful feature basic full-text search over content items. This works for simple websites or if your content model follows a web-centric approach. However, as websites grow, editors start reusing more and more content, and if the content model respects the website structure, it will quickly become a bottleneck. On the other hand, if the content model and website structure don't match, the search capabilities of headless CMSs stop being effective. And that's a good thing—the content model should primarily support the work of content editors and ensure proper content reuse.

A working approach that scales is to use an external search provider like Algolia or Azure Cognitive Search. They follow the microservices trend—are the best-of-breed tools for the job—and allow you to build search indexes based on the website structure rather than the content model. Both of the mentioned providers also constantly improve search algorithms with Machine Learning, so they don't just use density-based searching like many CMSs but allow search on your site to understand the context of what your visitors are looking for and provide a higher level of search result accuracy.



The diagram shows a typical landing page that consists of reusable content pieces. Only the landing page has a URL, so if the result of a search query contains any of the used content items, we want to navigate the user to the landing page URL. We achieve this by flattening the content structure to a web-centric search index.



In this flattening example using Kontent and Algolia, we process all (including nested) content items into searchable blocks that contain content displayed on the page.

```
// get all content
const content = await kontentClient.getAllContentFromProject();
// all items with a predefined slug property -> SEARCHABLE PAGES (indexed
objects)
const contentWithSlug = content.filter(item => item[config.kontent.slugCodename]);
// create a searchable structure based on the content structure
const searchableStructure = kontentClient.
createSearchableStructure(contentWithSlug, content);
...
createSearchableStructure(contentWithSlug: ContentItem[], allContent:
ContentItem[]): SearchableItem[] {
    const searchableStructure: SearchableItem[] = [];

    // process all items with slug into searchable items
    for (const item of contentWithSlug) {
      // searchable item structure - metadata
      let searchableItem: SearchableItem = {
        objectID: `${item.system.codename}_${item.system.language}`,
        id: item.system.id,
        codename: item.system.codename,
        ...
        content: []
      };

      searchableItem.content = this.getContentFromItem(item, [], [],
allContent);
      searchableStructure.push(searchableItem);
    }

    return searchableStructure;
  }
```

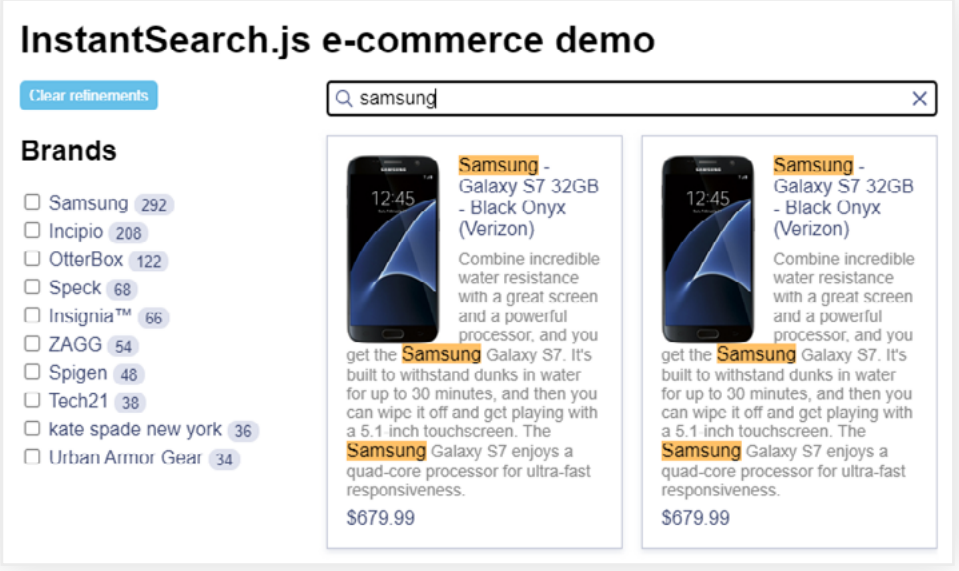
The flattened data are submitted to an Algolia index, which is configured to search through `content.contents, content.name`, and name fields:

```
async setupIndex() {
    let result = await this.index.setSettings({
      searchableAttributes: [“content.contents”, “content.name”, “name”],
      attributesForFaceting: [“content.codename”, “language”],
      attributesToSnippet: [‘content.contents:80’]
    }).wait();
  }
```

This index also supports multilingual search through facets and returns 80 characters of content around the found match.

Algolia is an external system, so all updates leading to partial or full search index rebuilds are typically handled by webhook notifications. Those updates include all used content items, even those that are reused and only added/removed from the parent page.

The front-end implementation uses an API to fetch search results from the search provider’s network. To speed up development, Algolia offers InstantSearch.js library that contains already implemented customizable components with search functionalities.



4 Headless CMS.

Any large website is serviced by a team of content editors and it's likely that the content is not exclusively created for the web channel only. Jamstack and the related tooling represent just one aspect of choosing a headless CMS—even though your client may only require a single channel for now, a website. Storing content in a headless CMS has the potential to free you from any content migrations in the future. It withstands any front-end changes and redesigns, technology changes, and so on.

Any headless CMS will do a good job when storing content for simple blog sites, but as the site grows, make sure to watch for:

- **Open/closed source**
  Most enterprise-level systems are closed-source. That doesn't necessarily ensure top quality and security but gives you a single entity that you can have a commercial relationship with. That entity is then responsible for the reputation of the product, which results in better service, higher reliability, faster responses, and quick bug fixes.

- **Hosting**
  Headless CMSs offer the following types of hosting:

  - **Self-hosted**
    You are responsible for hosting and maintaining the solution. That also means solving performance, scaling, geo-redundancy/disaster recovery, CDN, monitoring and incident reporting, WAF, security hardening for any outside communication; the list goes on and on. In most cases, you almost certainly don't want to do it.

  - **Managed hosting and IaaS**
    You host the CMS in the cloud using VM (IaaS), or the CMS vendor provides the hosting space for you (managed hosting). Some points from the previous paragraph are tackled by the provider (depending on the offering), some are still up to you, as you're the owner and maintainer of the space.

  - **SaaS**
    You register, select a plan, and use the product. Everything else is handled by the vendor.

    Note: Some CMSs like Sanity.io are a hybrid that requires you to self-host the content management front-end application.

  The right choice always depends on the specifics of each project. However, we're talking about large sites and microservices—you will need to focus on the front-end development, reliability of integrations, and communication between services. SaaS will handle the scaling, deployments, upgrades, security, and other rather unenjoyable tasks for you.

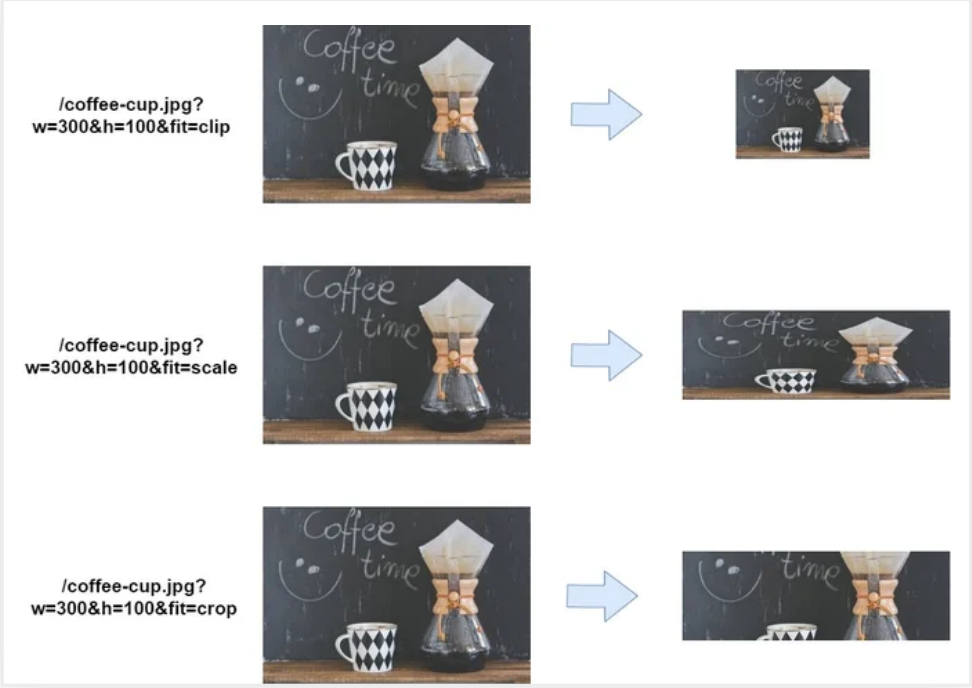
- **Developer experience**
  There is always an API, but building a site is always easier with an SDK or tools built for your target platform. Some tools like Next.js or Nuxt can work with any data

source, but frameworks like Gatsby, Gridsome, Jekyll, Statiq, and others require special plugins.

```
“dependencies”: {
      “@kentico/gatsby-kontent-components”: “^7.0.0”,
      “@kentico/gatsby-source-kontent”: “7.0.0”,
      “@kentico/kontent-smart-link”: “2.1.0”,
      “@rshackleton/gatsby-transformer-kontent-image”: “^2.1.0”,
      “gatsby-plugin-image”: “^1.11.0”
}
```

Package.json showing used plugins that simplify sourcing data from headless CMS Kontent

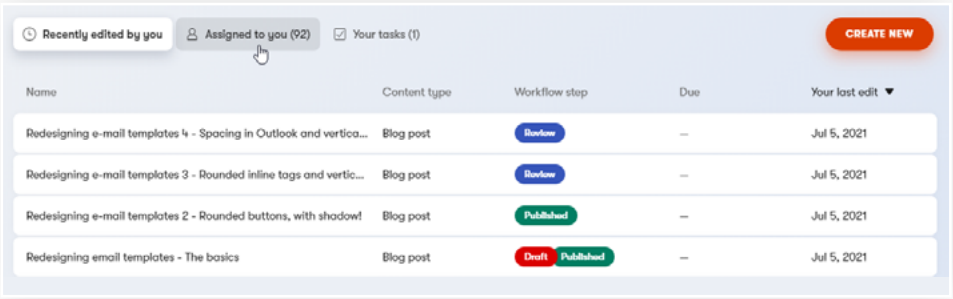
Also, pay attention to images and other media files. On large sites, the assets can easily grow to several GB that will be provided through the CMS's CDN (provided you're using SaaS). Most providers also offer an images API and SDK that help with preparing your assets for responsive sites or directly integrate with the target platform capabilities, e.g., Gatsby Image.



- **Content editor experience**

  Editors are used to traditional monolithic web-focused CMS platforms with WYSIWYG page editors which go directly against the content-first approach. However, the headless CMS needs to provide a friendly environment because if the content editor's journey is tough, the project will fail due to their lack of engagement.

  Oftentimes, editors are looking for the simplest features like keyboard shortcuts. But as their team grows, they will need tools for collaboration like inline commenting, the ability to add suggestions, safe real-time editing with auto-saving, advanced content workflow capabilities, and—of course—versions history and the ability to compare between versions. Having a personalized dashboard, a list of recently edited items, and a content calendar also helps them be productive.

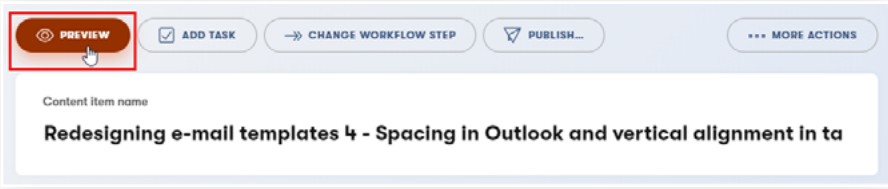


- **Visual website editing**

  Editors like to work visually and typically prefer to see their changes framed in the context of the website before publishing. There are three levels of this support:

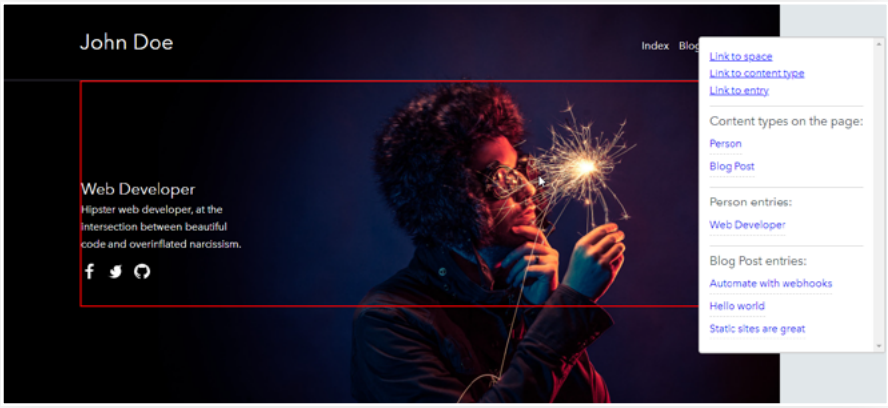
  - **Preview**

    There is a preview button that takes editors to a predefined URL on a preview version of the site that is either server-side rendered (Next.js), client-side rendered with Preview API (Nuxt), or incrementally rebuilt after every content change (Gatsby with Gatsby Cloud).



- **Links back to the CMS**

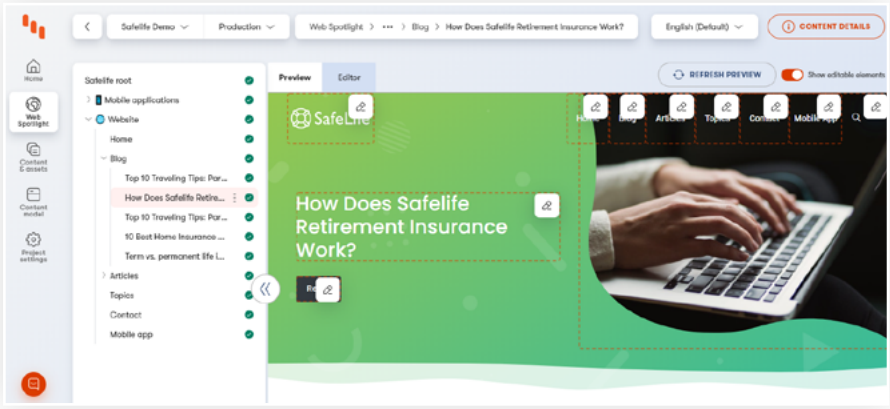
  On the preview version of the site, editors can see “Edit” buttons that take them directly to the editing interface of the backing content items and thus freeing them from having to find the respective content item in the CMS. This feature is available in many leading headless CMS platforms; the screenshot below is from Contentful.



- **Visual editing**

  Editors see the Jamstack site directly within the CMS and can edit respective content items and components without leaving the UI.

  Example: Kontent’s Web Spotlight, Storyblok

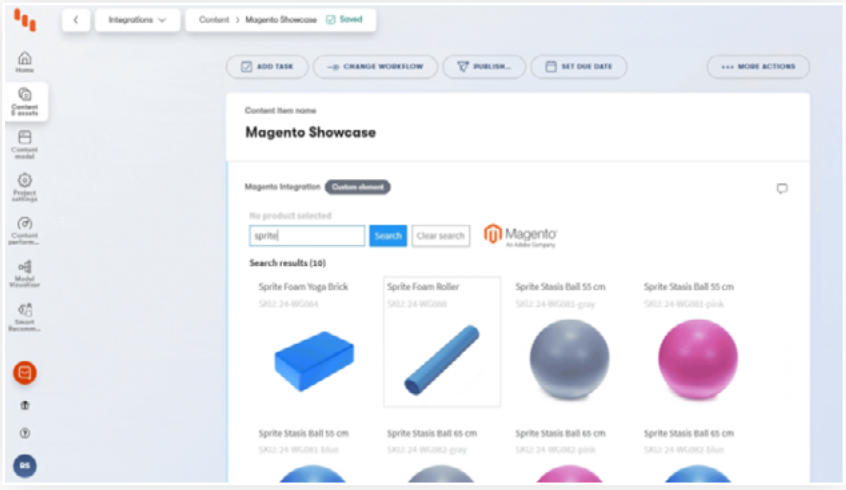


- **Performance & reliability**
  Most modern headless CMSs run in the cloud as multi-tenant applications. However, if your website grows or your client requires it, the vendor should be able to provide dedicated architecture. Jamstack sites that are statically generated are inherently more immune to data source failures, but the delivery of your content should still be handled by a CDN (very relevant for assets) so that even if the data provider fails, your website stays operational.

- **Integrations & data handling**
  Any larger site requires data and information from multiple sources or needs to communicate with multiple cloud services.

  - **External communication**
    Every headless CMS features webhooks for triggering actions of external systems. So make sure to check their granularity, reliability, and performance.

  - **UI extensibility**
    YouTube videos, Bynder images, Shopify products—editors need to work with a lot of different data types. The headless CMS should feature such integrations out of the box or provide a way to extend its UI to add such support.



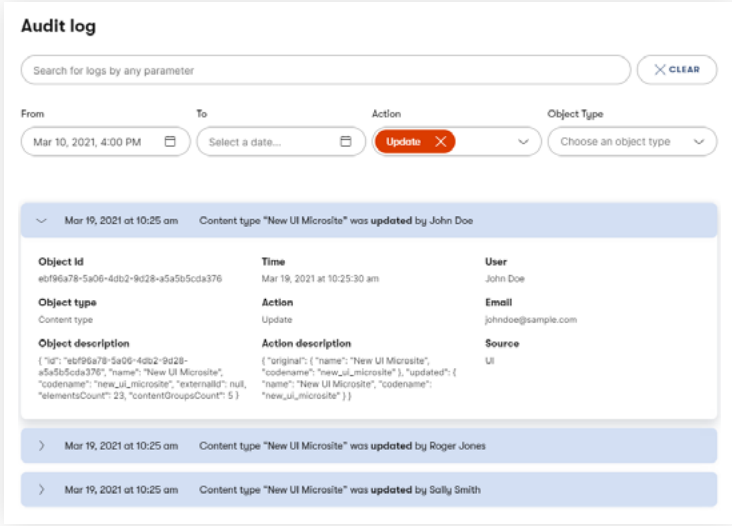
- **Management API**
  Serverless functions, form handlers, integrations—they all need a way to fetch and update data in the headless CMS. Typically, it's a Content Management API that is complemented by SDKs for multiple platforms. Check the scope of the API to make sure it supports your use cases.

- **Vendor locking**
  Clients who historically invested a lot of money into implementing legacy solutions that locked them for many years with a single vendor are now sensitive about this aspect. A good headless CMS features import/export functionalities (ideally into JSON) that allow you to take your data to another vendor should you need it.

- **Migrations**
  Hardly any project starts on a green field. The headless CMSs usually feature Content Management API and supporting tools to make data migrations possible.

- **Security**
  In the scope of security, large projects typically require features like SSO, flexibility in roles and permissions, and security certifications such as ISO 27001 and SOC 2 Type 2.

  As the number of users in the CMS grows, project managers require an Audit log to see what changes were done and by whom.



- **Support**
  Useful support engineers can save a lot of time in development, especially if they're experienced with the Jamstack tools you're using. The vendor typically guarantees response time within hours. To test this, simply ask a question and check what responses you'll receive and how soon they'll arrive.

- **Pricing**
  There are more or less three levels of pricing:

  - **Free**
    Typically self-hosted and open-source solutions.

    Examples: Strapi

  - **Low-cost solutions ($10-$500/mo)**
    Typically products built by individuals or small start-ups. They're great for small to mid-size projects but may not keep up with you if your site and requirements grow quickly.

    Examples: Prismic, Graph CMS, Storyblok

  - **Enterprise solutions ($999+/mo)**
    CMSs that are pure headless and ready to support large projects and organizations. They are prepared to fulfill requirements for dedicated architecture, have ISO certifications, offer SLAs, and can help you succeed by providing consultation support.

    Examples: Kontent, Contentful, Contentstack

  These are the typical differentiators that affect pricing:

  - # of project
  - # of users
  - # of roles
  - # of content types
  - # of content items
  - # of languages
  - Allowed bandwidth (and applied FUP)
  - Additional services like support, SLA, etc.

Additional functional features.

Headless CMS is the best-of-breed system for content management. Nevertheless, many developers expect it to handle other functional requirements of Jamstack sites, like forms management, email sending, tabular data management, and so on. Headless CMSs don't support any of these because they shouldn't. If your project requires such functionalities, try to pick a specialized service that fulfills your use case. See the table below that lists a few providers per each category:

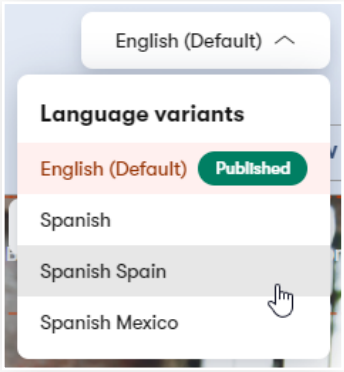
Forms	Netlify Forms Form.io Basin JotForm Your CRM (like Salesforce)
Email sending	Sendgrid Mailchimp Mailjet
Tabular data	Serverless databases - AWS Aurora - Cosmos DB Blobs - AWS S3 - Azure Blob Static JSON files
Custom Analytics	Google Tag Manager Custom Event Netlify Analytics Segment (CDP) Tracking (Google Analytics)

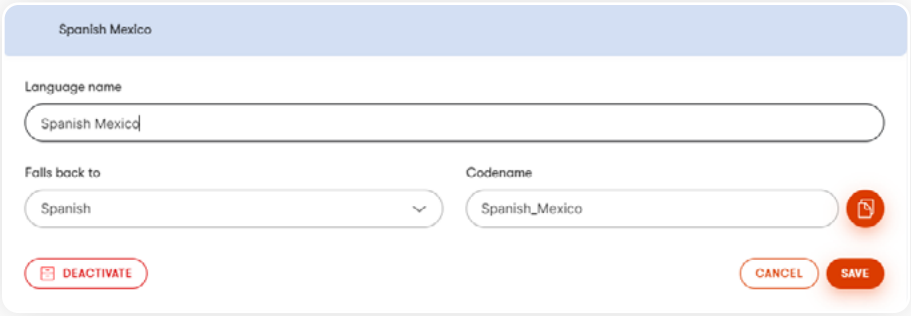
5 Multiple languages.

Multilingual support is a very underestimated feature and requires a lot of support on both the data source and site implementation sides. Most often, Jamstack sites are built with content from headless CMSs and their job is to fully support the multilingual capabilities listed below.

- **Multiple languages**

  The support for multiple languages needs to go beyond simple culture codes. There are dialects and regional specifics you may need to deal with as well as define their fallback language.





- **Content management**

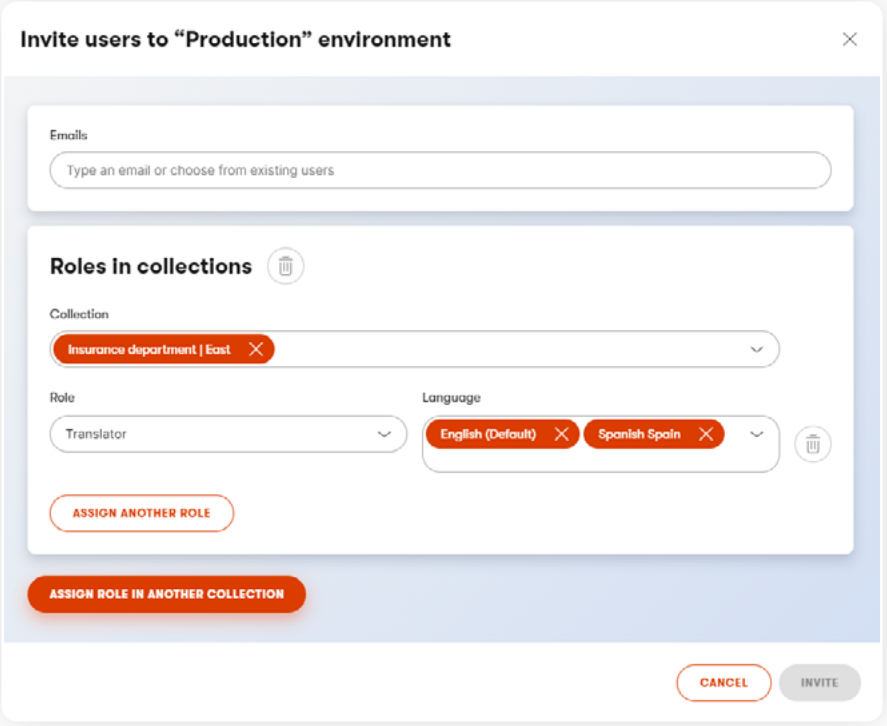
  On large sites with a lot of content, you will often deal with edge situations related to languages and translations, e.g.:

  - A content item references another content item that does not exist in the current language. Does that fulfill a “required” condition?

  - Rich text element links a page that is available only in a specific language, not the general fallback. Is this OK, and how are you going to render it on the website?

- A personalization variant of a component is not translated into the current language. Should you exclude it?

- Typically, these sites also have many users who maintain content in specific languages, and if we're talking about corporate sites, each country site may have different requirements for languages. The used CMS needs to be capable of granularly setting permissions.



The implementation in Jamstack is the easier part. Every CMS gives you the ability to filter content by language in its API or SDK. If you're using a front-end framework that requires a special plugin, verify how it handles translations and fallback languages.

For example, if your site is built with Gatsby, the Kontent source plugin adds two language-related fields to every content item `- preferred_language` and `system. language`:

```
{
          “node”: {
            “elements”: {
              “title”: {
                “value”: “Accounts Lek 24/7 cutting-edge support Spring non-
volatile”
              }
            },
            “preferred_language”: “cs-CZ”,
            “system”: {
              “language”: “en-US”
            }
          }
}
```

If you filter for `{preferred_language: {eq: “cs-CZ”}}` in the GraphQL query, you will get content items in the Czech language (if available) or their English fallbacks. The actual language of every item is in the `system.language` field, so you can decide whether or not to use it.

This distinction also allows you to inform website visitors that a linked item is only available in the fallback language:

Ahoj, jmenuju se Ondřej, pracuju jako Developer Advocate v DevRel týmu s Petr Švihlík (english).

6 Authentication & gated content.

First of all, let me explain two terms related to protecting assets on sites:

- **Authentication: Identifying a visitor**
- **Authorization: Deciding whether a user can access a specific resource**

In the scope of Jamstack sites, it's important to know whether we want to just authenticate visitors or also authorize them.

Typically, we aim for one of the following use cases:

- User profile, user data forms & other highly personal pages
- Content personalization based on rules (see the dedicated section below)
- Gated content

User profile, forms & other highly personal pages.

As I described in the Performance section, it doesn't make sense to pre-build every single page for large sites. With user profiles and other highly personal pages, it's almost impossible due to frequent changes in the data set. You're left with three options for fetching data:

- **Server-side on demand**
  With Next.js incremental static regeneration or Netlify's DPR, these pages can be rendered on demand. Rendering happens on the server, and the client waits for the response.

  Example: public profile page of a user, list of recently played games on a gaming portal, etc. Those are all pages that are user-specific, but still public.

  This is a code example from Next.js showing incremental static regeneration:

```
export async function getStaticProps({ params }) {
  return {
    props: {
      userData: await getUserData(params.urlSlug)
    },
    revalidate: 60
  }
}
```

- **Client-side**

  The particular page acts as a little SPA. It is served as a static page with JavaScript code that makes an API request on every page load. As soon as the server responds with data, we render the page.

  Example: user data management page, order history page, etc.

  This is a code example from Next.js showing gathering data dynamically on the client:

```
export default () => {
    const router = useRouter()
    const [ isReady, setIsReady ] = useState(false)
    const { userId } = router.query

    // get orders by UserID on client
    const { data, error } = useSWR(() => userId, getOrders)
     ...
    if (isReady && data && Array.isArray(data))
    {
        // render HTML
```

- **Server-side at request time**
  The use case is the same as for the client-side, but here it's the server that fetches the data at request time before responding. That causes a bit slower TTFB but removes the additional async request on the client.

  This is a code example from Next.js showing server-side data fetching:

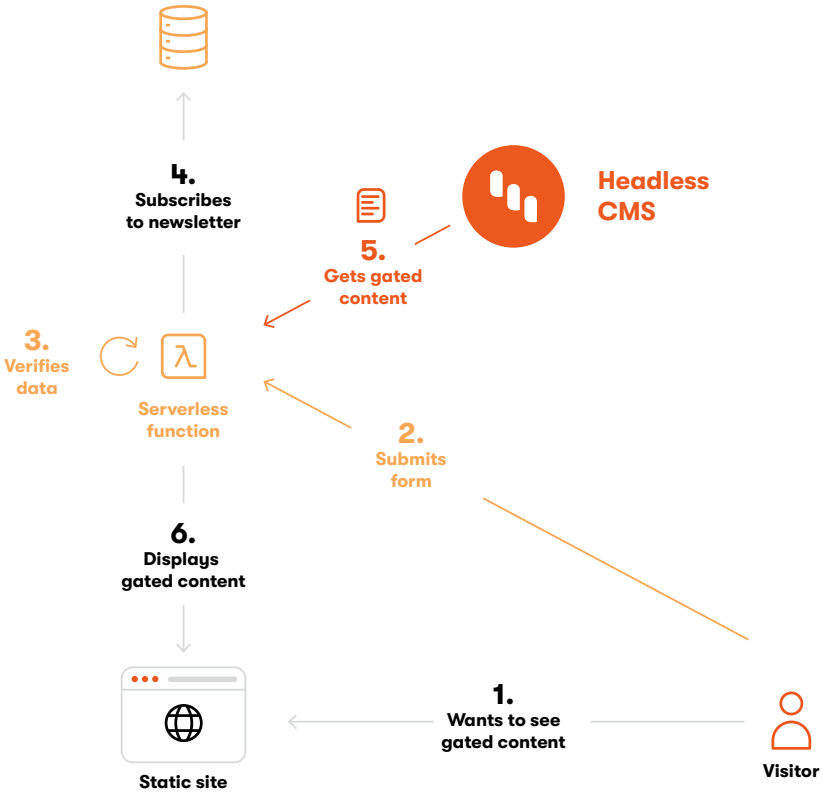
```
function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps(context) {
  const { userId } = context.query
  // Fetch data from external API
  const data = await getOrders(userId)

  // Pass data to the page via props
  return { props: { data } }
}

export default Page
```

The right solution highly depends on the use case. Nowadays, the trend is to go with server-side data fetching on the edge, as it removes the additional logic and processing on the client and saves one round-trip to the server. However, if the data on the page requires authentication, you'll need to go with API calls anyway and first ensure that the user is authenticated and has proper authorization.

Gated content.

A common example of a gate is an age-gate, often used when a user wishes to access products or content that require them to be older than a certain age, e.g., for purchasing alcohol. Gates are often used in lead generation. The intention is that a report or article is enticing enough for a user to provide contact details.

To implement gated content, we always need a serverless function or another form of server processing power. Both Next.js and Gatsby support serverless functions out of the box; for any other framework, you can just use them separately.



The serverless function needs to solve the following tasks:

- Verify that the user has completed the required action.

```
// this checks whether the user provided an email address
const data = req.body.email;
if (!email) {
return res.status(400).send({ message: 'Email not provided.' });
```

- Perform the desired action with user data (subscribe to a newsletter, verify the payment, etc.).

```
// subscribe the user to our newsletter
subscribe(email);
```

- Provide the gated content.

```
// get gated content from the headless CMS
const contentItem = await getGatedContentItem();
return res.status(200).send({
  gatedContent: contentItem.gated_content.value ?? '',
});
```

The gated resource can come from the same project within the headless CMS as public content or be stored in separate storage to protect project IDs and/or access keys.

If we want the gated content to be indexed, we need to take three more steps:

- Recognize Googlebot on the front end and provide the gated content directly.

```
// If we think this is a Google request then fetch content immediately
const isGoogle = navigator.userAgent.toLowerCase().includes('googlebot');
if (isGoogle) {
  const content = await fetchContent({ animal });
}
```

- Recognize Googlebot in the serverless function and provide the content.

```
const isGoogleAgent = req.headers?.['user-agent']?.toLowerCase()?.
includes('googlebot') ?? false;
if (isGoogleAgent) {
  let verified = false;
  try {
    // Verify Googlebot via reverse DNS lookup.
    const ip = (req.headers['x-forwarded-for'] || req.socket.remoteAddress) as
string | undefined;
    verified = await verifyGooglebot(ip);
  } catch (error) {
    …
  }
}
```

The Googlebot verification is based on the known IP addresses; the full implementation is available on GitHub.

- Mark the content as paid using schema.org attribute isAccessibleForFree.

You can see the complete working example of gated content with further details here.

7 Personalization & A/B testing.

Many developers see Jamstack as a good fit only for small sites because they don't see how personalization and similar features can work on a static site. In this chapter, I will explain both personalization and A/B testing, as they are very similar to a certain extent.

Personalization.

There are two types of personalization:

- **User-specific content**
  When you want to display personalized greetings, show the customer's last purchase with a link to "purchase again," and so on.

  Example: "Hey Ondrej, your last visit was: 1 week ago from France (IP: 75.122.14.151)"

- **Persona and group-specific content**
  Personalization is based on the information you know about the user but is applicable to a group of people, not an individual.

  Example: product recommendations, banner variant, etc.

Both need a server, serverless function, or client-side JS. The user-specific content requires you to know the specific user—they need to be authenticated or come to the site via a special link that contains their identifier.

The personalization based on persona or group has three parts:

1. First, you need to collect some data about the user. Typically, you monitor what they do on the site via a tracking script. For example, the user can be browsing through home printers or looking at gaming consoles.

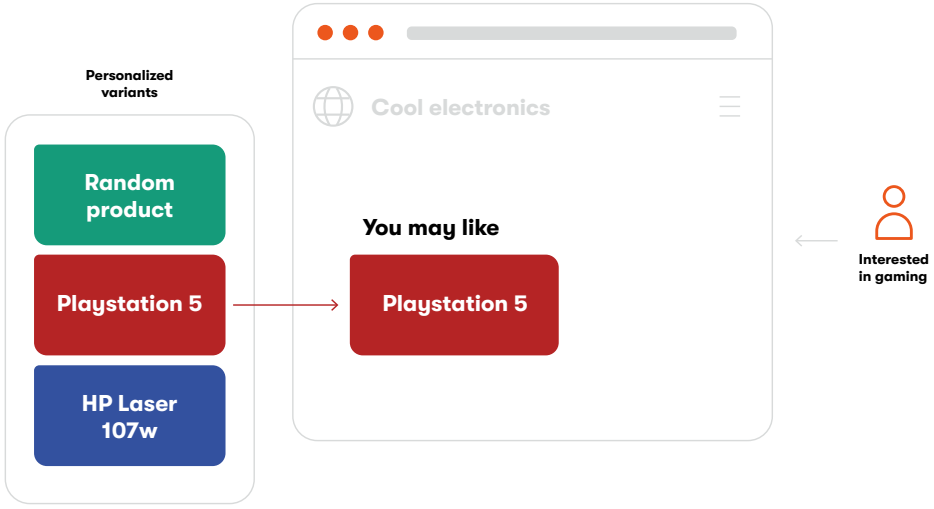
/gaming
/gaming/playstation-5
/gaming/playstation-4-slim
/gaming/playstation-vr-megapack
...

2. Then, when you've recorded enough activity, you evaluate this data and draw some conclusions about the user. In this example, you'd assign the user to "interested in printers" and "interested in gaming" personas respectively.

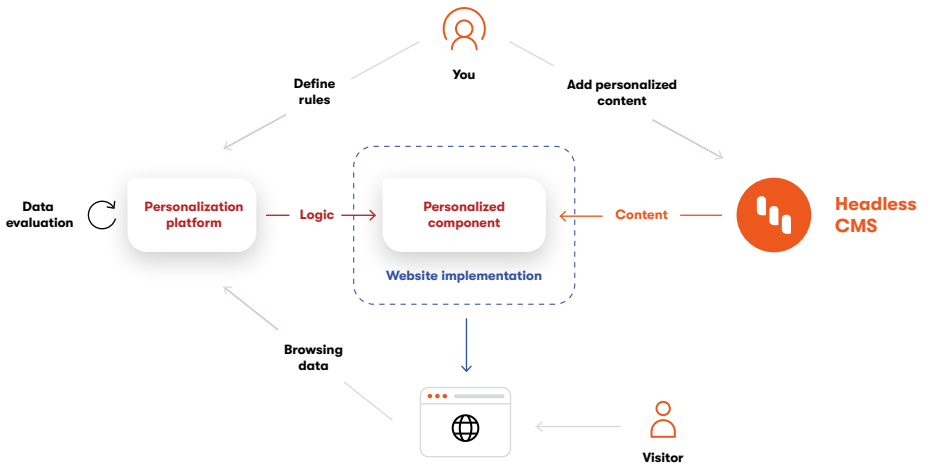
/gaming
/gaming/playstation-5
/gaming/playstation-4-slim
/gaming/playstation-vr-megapack
...

Interested in gaming

3. And finally, you can personalize the content on your site. Typically, you prepare multiple variants of some content and assign them to your personas. For example, a "recommended product" box will be showing HP Laser 107w for the "interested in printer" persona and a PlayStation for the "interested in gaming" persona.



Of course, in reality, personalization engines, including Ninetailed, Uniform, Salesforce Pardot, and others, are equipped with advanced techniques to understand the user and properly guide them using personalization to a successful conversion. The terminology and approach of each of these platforms are a bit different, but this is in general how it works.



So how we do this on a static site:

- **Client-side with async requests**
  Once the page loads your code, the browser makes an async request for the personalized content. Typically you ask the tracking system (like Pardot) for the identification of the content that should be displayed:

```
<script type="text/javascript" src="https://tracker.kontent.ai/dcjs/849473/354/dc.js?v=1627544857873"></script>
```

  The job of the tracking system is to identify the source of the request (specific user) and evaluate what type of content should be displayed. It responds with the content ID:

```
document.write("the_state_of_jamstack_2021_report");
```

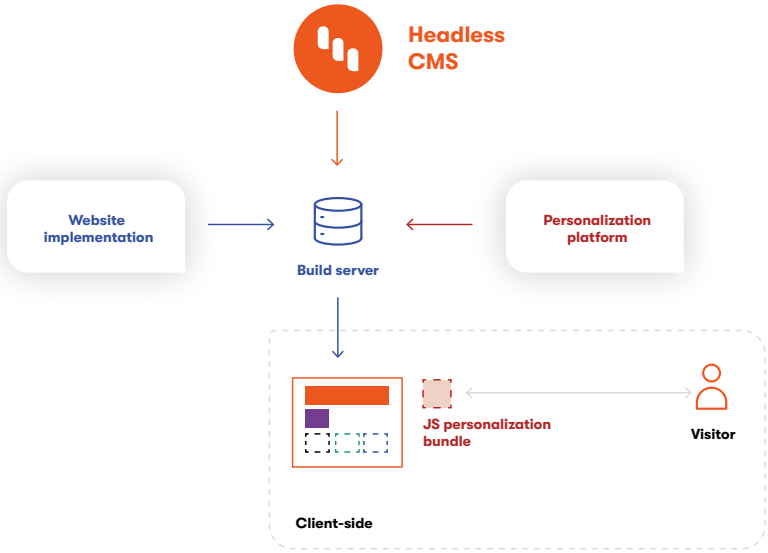
Then the browser makes another request to the used content storage (headless CMS), fetching and displaying the content:

```
fetch("https://deliver.kontent.ai/{projectId}/items/the_state_of_jamstack_2021_
report")
        .then(function (response) {
          return response.json();
        })
        .then(function (data) {
          var html = getHTML(itemToDisplay, data);
          container.innerHTML = html;
        })
```

The last step can be eliminated by pre-rendering all content variants into the static page and only displaying a single variant once we know which one. Find out more about the technical specifics or overall personalization process on the Kontent blog.

- **Fully client-side**

  Some providers like Uniform allow you to move the tracking, evaluation, and personalization to the client-side completely. Once you define the rules, the logic is bundled with your site's implementation and executed on the client.



Uniform provides a special UI component that handles all the steps automatically; you only need to provide the content variants:

```
export const PersonalizedHero = ({ item }: { item:
PersonalizedHeroData }) => {
  return (
    <Personalize
      variations={item.heros}
      component={Hero}
      trackingEventName="heroPersonalized"
      loadingMode={PersonalizedHeroLoading} // specify a
component for a potential loading state
    />
  );
};
```

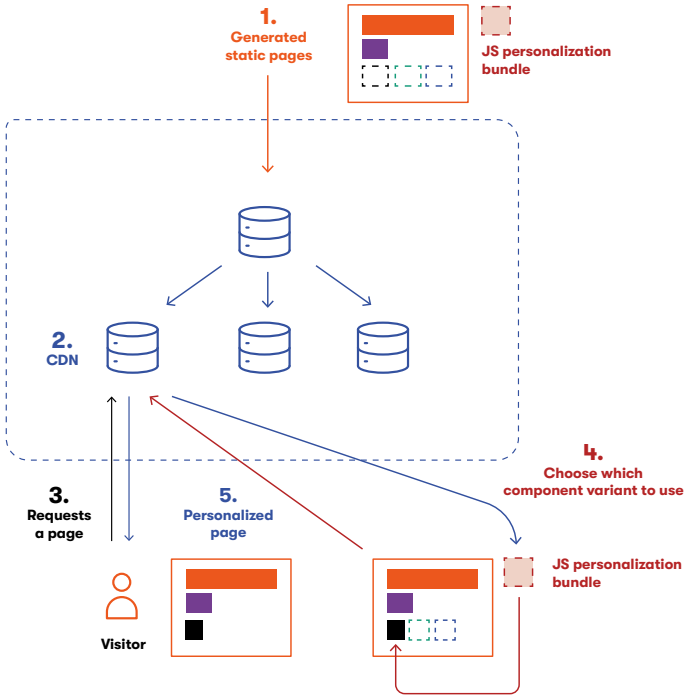
There are no additional requests necessary apart from the site's JS bundle.

The personalization rules are managed in the Uniform UI, and with every change, the system triggers a new site build using a webhook. Check out this guide in Uniform's docs which describes how this type of personalization works in detail.

- **Server-side/Edge**

  Both types of client-side personalization have one disadvantage—the async requests take some time. How much depends on your visitor's latency and device speed, but we're talking hundreds of milliseconds even in ideal cases.

  These days, you're able to leverage serverless functions that run on the edge, much closer to your visitors. They act as a proxy server—the request for a static page goes to them, they intercept the response and post-process the already generated page—they choose which variant of the pre-generated ones should be displayed.

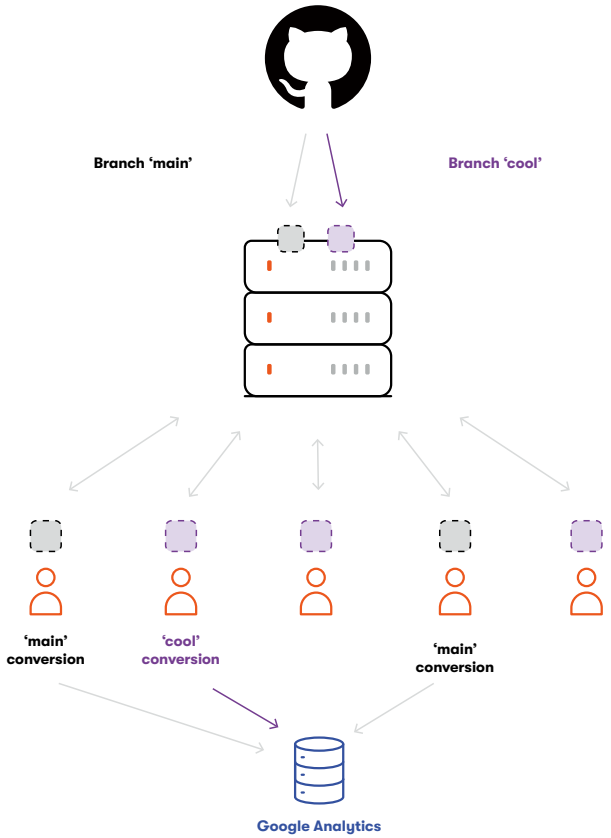


  They effectively do the client-side job on the CDN and use a small cookie to identify the visitor. You can also use a hybrid approach that first personalizes on the edge, and once the bundle is loaded, continues on the client-side.

A/B testing.

A/B testing is very similar to personalization, but the evaluation logic is replaced with a simpler decision logic. You can run an A/B test on the whole website traffic splitting it into equal halves, or you can decide to test only people from the US and so on. The important thing is to always show them the same variant once you've shown them one—this is achieved by storing a small cookie on the visitor's browser.

A/B testing is simpler to implement than personalization because you don't need to know either anything or very little about your visitors. That's why many Jamstack hosting providers offer tools for A/B tests, like Netlify that lets you split the traffic between two versions/branches of your site or Layer0 that also supports splitting traffic between multiple sites.



The results of such testing come from a specified goal, typically a conversion. On each version of the page, you need to adjust the Google Analytics tracker or similar code that logs successful conversions:

```
<script>
  ga('send', 'pageview', {
    'Branch':  '{{ getenv "BRANCH" }}'
  });
</script>
```

Note: This code sample comes from Netlify docs and shows how to include the branch name in the tracked conversion—in this case, a simple page view.

You can also handle A/B tests purely on the client-side with tools like Google Optimize. Check out this article for more info.

8 E-commerce.

All the previous sections about performance, search, personalization, and so on come together here. E-commerce requires you to combine all the features of the modern web to create the best visitor experience.

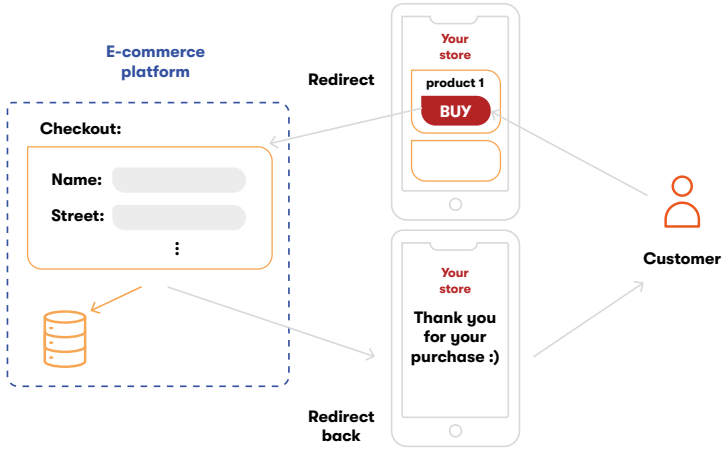
All articles about e-commerce tell you that the conversion ratio decreases exponentially with long page loads. Pre-generated sites bring performance out of the box, so it seems like a great fit for e-commerce. However, there are a few things you need to consider:

- Product catalog
- Shopping cart
- Checkout process
- Payment

With e-commerce, the situation is more complicated than with other services. For example, you expect a hosting provider to build & host your site just like you expect a search provider to maintain a search index and give you an API or tools to use on your site. E-commerce, or headless e-commerce, providers don't all offer the same type of integration.

- **Monoliths**
  If your client is already using a provider like Shopify, Prestashop, Magento, or a similar custom solution and they are happy with it, you'll likely be looking at ways to integrate these solutions with Jamstack.

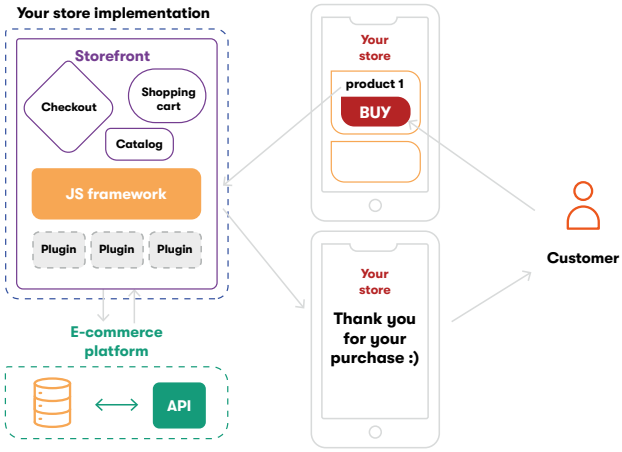
  - **Buy button**
    The platform lets you define a button or a larger section of products that you can add to your site using a generated code block. It works either client-side (similarly to an iframe) or is hosted on the provider's infrastructure and uses redirects.



Example: Shopify buy button

- **Storefront**

  If you need more freedom or feel like the JavaScript site add-on is not good enough, you can use Storefront—a layer, an implementation, on top of a specific tech stack that connects e-commerce platforms and front-end frameworks. It's like a plugin that provides e-commerce functionalities.



  Make sure to select the one that fits your tech stack. Very popular are Vue Storefront that extends Nuxt.js and connects with almost all e-commerce monoliths, Next.js Commerce and React StoreFront if you're using React and Next.js, and Gatsby plugin for Shopify.

- **Custom solution**

  If the monolith your client is using is not supported by the storefronts or you want to build the front end yourself, you can always use its API and connect it manually.

- **Checkout add-on**

  A good solution if you're looking for a simple way to sell just a few products. It's typically a simple buy button that redirects your visitors to a checkout page hosted by the provider.

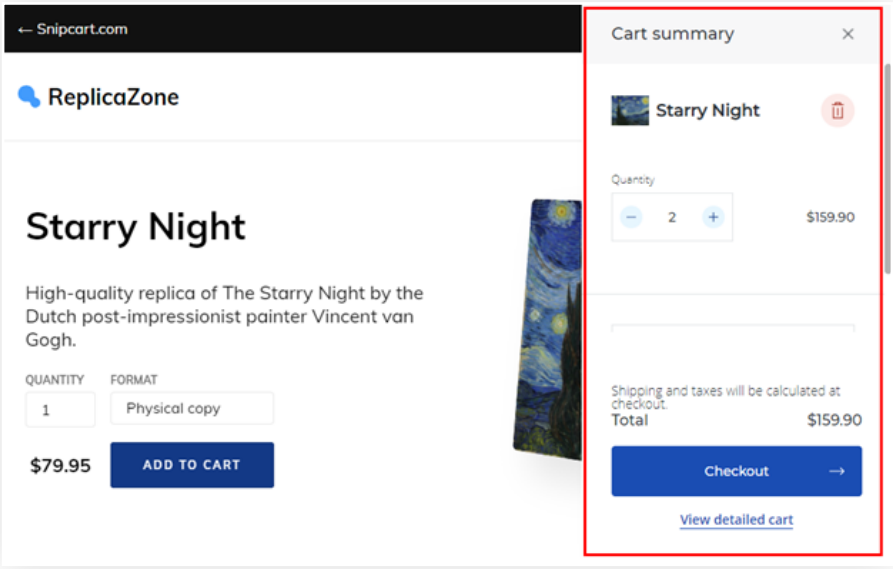
  Example: Stripe Checkout

- **Full add-on**

  The integration is done by a client-side JS bundle that you add to your site:

```
<script type="text/javascript" data-api-key="{YOUR_API_KEY}" src="https://cdn.
snipcart.com/scripts/snipcart.js"></script>
```

The JS code adds the shopping cart and checkout functionality in a component that slides over your website when needed.



In the case of Snipcart, all products in your store need to be rendered with special HTML attributes defining their identifiers and prices:

```
<button type="button" class="snipcart-add-item"
  data-item-name="Headphones"
  data-item-price="200.00"
  data-item-id="42"
  data-item-url="https://snipcart.com/headphones">
Add to cart
</button>
```

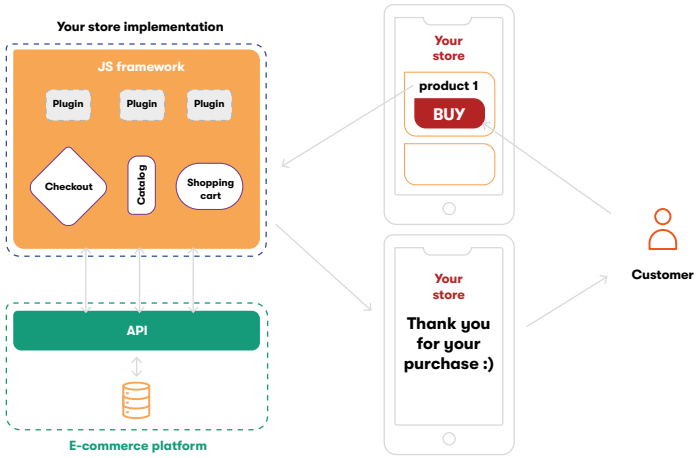
This is all happening on the client. Before the order is processed, Snipcart goes back to your site (`data-item-url`) and checks the HTML attributes of ordered products if they match the submitted order.

Another provider, Gumroad, requires you to store the products within their system, so the checks are done on the provider's side.

Example: Stripe, Gumroad

- **API-first**

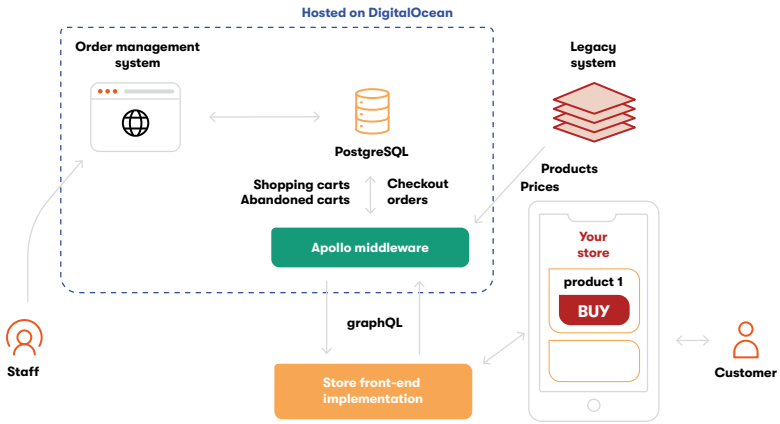
  Earlier, I mentioned monoliths that expose an API, but there are also pure headless providers that are API-first. The well-documented API and sample apps for multiple platforms are a standard, but some also provide GraphQL endpoint.



Example: Commercetools, Commerce Layer, Slatwall

- **Custom solution**

  In cases when monoliths or API-first platforms don't suit your needs, you can build the solution yourself. Yes, it will cost you a lot of development time, as you need to address all the aforementioned aspects (product catalog, shopping cart, checkout process, and payment) on both fronts—on the front end and back end. Even so, this approach is not that rare, and I've seen it successfully run in production multiple times. Take a look at the following diagram showing an architecture built by one of our clients:



If this is relevant to you, I'd recommend this article about combining data sources in a middleware layer that separates concerns and simplifies the front-end implementation.

I also recommend the Headless E-commerce guide published on Snipcart's blog that features this nice comparison:

Solution type	Website design	Content management	Product management	Cart & checkout	Store management
The Monolith	✓	✓	✓	✓	✓
The Add-On	⊘	⊘	✓	✓	✓
The Storefront	✓	⊘	⊘	✓	⊘
The API	⊘	⊘	✓	⊘*	✓
The Custom	⊘	⊘	⊘	⊘	⊘

*many API-first solutions offer beautiful cart/checkout frontend starters.

Just keep in mind that a row full of checkmarks does not mean it's the ultimate choice. Different clients, different development teams, and mainly different budgets since there are huge differences in pricing among e-commerce providers—these all require a specific approach. That's also why neither this section nor this comparison table tell you what the best solution is. However, considering you're reading this book and want to use Jamstack for your projects, your best bet will likely be the API or the Storefront approaches.

In conclusion.

In this book, I showed you that Jamstack can be used for large projects. Just like any other way of building websites, it has its disadvantages and can't be blindly used for everything. Especially larger websites with specific features require you and your team to evaluate the pros and cons like you would do for any typical client-server architectures based on .NET, Java, or PHP. But the fact is, Jamstack brings a lot of benefits including performance, security, simplicity of development, and openness of the communities, out of the box. And when I look at how rapidly it evolves, how static site generators have become true website-building frameworks, how providers race to introduce new services and improve the already existing ones, one thing is clear—Jamstack is not only a viable way of building websites but is also becoming a standard and will continue to grow in that sense.

Stay in touch.

 Twitter

 Discord

Kontent.

About Kontent.

Kontent by Kentico is a cloud-based headless CMS that features first-class integrations with all leading Jamstack frameworks and many sample apps to speed up your development.

Kontent represents the easiest way to manage content with Jamstack by allowing both developers and editors to focus on what they love. Developers can fetch content via CDN-backed API or GraphQL and process it using an open-source SDK for their preferred platform. Editors become more productive thanks to a personalized dashboard, multiple fully customizable workflows, and advanced collaboration features including comments, suggestions, and tasks.

Kontent is prepared to deliver on enterprise requirements like SLA, dedicated architecture, professional customer services, and training. In addition to that, the platform has three ISO certifications (ISO 9001, ISO 27001, and ISO 20000) and has passed the SOC 2 Type 2 examination.

SEE KONTENT IN ACTION →

“The best part of the Jamstack architecture is that it gives you freedom to choose the best-of-breed service for each of your requirements. For content management, there are many to choose from, but Kontent provides us with an unmatched balance of first-class customer support, developer features and SDKs, and market-leading collaborative editing features more akin to Google Docs than a typical headless CMS.”

Andy Thompson.
CTO, Luminary

About the author.

Ondrej Polesny is Developer Evangelist at Kontent. Always interested in problems that everyone claims have no solution, Ondrej enjoys building the architecture of components or applications, and figuring out how all parts fit together. He is constantly in touch with clients and the developer community which brings him closer to many interesting digital projects including, but not limited to, Jamstack.

Ondrej Polesny.
Developer Evangelist, Kentico Kontent

Twitter: @ondrabus

Website: ondrabus.com