

O'REILLY®

Compliments of  
**storyblok**

# Decoupled Applications and Composable Web Architectures

Building for Resilience,  
Flexibility, and Immediacy

Stefan Baumgartner

REPORT

Dear Reader,

We are excited to present to you the report “Decoupled Applications and Composable Web Architectures,” commissioned by Storyblok, a content management system (CMS) that empowers all teams to create and scale modern content experiences across any digital channel.

Many of our customers love us for our visual editor, enabling their marketing teams to create and publish content independently without having to talk to a product owner or create Jira tickets. Others for our custom workflows and native collaboration capabilities that enable their teams to work together faster on a single CMS. For us however, It all starts with composable architecture.

From our conception in 2017, we knew Storyblok had to be cloud-native, API-first, and composable. We believe these are core qualities every modern tech stack must have in order to give developers the flexibility to build great user experiences at scale while keeping up with fast-paced digital innovations.

In 2023, the need for flexibility, speed, and resilience in web development has become paramount. Companies that can deliver standout content experiences faster, develop more efficiently and enjoy a significant advantage over their peers. Those who don't will face increased maintenance costs and slow time-to-market as personalization and omnichannel strategies become increasingly complex.

We are proud to have been the only CMS recognized as the Customers' Choice in Gartner's Peer Insights 2023 report for web content management, validating our dedication to excellence. We are grateful to our customers, partners, and community members who've consistently named us as a CMS category leader on G2. However, the number we are most proud of is this: 582%.

That is the typical return on investment of a Storyblok customer over three years, as estimated by Forrester Consulting whom Storyblok commissioned to estimate its potential impact on existing and future customers using their Total Economic Impact™ (TEI) methodology.

One of the main factors contributing to this incredible return on investment is the efficiency and flexibility provided to companies that build off our composable architecture. We've seen firsthand how our customers like Oatly, T-Mobile, and Marc O'Polo have significantly improved their deployment speeds, reduced development and maintenance costs, and boosted engagement with exceptional content experiences built and scaled through headless architecture. Composable architecture doesn't just transform content and web applications. It drives business. Visit us at [Storyblok.com](https://Storyblok.com) to see for yourself.

As a company dedicated to empowering developers and technical practitioners with the transformative power of composable architecture, we believe this report holds immense value for you.

We hope you enjoy it.

Sincerely,  
Dominik Angerer  
CEO Storyblok



---

# Decoupled Applications and Composable Web Architectures

*Building for Resilience, Flexibility,  
and Immediacy*

*Stefan Baumgartner*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY**<sup>®</sup>

## **Decoupled Applications and Composable Web Architectures**

by Stefan Baumgartner

Copyright © 2023 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Acquisitions Editor:** Louise Corrigan

**Development Editor:** Jeff Bleiel

**Production Editor:** Kristen Brown

**Copyeditor:** Charles Roulmeliotis

**Interior Designer:** David Futato

**Cover Designer:** Randy Comer

**Illustrator:** Kate Dullea

August 2023: First Edition

### **Revision History for the First Edition**

2023-08-28: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Decoupled Applications and Composable Web Architectures*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Storyblok. See our [statement of editorial independence](#).

978-1-098-15144-7

[LSI]

---

# Table of Contents

<b>1. Composable Web Architectures.....</b>	<b>1</b>
The Four Aspects of a Web Application	2
Content Management with Headless and Decentralized Systems	5
Decoupling the Application Layer Through Single-Purpose Services	9
Frontend Flexibility: Pull Versus Push Deployment	12
Conclusion	17
<b>2. Developing and Operating Decoupled Applications.....</b>	<b>19</b>
Technology Lock-In, Technology Choices, and Rewrites	19
Rethinking Backend for Frontend	22
Serverless as a Rendering Layer	25
Frontend Composability	27
Third-Party Services	30
Conclusion	33



# Composable Web Architectures

The rise of the cloud has changed how we develop applications tremendously. While many organizations have slowly but steadily moved toward microservices, the traditional website or web application still relies on monolithic software being squeezed into containers in order to meet cloud native goals (such as high availability, high performance, resilience, scalability, and redundancy). But what works in cloud native applications can also work for websites and apps by embracing composable web architectures.

Composable web architecture emphasizes modularization, flexibility, and reusability of components. In this architecture, the various parts of the application (such as the frontend UI components, backend APIs, and data sources) are broken down into smaller, independent pieces that can be easily combined or replaced as needed.

The idea behind composable web architectures is to create a system that is flexible and adaptable to changing requirements or technology trends. Instead of building a monolithic application that is difficult to change or update, developers can focus on building smaller, more focused components that can be easily swapped out or updated as needed. This allows development teams to reduce the time to release significantly, as the way is paved to easily include and integrate new services.

One of the key benefits of composable web architectures is that they allow for greater flexibility in the technology stack, and reduce the chance of vendor lock-in. Because the various components of the application are independent, it is possible to use different

programming languages, frameworks, or tools for different parts of the application.

In this chapter, we will look at the benefits and trade-offs of composable web applications at large. We do so by dissecting the traditional monolith, and composing the remaining features in the frontend. In doing so, we discuss the trade-offs of “rolling your own” as opposed to using established vendors, and how integration with existing systems can work. After reading through this chapter, you will be able to migrate from your monolith to a composable web architecture, no matter what your original setup looked like.<sup>1</sup>

## The Four Aspects of a Web Application

Web applications can be separated into four different aspects or layers. Each aspect has different goals and requirements, as well as different tools to get the job done. In general, we identify the layers as:

### *Application*

The application layer includes specialized backend services that deal with concrete tasks like data processing and application state. This includes things like dynamic search, filters for product catalogs, and user management. Its main stakeholders are development teams.

### *Frontend*

The frontend layer is the direct interface to the users. It is responsible for delivering accessible and semantic HTML, showing fantastic designs using CSS, and creating engaging, dynamic user experiences with the addition of client-side JavaScript. Its main stakeholders are frontend and design teams.

### *Content management*

The content management layer deals with the organization of content, the creation of data structures, and convenient ways of editing said content. Its main goals are ease of use and flexibility in structure. The main stakeholders behind content management are e-marketing teams and content editors.

---

<sup>1</sup> For more radical, almost unapologetic takes on composable web architectures, check out Jamstack and MACH.



## *Runtime*

The runtime layer deals with deployment or hosting of all aspects of a web application. Its main goal is to find the right environment for the technology choices made by the other teams to ensure the availability, stability, and security of the system. Its main stakeholders are operation teams.

Sometimes organizations form roles, if not entire teams, with those exact four names, in order to take care of different parts of their product.

Usually, teams put in great effort to pick the right technology that suits their needs, which might be familiarity with a programming language, speed of development, ease of deployment, or ease of use. Many factors have to be taken into account, and monolithic architectures usually don't provide the silver bullet for all aspects. In fact, monoliths that work well in one aspect might have a negative impact on another.

Let's look at an example where a company has chosen to use WordPress, a traditional PHP-based blogging system and content management layer, as the main technology for their website. Let's assume that WordPress was mainly chosen because the e-marketing team loves the editor and flexibility in structuring content, or they are familiar with the SEO tools that come with the rich WordPress plug-in ecosystem. How does this decision affect all the other layers?

The *application* team now needs to write PHP to create new services. Maybe they can hook on existing functionality from WordPress already, or they find a plug-in that does the right things for them. But if the team can't find existing solutions, they need to deal with PHP as a programming language, even if this might not be their favorite tool or they lack experience in it. Even if they are fluent in PHP and it is their main programming language, they would have to understand WordPress internals and APIs to meaningfully get work done. This is not only limiting, but also means that the software written for this system is inevitably locked in with WordPress.

The *frontend* team is also stuck with the way WordPress produces HTML. WordPress lacks a templating engine out of the box, and developers need to be familiar with WordPress internals as well. Usually this includes lots of conditionals, branches, and partials that can be very frustrating. After all, WordPress has been designed as a blog system, even if it is capable of so much more. The main

job of a frontend team is to deliver accessible and semantic HTML. If the main way to produce HTML is so tightly coupled to the e-marketing team's choice of platform, they need to be experts of the platform, not of their craft. This again can be limiting, which ultimately results in broken or subpar applications.

Last, but not least, the *runtime* or operations team need to provide the right infrastructure and servers to run WordPress. This includes PHP, MySQL, Linux, and Apache or NGINX. While WordPress and its choice of technologies may be easy for small-scale websites, it may be really troublesome in the enterprise sector, where security, availability, and stability are the main driving factors for reliable hosting and operations of your applications.

For example, if we wanted to host WordPress in AWS, things could get very complicated. We would need to make sure that we use off-the-shelf service, follow separation of concerns, replicate data, scale automatically based on traffic, and distribute requests on several availability zones. The details would go beyond the scope of this report, but check out [the AWS documentation on WordPress hosting](#) for more details.

Of course, there are situations that require less setup than this WordPress example. But because requirements for any project change, your setup will likely need to adapt to prevent it from becoming complicated. This is especially true if you need a resilient and secure system.

As you can see, the simple decision of one stakeholder—in this case online marketing and their choice of content management system (CMS)—has a fundamental impact on all other aspects, and teams might not be able to fulfill their goals because of that technology choice. Similarly, how would other aspects be influenced if you chose a technology based on its runtime capabilities, or its frontend capabilities? What if the application and runtime team decide to go for .NET and Windows machines in the cloud only, and none of the available software matches the needs of the frontend team or content management team? Different trade-offs would have been made, but the trade-offs are there.

Note that those trade-offs might not be visible at first. Some technology choices might be good at first, but then prove to be unable to scale once your organizations grow or different stakeholders come into play. For example, a WordPress blog might be the right setup for a blog (with a team of editors and a single PHP developer) that runs on a cheap, WordPress-focused hosting solution. But if global availability becomes important, performance might be an issue. Or, the frontend might not scale with the needs of the newly attached UX team.

This is where *composable web architectures* come into play. The goal of composable web architectures is to develop decoupled applications, where each aspect is treated independently from others, making sure teams can make the right decisions on choosing the best technology for their case. In doing so, organizations are able to compose their applications from various technologies as needed, and always pick the best tool for the job.

Composable web architectures are created by splitting up tightly coupled connections through the introduction of JSON-based APIs. In the following sections of this chapter, we are going to dissect the monolith based on the four aspects. We will see how decoupling applications benefit each aspect, and how this can lead to highly composable architectures.

## Content Management with Headless and Decentralized Systems

Traditional CMSs have evolved into platforms, handling actual *content management* (including creation, editing, and publishing), as well as integration with additional services through plug-ins (*applications*) and rendering of user-facing *frontend* code.

**Figure 1-1** shows a traditional, monolithic approach to web applications. It presents a single CMS handling all tasks, including the application layer, the frontend using a rendering system, and the administration of the content:

- The circles *Administration* and *Frontend* show entry points to the main content area.
- The boxes *Content creation/modification/publishing* and *Application* indicate processes that deal with reading or writing content from the main storage. So does the *Rendering* box.
- The cylinders show storage capabilities: the main content database, as well as an additional database for extra application data. In some systems, all storage capabilities reside in the same database.

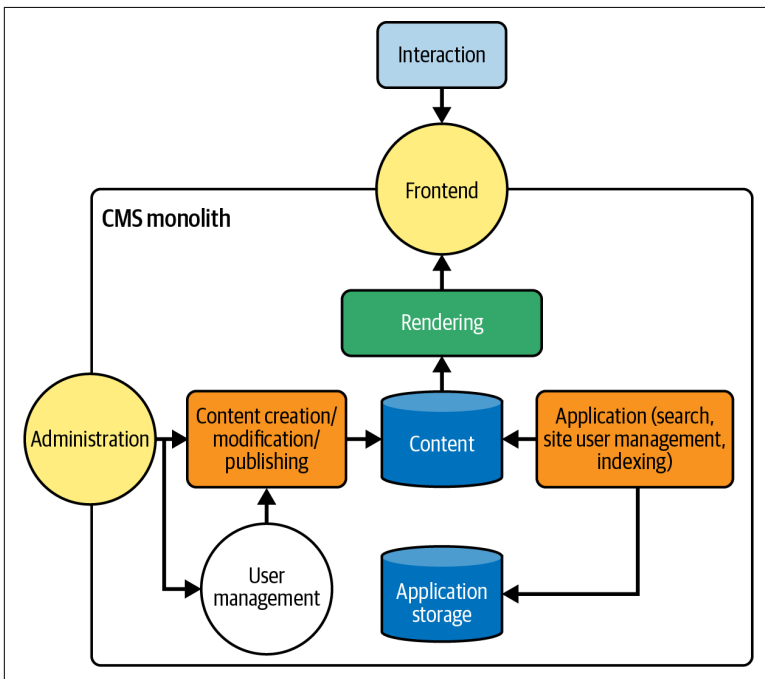


Figure 1-1. The monolith

As you can see, the main content storage is heavily connected to many processes. The obvious connections are to reading and writing through actual content management, as well as the rendering process toward the frontend. This is what a traditional CMS is doing: administration on one side, presentation on the other. The strong connection between all those points dictates all decisions for all other aspects. Furthermore, the data is structured to benefit content management, not necessarily the other processes attached to it.

We will now take a series of steps that will decouple the traditional monolith into a set of smaller, reusable, single-purpose services that can be composed in the frontend. The first step in decoupling applications is to detach the content management part from the rest of the application. **Figure 1-2** shows the CMS operating as its own entity, with enclosed data storage, and tailored processes for content editing, creation, and publishing. Instead of every process going directly to the content storage, the application backend talks via APIs to the headless CMS.

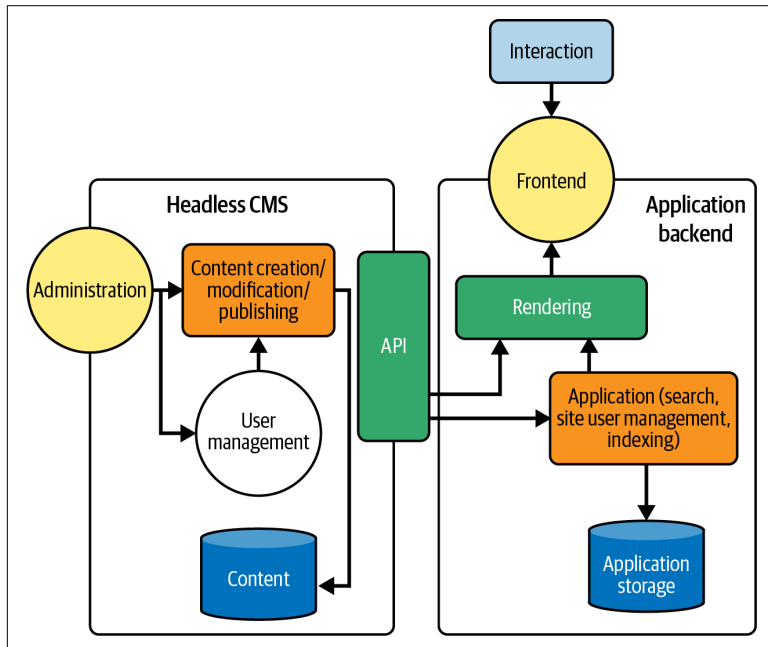


Figure 1-2. Decoupling the CMS part

To connect the application backend with the CMS, the CMS exposes a set of APIs, usually in the form of JSON-based HTTP endpoints. We call CMSs that use JSON APIs as their main output *headless content management systems*.

In a headless CMS architecture, the CMS is separated from the frontend website or application. The CMS only provides an API for accessing the content, while the website or application retrieves content from the API and renders it dynamically. You could say that the JSON API becomes the frontend of the CMS, nothing more.

For the main stakeholders of a CMS, the content editors, there are usually little to no change in how they work or maintain content. Even features like **visual editing**, which were a key argument for sticking to monolith CMS solutions, are now well supported in headless CMS, and arguably integrate even better.

Deciding on a headless CMS influences all other aspects. In the *frontend*, by separating the content from the presentation layer, a headless CMS allows developers more freedom of choice in picking their UI technology. This can be a JavaScript client-side framework like React, or a PHP-based application framework like Laravel, or something entirely different. UI rewrites become less of a hassle, and creating new elements on a website is entirely focused on the frontend. UI is also not only restricted to the web. The same APIs can be consumed to display content in a native app on iOS or Android.

The same benefits apply on the *application* side. A search service or product catalog generation can rely on the same APIs and can consume content without ever knowing where it comes from. This allows organizations to, for example, search services without switching the CMS underneath, or use the same search service to index multiple content sources.

It gets really interesting when we look at the *hosting* and operations aspect of introducing a headless CMS. Where a monolith has many points of access that all need different visibility, including a public-facing HTML output, the headless CMS only has two: the administration interface and the API endpoints. Access to the API can be given with fine-grained control, and the administration interface can be fully occluded to internal systems. This allows deployment of a headless CMS to be much more isolated; it can even run in an entirely different region or cloud provider altogether. With the introduction of a good cache layer on the API end of things, the headless CMS even becomes a satellite of the production system. The availability of the CMS does not dictate the availability of your website anymore.

When taking the API as the main provider for content of any kind, we gain the ability to decentralize services from the main application. In the context of composable web architectures, headless and decentralized systems provide the main point for composition.

Through their flexible APIs, we as website and web application providers can plug in functionality as needed.

## Decoupling the Application Layer Through Single-Purpose Services

Next, let's decouple the *application layer*. The application layer consists of everything beyond the display of managed content, services that are not necessarily part of the monolithic content platform. Any additional data source, data aggregation, user state, or just simple extra functionality belongs to the application layer. In a system like WordPress, this is the plug-in ecosystem, where you can enhance the core functionality.

Figure 1-3 shows how we can decouple the application layer from the frontend layer. The frontend now talks to the APIs from the CMS as well as the single-purpose services.

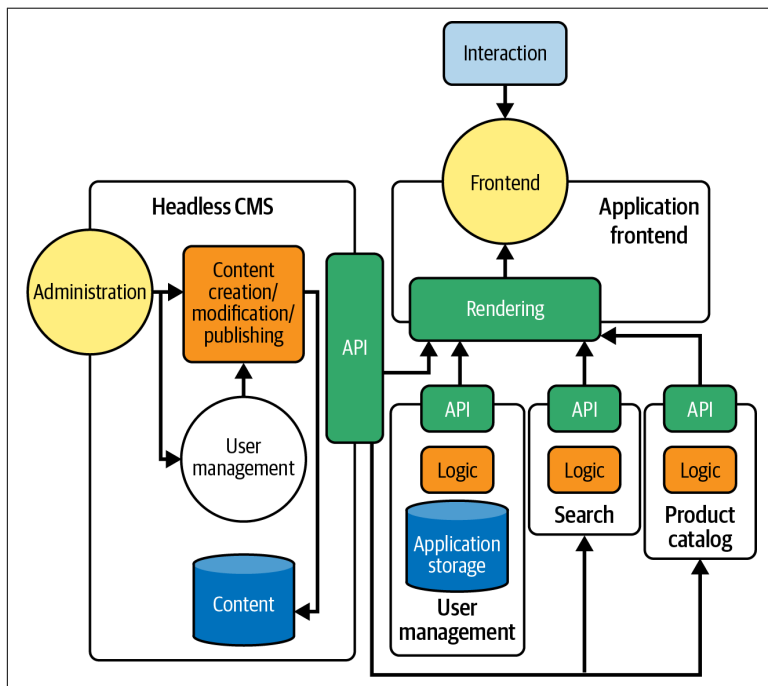


Figure 1-3. A decoupled application layer

Instead of deploying a single-purpose application backend, the rendering and interaction of content becomes its own entity, talking to APIs provided by *single-purpose services*. Single-purpose services are designed to perform a single function or task within an application or system. The goal of single-purpose services is that they are self-contained, work in isolation, and have little to no dependency on other services.

The architectural properties of single-purpose services are typically designed to maximize efficiency and performance for their specific function or task. Some key architectural properties of single-purpose services include:

#### *Modularity*

Each service performs a specific function or task. This makes it easier to maintain and update the service, as changes to one module do not affect other modules.

#### *Scalability*

Single-purpose services can handle increases in demand without experiencing performance issues. The service is distributed across multiple servers or instances, allowing it to handle a high volume of requests.

#### *Resilience*

Single-purpose services can recover from failures or errors without affecting the overall system. Redundant servers or instances can take over in the event of a failure.

#### *Lightweight*

Single-purpose services have minimal dependencies and a small footprint. This allows them to run efficiently and quickly, without consuming excessive resources or slowing down other services.

#### *API-based*

Single-purpose services have a well-defined interface for accessing the service. This allows for easy integration with other services.



**NOTE**

These architectural properties compare with the CAP theorem. CAP stands for *consistency*, *availability*, and *partition tolerance*. The theorem says that you can only pick two of the three CAP guarantees for distributed data access. Microsoft has an [excellent article](#) on CAP in the context of microservices.

Designing the *application* aspect as a group of single-purpose services allows the main stakeholder of that aspect—the developers—to work in their programming language of choice with the frameworks and libraries that they need. They can leverage all their knowledge and quickly create new features. They can also mix and match technologies based on their needs, and become fully independent of the technology choices of others: the microservice dream.

*Frontend* developers are not affected too much, as they consume “just” another API. Also, *content* editors don’t really see the impact of the change, which is just another indicator that those things maybe don’t belong together.

It has a huge impact on the *hosting* and operations part. What becomes visible when looking at [Figure 1-3](#) is that not all single-purpose services require a connection to the CMS at all. This further reduces dependencies and thus increases resilience.

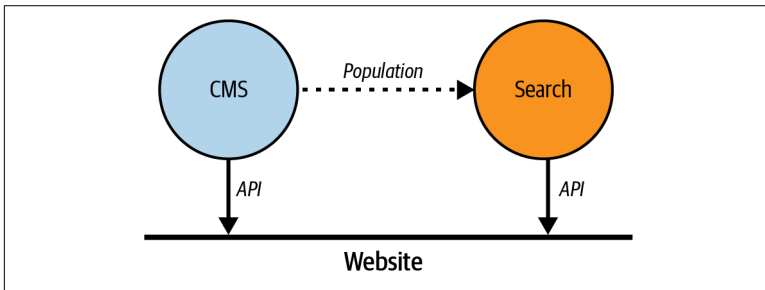


Be aware that when decoupling applications, there’s room for fragmentation and knowledge isolation. More APIs also mean more contracts to take care of.

Depending on a single-purpose service’s functionality, it can also be implemented as serverless functions. Take a contact form for example: retrieving the contact payload and putting it into a database/sending an email can be handled with a few lines of backend code. Function invocations can be paid per use, for the trade-off of higher costs per invocation and potentially longer startup times. For a contact form, which does not expect as much traffic as the rest of your application, this can reduce operation costs significantly.

Single-purpose services can be attached through third-party vendors. For example, both content management and search are traditionally seen as a unit, with the content stored in a CMS being scanned through the search indexer, but with single-purpose

services they can be split up. A SaaS CMS solution can store and provide content through APIs. Changes in content send events to a search SaaS solution; this populates the search index, which is then consumed by an API on the website itself, as seen in [Figure 1-4](#).



*Figure 1-4. A headless content management system being consumed by the website through its APIs*

The population of the index can happen independently from content consumption or search on the website. If the connection between the CMS and search is broken or defective, the website's connection can still function.

## Frontend Flexibility: Pull Versus Push Deployment

The last remaining piece of our original monolithic architecture is the *frontend*. This is the place where content is assembled, and HTML is finally delivered. With the frontend rendering completely decoupled from the rest, we suddenly gain flexibility in *how* we render content, again choosing the best tools for the job.

When a system delivers content to a client, it needs to go through a few steps and components:

### *Routing*

Content is accessed through URLs. A *router* parses query parameters and maps URLs to an internal representation. This will be used to fetch the right content and resources to produce the final result.

## Rendering

The *render* component of a web application takes the content associated with a URL and produces a machine-readable format. This can be HTML (traditional rendering), but also JSON (for the use of client-side frameworks), XML, or any other data exchange format. The goal is to get to a representation that can be consumed by a browser or by the application running inside a browser. Templating or client-side frameworks help developers reuse rendering components.

## Content retrieval

This step includes the retrieval of the actual content that will be rendered and subsequently delivered. This can include reading JSON data from disk or querying a database. We call the associated component *data storage*. Whatever the interface is, we get our data from there.

Those three components are inevitable when delivering web experiences. What's different is how they are arranged to produce results. Traditionally, web applications followed a so-called *pull architecture*, as shown in [Figure 1-5](#). The critical path of a client's request goes through the entire stack down to a query fetching the necessary results from a database. Every line in [Figure 1-5](#) has failure potential, causing the entire request to fail. Caches and resilience layers are mandatory.

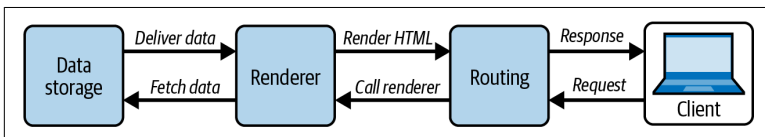
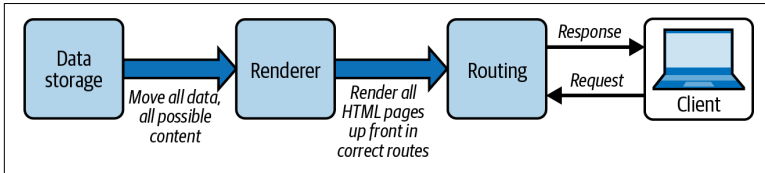


Figure 1-5. The traditional pull architecture

Pull architectures create fresh results for every request by going through every component. The benefits include contents that are never stale, the ability to include state, and the option for dynamic content per request. On the other hand, pull architectures are prone to performance problems due to the responsibilities of each request, and every connection between components includes the potential for a failed request, such as programming errors, networking errors, or infrastructure problems. Pull architectures need compute units and everything compute units imply.

In recent years, *push architectures*, such as Jamstack, have become more prominent and popular, as shown in [Figure 1-6](#). Instead of going through the entire stack with every request, push architectures map the entire available content up front, render the respective representations in static HTML files, and deploy the rendered results on static file storage. The critical path is reduced to a request that fetches HTML files from said storage.



*Figure 1-6. Push architectures such as Jamstack allow all content to be prerendered and deployed as static HTML files to simple file storage*

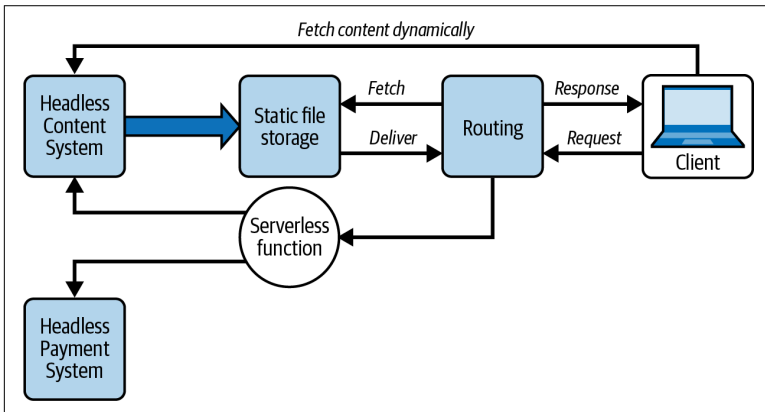
The routing unit’s only task is to map URLs to the right content, reducing the critical path to a minimum. Push architectures are easily deployed, cost-effective, and stable. The content is already finished, and there are no issues with problematic queries or programming errors. Push architectures need to know all eventualities up front, which means they lack significant support for dynamic content. Approaches like Jamstack<sup>2</sup> move dynamic, asynchronous content to the client side, which is a reasonable solution, but comes with different trade-offs, like a flash of unstyled content in client-side rendering, increased programming effort on the client side, unpredictable execution environments (browsers), and problems when dealing with sensitive content.

**NOTE**

One key aspect of hosting websites on static storage and distributing them globally via a content delivery network (CDN) is that this approach is really cheap in comparison to other servers. Update costs are moved to a build step, and global distribution happens in batches.

<sup>2</sup> See *Modern Web Development on the JAMstack* by Mathias Bilman and Phil Hawksworth (O’Reilly, 2019).

However, because composable web architectures rely on headless systems, they allow you to pick the right approach for each use case. With push architectures you prerender the majority of static content up front, and pull architectures use additional data on the client side or render pages with dynamic content through serverless functions. **Figure 1-7** illustrates how composable web architectures can consume APIs through various means. Headless systems can provide content for pregenerated pages, as well as dynamically created site and client-side rendered content.



*Figure 1-7. Composable web architectures take the best of all approaches*

A composable web architecture introduces different layers of resilience:

- Static content always works.
- Additional dynamic content fetched on the client side can be brittle and is allowed to fail.
- Content rendered via serverless functions can be dynamic, can include sensitive information, and can produce proper errors if things go wrong.

The underlying systems stay the same, stay self-contained, and follow a single purpose. Finally, we have arrived at the point where our application has become truly composable. Depending on our choice of technology or vendor, we are able to choose the best way possible for getting data to the frontend.

Figure 1-8 shows the final transformation. Here, we split up routing from the actual rendering and content retrieval, creating more flexibility in how we deliver content to our clients.

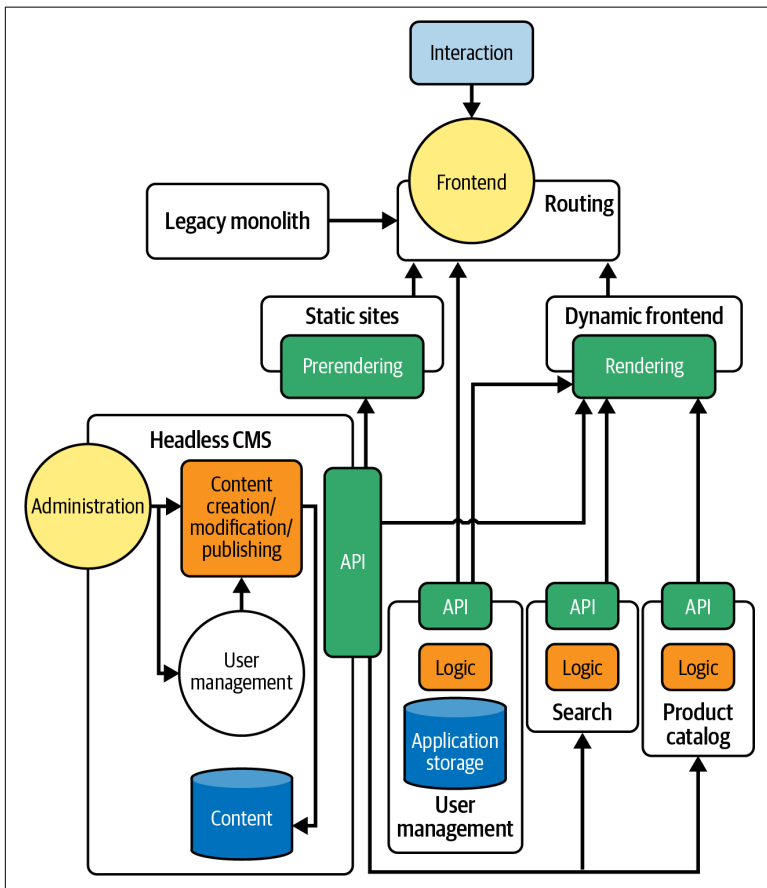


Figure 1-8. The last decoupling step

This allows us to mix and match different rendering approaches best suited to our needs. This can be static generation of web pages with little to no interactivity, server-rendered pages with dynamic data, or even legacy monoliths that are still attached to the new system. Also note that in Figure 1-8 we see that the routing layer also has direct access to the single-purpose services as well, which means that some of the content updates can happen entirely client side.

Since composable web architectures deal with flexibility of technology choices and the possibility to change parts of an application quickly, choosing a UI framework can be quite an commitment. With different deployment models (server-side rendered, client-side rendered, statically generated), we create options: for example, during the migration to a new design and maybe frontend technology, the old website can be statically generated and routed as a fallback, while the new content and design takes shape and replaces the old one.

## Conclusion

Every step toward composable web architectures shown in this chapter gears toward one common goal: independence and composition. Instead of having everything in one rigid, hard-to-maintain monolith, developers can pick their favorite technology and develop new features entirely independent from all other pieces of their system. This allows developers to ship new features fast, even dropping the backend implementation in favor of a new third-party service that might just do what they need. You compose your website from different bits and pieces, creating a single representation.

This also allows organizations to move away from certain features quickly. If a tool or service doesn't seem to cut it, drop it in favor of a new or better one. Technology lock-in becomes a thing of the past; so does the big bang rewrite. The frontend layer just becomes another piece of the puzzle, and UI rewrites don't sacrifice the architecture underneath.

In the following chapter we look at the implementation of composable web architectures in detail, and deal with the operational impact of such architectures.





# Developing and Operating Decoupled Applications

Composable web architectures assemble modular services into a full application. Integrating new services into a web user interface is easy, but how do we make sure that the user experience is coherent, robust, and consistent?

This puts a lot of responsibility on the frontend developers, who are now in control of assembling a web application from various bits and pieces. They need to make decisions on how to talk to APIs, what rules to implement to show state in the user interface, and how to deal with erroneous responses or worse: unavailable services.

In this chapter, we look at different aspects of developing and operating decoupled applications. We see development technologies that support composition and decrease fragmentation, and answer the most burning questions on operational costs.

## Technology Lock-In, Technology Choices, and Rewrites

Figure 2-1 shows a sample application built on a composable web architecture. This example is derived from real-world applications such as the [Dynatrace web portal](#). The website features five major outlets that are based on different technologies, with various teams working on them and different stakeholders deciding on content, features, and UX.

It includes the main website, using a custom rendering service connected to a headless CMS, a traditional blog monolith, a documentation website that is generated with a static site generator, and the careers portal, which is a Next.js app getting data from a recruitment portal. Next to it is a global search service that gets data from various sources.

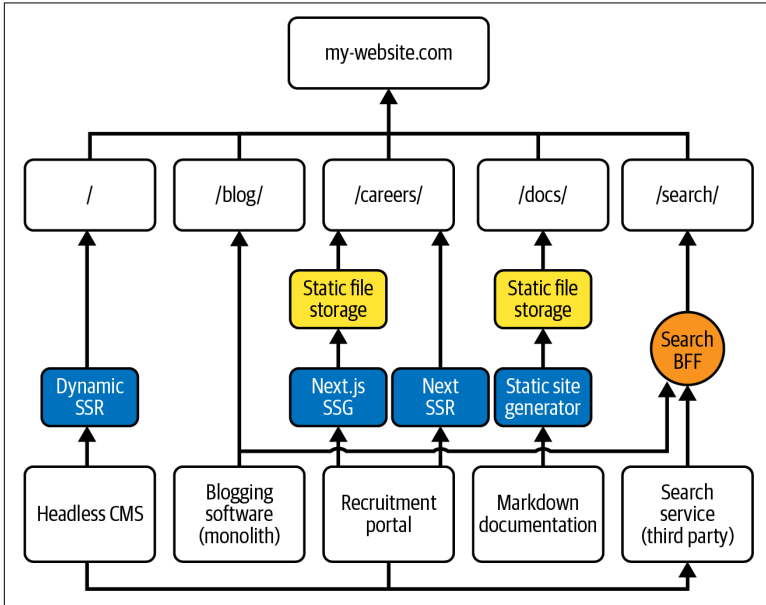


Figure 2-1. A web portal for a company website

The different outlets are:

1. The main website served under the root /. This outlet contains the main product and company information, and is highly optimized SEO content created by the online marketing team. They use dynamic server-side rendering written in PHP, with content coming as JSON via a headless CMS.
2. The company's blog is served under /blog/ and is based on monolithic blogging software that has been with the company for a decade. The content is maintained by the e-marketing team but contributed by the entire company. Allowing every person in the company to write articles and keeping every contributor's author biography are key features of this installation.

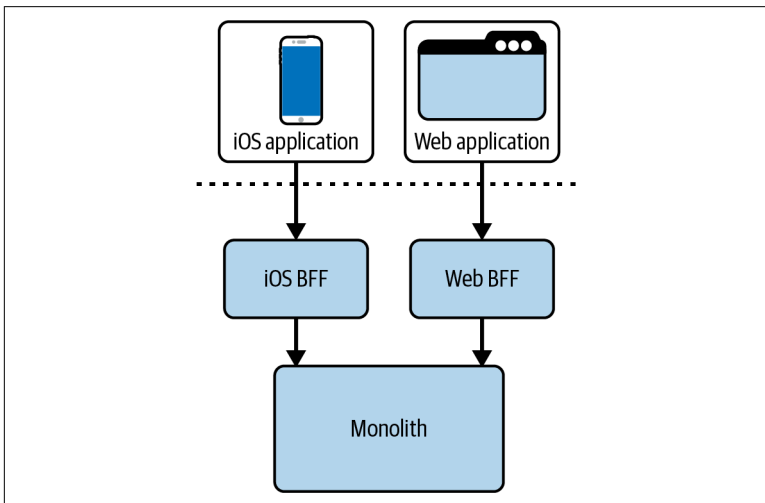
3. The job portal under `/careers/` is a Node.js-based Next.js application. `Next.js` is a React framework that allows for dynamic websites that are rendered on the server or on the client, or statically prerendered based on the page's demands. Next.js is connected to a recruitment portal that serves available job postings via a JSON API, and allows people to apply through a different set of APIs, including the upload of CVs as PDFs. Next.js is configured such that the available job postings are generated statically, while the overview (which features a dynamic filtering process) and application process (which needs to give feedback based on the state of the application) are rendered on the server with client-side interactions.
4. The product's documentation is written by technical writers within the the company's engineering department. They prefer to write their documentation via Markdown, and render a set of interlinked static web pages. It is served under `/docs/`.
5. The last piece of the website is the search, which is provided by a third-party search service. A background process written by the operations team sends JSON input from the main outlet, the job portal, and the documentation to the search service for indexing on a daily basis. The search service also features JSON APIs to query for results. Since the blogging monolith does not have a JSON output that can be used to index the blog via the search service, the developers created a serverless function to combine both the search service's JSON output as well as the search results from the blog in the form of a *Backend for Frontend* pattern. The search backend for frontend is reachable under `/search/` and either serves JSON output for dynamic results over a search field across all websites, or renders HTML if accessed directly.

Each technology has been carefully chosen by development teams to produce the best results for their use case and their users. They use the right tool for the job. Also, every technology can be replaced with others should requirements change, without influencing the adjacent outlets, effectively avoiding technology lock-in. Also, elements on the website, like a search field, can be used across the entire web application, independent from the actual tech stack underneath.

To develop a decoupled application like this without risking fragmentation requires a lot of attention to detail. Let's dig in.

## Rethinking Backend for Frontend

We see that the search service has been made robust and independent of technologies through the integration of a Backend for Frontend pattern. The term *Backend for Frontend* (BFF) originally describes a pattern where organizations create tailored APIs for various frontends, such as a mobile application, administration UI, or the website/web application itself. It originated in the early 2010s at [Soundcloud](#), and has been prominently described by [Sam Newman](#) in his series on microservices as well as [Lukasz Plotnicki](#) for Thoughtworks. [Figure 2-2](#) shows the original design for a BFF.



*Figure 2-2. The traditional use case and approach for BFF, creating tailored API layers for different user interfaces*

While BFFs have been traditionally meant to create tailored APIs for various outputs, they can also help in the migration process from a monolith to microservices. BFFs can be envisioned as a layer and contract for UI developers to consume stable APIs and to have consistent and controlled access to services underneath. With BFFs knowing how data should be consumed, they can also include tailored caching to minimize database access and unnecessary network traffic.

Figure 2-3 shows how a BFF consumes APIs from the old monolith as well as from newer services that have been attached to the API layer. Instead of every call going to the monolith, the BFF integrates different services into a single API layer.

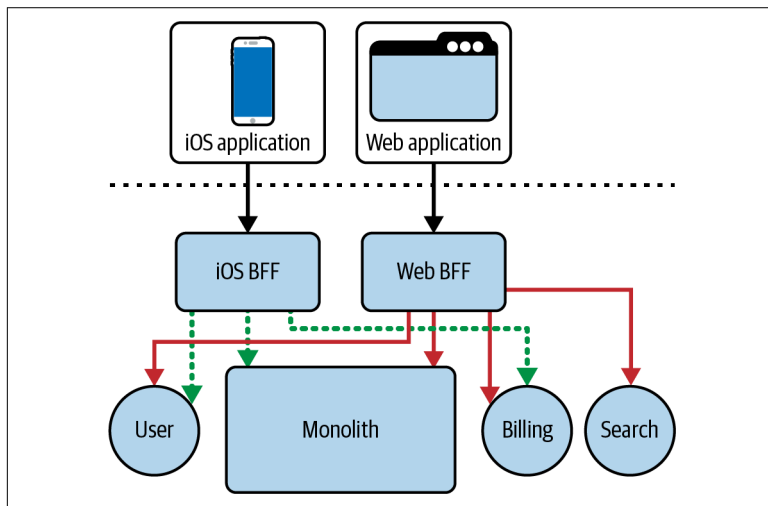


Figure 2-3. The BFF as a migration step to microservices

In the context of composable web architectures, BFFs reduce and tailor API surface for the single-purpose services as shown in Figure 2-4. Single-purpose services can originate from various vendors and thus be decentralized. They might work with different authentication methods or might expose too many endpoints for the purpose of the application. A BFF can tailor responses and offer normalized error handling, caching, and failovers for both first- and third-party services.

When considering resilience, using a BFF can involve trade-offs. A well-designed BFF can boost the resilience and performance of attached single-purpose services by delivering cached responses, which reduces traffic and possibly costs when using third-party SaaS products. In addition to caching responses, BFFs can also ensure that the frontend receives a normalized error response with the corresponding status codes in the case of failing requests. This allows developers to rely on global error handling and error boundaries to detect when something goes wrong, or to temporarily deactivate a feature such as search if backend calls fail.

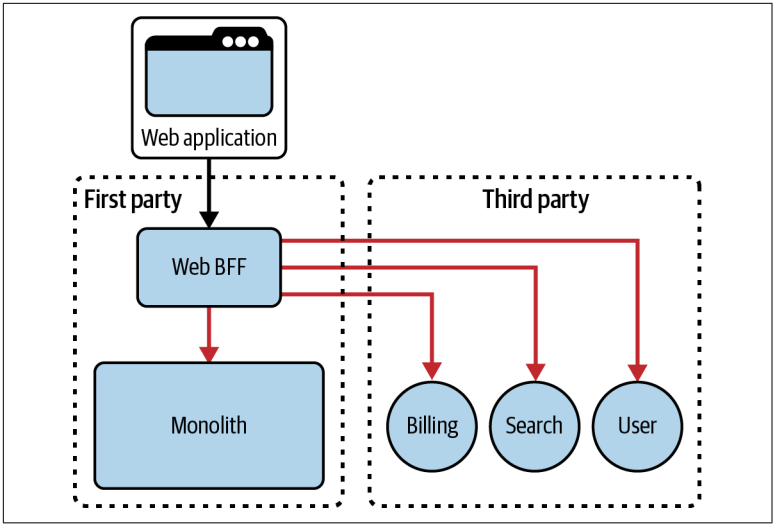



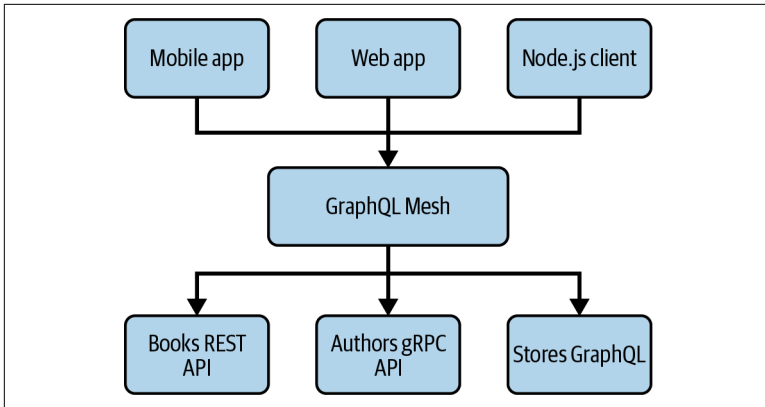
Figure 2-4. The BFF as an integration step for composable web architectures

However, BFFs can become a single point of failure, especially when deployed as a separate service. To mitigate this problem, BFFs can be implemented as small, self-contained serverless functions using services like [AWS Lambda](#) or [Cloudflare Workers](#). Splitting the responsibility into a set of independent functions enables BFFs to fail more gracefully by handling only single functions or even single invocations, rather than overwhelming the entire backend service with requests and potentially bringing down the entire architecture.

Another approach to tailoring responses is GraphQL. [GraphQL](#) is a query language that enables tailored responses in user interfaces, with the trade-off of potentially longer response times compared to regular API calls.

 GraphQL promises tailored requests and responses for every website, but has received some criticism in recent years for the extra effort organizations need to exert to ensure performance and secure access, avoid DDOS attacks, and produce reliable results. Make sure GraphQL is the right solution for your problem and organization.

A pattern that has emerged with GraphQL APIs is the use of a composition layer called **GraphQL Mesh**, which allows developers to plug in different first- and third-party GraphQL and REST APIs, making them available with a single query backend, as shown in **Figure 2-5**.



*Figure 2-5. GraphQL Mesh allows the connection of different sources with different protocols and provides a unified layer that is accessible via GraphQL*

BFFs and mesh layers also help with avoiding vendor lock-in. They serve as the mediator between connected services and your application. Should you decide to change one service for the other, you only need to adapt the BFF layer.



A mesh layer alone won't help you with vendor lock-in. There are mesh layer offerings as a service, which very much tie you to a specific vendor for your GraphQL data.

## Serverless as a Rendering Layer

Decoupling applications involves separating the frontend from the backend. This separation causes a clear cut between rendering HTML, which is done by many clients, and serving data, done by fewer servers. Decoupling can go as far as rendering entirely in the browser, with no server-side rendering step at all, which moves a lot of responsibility to the frontend development teams.

But this shift in responsibility is what makes this separation desirable. Frontend development has an entirely different operational complexity than backend development. Where frontend developers need to care about browser inconsistencies, user interactions, and UI state, backend developers care a lot about service availability, CPU usage, memory limits, and transaction times. Rendering proper HTML is one concern that a backend developer happily gets off their list, especially with the frontend becoming so complex in recent years.

However, assembling HTML on the client is not a silver bullet, as it can have severe impact on aspects such as performance. The [Netlify blog](#) has listed seven different rendering approaches, all with their benefits and caveats. So there is a need for rendering on the server side. But we still want to keep the benefits of shifting more responsibilities to frontend developers.

A solution for this problem that has been heavily adopted in recent years is serverless. Serverless functions allow developers to execute small, stateless tasks that can be wired to an HTTP interface, taking care of request/response patterns. Those tasks include the handling of contact forms, payments, search, etc. They can be written fully in JavaScript, which frontend developers are familiar with. Also, PHP scripts can be executed well in serverless environments, as PHP shares some of the traits of serverless.

**NOTE**

While serverless rendering is associated with long startup and execution times and potentially high costs, modern serverless platforms like Cloudflare Workers or Deno Deploy—commonly referred to as edge functions—execute rendering steps in mere milliseconds and stay affordable. In 2023, nobody needs to wait for serverless.

From an operations point of view, serverless functions also get rid of a lot of the concerns you usually associate with traditional application servers. Most serverless offerings treat each invocation in isolation, meaning that a request gets a very small dimensioned VM instance that works on a particular task and no other. Only if the task is done is the same instance allowed to work on another task. Concurrent requests are solved by spinning up multiple instances, as seen in [Figure 2-6](#). Concurrency happens through multiple



instances, effectively isolating each process from noisy neighbors or potential panic.

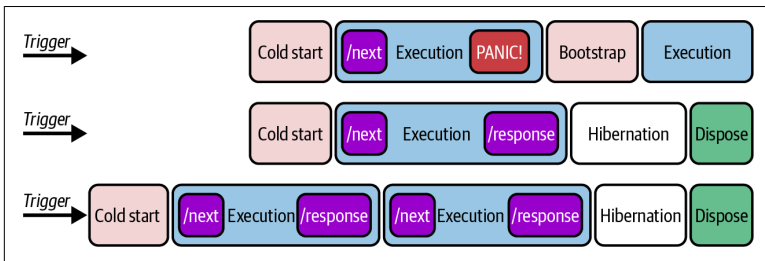


Figure 2-6. Serverless executions are run in isolation

The big benefit from this setup is that if an error occurs, and the function execution breaks the process, all other processes still run and produce results. With this, software becomes much more resilient. Serverless offerings allow for thousands of parallel instances per function, getting rid of most scaling issues. Last, but not least, serverless offerings only charge for what you use, which can have a tremendous benefit on your operational costs.

With serverless as a rendering layer and existing frameworks, frontend developers can write minimal backend functionality in programming languages they are familiar with, without dealing with concerns that usually come along when operating servers.

It was Chris Coyier from CSS Tricks who realized that with the availability of serverless, frontend developers have become **full stack developers** now, taking care (at least in part) of server-side tasks.

## Frontend Composability

When we integrate services like search or elements like navigation and footers to a decoupled application, we need to consider how we make this element available on all pages, no matter how they have been created. There are various ways to compose user interfaces, and as with any software architecture, one size does not fit all organizations. The appropriate way to assemble UI and thus compose largely depends on how teams are structured, how responsibilities are distributed, and what technology choices have been made.

Frontend composition has a strong relationship with micro-frontends, an architectural approach that involves breaking down

the frontend of an application into smaller, independent frontends, each with its own UI components and probably routing logic. The goal is to deploy frontends independently, allowing teams to release features end-to-end, and also to include several technologies from different teams on a single page so they can decide on their stack independently. Luca Mezzalana describes several approaches in his book *Building Micro-Frontends* (O'Reilly, 2021), and Michael Geers explains the concept on the eponymous website <https://micro-frontends.org>. Figure 2-7 shows how a UI is composed through micro-frontends. In the figure, the product display, the checkout, and the related products screen are all independent of each other. All features are produced by different teams.

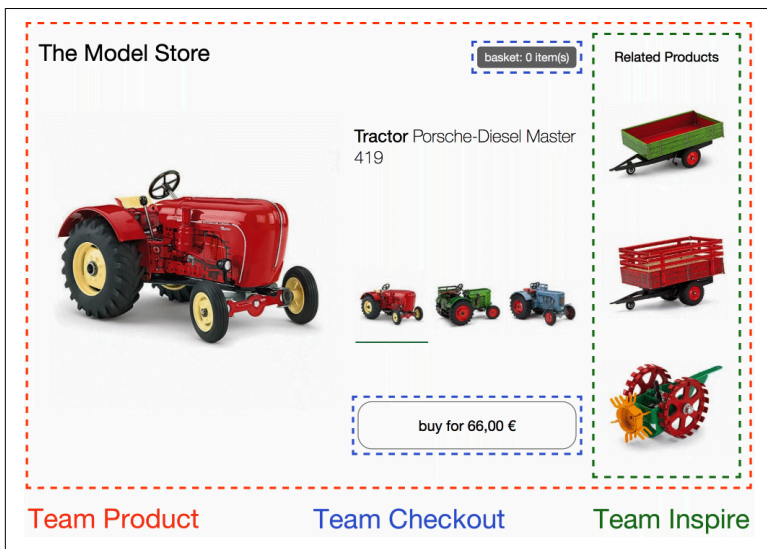


Figure 2-7. A micro-frontend example from Michael Geers's [micro-frontends.org](https://micro-frontends.org) website

Micro-frontends mostly deal with organizational setups and the architecture around them. Not all the principles and patterns described in this architecture are necessary for the composition of user interfaces.

The overall goal of frontend composability is to assemble isolated, self-contained elements of user interfaces that have a strong connection to the underlying service. For example, a search widget should be a drop-in element that takes care of everything related to search, including communicating with the right APIs and producing the

correct error messages. Ideally, this element should be independent of other changes in the user interface and able to be reused in other applications.

A great way to achieve composability is through component-based UI frameworks. In recent years, all popular UI frameworks such as React, Angular, and Vue have been based on a component model that favors composition. This allows developers to create reusable UI components that can be combined to form larger UI elements. Depending on the team's setup, it may decide to go for a single UI framework that allows for easy composition and clear component boundaries. UI frameworks are no longer exclusive to client-side rendering. Tools like [Next.js](#) demonstrate that the same UI code can be rendered statically, on demand on the server side, or dynamically on the client side.

*Design systems* and *component libraries* can build the foundation of frontend composition. Design systems describe look and feel, functional and perceptual patterns. They provide designers with a framework to define and create new UI components, and allow developers to derive optimized and reusable UI components. Alla Kohlmatova describes the goals of design systems in her book *Design Systems*. A component library can be seen as a concrete implementation of a design system either through basic web technologies—HTML, CSS, and JavaScript—or based on a UI framework, like React.

Picking React as an implementation basis for a component library is a reasonable choice: React is a component-based framework that favors composition, and it has patterns that allow an implementation that separates functionality from design. Developers can implement elements that work in different contexts (server-side rendered, statically generated, dynamic on the client side), and there's a really good ecosystem supporting developers with extra components that reduce effort for many complex scenarios.

But the question isn't how well React works. The more important question is how easily can React be replaced? If it comes to a rewrite, a change in frontend technologies, how much effort does it take to move to the new technology? Do we start from zero, or are there elements that can be carried over to the implementation?

An example is [IBM's Carbon Design System](#). For a company as big as IBM, teams inevitably have different technology choices for their applications. That's why the Carbon Design System is first and

foremost built upon reusable CSS classes that can then be used for different frameworks like Angular, Vue, React, Svelte, or even regular web components. If a new technology needs to be included, the basic styles and patterns still work. The HTML for those patterns has been defined, reviewed, and tested for accessibility. The “only” thing missing is a concrete implementation in said framework.

Using React-specific technologies for CSS like *styled components* would ultimately lead to lock-in. Styled components can't be reused in Vue, nor Angular, and most likely not in a technology that we don't know about yet. Basic CSS and HTML can be reused. The effort of moving to a new frontend system, or maybe even getting rid of frameworks entirely, is much lower if we can carry over as many elements of our design language and implementation as possible.

Having a simple CSS library with HTML examples also allows for the creation of static pages for sites that don't require modularity or a complex setup, as an example. They still have the same look and feel as the web application created on top of it.

All three elements, micro-frontends, UI frameworks, and design systems, can be combined in a single application. For example, the application performance monitoring company Dynatrace achieved frontend composability in their [AppEngine release](#) by allowing developers to build self-contained screens through a dedicated application framework and design system. They use React as a base technology for their components, providing reusable end-to-end elements with a design system called [Strato](#). Each “app” is encompassed by an application shell, which serves as a mediator between screens and provides globals like user information, URLs, etc., forming a micro-frontends architecture.

## Third-Party Services

If we look at the lower parts of [Figure 2-1](#) we see that a lot of the services used can be served and provided by third parties. The recruitment portal, search service, and headless CMS can be run by SaaS offerings, leaving only the blogging software as well as the build process for the documentation in the hands of the operations team.

Using SaaS offerings for your single-purpose services can come with a lot of advantages:

- There's a lower up-front cost, because SaaS is usually subscription based. Some of them offer free tiers to get your developers started and only require a professional or enterprise subscription later. Also, you don't run their servers, which allows you to bring down operating costs and maintenance.
- Not only can you neglect server maintenance, you also don't need to update the product itself. Not only do you get the latest and greatest features, but you also get security updates and bug fixes, which is arguably much more important to you and your organizations.
- Most SaaS offerings work with JSON-based APIs. In a world of decoupled applications, this is the most convenient and easiest way to integrate new features.
- SaaS offerings are designed to scale. They host multiple tenants on their solution already, and are prepared for high traffic and high load, operational aspects that you mostly don't want to deal with.

But SaaS offerings are no panacea. Your organization might have different requirements regarding privacy and data security than the service can offer. The service might not reach its SLAs but requires you to be always connected, and ultimately it may move at a different pace than your organization.

Using third-party SaaS offerings is a topic of trust and costs. Big organizations pay good money for enterprise deals, but are ultimately rewarded with high-availability SLAs and continuous development of the product itself. This investment can unburden operations teams tremendously, as they don't need to operate multiple clusters of differently purposed software, most of them potentially written in different programming languages and running on a variety of servers.

If buy-in to third-party services poses risk and uncertainty to you and your organization, remember a key feature of composable web architectures: minimizing technology lock-in. You can always remove an existing service for others, and while this requires some effort, it's significantly lower than changing entire systems.

If you go for SaaS offerings, evaluate vendors by asking the following questions:

- Does the SaaS offering allow you to export your data to a storage service like AWS S3? Maybe even a JSON dump of your entire content structure? This way you can still read a snapshot of your web application to keep your system going without needing a live connection to the third-party service.
- Do SaaS offerings build on open standards and conventions? For example, does your identity provider work with OAuth 2.0, so you only need to exchange APIs but not the authentication/authorization flow?
- How strong does the connection to the SaaS product need to be? Can you cache responses? Does your software fail if availability falters, or can you bypass unavailable APIs and still offer your users *some* content?
- How does the vendor introduce major updates to their platform? Do you have a migration period that allows you to move to the next version? Do they have some long-term support (LTS) model of their API? See whether their idea of LTS is aligned with your sprint cycles.
- How does the vendor respond to security incidents? Check their blog to see if there have been breaches and how they communicated them. Do they have an ISO 27001, PCI-DSS Level 1, or SOC 1/SSAE-16 certification? Do they regularly check against OWASP security threats? Do they dedicate a team to control their cloud infrastructure? Are they GDPR compliant? How do they handle sensitive data within the company? The vendor should have this information publicly available. And you can hold them accountable.

Using third-party services also helps reducing attack vectors and strengthens security. Your data is not in one place anymore, and at best it's also hidden behind a BFF. Mike Gaultieri at [Netlify](#) gives an overview on possible threats and risk and how they can be mitigated using composable web architectures.

## Conclusion

With composable web architectures, we can ensure that every part of our web project can be developed independently, enabling teams to get features shipped more quickly and with a lot less lead time. A truly decoupled application allows developers to create more resilient, failsafe, and sustainable experiences. If one piece of the system acts up, it doesn't take the rest of the web application with it. And as we've seen with the example in this chapter, composable web architectures are perfect for incremental adoption. Fostering and maintaining a good routing layer and strong URL structure, the old monolithic website can live alongside the newer web outlets for ages, and your users most likely won't notice. Chances are, if you move to a composable web architecture of decoupled applications, you will never have another big bang rewrite, but rather change single elements on your site or sub-branches of your information architecture piece by piece. A good, robust, and flexible design system helps with visual consistency, and can be another boost for your development and design teams alike.

But be aware that composable web architectures also require a different mindset. They embrace a diverse set of tools and technologies, even different programming languages and hosting environments. The goal is to get the right tool for the job. To ensure that this doesn't lead to endless technology fragmentation, set yourself some boundaries on what you can reasonably host and monitor. This also means including third-party software and SaaS solutions, as they can load a good deal of responsibility off your shoulders; they might ultimately be a lot cheaper than leaving the development and operation of said services within your organization. Watch out for the right SLAs and SLOs, and choose companies that communicate well, have standards in terms of security, privacy, and availability, and allow you to have a proper exit strategy. Technology buy-in is good, technology lock-in must be avoided.

Composable web architectures are the next step in the evolution of cloud-based microservices. They move a lot of responsibility to the frontend, with small backend functions that are easy to create and maintain. With a rich ecosystem of cloud hosting providers, single-purpose services, and easy-to-deploy backends, frontend developers have the power to get a full stack application shipped in no time.

## About the Author

---

**Stefan Baumgartner** is a developer and architect based in Austria. He is the author of *TypeScript in 50 Lessons* and *TypeScript Cookbook*, and runs a popular [TypeScript and technology blog](#). In his spare time, he organizes several meetups and conferences, like the Rust Linz meetup and the [European TypeScript conference](#). Stefan enjoys Italian food, Belgian beer, and British vinyl records. He is also an independent consultant and trainer for Rust and TypeScript at [oia.dev](#).