MICRO FRONTEND ARCHITECTURE

By Dr. Vishwanath Rao

When your front-end starts getting too big, you will want to split it in smaller parts and give each part to a different team to develop, but at the end, they all need to integrate somehow in the same front-end so the user can see it.

1. It is a concept of designing large web applications as a composition of small sub-applications.

2. The idea is very similar to micro services.

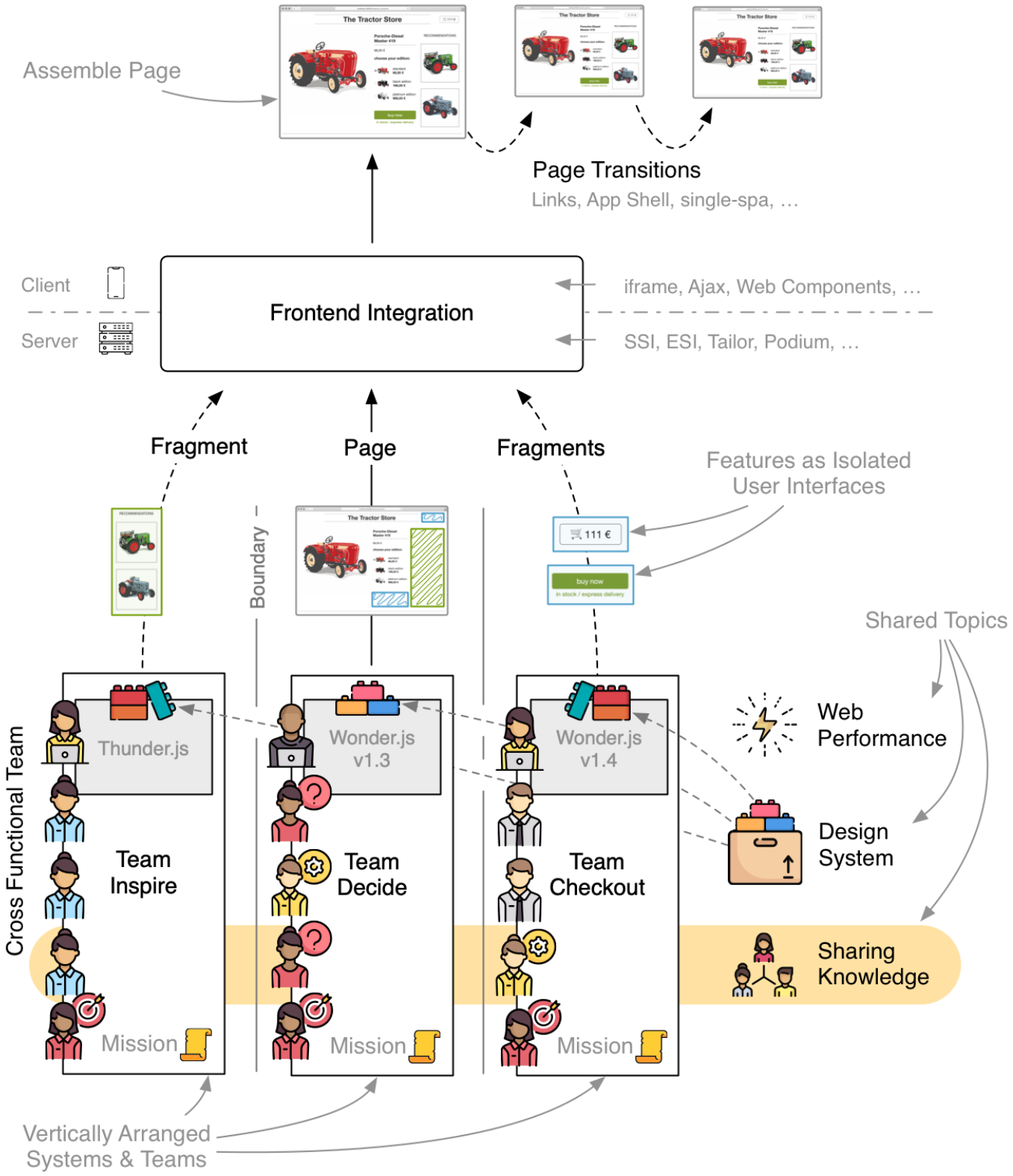
3. The difference is that micro services are independent backend services, but micro frontends, as the name suggests, are independent frontend components.

4. If we combine these two architectural patterns together, we can design self-contained applications.

5. It means that we can split large system vertically into separate parts, where in ideal world each part has its own frontend, backend services and database if needed.

6. It sounds like a perfect way to escape monolithical UI, in favour of independent and scalable components!

- **Faster feature development:** A team includes all skills to develop a feature. No coordination between separate frontend- and backend teams is required.
- **Easier frontend upgrades:** Each team owns its complete stack from frontend to database. Teams can decide to update or switch their frontend technology independently.
- **More customer focus:** Every team ships their features directly to the customer. No pure API teams or operation teams exist.


Assemble Page
Page Transitions
Links, App Shell, single-spa, …
Client
Server
Frontend Integration
iframe, Ajax, Web Components, …
SSI, ESI, Tailor, Podium, …
Fragment
Page
Fragments
Features as Isolated User Interfaces
Boundary
111 €
buy now
Shared Topics
Cross Functional Team
Thunder.js
Wonder.js v1.3
Wonder.js v1.4
Team Inspire
Team Decide
Team Checkout
Web Performance
Design System
Sharing Knowledge
Mission
Mission
Mission
Vertically Arranged Systems & Teams

Micro frontends are:

- smaller, more cohesive and maintainable codebases
- more scalable organisations with decoupled, autonomous teams
- the ability to upgrade, update, or even rewrite parts of the frontend in a more incremental fashion than was previously possible

Benefits

Incremental upgrades

Simple, decoupled codebases

Independent deployment

Autonomous teams

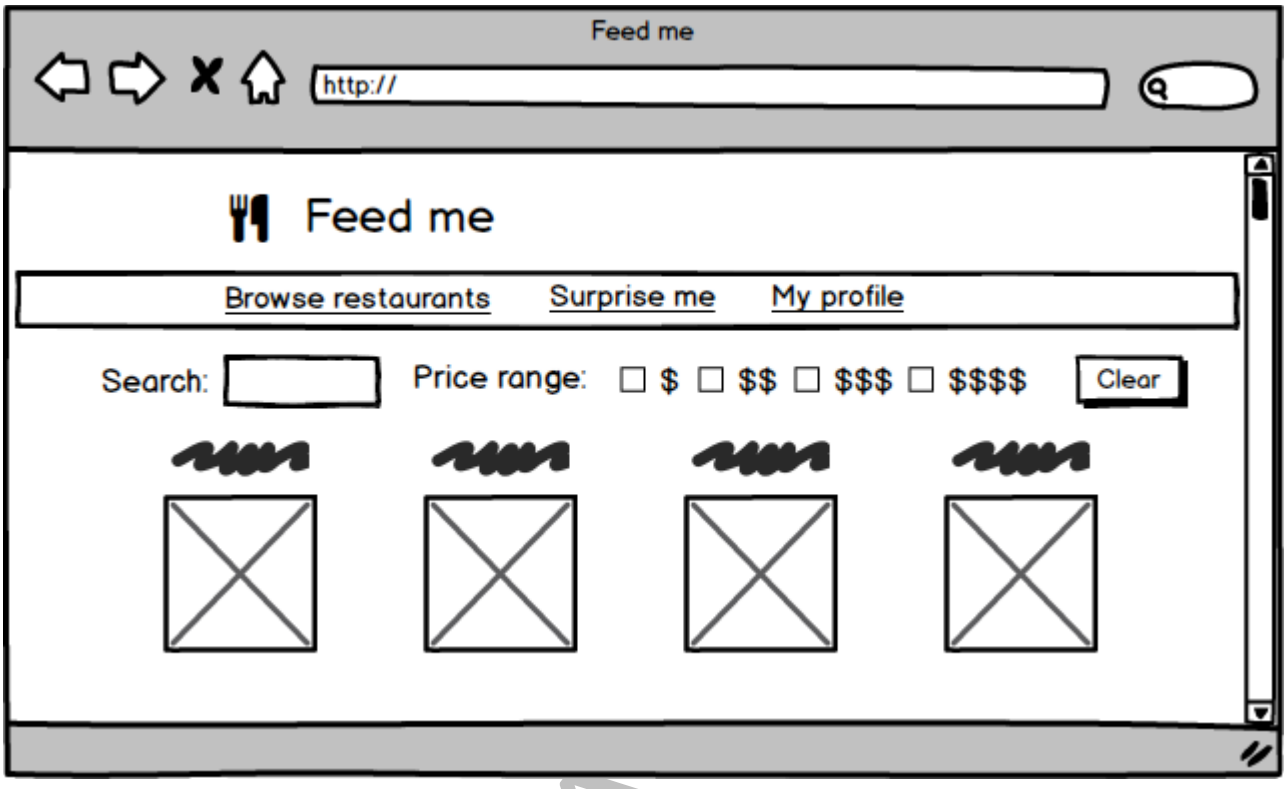
Micro frontends are all about slicing up big and scary things into smaller, more manageable pieces, and then being explicit about the dependencies between them.

Our technology choices, our codebases, our teams, and our release processes should all be able to operate and evolve independently of each other, without excessive coordination.

CASE STUDY TO ILLUSTRATE MICRO FRONTENDS

Imagine a website where customers can order food for delivery. On the surface it's a fairly simple concept, but there's a surprising amount of detail if you want to do it well:

- There should be a landing page where customers can browse and search for restaurants. The restaurants should be searchable and filterable by any number of attributes including price, cuisine, or what a customer has ordered previously
- Each restaurant needs its own page that shows its menu items, and allows a customer to choose what they want to eat, with discounts, meal deals, and special requests
- Customers should have a profile page where they can see their order history, track delivery, and customise their payment options

- 

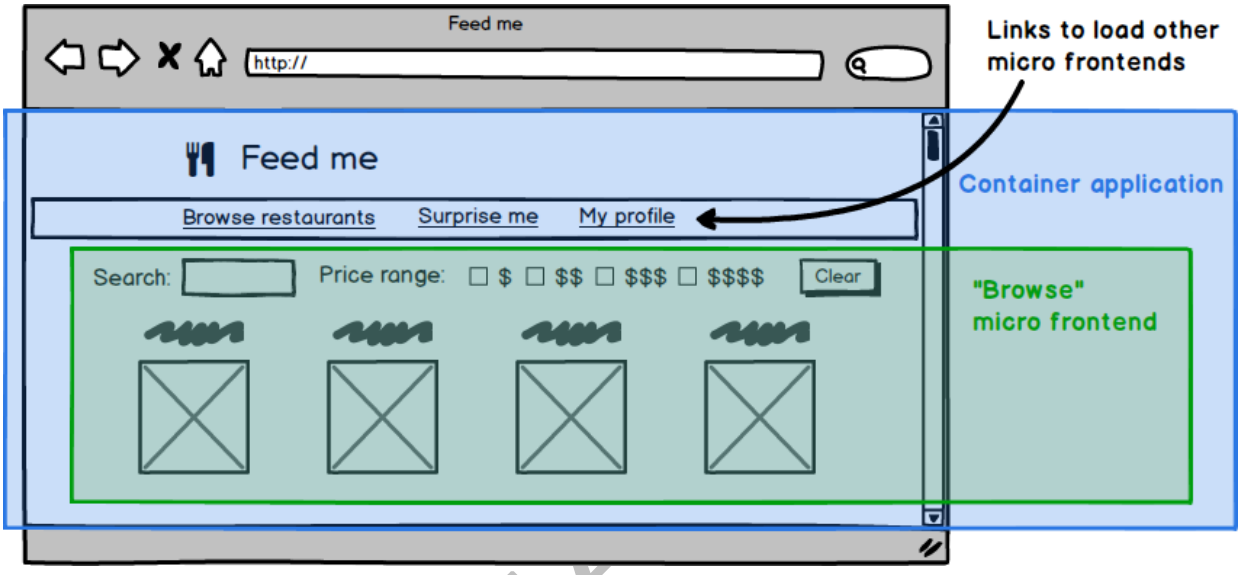
There is enough complexity in each page that we could easily justify a dedicated team for each one, and each of those teams should be able to work on their page independently of all the other teams. They should be able to develop, test, deploy, and maintain their code without worrying about conflicts or coordination with other teams. Our customers, however, should still see a single, seamless website.

Integration approaches

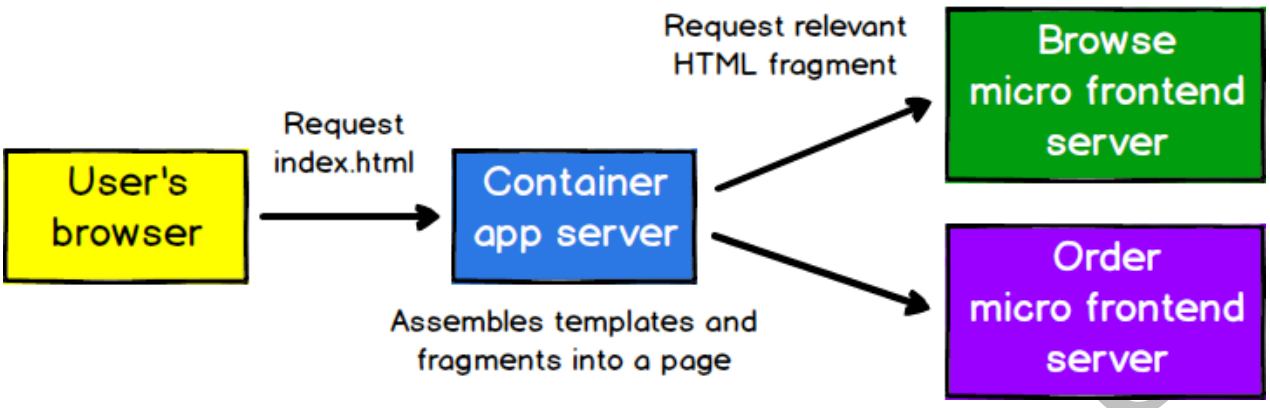
Given the fairly loose definition above, there are many approaches that could reasonably be called micro frontends. In this section we'll show some examples and discuss their tradeoffs. There is a fairly natural architecture that emerges across all of the approaches - generally there is a micro frontend for each page in the application, and there is a single **container application**, which:

- renders common page elements such as headers and footers
- addresses cross-cutting concerns like authentication and navigation

- brings the various micro frontends together onto the page, and tells each micro frontend when and where to render itself



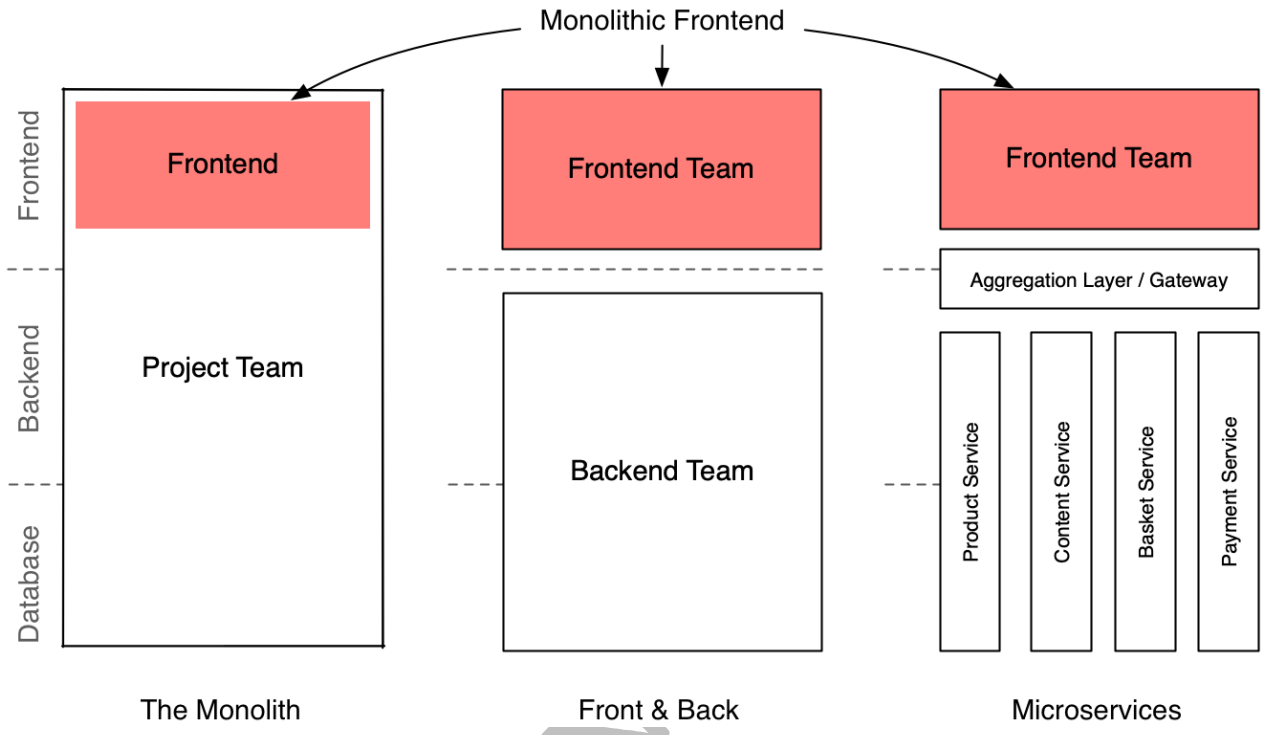
For even greater independence, there could be a separate server responsible for rendering and serving each micro frontend, with one server out the front that makes requests to the others. With careful caching of responses, this could be done without impacting latency.



SUMMARY

- Micro frontends is an architectural approach and not a specific technique
- Micro frontends removes the team barrier between frontend and backend developers by introducing cross functional teams.
- With the micro frontends approach the application gets divided into multiple vertical slices that span from database to user-interface.
- Each vertical system is smaller and more focused. It's thereby easier to understand, test and refactor than a monolith.
- Frontend technology is changing fast. Having an easy way to evolve your application is a valuable asset.
- It's a good pattern to set the team boundaries along the user journey and customer needs.
- Team should have a clear mission like: "Help the customer to find the product she is looking for."
- A team can own a complete page or deliver a piece of functionality via a fragment.
- A fragment is a mini application that is self-contained, which means it brings everything it needs with it.
- The micro frontends model typically comes with more code for the browser. It's important to address web performance from the start.
- There are multiple frontend integration techniques that work either on the client or on the browser.
- Having a shared design system helps to achieve a consistent look and feel across all team frontends.
- To make good vertical cuts it's important to know your companies domain well. Changing responsibilities afterwards works but creates friction.

IMPLEMENTATION



With micro frontends the application including the frontend gets split into smaller vertical systems. Compared to a monolith there are multiple benefits. A vertical system …

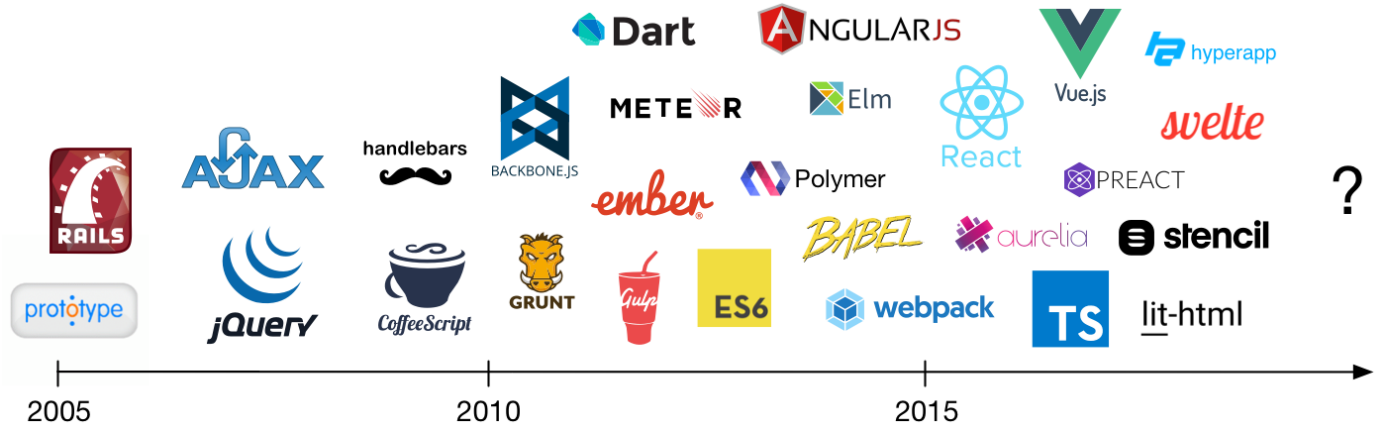
- is easier to test.
- is narrow in scope.
- has a smaller codebase.
- is easier to refactor.
- is independently deployable.
- is easier to understand and maintain.
- can use the technology that fits best.
- isolates the risk of failure to a smaller area.
- is more predictable because it does not share state with other systems.

Frontend development transformed from "making the html pretty with css" to a professional field of engineering.

To deliver good work a web developer nowadays needs to know topics like responsive design, usability, web performance, reusable components, testability, accessibility, security and the changes in web standards and their browser support.

The evolution of frontend tools, libraries and frameworks is closely related to the rising expectations for a frontend.

Tools in frontend development are changing fast. Being able to adapt, when it makes sense is important.



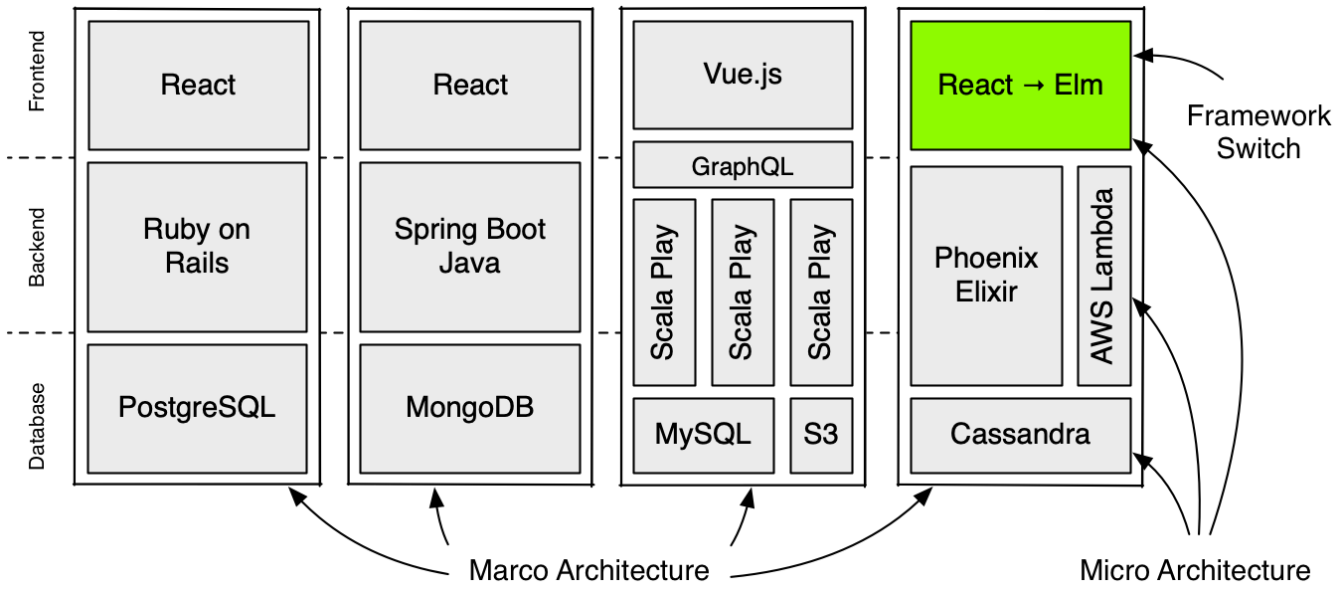
Since they are in full control of their technology stack (*Micro Architecture*), its easy for the team to make the decision and switch horses.

They don't have to coordinate with other teams.

The only thing they have to ensure is that they stay compatible with the previously agreed upon inter-team conventions (*Macro Architecture*).

These might include adhering to namespaces and supporting the chosen frontend integration technique.

You'll learn more about these conventions through the course of the book.Teams can decide about their internal architecture (micro architecture) on their own as long as they stay in the boundaries of the agreed upon macro architecture.



Self-contained

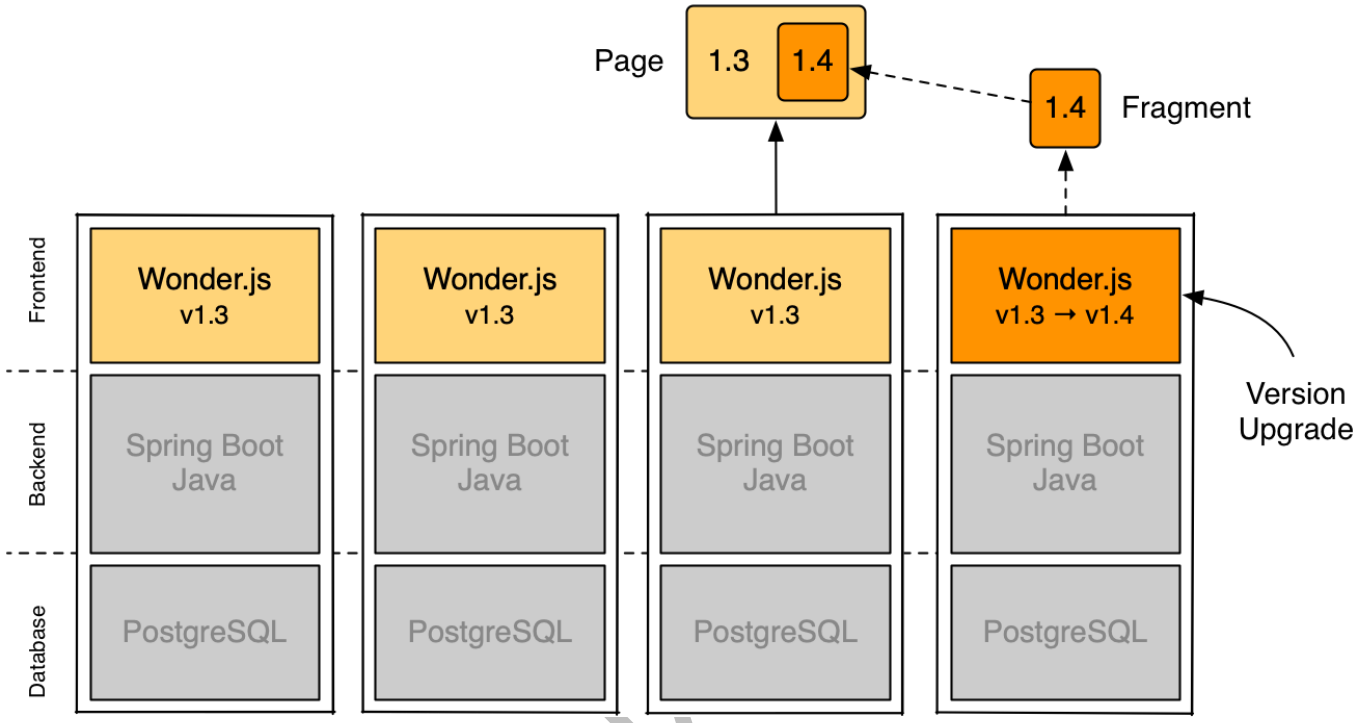
Pages and fragments are self-contained.

That means they bring their own markup, styles and scripts and should not have shared runtime dependencies.

This makes it possible for a team to deploy a new feature in a fragment without having to consult with other teams first.

An update may also come with an upgraded version of the JavaScript framework they are using. Because the fragment is isolated, this is not a big deal.

Fragments are self-contained and upgradable independently of the page they are embedded in.



ADVANTAGES

Good for medium to large projects

When the team exceeds that size it is worthwhile considering a team split.

Doing a vertical, micro frontend style, split is definitely an option you should look into.

I've worked on different micro frontends projects in the e-commerce field with 2 to 6 teams and 10 to 50 people in total.

For this project size the micro frontends model works pretty well. But its not limited to that size.

Works best on the web

Though the ideas behind micro frontends are not limited to a specific platform it works best on the web.

Here the openness of the web plays its strength.

Native monolith

Native applications for controlled platforms like iOS or Android are monolithic by design.

Composing and replacing functionality on the fly is not possible.

For updating a native app you have to build a single application bundle that's then submitted to Apples or Googles review process.

One possible way around this is to load parts of the application from the web via an embedded browser or WebView and keep the native part of the app to a minimum.

But when you have to implement native UI it's hard to have multiple end-to-end teams working on it without stepping on each others toes.

Multiple frontends per team

A team is also not limited to only have one frontend.

In e-commerce its common, that you have a front-office (customer facing) and a back-office (employee facing) side of your shop.

The team that builds the checkout for the end-user will e.g. also build the associated help desk functionality for the customer hotline or the WebView-based version of the checkout that can be embedded in a native app.

Productivity vs. overhead

Dividing your application into autonomous systems brings a lot of benefits but does not come for free.

Setup

When starting fresh you need to find good team boundaries, setup the systems and implement an integration strategy.

You need to establish common rules that all teams agree on like using namespaces. Its also important to provide ways for people to exchange knowledge between teams.

Organizational complexity

Having smaller vertical systems reduces the technical complexity of the individual systems. But running a distributed system adds its own complexity on top.

Compared to a monolithic application there is a new class of problems you have to think about. Which team gets paged on the weekend when it's not possible to add an item to the basket? The browser is a shared runtime environment.

A change from one team might have negative performance effects on the complete page. It's not always easy to find out who's responsible.

You will probably need an extra shared service for your frontend integration. Depending on your choice it might not come with a lot of maintenance work. But it's one more piece to think about.

DISADVANTAGES

Redundancy

Having multiple teams side-by-side that build and run their own stack introduces a lot of redundancy.

Every team needs to setup and maintain its own application server, build process, continuous integration pipeline and might ship redundant JavaScript/CSS code to the browser.

Here two examples where this is an issue:

- A critical bug in a popular library can't be fixed in one central place. All teams that use it must install and deploy the fix themselves.
- When one team has put in the work to make their build process twice as fast, the other teams don't benefit from this change. This team has to share this information with the others and they have to implement the same optimization on their own.

Consistency

This architecture requires all teams to have their own database to be fully independent. But sometimes one team needs data that another team owns.

In an online-store the product is a good example for this. All teams need to know what products the shop actually offers.

A typical solution for this is data replication using an event bus or a feed system.

One team owns the product data, the other teams replicate that data on a regular basis.

When one team goes down, the other teams are not affected and still have access to their local representation of the data.

But these replication mechanisms take time and introduce latency.

Thereby changes in price or availability might be inconsistent for brief periods of time.

A product that's promoted with a discount on the homepage might not have that discount in the shopping-cart.

When everything works as expected we are talking about delays in the region of milliseconds or seconds but when something goes wrong this duration can be longer.

Heterogeneity

Free technology choice is one of the biggest advantages that micro frontends introduces, but it's also one of the points that is discussed controversially.

Do I want all development teams do have a completely different technology stack? It makes it harder for developers to switch from one team to another or even exchange best practices.

But just *because you can* does not mean that *you have to* pick a different stack.

Even when all teams opt to use the same technologies the core benefits of autonomous version upgrades and less communication overhead remain

More frontend code

As stated earlier, sites that are build using micro frontends typically require more JavaScript and CSS code. Building fragments that can run in isolation introduces redundancy.

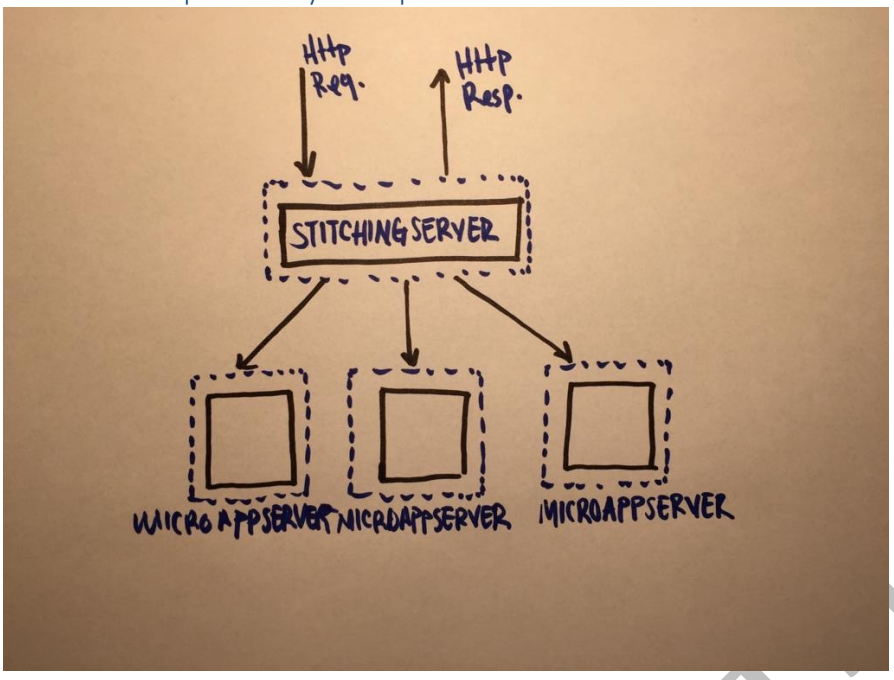
Client-side

- Orchestration
- Routing
- Isolation of micro-apps
- App to app communication
- Consistency between micro-app UIs

Server-side

- Server-side rendering
- Routing
- Dependency management

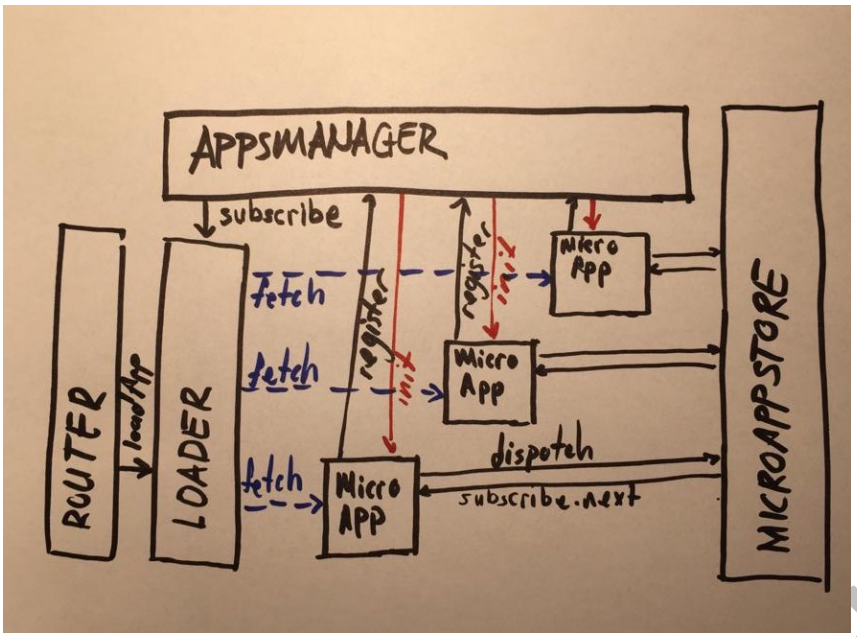
A flexible and powerful yet simple architecture



So, it worthed the wait all along this article! The basic elements and requirements of a micro frontends architecture finally started to reveal itself!

With the guidance of these requirements and concerns, I started to develop a solution which I named as ***microfe***. Here I will describe the architectural goal of this project by underlining its main components in an abstract manner.

It is easy to start with client-side and it has three separate backbone-structures: *AppsManager*, *Loader*, *Router* and one extra *MicroAppStore.*



AppsManager

AppsManager is the core of client-side micro-app orchestration.

The main functionality of AppsManager is to create the dependency tree.

When all of the dependencies of a micro-app are resolved, it instantiates the micro-app.

Loader

Another important part of client-side micro-app orchestration is the Loader.

The responsibility of the loader is fetching the unresolved micro-apps from the server-side.

Router

To solve client-side routing I introduced the Router into ***microfe***.

Unlike the common client-side routers, the ***microfe*** router has limited functionalities, It does not resolve the pages but micro-apps. Let's say we have an URL /content/detail/13 and a

ContentMicroApp. In that case, the ***microfe*** router will resolve the URL up to /content/* and it will call *ContentMicroApp* /detail/13 URL part.

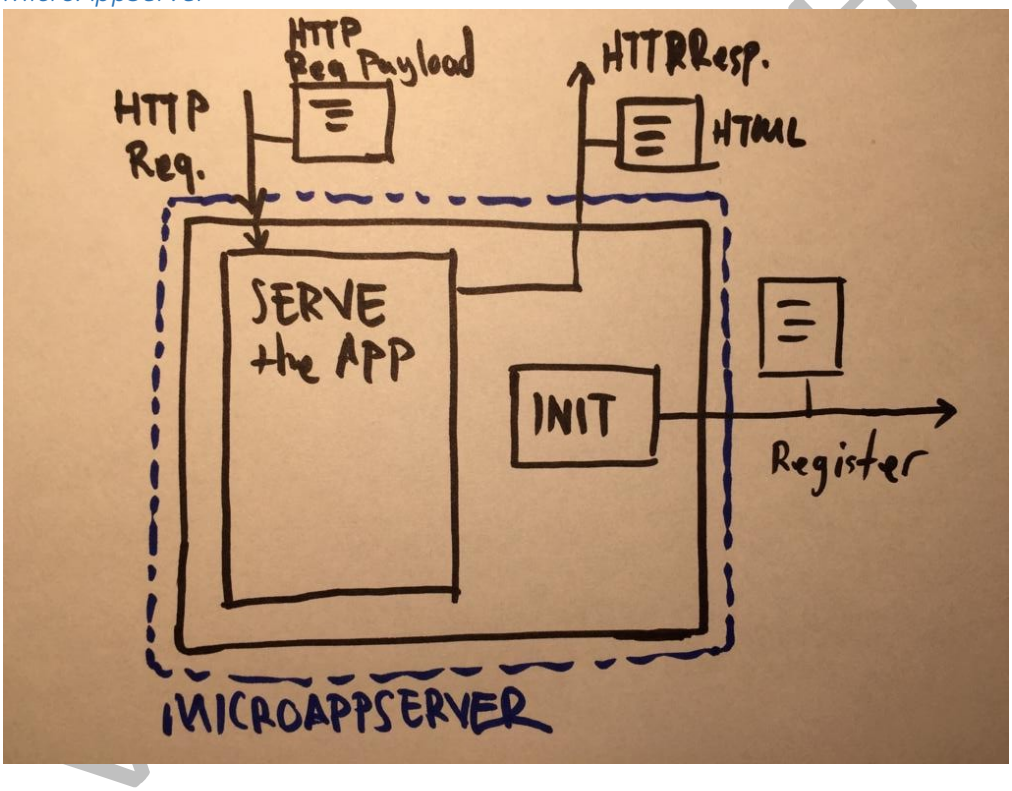
MicroAppStore

To solve micro-app to micro-app client-side communication I introduced MicroAppStore into ***microfe.***

It has the similar functionalities of Redux library with a difference: It is resilient to asynchronous data structure changes and reducer declarations.

The server-side part can be a little bit more complicated in implementation but simpler in structure. It consists of only two main part *StitchingServer* and lots of *MicroAppServer*.

MicroAppServer



Bare minimum functionality of a *MicroAppServer* can be summarized as *init* and *serve*.

While a *MicroAppServer* booting up first thing it should do is calling *SticthingServer* register endpoint with a *micro-app declaration* which defines the micro-app *dependencies*, *type*, and *URL schema* of *MicroAppServer*.

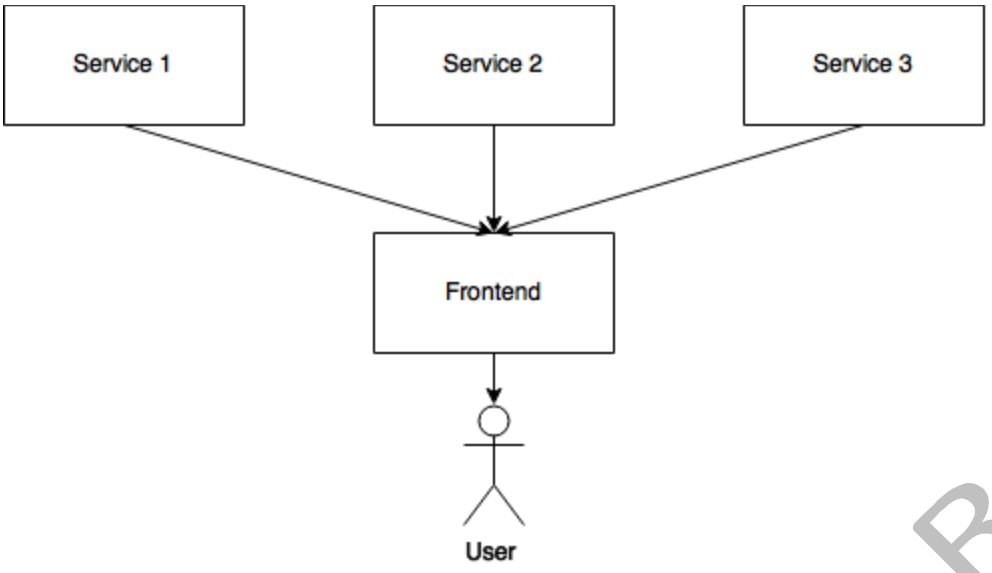
I think there is no need to mention about serve functionality since there is nothing special about it.

StitchingServer

StitchingServer provides a *register* endpoint for *MicroAppServers*. When a *MicroAppServer* registers itself to *StichingServer*, *StichingServer* records the declaration of the *MicroAppServer*

The monolithic frontend in the microservices architecture

When you are implementing a microservices architecture you want to keep services small. This should also apply to the frontend.
When you have a big monolithic frontend that can't be split up easily, you have to think about making it smaller.
You can decompose the frontend into separate components independently developed by different teams.
You have recently identified a couple of self-contained features and created microservices to provide each functionality. Your former monolith has been carved down to bare essentials for providing the user interface, which is your public facing web frontend. This microservice only has one functionality which is providing the user interface. It can be scaled and deployed separate from the other backend services.



With a monolithic frontend you never get the flexibility to scale across teams as promised by microservices.

Besides not being able to scale, there is also the classical overhead of a separate backend and frontend team.

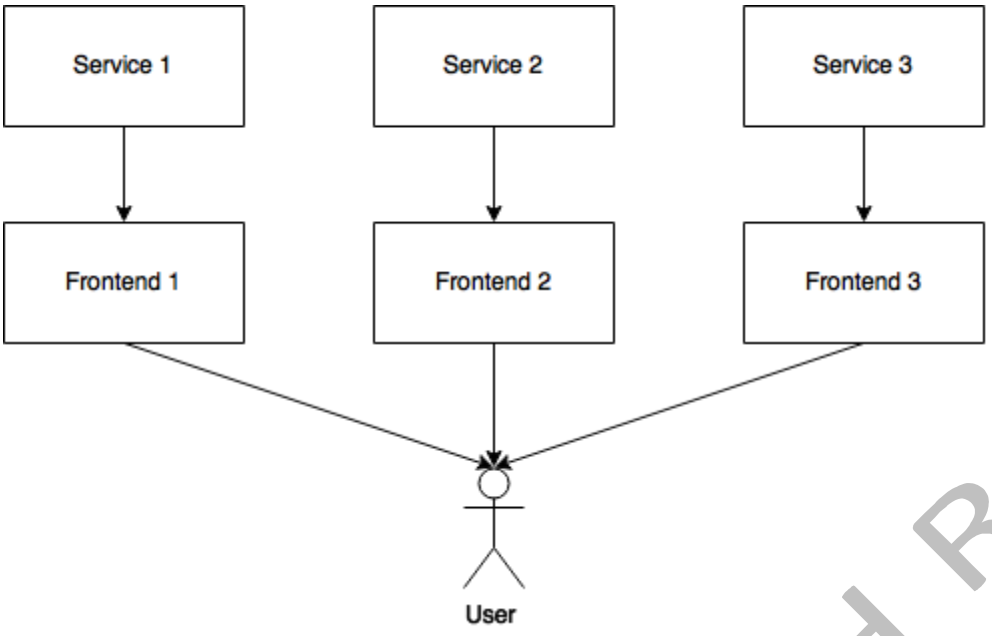
Each time there is a breaking change in the API of one of the services, the frontend has to be updated.

Especially when a feature is added to a service, the frontend has to be updated to ensure your customers can even use the feature.

If you have a frontend small enough it can be maintained by a team which is also responsible for one or more services which are coupled to the frontend.

This means that there is no overhead in cross team communication. But because the frontend and the backend can not be worked on independently, you are not really doing microservices. F

or an application which is small enough to be maintained by a single team it is probably a good idea not to do microservices.



Often you are unable to separate your web application into multiple entirely separate applications.

A consistent look and feel has to be maintained and the application should behave as single application.

However the application and the development team are big enough to justify a microservices architecture.

Examples of such big client facing applications can be found in online retail, news, social networks or other online platforms.

Share code

You can share code to make sure that the look and feel of the different frontends is consistent.

However then you risk coupling services via the common code.

This could even result in not being able to deploy and release separately.

It will also require some coordination regarding the shared code.

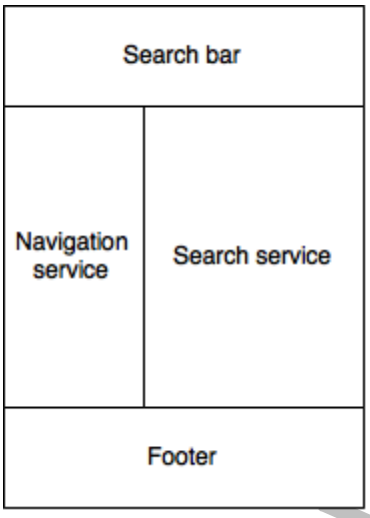
Composite frontend

It is possible to compose your frontend out of different components.

Each of these components could be maintained by a separate team and deployed independent of each other.

Again it is important to split along bounded contexts to limit the API surface between the components.

The image below shows an example of such a composite frontend.



Admittedly this is an idea we already saw in portlets during the SOA age.

However, in a microservices architecture you want the frontend components to be able to deploy fully independently and you want to make sure you do a clean separation which ensures there is no or only limited two way communication needed between the components.

Six ways to implement a micro-frontends architecture

- Basic: Application Distribution Routing -> Route Distribution Application
  - Backend: Function Call -> Remote Call
  - Front End: Component Call -> Application Call
- Route-dispatch micro-frontends
- Create a container with iFrame
- Homemade micro-frontends framework
- Combined integration: Widging applications
- Pure Web Components technology build
- Build with Web Components
  - Integrating existing frameworks in Web Components
  - Web Components integrated into existing frameworks
- Compound type micro-frontends

PATTERNS SUPPORTED IN MICRO FRONTEND ARCHITECTURE

Microfrontends are not a new thing, but certainly a recent trend.

Coined in 2016, the pattern slowly gained popularity as problems started to appear when developing large scale web apps.

In this article, we'll go over the different patterns for creating microfrontends, their advantages and drawbacks, as well as implementation details and examples for each of the presented methods.

I will also argue that microfrontends come with some inherited problems that may be solved by going even a step further — into a region that could be either called Modulith or Siteless UI, depending on the point of view.

But let's go step by step. We start our journey with a historic background.

Background

When the web (i.e., HTTP as transport and HTML as representation) started there was no notion of “design” or “layout”.

Instead, text documents have been exchanged. The introduction of the `<img>` tag changed that all.

Together with `<table>` designers could declare war on good taste.

Nevertheless, one problem arises quite quickly: How is it possible to share a common layout across multiple sites? For this purpose two solutions have been proposed:

1. Use a program to dynamically generate the HTML (slowish, but okay — especially with the powerful capabilities behind the CGI standard)
2. Use mechanisms already integrated into the web server to replace common parts with other parts

While the former lead to C and Perl web servers, which became PHP and Java then converted to C# and Ruby, before finally emerging at Elixir and Node.js, the latter wasn’t really after 2002.

The web 2.0 also demanded more sophisticated tools, which is why server-side rendering using full-blown applications dominated for quite while.

Until Netflix came and told everyone to make smaller services to make cloud vendors rich.

Ironically, while Netflix would be ready for their own data centers they are still massively coupled to cloud vendors such as AWS, which also hosts most of the competition including Amazon Prime Video.

The Patterns

In the following, we'll look at some of the patterns that are possible for actually realizing a microfrontend architecture.

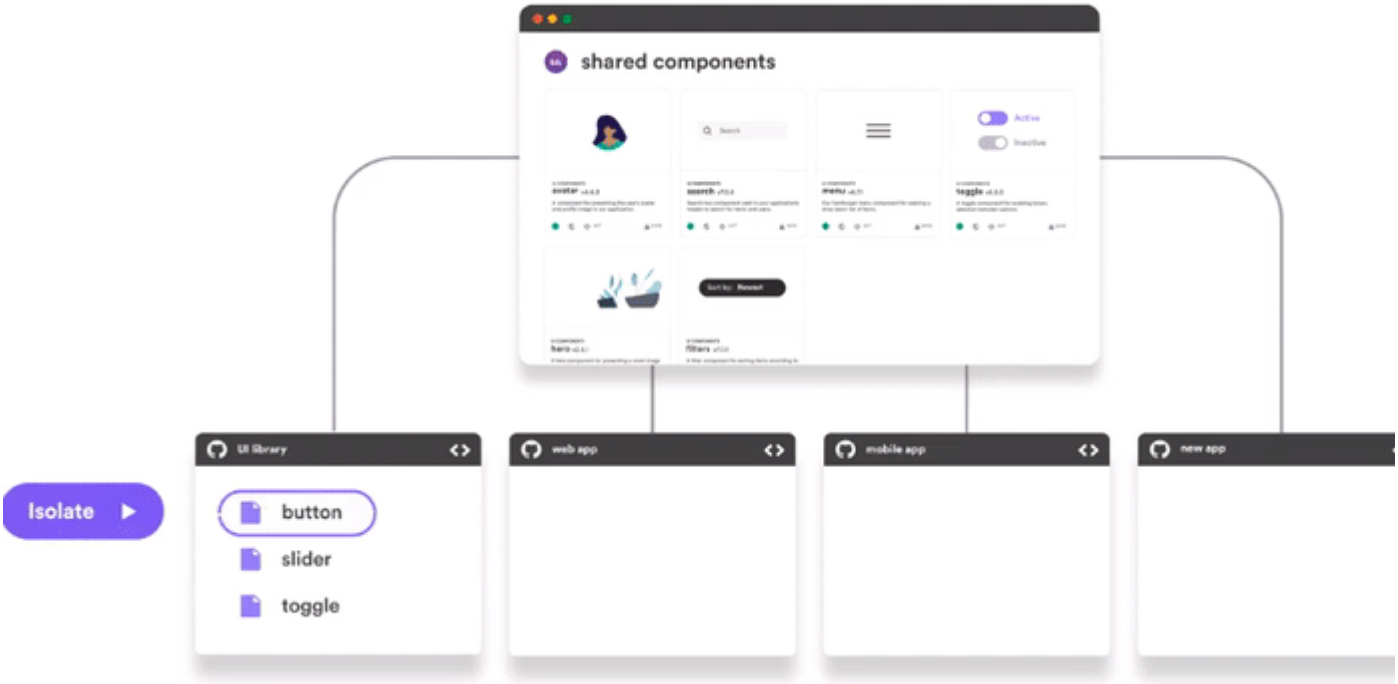
We'll see that "it depends" is in fact the right answer when somebody asks: "what is the right way to implement microfrontends?". It very much depends on what we are after.

Each section contains a bit of example code and a very simple snippet (sometimes using a framework) to realize that pattern for a proof of concept or even an MVP.

In the end, I try to provide a small summary to indicate the target audience according to my personal feelings.

Regardless of the pattern you choose, when integrating separate projects, keeping a consistent UI is always a challenge.

Use tools like **Bit** (Github) to share and collaborate on UI components across your different microservices.



Sharing components across projects with Bit

The Web Approach

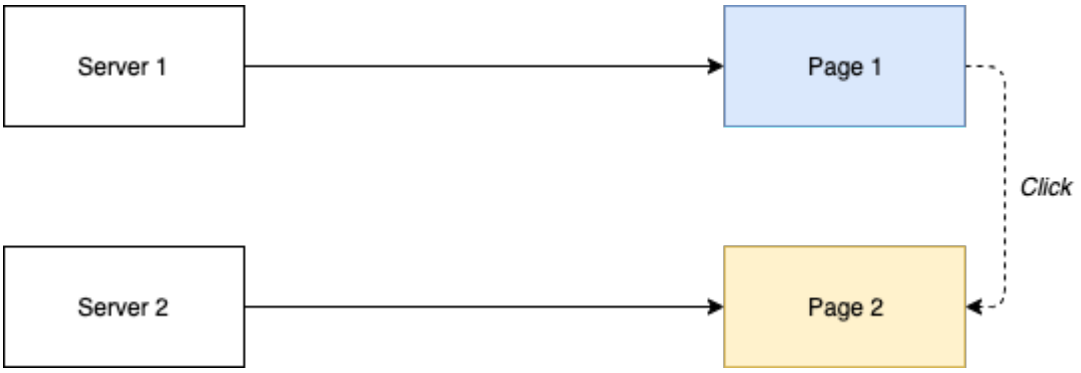
The most simple method of implementing microfrontends is to deploy a set of small websites (ideally just a single page), which are just linked together.

The user goes from website to website by using the links leading to the different servers providing the content.

To keep the layout consistent a pattern library may be used on the server.

Each team can implement the server-side rendering as they desire.

The pattern library must also be usable on different platforms.



Pattern: Web Approach

Using the web approach can be as simple as deploying static sites to a server.

This could be done with a Docker image as follows:

Obviously, we are not restricted to use a static site.

We can apply server-side rendering, too.

Changing the nginx base image to, e.g., ASP.NET Core allows us to use ASP.NET Core for generating the page.

But how is this different to the frontend monolith? In this scenario, we would, for example, take a given microservice, exposed via a web API (i.e., returning something like JSON) and change it to return rendered HTML instead.

Logically, microfrontends in this world are nothing more than a different way of representing our API. Instead of returning "naked" data we generate the view already.

In this space we find the following solutions:

- nginx
- Docker

What are pros and cons of this approach?

- **Pro**: Completely isolated
- **Pro**: Most flexible approach
- **Pro**: Least complicated approach
- **Con**: Infrastructure overhead
- **Con**: Inconsistent user experience
- **Con**: Internal URLs exposed to outside

Server-Side Composition

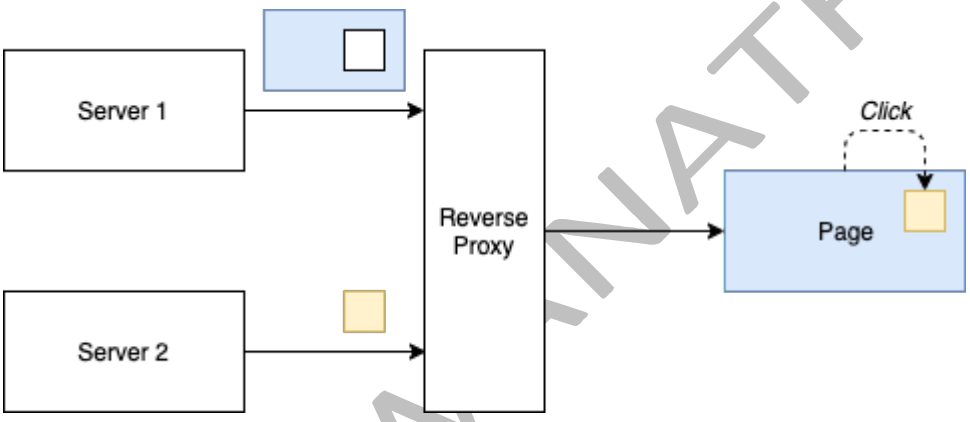
This is the *true* microfrontend approach.

Why? As we have seen, microfrontends have been supposed to be run server-side.

As such the whole approach works independently for sure.

When we have a dedicated server for each small frontend snippet we may really call this **micro** frontend.

In a diagrammatic form, we may end up with a sketch like the one below.



Pattern: Server-Side Composition

The complexity of this solution lies totally in the reverse proxy layer.

How the different smaller sites are combined into one site can be tricky.

Especially things like caching rules, tracking, and other tricky bits will bite us at night.

In some sense, this adds a kind of gateway layer to the first approach.

The reverse proxy combines different sources into a single delivery.

While the tricky bits certainly need to (and can) be solved somehow.

A little bit more powerful would be something like using Varnish Reverse Proxy.

In addition, we find that this is also a perfect use case for ESI (abbr. Edge-Side Includes) — which is the (much more flexible) successor to the historic Server-Side Includes (SSI).

A similar setup can be seen with the Tailor backend service, which is a part of Project Mosaic.

In this space we find the following solutions:

- Project Mosaic
- Podium

What are pros and cons of this approach?

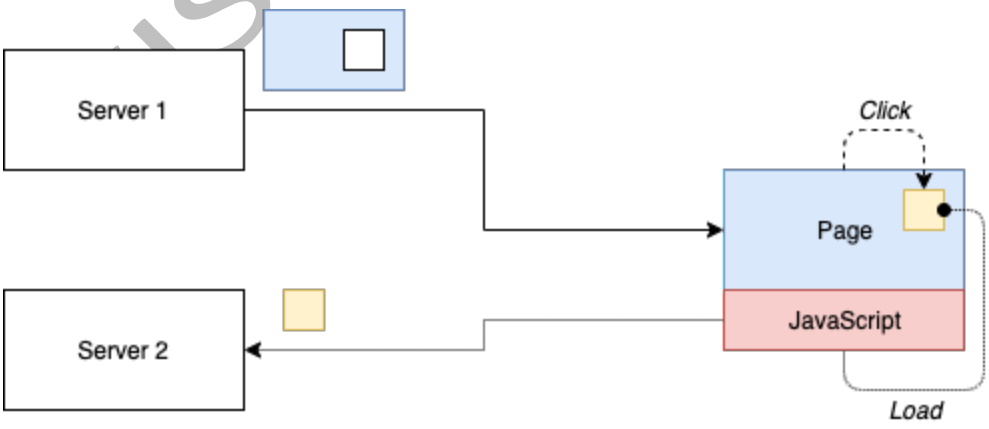
- **Pro**: Completely isolated
- **Pro**: Looks embedded to user
- **Pro**: Very flexible approach
- **Con**: Enforces coupling between the components
- **Con**: Infrastructure complexity
- **Con**: Inconsistent user experience

Client-Side Composition

At this point one may be wondering: Do we need the reverse proxy? As this is a backend component we may want to avoid this altogether.

The solution is client-side composition.

In the simplest form this can be implemented with the use of `<iframe>` elements. Communication between the different parts is done via the `postMessage` method.



Pattern: Client-Side Composition

Note: The JavaScript part may be replaced with "browser" in case of an `<iframe>`. In this case the potential interactivity is certainly different.

As already suggested by the name, this pattern tries to avoid the infrastructure overhead coming with a reverse proxy. Instead, since microfrontends contain already the term "frontend" the whole rendering is left to the client. The advantage is that starting with this pattern serverless may be possible. In the end the whole UI could be uploaded to, e.g., a GitHub pages repository and everything just works.

As outlined, the composition can be done with quite simple methods, e.g., just an `<iframe>`. One of the major pain points, however, is how such integrations will look to the end user. The duplication in terms of resource needs is also quite substantial. A mix with pattern 1 is definitely possible, where the different parts are being placed on independently operated web servers.

Nevertheless, in this pattern knowledge is required again — so component 1 already knows that component 2 exists and needs to be used. Potentially, it even needs to know *how* to use it.

Considering the following parent (i.e., delivered application or website):

https://gist.github.com/dddbf80773ed8df03fcf20679f30c835

We can write a page that enables the direct communication path:

If we do not consider frames an option we could also go for web components. Here, communication could be done via the DOM by using custom events. However, already at this point in time it may make sense to consider client-side rendering instead of client-side composition; as rendering implies the need for a JavaScript client (which aligns with the web component approach).

In this space we find the following solutions:

- Web Components
- PostMate

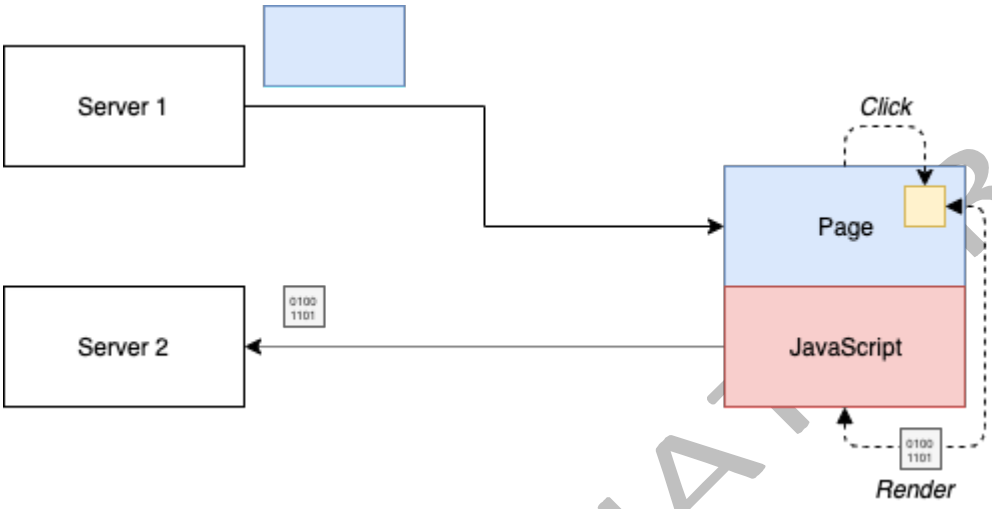
What are pros and cons of this approach?

- **Pro**: Completely isolated
- **Pro**: Looks embedded to user
- **Pro**: Serverless possible
- **Con**: Enforces coupling between the components
- **Con**: Inconsistent user experience
- **Con**: May require JavaScript / no seamless integration

Client-Side Rendering

While client-side composition may work without JavaScript (e.g., only using frames that do not rely on communication with the parent or each other), client-side rendering will fail without JavaScript. In this space we already start creating a framework in the composing application. This framework has to be respected by all microfrontends brought in. At least they need to use it for being mounted properly.

The pattern looks as follows.



Pattern: Client-Side Rendering

Quite close to the client-side composition, right? In this case the JavaScript part may not be replaced. The important difference is that server-side rendering is in general off the table. Instead, pieces of data are exchanged, which are then transformed into a view.

Depending on the designed or used framework the pieces of data may determine the location, point in time, and interactivity of the rendered fragment. Achieving a high degree of interactivity is no problem with this pattern.

In this space we find the following solutions:

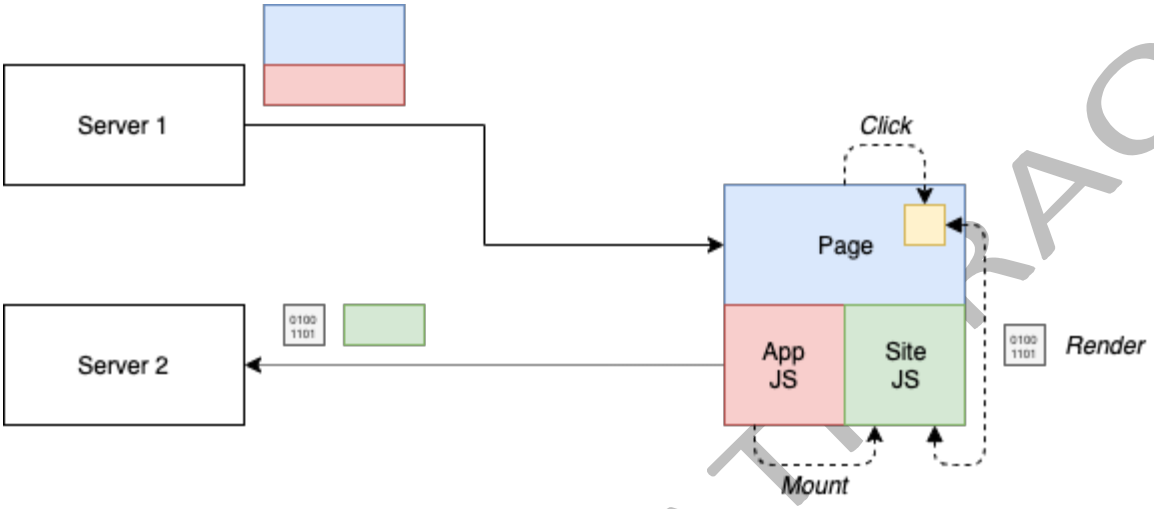
- Web Components
- Luigi

What are pros and cons of this approach?

- **Pro**: Enforces separation of concerns
- **Pro**: Provides loose coupling of the components
- **Pro**: Looks embedded to the user
- **Con**: Requires more logic in the client
- **Con**: Inconsistent user experience
- **Con**: Requires JavaScript

SPA Composition

Why should we stop at client-side rendering using a single technology? Why not just obtain a JavaScript file and run it besides all the other JavaScript files? The benefit of this is the potential use of multiple technologies side-by-side.



Pattern: SPA Composition

It is up for debate if running multiple technologies (independent if its in the backend or the frontend — granted, in the backend it may be more "acceptable") is a good thing or something to avoid, however, there are scenarios where multiple technologies need to work together.

From the top of my head:

- Migration scenarios
- Support of a specific third-party technology
- Political issues
- Team constraints

Either way, the emerged pattern could be drawn as below.

So what is going on here? In this case delivering just some JavaScript with the app shell is no longer optional — instead, we need to deliver a framework that is capable of orchestrating the microfrontends.

The orchestration of the different modules boils down to the management of a lifecycle: mounting, running, unmounting. The different modules can be taken from independently running servers, however, their location must be already known in the application shell.

Implementing such a framework requires at least some configuration, e.g., a map of the scripts to include:

The lifecycle management may be more complicated than the script above. Thus a module for such a composition needs to come with some structure applied — at least an exported `mount` and `unmount` function.

In this space we find the following solutions:

- Single SPA
- Frint.js

What are pros and cons of this approach?

- **Pro**: Enforces separation of concerns
- **Pro**: Gives developers much freedom
- **Pro**: Looks embedded to the user
- **Con**: Enforces duplication and overhead
- **Con**: Inconsistent user experience
- **Con**: Requires JavaScript

Siteless UIs

This topic deserves its own article, but since we are listing all the patterns I don't want to omit it here. Taking the approach of SPA composition all we miss is a decoupling (or independent centralization) of the script sources from the services, as well as a shared runtime.

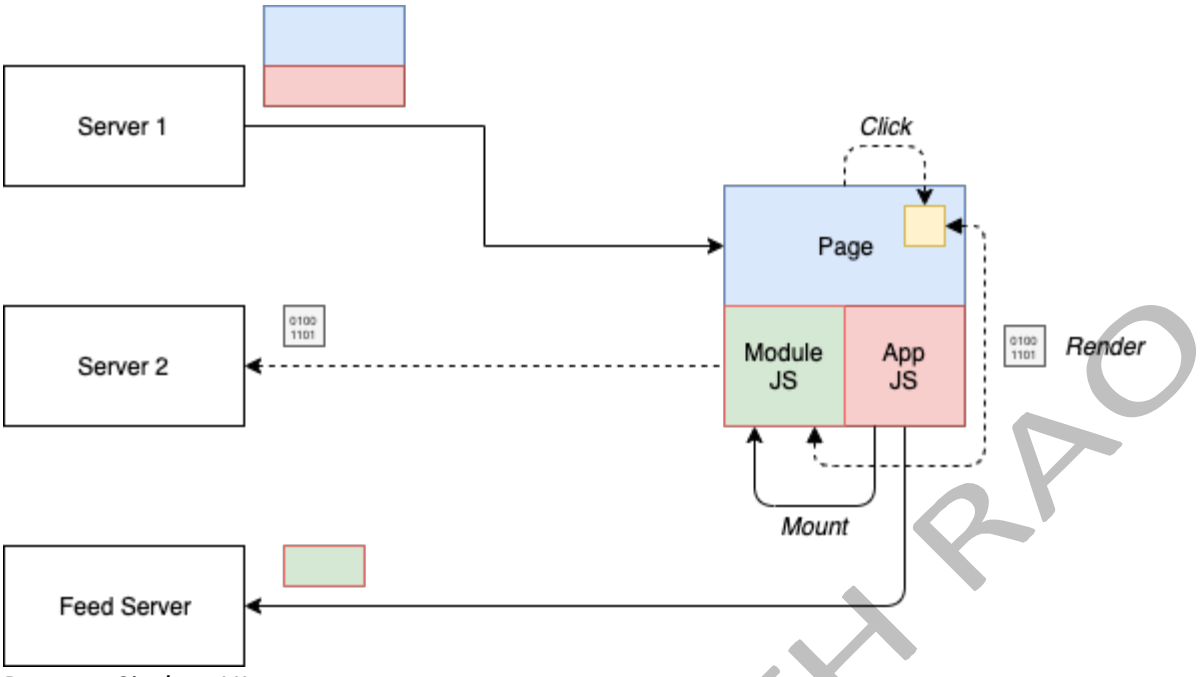
Both things are done for a reason:

- The decoupling makes sure that UI and service responsibilities are not mixed up; this also enables serverless computing
- The shared runtime is the cure for the resource intense composition given by the previous pattern

Both things combined yield benefits to the frontend as "serverless functions" did for the backend. They also come with similar challenges:

- The runtime cannot be just updated — it must stay / remain consistent with the modules
- Debugging or running the modules locally require an emulator of the runtime
- Not all technologies may be supported equally

The diagram for siteless UIs looks as follows.



Pattern: Siteless UIs

The main advantage of this design is that sharing of useful or common resources is supported. Sharing a pattern library makes a lot of sense.

All in all the architecture diagram looks quite similar to the SPA composition mentioned earlier. However, the feed service and the coupling to a runtime bring additional benefits (and challenges to be solved by any framework in that space).

The big advantage is that once these challenges are cracked the development experience is supposed to be excellent. The user experience can be fully customized, treating the modules as flexible opt-in pieces of functionality. A clear separation between feature (the respective implementation) and permission (the right to access the feature) is thus possible.

One of the easiest implementations of this pattern is the following:

This uses a global variable to share the API from the app shell. However, we already see several challenges using this approach:

- What if one module crashes?
- How to share dependencies (to avoid bundling them with each module, as shown in the simple implementation)?
- How to get proper typing?
- How is this being debugged?
- How is proper routing done?

Implementing all of these features is a topic of its own. Regarding the debugging we should follow the same approach as all the serverless frameworks (e.g., AWS Lambda, Azure

Functions) do. We should just ship an emulator that behaves like the real thing later on; except that it is running locally and works offline.

In this space we find the following solutions:

- Piral

What are pros and cons of this approach?

- **Pro**: Enforces separation of concerns
- **Pro**: Supports sharing of resources to avoid overhead
- **Pro**: Consistent and embedded user experience
- **Con**: Strict dependency management for shared resources necessary
- **Con**: Requires another piece of (potentially managed) infrastructure
- **Con**: Requires JavaScript

A Framework for Microfrontends

Finally, we should have a look at how one of the provided frameworks can be used to implement microfrontends. We go for Piral as this is the one I'm most familiar with.

In the following we approach the problem from two sides. First, we start with a module (i.e., microfrontend) in this context. Then we'll walk over creating an app shell.

For the module we use my Mario5 toy project. This is a project that started several years ago with a JavaScript implementation of Super Mario called "Mario5". It was followed up by a TypeScript tutorial / rewrite named "Mario5TS", which has been kept up-to-date since then.

For the app shell we utilize the sample Piral instance. This one shows all concepts in one sweep. It's also always kept up-to-date.

Let's start with a module, which in the Piral framework is called a **pilet**. At its core, a pilet has a JavaScript root module which is usually located in *src/index.tsx*.

A Pilet

Starting with an empty pilet gives us the following root module:

We need to export a specially named function called `setup`. This one will be used later to integrate the specific parts of our application.

Using React we could for instance register a menu item or a tile to be always displayed:

Since our tile requires some styling we also add a stylesheet to our pilet. Great, so far so good. All the directly included resources will be *always* available on the app shell.

Now its time to also integrate the game itself. We decide to put it in a dedicated page, even though a modal dialog may also be cool. All the code sits in *mario.ts* and works against the standard DOM — no React yet.

As React supports also manipulation of hosted nodes we use a reference hook to attach the game.

Theoretically, we could also add further functionality such as resuming the game or lazy loading the side-bundle containing the game. Right now only sounds are lazy loaded through calls to the `import()` function.

Starting the pilet will be done via

https://gist.github.com/0678ac99f4653843159cf8e54ad06422

which uses the Piral CLI under the hood. The Piral CLI is always installed locally, but could be also installed globally to get commands such as `pilet debug` directly available in the command line.

Building the pilet can be also done with the local installation.

The App Shell

Now its time to create an app shell. Usually, we would already have an app shell (e.g., the previous pilet was created already for the sample app shell), but in my opinion its more important to see how development of a module goes.

Creating an app shell with Piral is as simple as installing `piral`. To make it even more simple the Piral CLI also supports scaffolding of a new app shell.

Either way we will most likely end up with something like this:

Here we do three things:

1. We set up all the imports and libraries
2. We create the Piral instance; supplying all functional options (most importantly declaring where the pilets come from)
3. We render the app shell using components and a custom layout that we defined

The actual rendering is done from React.

Building the app shell is straight forward — in the end its a standard bundler (Parcel) to process the whole application. The output is a folder containing all files to be placed on a webserver or static storage.

Points of Interest

Coining the term “Siteless UI” potentially requires a bit of explanation. I’ll try to start with the name first: As seen, it is a direct reference to “Serverless Computing”. While serverless may also be a good term for the used technology, it may also be misleading and wrong. UIs can generally be deployed on serverless infrastructures (e.g., Amazon S3, Azure Blob Storage, Dropbox). This is one of the benefits of “rendering the UI on the client” instead of doing server-side rendering.

However, I wanted to follow the approach “having an UI that cannot live without a host” kind a thing. Same as with serverless functions, which require a runtime sitting somewhere and could not start otherwise.

Let’s compare the similarities. First, let’s start of with the conjunction that microfrontends should be to frontend UIs what microservices have been to backend services. In this case we should have:

- can start standalone
- provide independent URLs
- have independent lifetime (startup, recycle, shutdown)
- do independent deployments
- define an independent user / state management
- run as a dedicated webserver somewhere (e.g., as a docker image)
- if combined, we use a gateway-like structure

Great, certainly some things that apply here and there, however, note that some of this contradicts with SPA composition and, as a result, with siteless UIs, too.

Now let’s compare this to an analogy that we can do if the conjecture changes to: siteless UIs should be to frontend UIs what serverless functions have been to backend services. In this case we have:

- require a runtime to start
- provide URLs within the environment given by the runtime
- are coupled to the lifetime defined by the runtime
- do independent deployments
- defined partially independent, partially shared (but controlled / isolated) user / state management
- run on a non-operated infrastructure somewhere else (e.g., on the client)

If you agree that this reads like a perfect analogy — awesome! If not, please provide your points in the comments blow. I’ll still try to see where this is going and if this seems like a great idea to follow. Much appreciated!

Further Reading

The following posts and articles may be useful to get a complete picture:

- Bits and Pieces, 11/2019
- dev.to, 11/2019
- LogRocket, 02/2019
- Micro frontends — a microservice approach to front-end web development
- Micro Front-Ends: Available Solutions
- Exploring micro-frontends
- 6 micro-front-end types in direct comparison: These are the pros and cons (translated)

Conclusion

Microfrontends are not for everyone. Hell, they are not even for every problem. But what technology is? They certainly fill a spot. Depending on the specific problem one of the patterns may be applicable. It's great that we have many diverse and fruitful solutions available. Now the only problem is to choose — and choose wisely!

Micro Front-Ends: Available Solutions

following areas of evaluation based on my requirements and needs.

- **Implementation, complexity, and integration.** The process is simple and doesn't require to learn new complex skills or framework.
- **Code isolation and separation of concerns.** Code and styles are isolated to prevent conflicts or unintended overrides.
- **Separate deployment and team owners.** Each micro front-end is deployed and managed separately to prevent downtime and reduce deployment time.
- **Support different technologies.** We can use different technologies: React, Angular; and the same one with different versions.
- **Cross-browser compatibility.** The solution is cross-browser compatible including support for IE11.

- **Performance.** Browser strategically reloads to prevent memory pollution, network saturation, among others.

Alternatives

Web Components

One of the most popular and trending alternatives is the use of Web Components to wrap and isolate technologies. This alternative provides new custom HTML elements that can be easily integrated into any application.

Pros

- Cleaner implementation using JavaScript
- Support code and style isolation
- Support separate deployments and teams
- Support different technologies for rendering the UI
- Uses browser native's API

Cons

- Not fully cross-browser compatible — I haven't found a way to make it work in IE 11 with the v1 specifications and polyfills available —
- The road map for full cross-browser compatibility is not clear yet
- Limited documentation for the polyfills
- Performance degradation could exist if too many components are imported

Single-Spa

Another popular solution with one vision in mind: using new frameworks, without re-writing your existing app.

Pros

- Support code and style isolation
- Efficiently adds and removes components to the DOM
- Well documented
- Support different technologies
- Cross-browser compatible

Cons

- Do not support separate deployments
- Requires significant effort at the integration level
- Removing from the DOM doesn't remove memory usage — primary JavaScript code in memory —
- Because is consider a meta-framework, there is some learning and restrictions

<iframe/>

This approach is another popular solution that uses old-fashioned methods. In this case, iframes are used to isolate parts of the application altogether.

Pros

- Content is truly and completely decoupled
- Easy to use

Cons

- Not performance gain
- Resizing and scrollbars become a big issue
- Usability issues
- Bring security risks

Vanilla JavaScript and Lazy Loading

This approach requires appending a new JavaScript and CSS file into the DOM. Typically, the script will create a new HTML element to render the feature or the element must exist with a unique ID. There are many other approaches, for example, using server-side rendering to get the HTML from an API server.

Pros

- Code isolation via JavaScript encapsulation
- Easy to use
- Support separate deployments
- Supported by all browsers
- Components can be added on-demand (lazy-loading)

Cons

- Some performance degradation when loading multiples
- Manual memory management to prevent degradation

Pattern based Rendering

Server-side rendering is getting more and more traction thanks to React and its built-in server-side hydration feature. But it's not the only solution to deliver a fast experience to the user with a super fast time-to-first-byte (TTFB) score: Pre-rendering is also a pretty good strategy. What's the difference between these solutions and a fully client-rendered application?

There are three types of rendering that applications can support

- Client- rendered application
- Server- Side Rendering (SSR)
- Pre- Rendering.

Client-rendered Application

Since frameworks like Angular, Ember.js, and Backbone exists, front-end developers have tended to render everything client-side. Thanks to Google and its ability to "read" JavaScript, it works pretty well, and it's even SEO friendly.

With a client-side rendering solution, you redirect the request to a single HTML file and the server will deliver it without any content (or with a loading screen) until you fetch all the JavaScript and let the browser compile everything before rendering the content.

Under a good and reliable internet connection, it's pretty fast and works well. But it can be a lot better, and it doesn't have to be difficult to make it that way. That's what we will see in the following sections.

Server-side Rendering (SSR)

An SSR solution is something we used to do a lot, many years ago, but tend to forget in favor of a client-side rendering solution.

With **old** server-side rendering solutions, you built a web page—with PHP for example—the server compiled everything, included the data, and delivered a fully populated HTML page to the client. It was fast and effective.

But… every time you navigated to another route, the server had to do the work all over again: Get the PHP file, compile it, and deliver the HTML, with all the CSS and JS delaying the page load to a few hundred ms or even whole seconds.

What if you could do the first page load with the SSR solution, and then use a framework to do dynamic routing with AJAX, fetching only the necessary data?

This is why SSR is getting more and more traction within the community because React popularized this problem with an easy-to-use solution: The `RenderToString` method.

This new kind of web application is called a *universal app* or an *isomorphic app*. There's still some controversy over the exact meanings of these terms and the relationship between them, but many people use them interchangeably.

Anyway, the advantage of this solution is being able to develop an app server-side and client-side with the same code and deliver a really fast experience to the user with custom data. The disadvantage is that you need to run a server.

SSR is used to fetch data and pre-populate a page with custom content, leveraging the server's reliable internet connection. That is, the server's own internet connection is better than that of a user with lie-fi), so it's able to prefetch and amalgamate data before delivering it to the user.

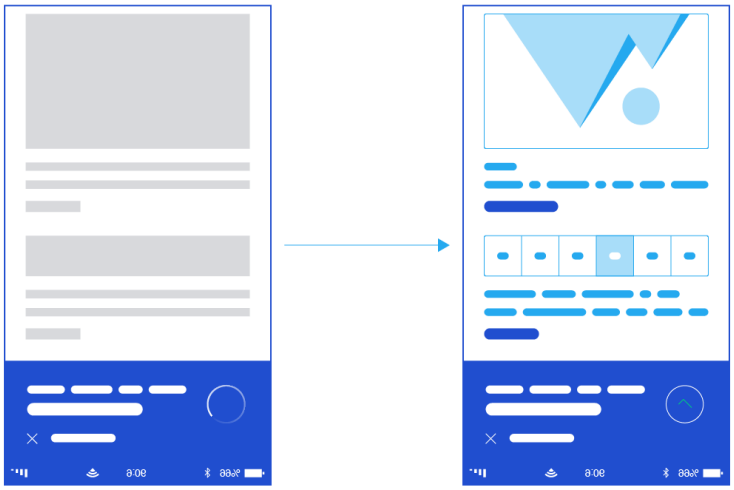
With the pre-populated data, using an SSR app can also fix an issue that client-rendered apps have with social sharing and the OpenGraph system. For example, if you have only one `index.html` file to deliver to the client, they will only have one type of metadata—most likely your homepage metadata. This won't be contextualized when you want to share a different route, so none of your routes will be shown on other sites with their proper user content (description and preview picture) that users would want to share with the world.

Pre-rendering

The mandatory server for a universal app can be a deterrent for some and may be overkill for a small application. This is why pre-rendering can be a really nice alternative.

I discovered this solution with Preact and its own CLI that allows you to compile all pre-selected routes so you can store a fully populated HTML file to a *static* server. This lets you deliver a super-fast experience to the user, thanks to the Preact/React hydration function, without the need for Node.js.

The catch is, because this isn't SSR, you don't have user-specific data to show at this point—it's just a static (and somewhat generic) file sent directly on the first request, as-is. So if you have user-specific data, here is where you can integrate a beautifully designed skeleton to show the user their data is coming, to avoid some frustration on their part:



There is another catch: In order for this technique to work, you still need to have a proxy or something to redirect the user to the right file.

Why?

With a single-page application, you need to redirect all requests to the root file, and then the framework redirects the user with its built-in routing system. So the first page load is always the same root file.

In order for a pre-rendering solution to work, you need to tell your proxy that some routes need specific files and not always the root `index.html` file.

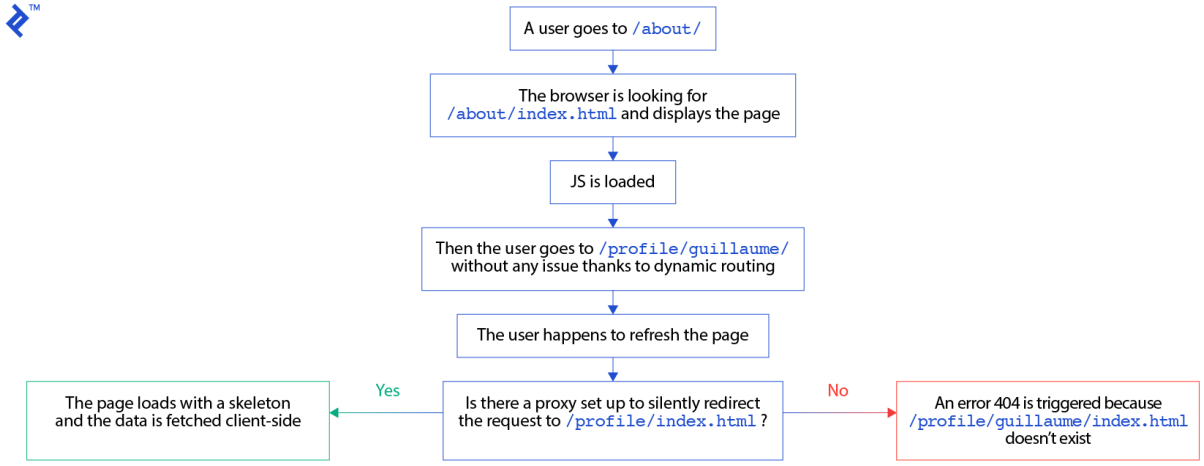
For example, say you have four routes (`/`, `/about`, `/jobs`, and `blog`) and all of them have different layouts. You need four different HTML files to deliver the skeleton to the user that will then let React/Preact/etc. rehydrate it with data. So if you redirect all those routes to the root `index.html` file, the page will have an unpleasant, glitchy feel during loading, whereby the user will see the skeleton of the wrong page until it finishes loading and replaces the layout. For example, the user might see a homepage skeleton with only one column, when they had asked for a different page with a Pinterest-like gallery.

The solution is to tell your proxy that each of those four routes needs a specific file:

- `https://my-website.com` → Redirect to the root `index.html` file
- `https://my-website.com/about` → Redirect to the `/about/index.html` file
- `https://my-website.com/jobs` → Redirect to the `/jobs/index.html` file
- `https://my-website.com/blog` → Redirect to the `/blog/index.html` file

This is why this solution can be useful for small applications—you can see how painful it would be if you had a few hundred pages.

Strictly speaking, it's not mandatory to do it this way—you could just use a static file directly. For example, `https://my-website.com/about/` will work without any redirection because it will automatically search for an `index.html` inside its directory. But you need this proxy if you have param urls—`https://my-website.com/profile/guillaume` will need to redirect the request to `/profile/index.html` with its own layout, because `profile/guillaume/index.html` doesn't exist and will trigger a 404 error.



In short, there are three basic views at play with the rendering strategies described above: A loading screen, a skeleton, and the fu

Universal Rendering

Universal rendering, or server-side rendering, involves sending the fully rendered application in the initial payload to the browser and loading the application code afterward. This shortens the time the user will have to wait before something appears on screen. Brian demonstrates what happens without universal rendering and talks about how to solve these problems

- **Introduction**
  Explains why universal rendering is beneficial, the different techniques of doing universal rendering (SSR, SSG, pre-rendering) and when to use what technique.
- **Learning Material**
  - SEO benefits of universal rendering
  - Performance benefits of SSR
  - How to implement SSR
- **Tools**
  - SSR frameworks & libraries
  - Pre-rendering services & tools
  - Static site generators

Benefits

SEO

Modern frontends (React, Vue, Angular, ...) use JavaScript to load and display content. Such JavaScript-generated-content is invisible to crawlers that don't execute JavaScript. Most crawlers (search engines and social sites) don't execute JavaScript.

The Google crawler is the only one that can successfully index JavaScript-generated-content. But it has limitations. (Mainly around delayed indexing and client-side routing, see Learning Material.)

If you want your content to be crawled by all other search engines (Bing, Baidu, DuckDuckGo, etc.), then your content needs to be included in your website's HTML.

SMO

The crawler of social media sites (Facebook, Twitter, ...) don't execute JavaScript and rely on HTML exclusively.

If you want your website to be correctly previewed when a user shares your website, then the corresponding information needs to be included in your website's HTML.

(SMO means "Social Media Optimization".)

Performance

Rendering your website's pages to HTML decreases the perceived loading time: Once the HTML is loaded, content can already be displayed before any JavaScript is loaded/executed.

The improvement is considerable on mobile where loading and executing JavaScript is much slower.

Techniques

Server-Side Rendering (SSR), Pre-Rendering, and Static Site Generators (SSG) are techniques to render JavaScript-generated-content to HTML. Making the content visible to crawlers and improving performance.

There are two ways to render JavaScript-generated-content to HTML:

- **Directly render to HTML**
  Modern view libraries (React, Vue, Angular, ...) can render views to HTML (in addition to be able to render views to the DOM). (E.g. a React component can be rendered to HTML with `require('react-dom/server').renderToStaticMarkup()`.)
- **Render to HTML via headless browser**
  A headless browser runs your website's JavaScript, the website's pages are rendered to the DOM of the headless browser, and HTML is automatically generated from the resulting DOM.

Leading to the following techniques:

- **Server-Side Rendering (SSR)**
  Directly render your website's pages to HTML at request-time: Every time a user requests a page, the server renders the page directly to HTML.
  SSR is the most reliable option if your HTML changes frequently. (If your website's content may change after deploy-time, e.g. if you website's content is generated by users.)
- **Pre-Rendering**
  A headless browser crawls your website, executes the website's JavaScript, and generates HTML upon the resulting DOM.
- **Static Site Generators (SSG)**
  A static site is a website that doesn't have any server code: The website is composed of static browser assets only (HTML, CSS, JavaScript, images, fonts, etc.). Some SSG are able to render your views to HTML at build-time: When your website is built, each page is rendered to a HTML file that includes your page's content.
  If your content only changes at deploy-time, then using a SSG is an option.

OPTIONAL REFERENCES

SSR

Frameworks

- Next.js - The most popular SSR tool.
- After.js - Similar to Next.js but with routing based on React Router.
- React Server
- Reframe - Flexible web framework. It does SSR by default and can be used as SSG.
- Fusion.js - Plugin-based universal web framework maintained by Uber.

Libraries

- Goldpage - A do-one-thing-do-it-well library that supports all app types; "SPA", "SSR", "Static Website", etc.
- Razzle - Handles the building. You do the rest.
- React Universal Component - Utility to code split your SSR app.
- Rogue.js - SSR utilities focused on flexibility. First-class support for React Router, Apollo GraphQL, Redux, Emotion, and Styled-Components. The build step is up to you (but you can use Razzle.)

Boilerplates

- cra-ssr - SSR app boilerplate based on CRA (without having ejected).

SSG

- Gatsby.js - SSG based on React and GraphQL.
- React Static - SSG based on React and focused on simplicity.
- Goldpage - A do-one-thing-do-it-well library that supports all app types; "SPA", "SSR", "Static Website", etc.
- Phenomic - SSG based on a flexible plugin system.
- Next.js - Although primarily focused on SSR, Next.js can also generate static sites.
- Reframe - Flexible web framework. It does SSR by default and can be used as SSG.

Pre-Rendering

Dynamic Pre-Rendering

Automatically and regularly render your deployed website to HTML.

SaaS

- Prerender.io

- SEO4Ajax
- Prerender.cloud
- SEO.js
- BromBone

Libraries

- Prerender.io Node Server - The prerender.io Node Server is open source.

Static Pre-Rendering

Some static pre-renderers, instead of generating HTML upon a generated DOM, directly render your pages to HTML.

- Prerender SPA Plugin - Uses Puppeteer to crawl & render your pages.
- react-snap - Uses Puppeteer to crawl & render your pages.
- prep - Uses Chromeless to crawl & render your pages.
- SSG webpack plugin - Directly render your pages to HTML. You provide render functions and routes. All routes are rendered at build-time using the render functions you provided. Also has a crawl mode to use a headless browser to automatically discover your website's URLs.
- React Snapshot - Pre-renders React apps at build-time. Uses `require('react-dom/server').renderToString` to directly render the HTML. Uses JSDOM as headless browser to automatically discover your app's URLs.

Sites vs. Apps defined: the Documents-to-Applications Continuum.

There seems to be a fundamental misunderstanding being perpetuated in the web community that there is no difference between what we call a web site and what we call a web application.

What is a site?

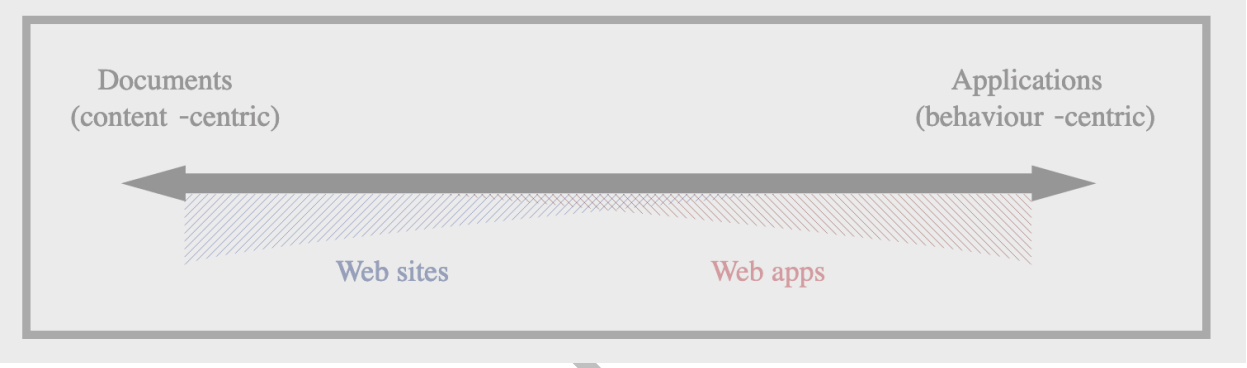
Web sites are documents; they are content-centric. Sites are geared towards content consumption.

What is an app?

Web apps are tools; they are behaviour-centric. Apps are geared towards content creation and manipulation.

it is a site; it was content-centric to begin with. Any behaviour that was added—hopefully via progressive enhancement

The Documents-to-Applications Continuum



When a product falls closer to the documents side of the continuum, we can use progressive enhancement to layer features and interactions on top of the content-based core while keeping that core accessible to the largest number of people possible. These progressively enhanced features usually either layer advanced formatting or layout on these documents or add fanciful interactions for navigating within or between them. We can make the content adapt to different screen sizes and make the limited interactions that are used to navigate the content adapt to different input mechanisms. This isn't an easy task, but nor is it impossible.

As products shift from the documents side of the spectrum to the applications side, however, implementing progressive enhancement gets harder. In fact, it might become entirely meaningless or impossible. How would you gracefully degrade an online image editor, for example? How would an image editor work on feature handsets without graphical displays? What content would you actually fall back to displaying?

In order to create exemplary user experiences, we need to maintain focus. This focus has to be placed squarely on meeting the needs of our users in the best way possible. No product team on Earth has the resources to create applications that provide the best possible user experience for every user.

Given unlimited time and resources, we could optimize the user experience of our apps on every device and platform known to humankind. However, given limited time and budget we have to work with in the real world, we must be selective with our audience, problem domain, platforms and devices. We do this not to exclude people unnecessarily but because we realize that including everyone and giving everyone a great user experience is impractical.

Progressive web applications

Progressive web applications (**PWAs**) are a type of application software delivered through the web, built using common web technologies including HTML, CSS and JavaScript. They are intended to work on any platform that uses a standards-compliant browser. Functionality includes working offline, push notifications, and device hardware access, enabling creating user experiences similar to native applications on desktop and mobile devices. Since they are a type of webpage or website known as a web application

Characteristics Of A Progressive Web App

Before we jump into the code, it is important to understand that progressive web apps have the following characteristics:

- **Progressive**. By definition, a progressive web app must work on any device and enhance progressively, taking advantage of any features available on the user's device and browser.
- **Discoverable**. Because a progressive web app is a website, it should be discoverable in search engines. This is a major advantage over native applications, which still lag behind websites in searchability.
- **Linkable**. As another characteristic inherited from websites, a well-designed website should use the URI to indicate the current state of the application. This will enable the web app to retain or reload its state when the user bookmarks or shares the app's URL.
- **Responsive**. A progressive web app's UI must fit the device's form factor and screen size.
- **App-like**. A progressive web app should look like a native app and be built on the application shell model, with minimal page refreshes.
- **Connectivity-independent**. It should work in areas of low connectivity or offline (our favorite characteristic).
- **Re-engageable**. Mobile app users are more likely to reuse their apps, and progressive web apps are intended to achieve the same goals through features such as push notifications.

- **Installable**. A progressive web app can be installed on the device's home screen, making it readily available.
- **Fresh**. When new content is published and the user is connected to the Internet, that content should be made available in the app.
- **Safe**. Because a progressive web app has a more intimate user experience and because all network requests can be intercepted through service workers, it is imperative that the app be hosted over HTTPS to prevent man-in-the-middle attacks.

Reliable

When launched from the user's home screen, service workers enable a Progressive Web App to load instantly, regardless of the network state.

A service worker, written in JavaScript, is like a client-side proxy and puts you in control of the cache and how to respond to resource requests. By pre-caching key resources you can eliminate the dependence on the network, ensuring an instant and reliable experience for your users.

Fast

53% of users will abandon a site if it takes longer than 3 seconds to load! And once loaded, users expect them to be fast—no janky scrolling or slow-to-respond interfaces.

Engaging

Progressive Web Apps are installable and live on the user's **home screen**, without the need for an app store. They offer an **immersive full screen** experience with help from a web app manifest file and can even re-engage users with web **push notifications**.

The Web App Manifest allows you to control how your app appears and how it's launched. You can specify home screen icons, the page to load when the app is launched, screen orientation, and even whether or not to show the browser chrome.

Mobile & Desktop

Using responsive design techniques, PWAs work on both mobile **and** desktop, using a single code base between platforms. If you're considering writing a native app, take a look at the benefits that a PWA offers.

Top PWA Projects

1. Trivago Hotel Booking
2. Pinterest
3. Tinder
4. 9Gag
5. OLX
6. Starbucks
7. Forbes

Web Components

Web Components consist of the following elements (also called specifications):

- **Custom Elements,** which define how to develop and use new elements of DOM tree,
- **Shadow DOM**, thanks to which you can hide the details of implementation (use encapsulations),
- **ES Modules**, which allow to use earlier defined JS documents in other documents,
- **HTML Template**, thanks to which we may define fragments which are initiated when they are necessary.

Custom Elements v1: Reusable Web Components

Introduction

Note: This article describes the new Custom Elements spec. If you've been using custom elements, chances are you're familiar with the version 0 that shipped in Chrome 33. The

concepts are the same, but the version 1 spec has important API differences. Keep reading to see what's new or check out the section on History and browser support for more info.

The browser gives us an excellent tool for structuring web applications. It's called HTML. You may have heard of it! It's declarative, portable, well supported, and easy to work with. Great as HTML may be, its vocabulary and extensibility are limited. The HTML living standard has always lacked a way to automatically associate JS behavior with your markup... until now.

Custom elements are the answer to modernizing HTML, filling in the missing pieces, and bundling structure with behavior. If HTML doesn't provide the solution to a problem, we can create a custom element that does. **Custom elements teach the browser new tricks while preserving the benefits of HTML**.

Defining a new element

To define a new HTML element we need the power of JavaScript!

The `customElements` global is used for defining a custom element and teaching the browser about a new tag. Call `customElements.define()` with the tag name you want to create and a JavaScript `class` that extends the base `HTMLElement`.

Example - defining a mobile drawer panel, `<app-drawer>`:

```
class AppDrawer extends HTMLElement {...}
window.customElements.define('app-drawer', AppDrawer);

// Or use an anonymous class if you don't want a named constructor in current
scope.
window.customElements.define('app-drawer', class extends HTMLElement {...});
```

Example usage:

```
<app-drawer></app-drawer>
```

It's important to remember that using a custom element is no different than using a `<div>` or any other element. Instances can be declared on the page, created dynamically in JavaScript, event listeners can be attached, etc. Keep reading for more examples.

Defining an element's JavaScript API

The functionality of a custom element is defined using an ES2015 `class` which extends `HTMLElement`. Extending `HTMLElement` ensures the custom element inherits the entire DOM API and means any properties/methods that you add to the class become part of the element's DOM interface. Essentially, use the class to create a **public JavaScript API** for your tag.

Example - defining the DOM interface of `<app-drawer>`:

```
class AppDrawer extends HTMLElement {

  // A getter/setter for an open property.
  get open() {
    return this.hasAttribute('open');
  }

  set open(val) {
    // Reflect the value of the open property as an HTML attribute.
    if (val) {
      this.setAttribute('open', '');
    } else {
      this.removeAttribute('open');
    }
    this.toggleDrawer();
  }

  // A getter/setter for a disabled property.
  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    // Reflect the value of the disabled property as an HTML attribute.
    if (val) {
      this.setAttribute('disabled', '');
    } else {
      this.removeAttribute('disabled');
    }
  }

  // Can define constructor arguments if you wish.
  constructor() {
    // If you define a constructor, always call super() first!
    // This is specific to CE and required by the spec.
    super();

    // Setup a click listener on <app-drawer> itself.
    this.addEventListener('click', e => {
      // Don't toggle the drawer if it's disabled.
      if (this.disabled) {
        return;
      }
      this.toggleDrawer();
    });
  }

  toggleDrawer() {
    ...
  }
}

customElements.define('app-drawer', AppDrawer);
```

In this example, we're creating a drawer that has an `open` property, `disabled` property, and a `toggleDrawer()` method. It also reflects properties as HTML attributes.

A neat feature of custom elements is that **`this` inside a class definition refers to the DOM element itself** i.e. the instance of the class. In our example, `this` refers to `<app-drawer>`. This ( ) is how the element can attach a `click` listener to itself! And you're not limited to event listeners. The entire DOM API is available inside element code. Use `this` to access the element's properties, inspect its children (`this.children`), query nodes (`this.querySelectorAll('.items')`), etc.

Rules on creating custom elements

1. The name of a custom element **must contain a dash (-)**. So `<x-tags>`, `<my-element>`, and `<my-awesome-app>` are all valid names, while `<tabs>` and `<foo_bar>` are not. This requirement is so the HTML parser can distinguish custom elements from regular elements. It also ensures forward compatibility when new tags are added to HTML.
2. You can't register the same tag more than once. Attempting to do so will throw a `DOMException`. Once you've told the browser about a new tag, that's it. No take backs.
3. Custom elements cannot be self-closing because HTML only allows a few elements to be self-closing. Always write a closing tag (`<app-drawer></app-drawer>`).

Custom element reactions

A custom element can define special lifecycle hooks for running code during interesting times of its existence. These are called **custom element reactions**.

Name	Called when
`constructor`	An instance of the element is created or upgraded. Useful for initializing state, setting up event listeners, or creating a shadow dom. See the spec for restrictions on what you can do in the `constructor`.
`connectedCallback`	Called every time the element is inserted into the DOM. Useful for running setup code, such as fetching resources or rendering. Generally, you should try to delay work until this time.
`disconnectedCallback`	Called every time the element is removed from the DOM. Useful for running clean up code.
`attributeChangedCallback(attrName, oldVal, newVal)`	Called when an observed attribute has been added, removed, updated, or replaced. Also called for initial values when an element is created by the parser, or upgraded. **Note:** only attributes listed in the `observedAttributes` property will receive this callback.

Name	Called when
`adoptedCallback`	The custom element has been moved into a new `document` (e.g. someone called `document.adoptNode(el)`).

Note: The browser calls the `attributeChangedCallback()` for any attributes whitelisted in the `observedAttributes` array (see Observing changes to attributes). Essentially, this is a performance optimization. When users change a common attribute like `style` or `class`, you don't want to be spammed with tons of callbacks.

Reaction callbacks are synchronous. If someone calls `el.setAttribute()` on your element, the browser will immediately call `attributeChangedCallback()`. Similarly, you'll receive a `disconnectedCallback()` right after your element is removed from the DOM (e.g. the user calls `el.remove()`).

Example: adding custom element reactions to `<app-drawer>`:

```
class AppDrawer extends HTMLElement {
  constructor() {
    super(); // always call super() first in the constructor.
    ...
  }
  connectedCallback() {
    ...
  }
  disconnectedCallback() {
    ...
  }
  attributeChangedCallback(attrName, oldVal, newVal) {
    ...
  }
}
```

Define reactions if/when it make senses. If your element is sufficiently complex and opens a connection to IndexedDB in `connectedCallback()`, do the necessary cleanup work in `disconnectedCallback()`. But be careful! You can't rely on your element being removed from the DOM in all circumstances. For example, `disconnectedCallback()` will never be called if the user closes the tab.

Properties and attributes

Reflecting properties to attributes

It's common for HTML properties to reflect their value back to the DOM as an HTML attribute. For example, when the values of `hidden` or `id` are changed in JS:

```
div.id = 'my-id';
div.hidden = true;
```

the values are applied to the live DOM as attributes:

```
<div id="my-id" hidden>
```

This is called "reflecting properties to attributes". Almost every property in HTML does this. Why? Attributes are also useful for configuring an element declaratively and certain APIs like accessibility and CSS selectors rely on attributes to work.

Reflecting a property is useful anywhere you want to **keep the element's DOM representation in sync with its JavaScript state**. One reason you might want to reflect a property is so user-defined styling applies when JS state changes.

Recall our `<app-drawer>`. A consumer of this component may want to fade it out and/or prevent user interaction when it's disabled:

```
app-drawer[disabled] {
  opacity: 0.5;
  pointer-events: none;
}
```

When the `disabled` property is changed in JS, we want that attribute to be added to the DOM so the user's selector matches. The element can provide that behavior by reflecting the value to an attribute of the same name:

```
...

get disabled() {
  return this.hasAttribute('disabled');
}

set disabled(val) {
  // Reflect the value of `disabled` as an attribute.
  if (val) {
    this.setAttribute('disabled', '');
  } else {
    this.removeAttribute('disabled');
  }
  this.toggleDrawer();
}
```

Observing changes to attributes

HTML attributes are a convenient way for users to declare initial state:

```
<app-drawer open disabled></app-drawer>
```

Elements can react to attribute changes by defining a `attributeChangedCallback`. The browser will call this method for every change to attributes listed in the `observedAttributes` array.

```
class AppDrawer extends HTMLElement {
  ...
```

```
  static get observedAttributes() {
    return ['disabled', 'open'];
  }

  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    if (val) {
      this.setAttribute('disabled', '');
    } else {
      this.removeAttribute('disabled');
    }
  }

  // Only called for the disabled and open attributes due to
observedAttributes
  attributeChangedCallback(name, oldValue, newValue) {
    // When the drawer is disabled, update keyboard/screen reader behavior.
    if (this.disabled) {
      this.setAttribute('tabindex', '-1');
      this.setAttribute('aria-disabled', 'true');
    } else {
      this.setAttribute('tabindex', '0');
      this.setAttribute('aria-disabled', 'false');
    }
    // TODO: also react to the open attribute changing.
  }
}
```

In the example, we're setting additional attributes on the `<app-drawer>` when a `disabled` attribute is changed. Although we're not doing it here, you could also **use the `attributeChangedCallback` to keep a JS property in sync with its attribute**.

Element upgrades

Progressively enhanced HTML

We've already learned that custom elements are defined by calling `customElements.define()`. But this doesn't mean you have to define + register a custom element all in one go.

Custom elements can be used *before* their definition is registered.

Progressive enhancement is a feature of custom elements. In other words, you can declare a bunch of `<app-drawer>` elements on the page and never invoke `customElements.define('app-drawer', ...)` until much later. This is because the browser treats potential custom elements differently thanks to unknown tags. The process of calling `define()` and endowing an existing element with a class definition is called "element upgrades".

To know when a tag name becomes defined, you can use `window.customElements.whenDefined()`. It returns a Promise that resolves when the element becomes defined.

```
customElements.whenDefined('app-drawer').then(() => {
  console.log('app-drawer defined');
});
```

Example - delay work until a set of child elements are upgraded

```
<share-buttons>
  <social-button type="twitter"><a href="...">Twitter</a></social-button>
  <social-button type="fb"><a href="...">Facebook</a></social-button>
  <social-button type="plus"><a href="...">G+</a></social-button>
</share-buttons>
```

```
// Fetch all the children of <share-buttons> that are not defined yet.
let undefinedButtons = buttons.querySelectorAll(':not(:defined)');

let promises = [...undefinedButtons].map(socialButton => {
  return customElements.whenDefined(socialButton.localName);
));

// Wait for all the social-buttons to be upgraded.
Promise.all(promises).then(() => {
  // All social-button children are ready.
});
```

Note: I think of custom elements as being in a state of limbo before they're defined. The spec defines an element's state as `undefined`, `uncustomized`, or `custom`. Built-in elements like `<div>` are always `defined`.

Element-defined content

Custom elements can manage their own content by using the DOM APIs inside element code. Reactions come in handy for this.

Example - create an element with some default HTML:

```
customElements.define('x-foo-with-markup', class extends HTMLElement {
  connectedCallback() {
    this.innerHTML = "<b>I'm an x-foo-with-markup!</b>";
  }
  ...
});
```

Declaring this tag will produce:

```
<x-foo-with-markup>
 <b>I'm an x-foo-with-markup!</b>
</x-foo-with-markup>
```

Note: Overwriting an element's children with new content is generally not a good idea because it's unexpected. Users would be surprised to have their markup thrown out. A better way to add element-defined content is to use shadow DOM, which we'll talk about next.

Creating an element that uses Shadow DOM

Note: I'm not going to cover the features of Shadow DOM in this article, but it's a powerful API to combine with custom elements. By itself, Shadow DOM is composition tool. When it's used in conjunction with custom elements, magical things happen.

Shadow DOM provides a way for an element to own, render, and style a chunk of DOM that's separate from the rest of the page. Heck, you could even hide away an entire app within a single tag:

```
<!-- chat-app's implementation details are hidden away in Shadow DOM. -->
<chat-app></chat-app>
```

To use Shadow DOM in a custom element, call `this.attachShadow` inside your `constructor`:

```
let tmpl = document.createElement('template');
tmpl.innerHTML = `
  <style>:host { ... }</style> <!-- look ma, scoped styles -->
  <b>I'm in shadow dom!</b>
  <slot></slot>
`;

customElements.define('x-foo-shadowdom', class extends HTMLElement {
  constructor() {
    super(); // always call super() first in the constructor.

    // Attach a shadow root to the element.
    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.appendChild(tmpl.content.cloneNode(true));
  }
  ...
});
```

Note: In the above snippet we use a `template` element to clone DOM, instead of setting the `innerHTML` of the `shadowRoot`. This technique cuts down on HTML parse costs because the content of the template is only parsed once, whereas calling `innerHTML` on the `shadowRoot` will parse the HTML for each instance. We'll talk more about templates in the next section.

Example usage:

```
<x-foo-shadowdom>
  <p><b>User's</b> custom text</p>
</x-foo-shadowdom>

<!-- renders as -->
<x-foo-shadowdom>
  #shadow-root
```

```
    <b>I'm in shadow dom!</b>
    <slot></slot> <!-- slotted content appears here -->
</x-foo-shadowdom>
```

Creating elements from a `<template>`

For those unfamiliar, the `<template>` element allows you to declare fragments of the DOM which are parsed, inert at page load, and can be activated later at runtime. It's another API primitive in the web components family. **Templates are an ideal placeholder for declaring the structure of a custom element**.

Example: registering an element with Shadow DOM content created from a `<template>`:

```
<template id="x-foo-from-template">
  <style>
    p { color: green; }
  </style>
  <p>I'm in Shadow DOM. My markup was stamped from a &lt;template&gt;.</p>
</template>

<script>
  let tmpl = document.querySelector('#x-foo-from-template');
  // If your code is inside of an HTML Import you'll need to change the above
line to:
  // let tmpl = document.currentScript.ownerDocument.querySelector('#x-foo-
from-template');

  customElements.define('x-foo-from-template', class extends HTMLElement {
    constructor() {
      super(); // always call super() first in the constructor.
      let shadowRoot = this.attachShadow({mode: 'open'});
      shadowRoot.appendChild(tmpl.content.cloneNode(true));
    }
    ...
  });
</script>
```

These few lines of code pack a punch. Let's understand the key things going on:

1. We're defining a new element in HTML: `<x-foo-from-template>`
2. The element's Shadow DOM is created from a `<template>`
3. The element's DOM is local to the element thanks to the Shadow DOM
4. The element's internal CSS is scoped to the element thanks to the Shadow DOM

Styling a custom element

Even if your element defines its own styling using Shadow DOM, users can style your custom element from their page. These are called "user-defined styles".

```
<!-- user-defined styling -->
<style>
  app-drawer {
```

```
    display: flex;
  }
  panel-item {
    transition: opacity 400ms ease-in-out;
    opacity: 0.3;
    flex: 1;
    text-align: center;
    border-radius: 50%;
  }
  panel-item:hover {
    opacity: 1.0;
    background: rgb(255, 0, 255);
    color: white;
  }
  app-panel > panel-item {
    padding: 5px;
    list-style: none;
    margin: 0 7px;
  }
</style>

<app-drawer>
  <panel-item>Do</panel-item>
  <panel-item>Re</panel-item>
  <panel-item>Mi</panel-item>
</app-drawer>
```

You might be asking yourself how CSS specificity works if the element has styles defined within Shadow DOM. In terms of specificity, user styles win. They'll always override element-defined styling. See the section on Creating an element that uses Shadow DOM.

Pre-styling unregistered elements

Before an element is upgraded you can target it in CSS using the `:defined` pseudo-class. This is useful for pre-styling a component. For example, you may wish to prevent layout or other visual FOUC by hiding undefined components and fading them in when they become defined.

Example - hide `<app-drawer>` before it's defined:

```
app-drawer:not(:defined) {
  /* Pre-style, give layout, replicate app-drawer's eventual styles, etc. */
  display: inline-block;
  height: 100vh;
  opacity: 0;
  transition: opacity 0.3s ease-in-out;
}
```

After `<app-drawer>` becomes defined, the selector (`app-drawer:not(:defined)`) no longer matches.

Extending elements

The Custom Elements API is useful for creating new HTML elements, but it's also useful for extending other custom elements or even the browser's built-in HTML.

Extending a custom element

Extending another custom element is done by extending its class definition.

Example - create `<fancy-app-drawer>` that extends `<app-drawer>`:

```
class FancyDrawer extends AppDrawer {
  constructor() {
    super(); // always call super() first in the constructor. This also calls
the extended class' constructor.
    ...
  }

  toggleDrawer() {
    // Possibly different toggle implementation?
    // Use ES2015 if you need to call the parent method.
    // super.toggleDrawer()
  }

  anotherMethod() {
    ...
  }
}

customElements.define('fancy-app-drawer', FancyDrawer);
```

Extending native HTML elements

Let's say you wanted to create a fancier `<button>`. Instead of replicating the behavior and functionality of `<button>`, a better option is to progressively enhance the existing element using custom elements.

A **customized built-in element** is a custom element that extends one of the browser's built-in HTML tags. The primary benefit of extending an existing element is to gain all of its features (DOM properties, methods, accessibility). There's no better way to write a progressive web app than to **progressively enhance existing HTML elements**.

Note: Only Chrome 67 supports customized built-in elements (status) right now. Edge and Firefox will implement it, but Safari has chosen not to implement it. This is unfortunate for accessibility and progressive enhancement. If you think extending native HTML elements is useful, voice your thoughts on 509 and 662 on Github.

To extend an element, you'll need to create a class definition that inherits from the correct DOM interface. For example, a custom element that extends `<button>` needs to inherit from `HTMLButtonElement` instead of `HTMLElement`. Similarly, an element that extends `<img>` needs to extend `HTMLImageElement`.

Example - extending `<button>`:

```
// See https://html.spec.whatwg.org/multipage/indices.html#element-interfaces
// for the list of other DOM interfaces.
class FancyButton extends HTMLButtonElement {
  constructor() {
    super(); // always call super() first in the constructor.
    this.addEventListener('click', e => this.drawRipple(e.offsetX,
e.offsetY));
  }

  // Material design ripple animation.
  drawRipple(x, y) {
    let div = document.createElement('div');
    div.classList.add('ripple');
    this.appendChild(div);
    div.style.top = `${y - div.clientHeight/2}px`;
    div.style.left = `${x - div.clientWidth/2}px`;
    div.style.backgroundColor = 'currentColor';
    div.classList.add('run');
    div.addEventListener('transitionend', e => div.remove());
  }
}

customElements.define('fancy-button', FancyButton, {extends: 'button'});
```

Notice that the call to `define()` changes slightly when extending a native element. The required third parameter tells the browser which tag you're extending. This is necessary because many HTML tags share the same DOM interface. `<section>`, `<address>`, and `<em>` (among others) all share `HTMLElement`; both `<q>` and `<blockquote>` share `HTMLQuoteElement`; etc.. Specifying `{extends: 'blockquote'}` lets the browser know you're creating a souped-up `<blockquote>` instead of a `<q>`. See the HTML spec for the full list of HTML's DOM interfaces.

Note: Extending `HTMLButtonElement` endows our fancy button with all the DOM properties/methods of `<button>`. That checks off a bunch of stuff we don't have to implement ourselves: `disabled` property, `click()` method, `keydown` listeners, `tabindex` management. Instead, our focus can be progressively enhancing `<button>` with custom functionality, namely, the `drawRipple()` method. Less code, more reuse!

Consumers of a customized built-in element can use it in several ways. They can declare it by adding the `is=""` attribute on the native tag:

```
<!-- This <button> is a fancy button. -->
<button is="fancy-button" disabled>Fancy button!</button>
```

create an instance in JavaScript:

```
// Custom elements overload createElement() to support the is="" attribute.
let button = document.createElement('button', {is: 'fancy-button'});
button.textContent = 'Fancy button!';
```

```
button.disabled = true;
document.body.appendChild(button);
```

or use the `new` operator:

```
let button = new FancyButton();
button.textContent = 'Fancy button!';
button.disabled = true;
```

Here's another example that extends `<img>`.

Example - extending `<img>`:

```
customElements.define('bigger-img', class extends Image {
  // Give img default size if users don't specify.
  constructor(width=50, height=50) {
    super(width * 10, height * 10);
  }
}, {extends: 'img'});
```

Users declare this component as:

```
<!-- This <img> is a bigger img. -->
<img is="bigger-img" width="15" height="20">
```

or create an instance in JavaScript:

```
const BiggerImage = customElements.get('bigger-img');
const image = new BiggerImage(15, 20); // pass constructor values like so.
console.assert(image.width === 150);
console.assert(image.height === 200);
```

Misc details

Unknown elements vs. undefined custom elements

HTML is lenient and flexible to work with. For example, declare `<randomtagthatdoesntexist>` on a page and the browser is perfectly happy accepting it. Why do non-standard tags work? The answer is the HTML specification allows it. Elements that are not defined by the specification get parsed as `HTMLUnknownElement`.

The same is not true for custom elements. Potential custom elements are parsed as an `HTMLElement` if they're created with a valid name (includes a "-"). You can check this in a browser that supports custom elements. Fire up the Console: Ctrl+Shift+J (or Cmd+Opt+J on Mac) and paste in the following lines of code:

```
// "tabs" is not a valid custom element name
document.createElement('tabs') instanceof HTMLUnknownElement === true
```

```
// "x-tabs" is a valid custom element name
document.createElement('x-tabs') instanceof HTMLElement === true
```

API reference

The `customElements` global defines useful methods for working with custom elements.

`define(tagName, constructor, options)`

Defines a new custom element in the browser.

Example

```
customElements.define('my-app', class extends HTMLElement { ... });
customElements.define(
  'fancy-button', class extends HTMLButtonElement { ... }, {extends:
'button'});
```

`get(tagName)`

Given a valid custom element tag name, returns the element's constructor. Returns `undefined` if no element definition has been registered.

Example

```
let Drawer = customElements.get('app-drawer');
let drawer = new Drawer();
```

`whenDefined(tagName)`

Returns a Promise that resolves when the custom element is defined. If the element is already defined, resolve immediately. Rejects if the tag name is not a valid custom element name.

Example

```
customElements.whenDefined('app-drawer').then(() => {
  console.log('ready!');
});
```

History and browser support

If you've been following web components for the last couple of years, you'll know that Chrome 36+ implemented a version of the Custom Elements API that uses `document.registerElement()` instead of `customElements.define()`. That's now considered a deprecated version of the standard, called v0. `customElements.define()` is the new hotness and what browser vendors are starting to implement. It's called Custom Elements v1.

If you happen to be interested in the old v0 spec, check out the html5rocks article.

Browser support

Chrome 54 (status), Safari 10.1 (status), and Firefox 63 (status) have Custom Elements v1. Edge has begun development.

To feature detect custom elements, check for the existence of `window.customElements`:

```
const supportsCustomElementsV1 = 'customElements' in window;
```

Polyfill

Until browser support is widely available, there's a standalone polyfill available for Custom Elements v1. However, we recommend using the webcomponents.js loader to optimally load the web components polyfills. The loader uses feature detection to asynchronously load only the necessary pollyfills required by the browser.

Note: If your project transpiles to or uses ES5, be sure to see the notes on including custom-elements-es5-adapter.js in addition to the polyfills.

Install it:

```
npm install --save @webcomponents/webcomponentsjs
```

Usage:

```
<!-- Use the custom element on the page. -->
<my-element></my-element>

<!-- Load polyfills; note that "loader" will load these async -->
<script src="node_modules/@webcomponents/webcomponentsjs/webcomponents-
loader.js" defer></script>

<!-- Load a custom element definitions in `waitFor` and return a promise -->
<script type="module">
  function loadScript(src) {
    return new Promise(function(resolve, reject) {
      const script = document.createElement('script');
      script.src = src;
      script.onload = resolve;
      script.onerror = reject;
      document.head.appendChild(script);
    });
  }

  WebComponents.waitFor(() => {
    // At this point we are guaranteed that all required polyfills have
    // loaded, and can use web components APIs.
    // Next, load element definitions that call `customElements.define`.
    // Note: returning a promise causes the custom elements
    // polyfill to wait until all definitions are loaded and then upgrade
    // the document in one batch, for better performance.
    return loadScript('my-element.js');
```

```
  });
</script>
```

Document Object Model

DOM - It's a way of representing a structured document via objects. It is cross-platform and language-independent convention for representing and interacting with data in HTML, XML, and others. Web browsers handle the DOM implementation details, so we can interact with it using JavaScript and CSS.

VIRTUAL DOM

Virtual DOM is any kind of representation of a real DOM. Virtual DOM is about avoiding unnecessary changes to the DOM, which are expensive performance-wise, because changes to the DOM usually cause re-rendering of the page. It allows to collect several changes to be applied at once, so not every single change causes a re-render, but instead re-rendering only happens once after a set of changes was applied to the DOM.

SHADOW DOM

Shadow DOM is mostly about encapsulation of the implementation. A single custom element can implement more-or-less complex logic combined with more-or-less complex DOM. *Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree.*

Shadow DOM is a tool used to build component-based apps and websites. Shadow DOM comes in small pieces, and it doesn't represent the whole Document Object Model. We can see it as a subtree or as a separate DOM for an element. Shadow DOM can be imaged like bricks from which the DOM is created.

The main difference between DOM and Shadow DOM is how it's created and how it behaves. Normally DOM nodes which we create are placed inside other elements, like in the tree we saw before. In the case of Shadow DOM, we create a scoped tree, which is connected to the element but separated from the children elements. It's called shadow tree and the element it's attached to

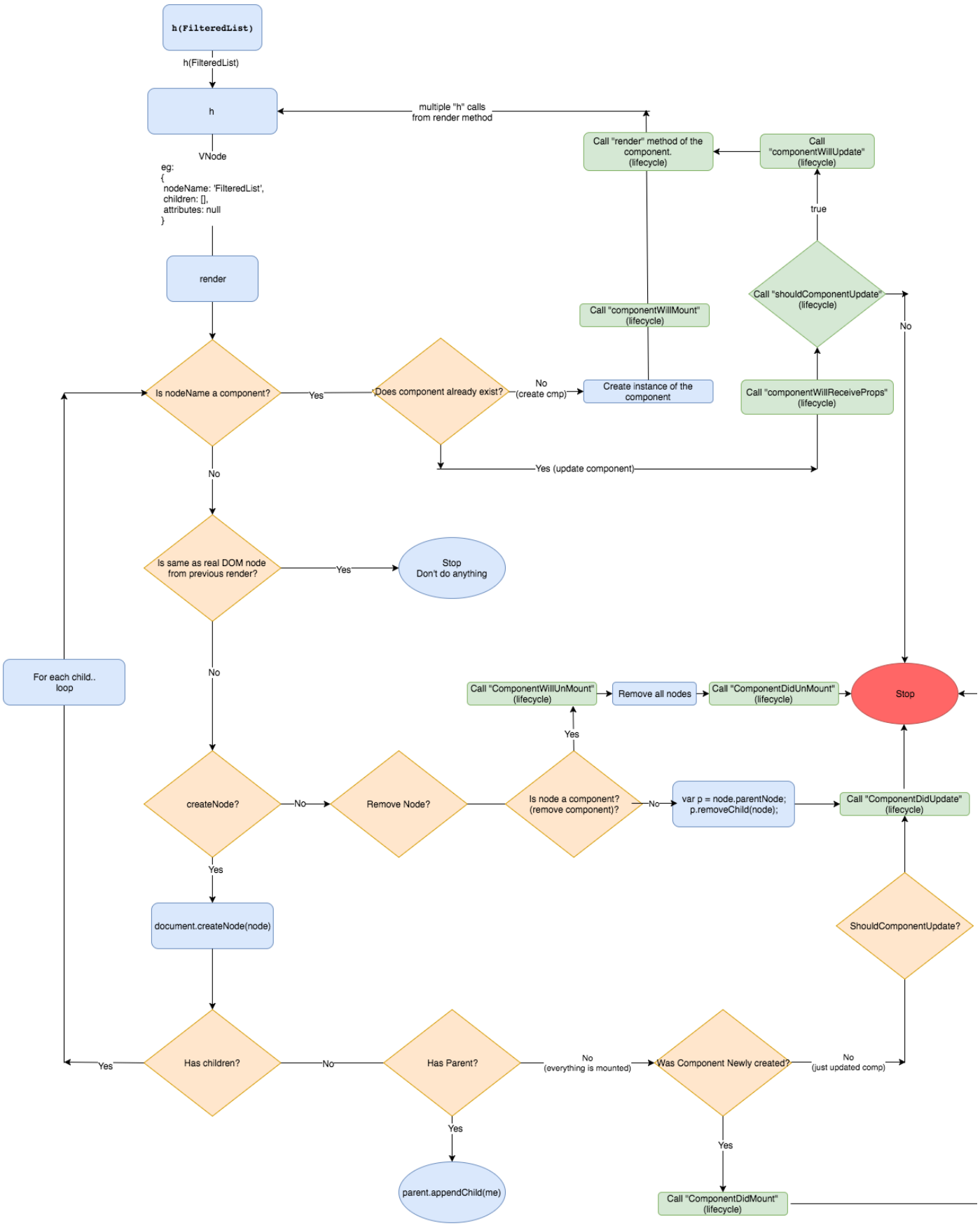
is called shadow host. And here we come to a great advantage of the Shadow DOM, everything which we will add to Shadow DOM is local, even styles.

why Shadow DOM is so useful and what issues it solves. First of all, it isolates the DOM, so the DOM of the component is a separate element which won't appear in a global DOM. Another issue it helps with is scoping of the CSS, which means styles created inside the single Shadow DOM element are isolated and stays in the scope of that Shadow DOM. It simplifies styling a lot as we don't have to worry a lot about naming space and we can use simple selectors and class names. Also, we can think of the application as it is built from chunks (it is based on the components actually) and not as a one massive, global object.

Shadow DOM can affect the performance of the application. As said at the beginning of the article, there are a lot of performance issues while we want to manipulate the DOM, because every change will make a re-rendering of the whole object. In the case of Shadow DOM browser knows which part should be updated.

Differences between Shadow DOM and Virtual DOM

The only thing which is common for both is that they help with performance issues. Both create a separate instance of the Document Object Model; besides this, both concepts are different. Virtual DOM is creating a copy of the whole DOM object, and Shadow DOM creates small pieces of the DOM object which has their own, isolated scope for the element they represent.


h(FilteredList)
h(FilteredList)
h
multiple "h" calls
from render method
VNode
eg:
{
nodeName: 'FilteredList',
children: [],
attributes: null
}
render
Call "render" method of the component.
(lifecycle)
Call "componentWillUpdate"
(lifecycle)
true
Call "shouldComponentUpdate"
(lifecycle)
No
Call "componentWillMount"
(lifecycle)
Is nodeName a component?
Yes
Does component already exist?
No
(create cmp)
Create instance of the component
Call "componentWillReceiveProps"
(lifecycle)
Yes (update component)
No
Is same as real DOM node
from previous render?
Yes
Stop
Don't do anything
For each child..
loop
No
Call "ComponentWillUnMount"
(lifecycle)
Remove all nodes
Call "ComponentDidUnMount"
(lifecycle)
Stop
Yes
createNode?
No
Remove Node?
Is node a component?
(remove component)?
No
var p = node.parentNode;
p.removeChild(node);
Call "ComponentDidUpdate"
(lifecycle)
Yes
document.createNode(node)
ShouldComponentUpdate?
Has children?
Yes
No
Has Parent?
No
(everything is mounted)
Was Component Newly created?
No
(just updated comp)
Yes
parent.appendChild(me)
Yes
Call "ComponentDidMount"
(lifecycle)

Shadow DOM

Heads Up! It's all about the V1 Spec.

This document is more yours than it is mine. It makes me happy that it has been able to help people. To do better I moved this document from the original gist to this repo so multiple people can work together and improve it.

If you like what you find here, please create issues with ideas as to what more can we add to this repository. Like examples, images, graphical representations for the terminologies, etc. via issues. Let's make everyone love the platform :).

Shadow DOM is designed as a tool for building component-based apps. Therefore, it brings solutions for common problems in web development:

- **Isolated DOM**: A component's DOM is self-contained (e.g. `document.querySelector()` won't return nodes in the component's shadow DOM).
- **Scoped CSS**: CSS defined inside shadow DOM is scoped to it. Style rules don't leak out and page styles don't bleed in.
- **Composition**: Design a declarative, markup-based API for your component.
- **Simplifies CSS** - Scoped DOM means you can use simple CSS selectors, more generic id/class names, and not worry about naming conflicts.
- **Productivity** - Think of apps in chunks of DOM rather than one large (global) page.

What's New?

Translations

Chinese - https://github.com/Tencent/omi/blob/master/tutorial/shadow-dom-in-depth.cn.md - Thanks @eyea.

- Examples. I am putting examples that'll help everyone understand it better, step by step. Check them out.

Let me know if a "Everything you need to know about Custom Elements" document like this one would help you. If so, I'll put one up .

Browser Support

- Chrome : Works
- Firefox : Works
- Opera : Works
- Safari : Works but few things are buggy.
- Edge : Under Consideration.

Comprehensive browser support info can be found here: https://caniuse.com/#feat=shadowdomv1.

Introduction

In a nutshell, Shadow DOM enables local scoping for HTML & CSS.

Shadow DOM fixes CSS and DOM. It introduces scoped styles to the web platform. Without tools or naming conventions, you can bundle CSS with markup, hide implementation details, and author self-contained components in vanilla JavaScript. - https://developers.google.com/web/fundamentals/web-components/shadowdom

It's like its own little world which hardly affects or gets affected by the outside world.

It's what you write as a **component author** to abstract away the implementation details of your component. It can also decide what to do with the user-provided **light DOM**.

Terminologies

- DOM : What we get over the wire (or wireless :|) is a string of text. To render something on the screen, the browsers have to parse that string of text and convert it into a data model so it can understand things better. It also preserves the hierarchy from the original string by putting those parsed objects in a tree structure.

We need to do that to make the machines understand our documents better. This tree like data model of our document is called Document Object Model.

- Component Author : The person who creates a component and defines how it works. Generally the person who writes a lot of shadow DOM code. Example - Browser vendors create the `input` element.

- Component User : Well, they use components built by authors. They can pass in light DOM and set attributes and properties on your component. They can even extend the internals of a component if they want. Example - we, the users who use the `input` element.

- Shadow Root: It's what gets attached to an element to give that element its shadow DOM. Technically it's a non-element node, a special kind of *DocumentFragment*.

```
<custom-picture>
```

```
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ISOLATION
    #shadow-root
        ...
    ____________________________________________ DOCUMENT FRAGMENT

    <!--LIGHT DOM-->
</custom-picture>
```

Throughout the document, I have put shadow root inside those weird ASCII boundaries. This will put more emphasis on thinking how they are actually document fragments that have a wall around them.

- Shadow Host: The element to which the shadow root gets attached. A host can access its shadow root via a property on itself: `.shadowRoot`.

- Shadow Tree : All the elements that go into the Shadow Root, which is scoped from outside world, is called Shadow Tree.

The elements in a shadow tree are **not** descendants of the shadow host in general (including for the purposes of Selectors like the descendant combinator) - Spec

- Light DOM: - The set of DOM elements we can sandwich between the opening and closing tags. - The DOM that lives outside shadow DOM. - The DOM, the *user* of your element writes. - The DOM that is the actual children of your element.

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
    ___________________________

    <!--Light DOM-->
    <img src="https://path.to/a-kitten.png">
    <cite>A Nice Kitten!</cite>
    <!--Light DOM ends-->
</custom-picture>
```

- DocumentFragment:

The DocumentFragment interface represents a minimal document object that has no parent. It is used as a lightweight version of Document to store a segment of a document structure comprised of nodes just like a standard document. The key difference is that because the document fragment isn't part of the actual DOM's structure, changes made to the fragment don't affect the document, cause reflow, or incur any performance impact that can occur when changes are made. - [MDN]

How to create Shadow DOM?

```
<div class="dom"></div>
let el = document.querySelector('.dom');
```

```
el.attachShadow({ mode: 'open' });
// Just like prototype & constructor bi-directional references, we have...
el.shadowRoot; // the shadow root.
el.shadowRoot.host; // the element itself.

// Put something in shadow DOM
el.shadowRoot.innerHTML = 'Hi I am shadowed!';

// Like any other normal DOM operation.
let hello = document.createElement('span');
hello.textContent = 'Hi I am shadowed but wrapped in span';
el.shadowRoot.appendChild(hello);
```

Off Topic Question - Could we use `append()` instead of `appendChild()` ?

Yes! But here are the differences from MDN.

- `ParentNode.append()` allows you to also append DOMString object, whereas `Node.appendChild()` only accepts Node objects.
- `ParentNode.append()` has no return value, whereas `Node.appendChild()` returns the appended Node object.
- `ParentNode.append()` can append several nodes and strings, whereas `Node.appendChild()` can only append one node.

What happens if we use an `input` element instead of the div to attach the shadow DOM?

Well, it doesn't work. Because the browser already hosts its own shadow DOM for those elements. Bunch of red colored english alphabets will be thrown at console's face.

Notes & Tips

- Shadow DOM cannot be removed once created; it can only be replaced with a new one.
- If you are creating a custom element, you should be creating the shadowRoot in its constructor. It can also be probably called in `connectedCallback()` but I am not sure if that introduces performance problems or any other problems. ♂□
- To see how browsers implement shadow DOM for elements like `input` or `textarea`, Go to `DevTools > Settings > Elements > [x] Show user agent shadow DOM`.

Shadow DOM Modes

Open

You saw the `{mode: "open"}` in the `attachShadow()` method right? Yeah! That's it. What open mode does is that it provides a way for us to reach into the shadow DOM to access the element's contents. It also lets us access the host element from within the shadow DOM.

This is done by the two implicit properties created when we call `attachShadow()` in `open` mode.

1. The element gets a property called `shadowRoot` which points to the shadow DOM being attached to it.
2. The `shadowRoot` gets a property called `host` pointing to the element itself.

```
// From the "How to create shadow DOM" example
el.attachShadow({ mode: 'open' });
// Just like prototype & constructor bi-directional references, we have...
el.shadowRoot; // the shadow root.
el.shadowRoot.host; // the el itself.
```

Closed

Pass `{mode: "closed"}` to `attachShadow()` to create a closed shadow DOM. It makes the shadow DOM inaccessible from JS.

```
el.shadowRoot; // null
```

So which one should I use?

Almost always use `open` mode shadow DOMs because they make it possible for both the component author and user to change things how they want.

Remember we did `el.shadowRoot` stuff up there? Yeah! That won't work with `closed` mode. The element doesn't get any reference to its shadow DOM, which is a problem when you want to access the shadow DOM for manipulation.

```
class CustomPicture extends HTMLElement {
  constructor() {
    this.attachShadow({ mode: 'open' }); // this.shadowRoot exists. Add or
remove stuff in there using this ref.
    this.attachShadow({ mode: 'closed' }); // this.shadowRoot returns null.
Bummer!
  }
}
// You could always do the following in your constructor.
// but it totally defies the purpose of using closed mode.
this._shadowRoot = this.attachShadow({ mode: 'closed' });
```

Also, closed mode isn't a security mechanism. It just gives a fake sense of security. Nobody can stop someone from modifying how `Element.prototype.attachShadow()` works.

Styles inside Shadow DOM.

- They are scoped.
- They don't leak out.
- They can have simple names.
- They are cool, be like them.

```
<custom-picture>
    #shadow-root
```

```
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        <style>
            /*Applies only to spans inside shadow DOM. Doesn't leak out.*/
            span {
                color: red;
            }
        </style>
        <span>Hello!</span>
        ________________________________________________________________________
</custom-picture>
```

Oh! oh! So can we also include a stylesheet?

Yeeeeaaaah.. but not in all browsers.

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
        <!--All styles coming from custom-picture.css will be scoped inside
this shadow root-->
        <link rel="stylesheet" href="custom-picture.css">
        <span>Hello!</span>
    ____________________________________________________
```

Does it get affected by global CSS?

Yeah. In some ways. Only the properties that are inherited will make their way through the shadow DOM boundary. Examples:

- color
- background
- font-family, etc.

The * selector also affects things because * means all elements and that includes the element to which you are attaching the shadow root to (the host element). Things which get applied to the host and can be inherited will pass the shadow DOM boundary to apply to inner elements.

Styles Terminologies

- :host: targets the host element. BUT!

```
/* winner */
custom-picture {
    background: red;
}
/* loser */
#shadow-root
    <style>
        :host {
            background: green;
        }
    </style>
```

- :host(`<selector>`): Does the component **host** match the **selector** ? Basically, allows us to target different states of the same host. Examples:

```
:host([disabled]) {
  ...;
}

:host(:focus) {
  ...;
}

:host(:focus) span {
  /*change all spans inside the element when the host has focus*/
}
```

- :host-context(`<selector>`): Is the **host** a descendant of **selector** ? Let us change a component's styles based on how the parent looks. General application could be in theming. Examples:

```
:host-context(.light-theme) {
  background: lightgray;
}

:host-context(.dark-theme) {
  background: darkgray;
}

/*You can also do...*/
:host-context(.aqua-theme) > * {
  color: aqua; /* lame */
}
```

A note about :host() and :host-context()

Both of these functional pseudo classes can only take `<compound-selector>` but not a combinator like space or ">" or "+" etc. In case of `:host()` that means you can only choose the attributes and other aspects of the host element.

In case of `:host-context()` that means you can choose the attributes and other aspects of one particular element which is an ancestor of the `:host`. Only one!

Do the modes in `attachShadow()` affect how styles get applied/cascaded?

Nope! Only affects how we do things in JS.

How does the user-agent styles get applied to even the shadow DOM elements?

Based on how shadow root or in general DocumentFragment works, user-agent styles (global) shouldn't have applied to all the elements inside a shadow root. So how do they work?

From the spec...

`Window`s gain a private slot `[[defaultElementStylesMap]]` which is a map of local names to stylesheets. This makes it possible to write elements inside shadow root and still get the default browser styles applied to them.

Which styles to put in shadow DOM ?

The purpose of having styles inside a shadow DOM is to just have default styles and provide hooks via CSS custom properties so component users can make changes to those default styles via, [CSS custom properties](#) (a.k.a CSS variables).

```
<business-card>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
        <h1 class="card-title">Hardcoded Title - </h1>
    ---------------------------------------------------
</business-card>
/*Inside shadow DOM*/
.card-title {
  color: var(--card-title-color, #000);
}

/*Component users can then override this color as*/
business-card {
  --card-title-color: magenta;
}
```

Notes from the Specs

- A shadow host is outside of the shadow tree it hosts, and so would ordinarily be untargettable by any selectors evaluated in the context of the shadow tree (as selectors are limited to a single tree), but it is sometimes useful to be able to style it from inside the shadow tree context.
- To work around that problem, the shadow host is treated as replacing the shadow root node.
- When considered within its own shadow trees, the shadow host is featureless. Only the `:host`, `:host()`, and `:host-context()` pseudo classes are allowed to match it.

Events in Shadow DOM

Events work towards maintaining the encapsulation provided by the shadow DOM. Essentially, if an event occurs somewhere in the shadow DOM, to outside world it'll look as if that event has triggered from the host element itself and not a specific part of the shadow DOM. This is called **re-targeting of the event**.

Inside the shadow DOM, however, the events aren't retargeted and we can find out which specific element an event was associated with.

Say our flattened DOM tree looks like this:

```
<body>
    <custom-picture>
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        #shadow-root
            <button> Hello </button>
        ---------------------------------
    </custom-picture>
</body>
```

On click of the button, to body, or anywhere outside custom-picture, the `event.target` will point to the `<custom-picture>` itself.

If the shadow tree is open, calling `event.composedPath()` will return an array of nodes that the event traveled through.

Inside `<custom-picture>` however, the event target will be the button which was really clicked.

Most events bubble out of the shadow DOM boundary, and when they do they get re-targered. Some events aren't allowed to pass that boundary. Precisely these:

- abort
- error
- select
- change
- load
- reset
- resize
- scroll
- selectstart

Slots

Slots are a pretty big thing in Shadow DOM.

Slots are placeholders inside your component that users can fill with their own markup. - https://developers.google.com/web/fundamentals/web-components/shadowdom#slots

When creating custom components, we want to be able to provide only the necessary markup that goes into a particular component and use/group/style that as we want to as component authors.

The DOM that a component user provides is called light DOM and slots are the way we arrange, style, and group those elements.

There are two aspects of a slot:

- **Light DOM Elements :** They say which slot they wanna go into.

```
<custom-picture>
    <!--Light DOM <img> saying it should be put into the "profile-picture"
slot-->
    <img src="assets/user.svg" slot="profile-picture">
</custom-picture>
```

- **The Actual Slot :** The `<slot>` element, residing somewhere in shadow DOM with a name for itself to be found by light DOM.

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
        <slot name="profile-picture">
            <!--The <img> from the light DOM gets rendered here!-->
        </slot>
    ____________________________________________
```

A very important note about how light DOM and slots play together.

Elements are allowed to "cross" the shadow DOM boundary when a <slot> invites them in. These elements are called distributed nodes. Conceptually, distributed nodes can seem a bit bizarre. Slots don't physically move DOM; they render it at another location inside the shadow DOM. - https://developers.google.com/web/fundamentals/web-components/shadowdom#slots

What if I don't provide the `slot` attribute in the `<img>` in `<custom-picture>`?

You'll see nothing rendered. Here's why:

1. A host element that has shadow DOM, only renders stuff that goes inside its shadow DOM.
2. In order to get light DOM elements rendered, they need to be part of the shadow DOM.
3. The way we make them part of the shadow DOM is by putting them in slots.
4. In our example above, there's no element in light DOM that wants to go into a slot named `profile-picture`.
5. Since there's no one, the `<img>` from light DOM is not rendered.
6. Takeaway: named slots accommodate only those light DOM elements which specify they want to go into that particular slot.

What if I want to render all elements that don't say where they should go?

This will require us to have a general purpose slot in our shadow DOM. A slot without a name. Example -

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
```

```
        <!--General purpose slot, render every element from light DOM that
doesn't mention a slot name, here.-->
        <slot>
            <!--The <img> from the light DOM gets rendered here!-->
        </slot>

    ________________________________________
    <!--Light DOM-->
    <img src="assets/user.svg">
</custom-picture>
```

What if I add two unnamed slots?

Woah! Actually, we can have duplicate unnamed or named slots but they are essentially ignored, since light DOM elements will go into the first slot they match.

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
        <slot>
            <!--The <img> from the light DOM gets rendered here! Winner!-->
        </slot>
        <slot>
            <!--Doesn't come here!-->
        </slot>

    ________________________________________
    <!--Light DOM-->
    <img src="assets/user.svg">
</custom-picture>
```

What if there's a slot but no light DOM elements want to go there?

Nothing will be rendered unless there's fallback content provided by the slot itself. Providing fallback content is easy:

```
#shadow-root
    <slot name="nobody-comes-here">
        <h1> I'll show up when no slot content is provided!</h1>
    </slot>

    <style>
        /*And that fallback can be styled from within the shadow DOM just
like we do styles*/
        slot[name="nobody-comes-here"] h1 {
            color: #bada55;
        }
    </style>
```

So what are slotted elements and how can we style them?

Light DOM elements that go into a slot are called slotted elements. As mentioned above, these are also called distributed elements which cross the shadow DOM boundary.

These slotted elements can be styled using the `::slotted()` functional pseudo element. The syntax is as follows:

```
::slotted(<compound-selector >) {
  /* styles */
}
```

Example:

```
<custom-picture>
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    #shadow-root
        <slot>
            <!--The <img> from the light DOM gets rendered here!-->
        </slot>

        <style>
            /* find the slotted image and set their width and height */
            ::slotted(img) {
                width: 256px;
                height: 256px;
            }
        </style>
    ____________________________________________
    <!--Light DOM-->
    <img src="assets/user.svg">
</custom-picture>
```

Here's how the spec formally defines it:

The ::slotted() pseudo-element represents the elements assigned, after flattening, to a slot. This pseudo-element only exists on slots.

Flattened trees are here.

An important thing to remember is that only direct children of the host element can be assigned to a slot. For example:

```
<custom-picture>
    <div class="picture-wrapper">
        <!--This won't work! Slots can't pick descendants out of the host
element's light DOM tree and put them in.-->
        <img src="assets/user.svg" slot="assign-me" />
    </div>
</custom-picture>
```

I, however, don't know why that's not possible and what the reasons are behind it, so I created this bug.

How can I pass a light DOM element multiple levels down?

It may sound like we don't need to think about this scenario but it's often required.

```
<parent-element>
```

```
    <!--parent-element uses child-element in its shadow DOM and we want this
span to render inside that child-element's shadow DOM-->
    <span slot="parent-slot">Finally</span>
</parent-element>
```

Here's what our custom elements look like:

```
class ParentElement extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
        <!--We specify a slot property on the slot itself. Which specifies
where it goes in the child-element's shadow DOM-->
        <child-element>
            <slot name="parent-slot" slot="child-slot"></slot>
        </child-element>
    `;
  }
}

class ChildElement extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <slot name="child-slot">
        <!--The span with the thext "Finally" gets rendered here!-->
      </slot>
    `;
  }
}

window.customElements.define('parent-element', ParentElement);
window.customElements.define('child-element', ChildElement);
```

Slots have JavaScript API

- To find the elements that went into a slot - `slot.assignedNodes()`;
- To find out which slot an light DOM element is assigned to - `element.assignedSlot`;

ISOLATED CSS

The CSS `isolation` property defines whether or not an element will be turned into a new stacking context.

The `isolation` property is used to *isolate* a group of elements so that they do not blend with their backdrop.

When an element is isolated using the `isolation` property, a new stacking context is created. Elements inside that context will not blend with the element's backdrop anymore.

The `isolation` property can be used in conjunction with the `mix-blend-mode` property.

If you are using the `background-blend-mode` property, the `isolation` property is not needed since background layers must not blend with the content that is behind the element, instead they must act as if they are rendered into an isolated group (the element itself).

You can isolate elements to prevent them from blending with their backdrop using the `isolation` property. However, everything in CSS that creates a stacking context must be considered an isolated group. HTML elements themselves should not create groups. This implies that even if you do not explicitly isolate elements using `isolation`, they might still be isolated if a stacking-context-creating property is used.

- **Syntax:**
- `isolation: auto | isolate`

- **Initial:** auto
- **Applies To:** All elements. In SVG, it applies to container elements, graphics elements and graphics referencing elements.
- **Animatable:** no

Values

auto
: By default, elements get this value which implies that they are not isolated.

  However, even if the `isolation` property is set to `auto`, a group of elements can still be isolated if operations that cause the creation of stacking context are performed.

isolate
: This creates a new stacking context on an element, and isolates the group. However, elements inside the same stacking context will blend with their backdrop that lies within that context.

Examples

In the following example, we have a piece of text wrapped in a text wrapper, positioned on top of an image.

```
<div class="container">
    <img src="isolation-destination.jpg" alt="Yellow tree.">
    <div class="text-wrapper">
        <h1>SUNSHINE</h1>
    </div>
</div>
```

The text is blended with its backdrop using the `mix-blend-mode` property with the blend mode value `overlay`.

```
h1 {
    mix-blend-mode: overlay;
}
```

add this property extra

We can isolate the text from its backdrop by using the `isolation` property on the text wrapper—this will isolate the content of the wrapper (i.e. our text), and prevent it from blending with the image.

```
.text-wrapper {
    isolation: isolate;
}
```

```
DEMO
----
Isolated CSS/EXAMPLE1.HTML
```
Example2.html
Example3.html

<blend-mode>

The `<blend-mode>` CSS data type defines the formula that must be used to mix the colors of an element with its **backdrop**.

What Is Compositing?

Compositing is the combining of a graphic element with its *backdrop*.

A **backdrop** is **the content behind the element** and is what the element is composited with.

Compositing defines how what you want to draw will be blended with what is already drawn on the canvas. The **source** is what you want to draw, and the **destination** is what is already drawn (the backdrop).

So, if you have two elements, and these elements overlap, you can think of the element on top as being the source, and the parts of the element behind that lie beneath it, will be the destination.

Using different composite operations (there are 16 operations), you can specify which parts of the two overlapping elements will be drawn, and which will not.

REFER BELOW MENTIONED IMAGE BEFORE CONTINUATION.

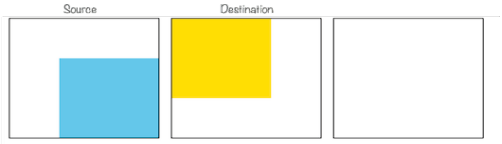
These composite operations are known as **Porter Duff compositing operations**. These operations specify what portions of the source and destination will be drawn, and blend modes specify how how the colors from the graphic element (source) and the backdrop (destination) interact. The illustrations in the above image are from the Compositing and Blending spec. In HTML5 Canvas context, these oprations are specified using the `globalCompositeOperation` property, and can be used to clip backgrounds to specific shapes, such as text, thus creating the effect of texture-filled text in Canvas. I have written about this process in this article over at Codrops.

Together, Porter Duff Compositing and blending form the overall compositing operation for intersecting elements. According to the specification, “typically, the blending step is performed first, followed by the Porter-Duff compositing step. In the blending step, the resultant color from the mix of the element and the the backdrop is calculated. The graphic element’s color is replaced with this resultant color. The graphic element is then composited with the backdrop using the specified compositing operator.”

Therefore, the way two intersecting or overlapping elements are handled is by blending their colors based on a blend mode, and then drawing only the parts specified by the composite operation.

Source Destination

Clear
No regions are enabled

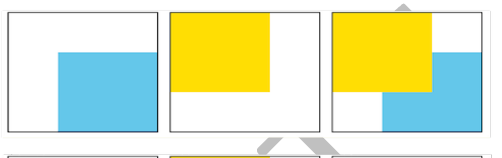


Copy
Only the source will be present

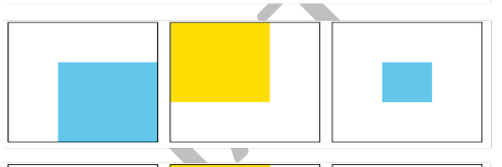
Destination
Only the destination will be present

Source-Over
Source is placed over the destination

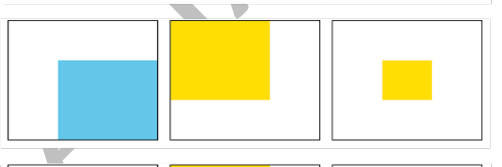
Destination-Over
Destination is placed over the source



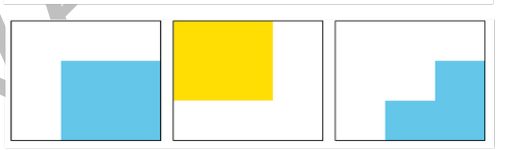
Source-In
The source that overlaps the destination, replaces the destination



Destination-In
The destination that overlaps the source, replaces the source



Source-out
Source is placed where it falls outside the destination



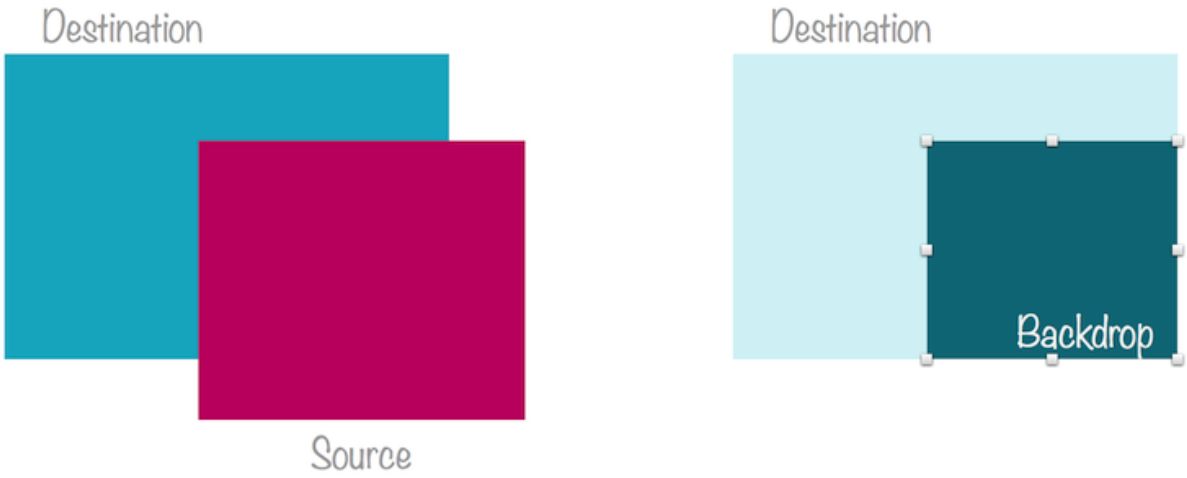
Destination-out
Destination is placed where it falls outside the source

Source-atop
Source which overlaps the destination, replaces the destination. Destination is placed elsewhere

Destination-atop
Destination which overlaps the source, replaces the source. Source is placed elsewhere

XOR
Destination which overlaps the source, replaces the source. Source is placed elsewhere

A backdrop is the content behind the element—known as *the source element*—and is what the element is composited with. The *destination* element is the element that lies behind the source element, and which the source overlaps with. The backdrop is the area where the color blending is done between the source and the destination.



The backdrop is the area where the color blending is done between the source and the destination.

A blend mode can be used to specify how an element's background image(s) and color(s) blend together—using the `background-blend-mode` property, and/or how two elements blend together—using the `mix-blend-mode` property.

Values

`normal`
: This is the default mode which specifies no blending. The blending formula simply selects the source color.

`multiply`
: The source color is multiplied by the destination color and replaces the destination. The resultant color is always at least as dark as either the source or destination color. **Multiplying any color with black results in black.** **Multiplying any color with white preserves the original color.**

`screen`
: Multiplies the complements of the backdrop and source color values, then complements the result. The result color is always at least as light as either of the two constituent

colors. **Screening any color with white produces white; screening with black leaves the original color unchanged.** The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.

`overlay`
: Multiplies or screens the colors, depending on the backdrop color value. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced but is mixed with the source color to reflect the lightness or darkness of the backdrop.

`darken`
: Selects the darker of the backdrop and source colors. The backdrop is replaced with the source where the source is darker; otherwise, it is left unchanged.

`lighten`
: Selects the lighter of the backdrop and source colors. The backdrop is replaced with the source where the source is lighter; otherwise, it is left unchanged.

`color-dodge`
: Brightens the backdrop color to reflect the source color. Painting with black produces no changes.

`color-burn`
: Darkens the backdrop color to reflect the source color. Painting with white produces no change.

`hard-light`
: Multiplies or screens the colors, depending on the source color value. The effect is similar to shining a harsh spotlight on the backdrop.

`soft-light`
: Darkens or lightens the colors, depending on the source color value. The effect is similar to shining a diffused spotlight on the backdrop.

`difference`
: Subtracts the darker of the two constituent colors from the lighter color. Painting with white inverts the backdrop color; painting with black produces no change.

`exclusion`
: Produces an effect similar to that of the Difference mode but lower in contrast. Painting with white **inverts the backdrop color**; painting with black produces no change.

`hue`
: Creates a color with the hue of the source color and the saturation and luminosity of the backdrop color.

`saturation`
: Creates a color with the saturation of the source color and the hue and luminosity of the backdrop color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.

`color`
: Creates a color with the hue and saturation of the source color and the luminosity of the backdrop color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images or tinting color images.

`luminosity`
: Creates a color with the luminosity of the source color and the hue and saturation of the backdrop color. This produces an inverse effect to that of the Color mode.
This mode is the one you can use to create monchorome “tinted” image effects like the ones you can see in different website headers.

Examples

The following example specifies a blend mode to be used to blend the background image and background color of an element:

```
.el {
    background-image: url('path/to/image.jpg');
    background-color: palevioletred;
    background-blend-mode: luminosity;
}
```

Browser Support

Refer to the `background-blend-mode` and `mix-blend-mode` property entries for information about browser support.

mix-blend-mode

The `mix-blend-mode` property is used to specify the `blend mode` for blending an element with its backdrop.

A backdrop is the content behind the element—known as *the source element*—and is what the element is composited with. The *destination* element is the element that lies behind the source element, and which the source overlaps with. The backdrop is the area where the color blending is done between the source and the destination.

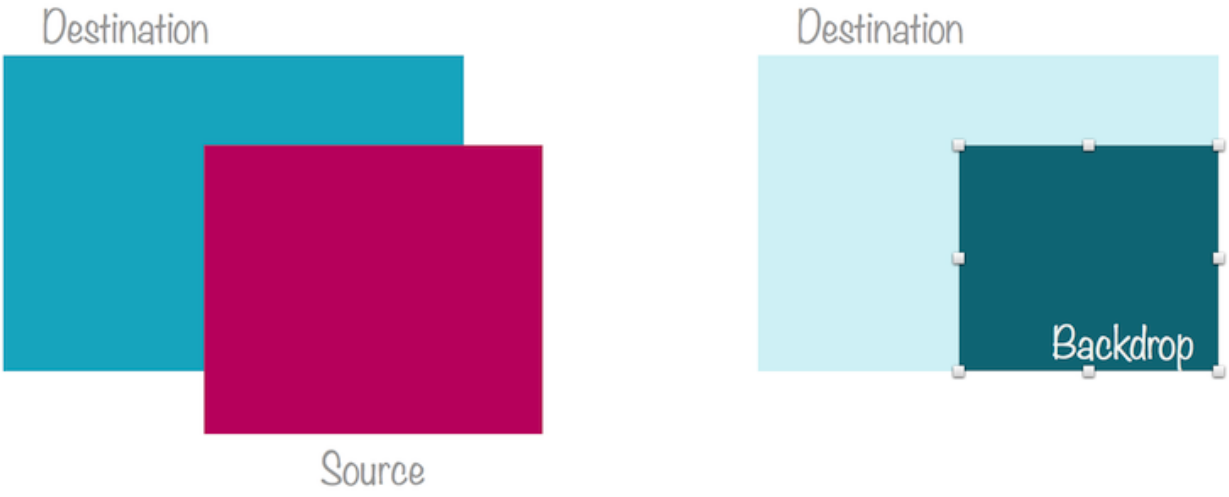
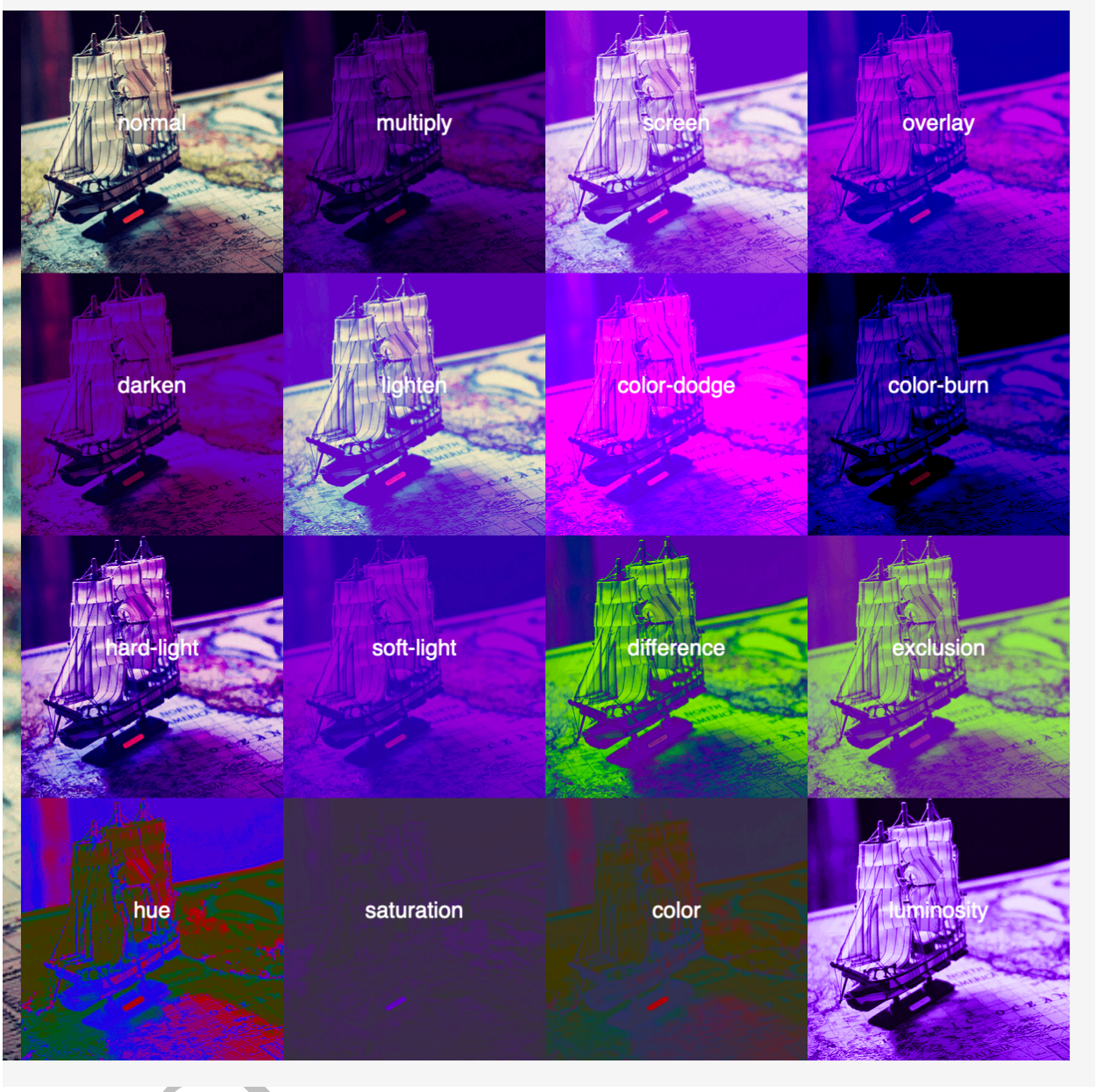


Image showing the area that defines an element's backdrop. (Source)

The backdrop is the area where the color blending is done between the so

urce and the destination.

The `mix-blend-mode` property can be used to mix together any element with its backdrop. For example, you can mix text with whatever (images, for example) it overlaps with, images with other images, a fixed header with the scrolling content of the page, etc.



COHERENT UI

Coherent UI is a graphical User Interface system specially designed for games and native apps. Game developers and UI artists can use standard modern HTML, CSS, and JavaScript to design and implement game interface and interaction. Coherent UI makes easy connecting gameplay, game objects, and systems to the UI. Game developers can focus on implementing core gameplay while quickly facilitating existing skills to implement the UI, as there is no special knowledge needed to use the system. Coherent UI is easy to integrate, and developers and artists can start using the library to make beautiful interfaces in a few minutes.

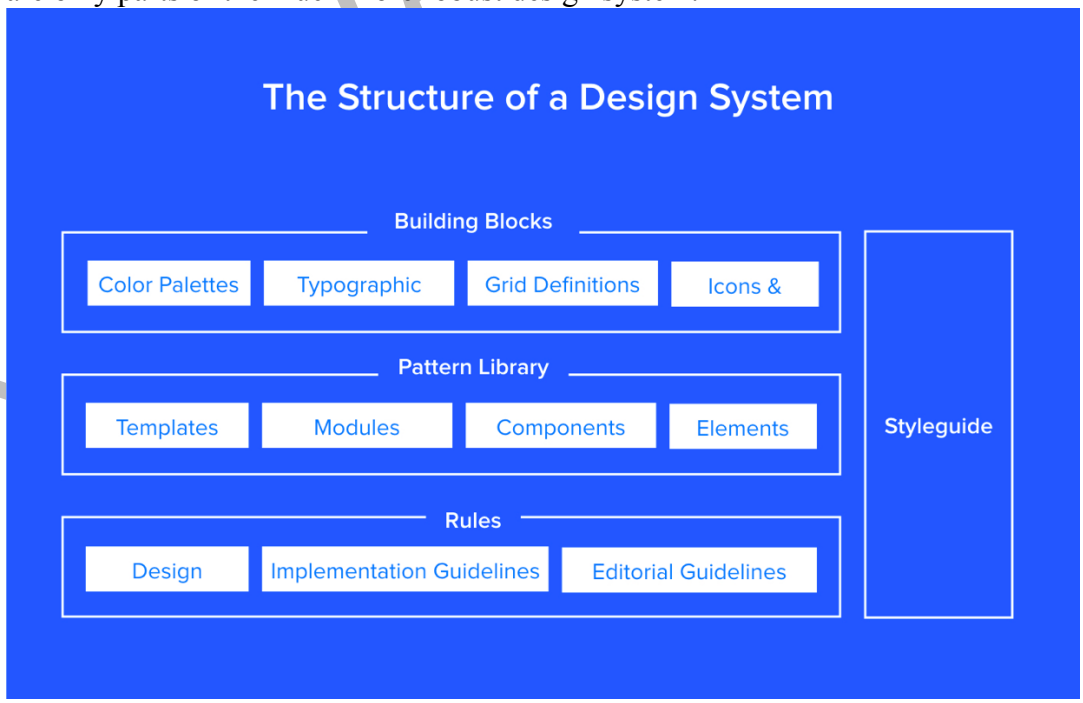
Graphical User Interface system
Product category:
IT- Gaming

- Coherent GT
- Coherent UI
- Coherent .NET
- Coherent UI 1.x

Design system definitions

First, let's get the definitions straight so we can learn how our concepts are connected and how they can be used to create better products.

- **Design System** – the complete set of design standards, documentation, and principles along with the toolkit (UI patterns and code components) to achieve those standards.
- **Pattern Library** – A subclass in the design system, this is the set of design patterns for use across a company.
- **Style guide** – Another subclass in the design system, this static documentation describes the design system itself: how products should look and feel, use cases for UI patterns, correct typographic scales, etc.
- Bottom line: A pattern library (e.g. set of symbols and assets in Sketch) and style guide are only parts of the much more robust design system.



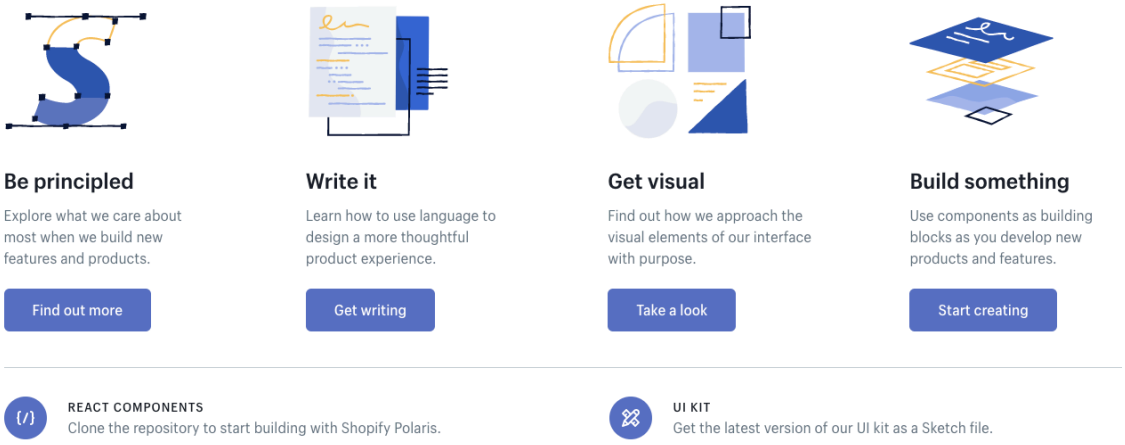
How it all connects

First, keep your focus broad and think about the idea of design systems.

At the broadest level, a design system is a living entity containing the common linguistics, principles, and tools to help teams build products coherently. As Nathan Curtis says, a "design system isn't a project, it's a product serving products".

Shopify's Polaris Design System is one of the best examples. On the homepage, they've already broken down all the core sections:

- **Product principles** – What is the purpose and soul behind all the products?
- **Written content –** How should the product's interface copy look and feel?
- **Visual properties** – What should the "skin" of the product look and feel like?
- **Components** – What are the UI patterns and code components needed to build products coherently across devices?



Creating a design system

Now that you know what these terms mean and how they work together, let's quickly review how to build a design system. Here's a quick overview of the steps involved from our 50-page e-book *Creating a Design System: The 100-Point Process Checklist*.

1. Create the UI inventory: First list and describe all of the design patterns currently used in your interface and note the inconsistencies therein.

2. Get support of the organization: Present your findings and explain the utility of a common design language to everyone. As explained in our Evangelizing Design Systems templates, estimate the number of design and engineering hours wasted on redundant work and how product coherence can improve NPS scores.

3. Establish design principles: Codify your practices. You're now starting to work on the style guide for the design system.

4. Build the color palette: When building the UI inventory, we found 116 different shades of grey that needed consolidation. Create the palette and its naming convention.

5. Build the typographic scale: You can optimize the scale to serve existing styles, or you might try to build a harmonious scale using the golden ratio or major second. When building the scale, don't forget that you're not only setting the size of the font, but also weight, line-height and other properties.

6. Implement icons library and other styles: Decide which icons from the UI inventory will become part of the design system, then standardize the implementation.

7. Start building your first patterns: This is the task that will never end. Patterns should always either reflect the truth about the product, or reflect the aspirational state of the product in the near future.

What is a pattern library?

A pattern library is a collection of user interface design elements. The site UI-Patterns describes these user interface design patterns as:

Recurring solutions that solve common design problems.

Still confused? Well, that is not surprising. Web designers like to make things sound more complicated than they are!

Essentially a pattern library is a collection of design elements that appear multiple times on a site. Typical examples might include:

- Slideshows
- Navigation
- Social media features
- News Listings
- Related links
- Carousels

The list could and does go on.

A pattern library, documents all of these ‘patterns’ (also often known as modules) and defines how they behave, what they look like and how they are coded.

A pattern library is a collection of design elements that can be reused across a website.

Examples of Pattern libraries that you might want to check out include:

- Mailchimp
- BBC Gel
- Starbucks
- Yahoo!

Of course, pattern libraries do not spontaneously appear, they need creating, and that takes effort. Why then is it worth your time to create a pattern library?

Your pattern library should be a part of a larger design system. Learn more about design systems and get help creating them.

Learn about design systems

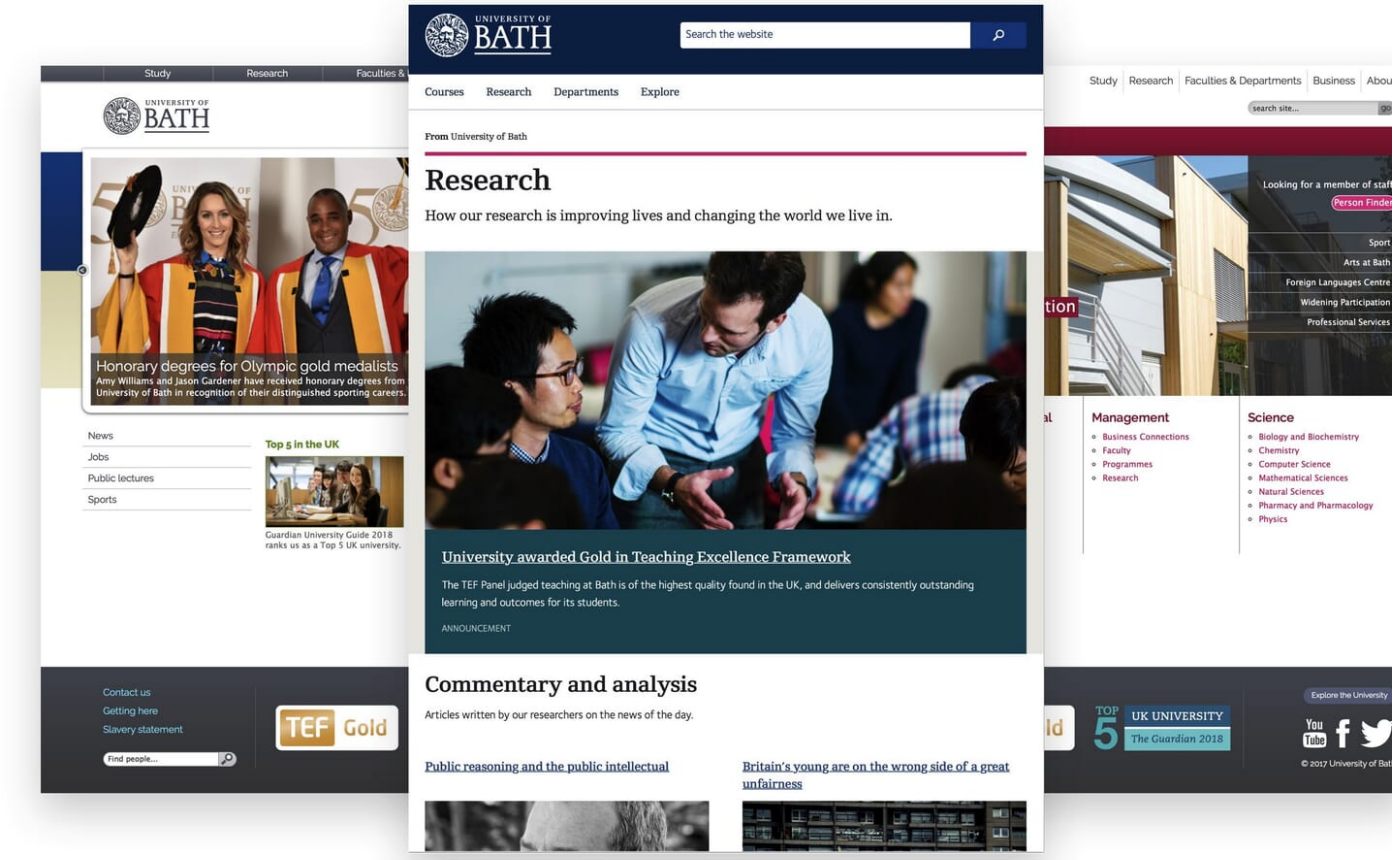
Why you need a pattern library?

Once a website reaches a certain size and complexity (especially if many subsites are involved) the argument for a pattern library are overwhelming. These benefits are three-fold:

A Pattern library ensure a consistent user interface

Big sites are developed by different people over a prolonged period and revised regularly. That almost always leads to a fragmented user interface unless there is something in place to ensure consistency.

You only need to visit any large site to see examples of this. Navigation shifts position, form elements are formatted differently and even typography changes. That happens because it is easier to guess how a button might look than find out how somebody styled it previously. A pattern library changes this by offering a straightforward way to duplicate existing design and functionality on any page of the site.



On larger sites it is easy for inconsistencies to creep in when there isn't a pattern library to set standards.

A pattern library facilitates reusability

Large organisations often have multiple web teams working across the company reporting into different departments. Often these teams work in isolation and so end up reinventing the wheel at considerable cost.

Having a central pattern library developed in collaboration between all of these web professionals means that the organisation can reuse functionality and design, so keeping costs down.

If one web developer creates a new pattern for a particular requirement in their area of responsibility, this can now be shared with the whole group and is also permanently available for future projects.

A pattern library lets you build from existing elements like building something out of predefined Lego bricks.

Tweet this

Once the majority of patterns are in place creating a new site or subsection becomes a mere matter of combining these patterns, in much the same way you build something out of existing Lego bricks.

Tips for creating a pattern library

As you will have seen from the examples posted above, there is nothing particularly special about a pattern library. It is essentially a collection of elements, their associated code and a few notes.

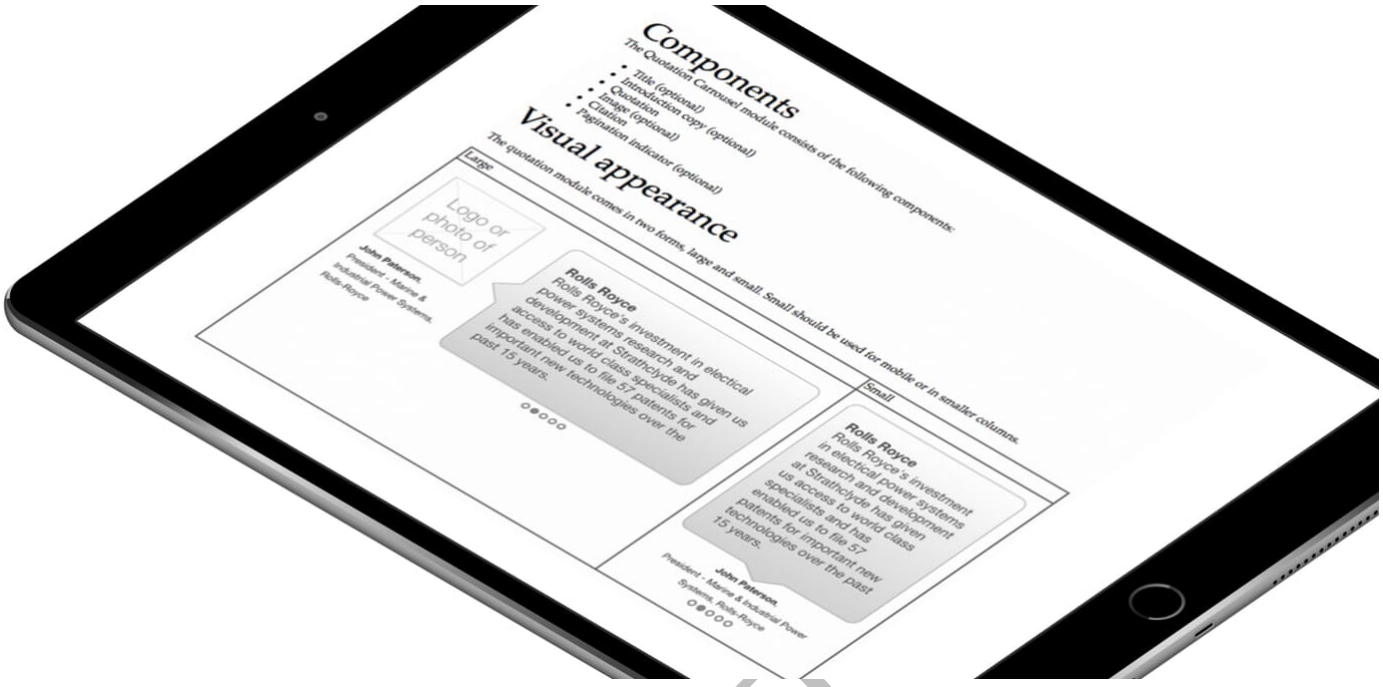
How you implement a pattern library is entirely up to you. However, I thought it might be useful if I share a few of the things I have discovered about working with pattern libraries.

Think about your pattern library from the start

The temptation is only to document a pattern library once you have built the site. However, this somewhat undermines the point of having a pattern library.

When working on a pattern library, I tend to put the skeleton together before anybody writes a line of code. We create a library featuring wireframes of individual patterns, notes on how that pattern works and other considerations, while still at the prototyping phase. That is helpful for the designer and developer, acting as a functional specification of sorts.

As the developer starts writing code this pattern can then get fleshed out with the final design and associated code. This approach is considerably easier than putting everything together at the end and also allows you to reuse patterns as you build the site.



Make sure your patterns are responsive

It should go without saying these days, but put careful consideration into how patterns will respond across multiple devices. When showing the visual appearance of a pattern, make sure you demonstrate how it responds at different sizes.

That is not just useful for mobile devices, but also for when you use the pattern in various contexts. For example, a news listing may include a thumbnail when being displayed in the main body of a page but drops the thumbnail when being shown in a narrower side column.

Define the components of a pattern

Many patterns can be made up of multiple components. For example, a news listing could include:

- A title
- A description
- A thumbnail
- The date
- The author

When defining a pattern, it is important to list these components and also whether they are required or not. For example, do you need a description on a news listing? If not, what happens to the design if a description is not present?

Careful consideration needs to be given to these various permutations as they can become quite complicated if not thought about carefully.

Describe how your pattern will function

If a pattern library is also going to act as a functional specification for developers, you need to put a lot of thought into how the pattern will work. Where is the data coming from to populate fields? What happens when the user clicks a button or link? How does a carousel operate on a mobile device?

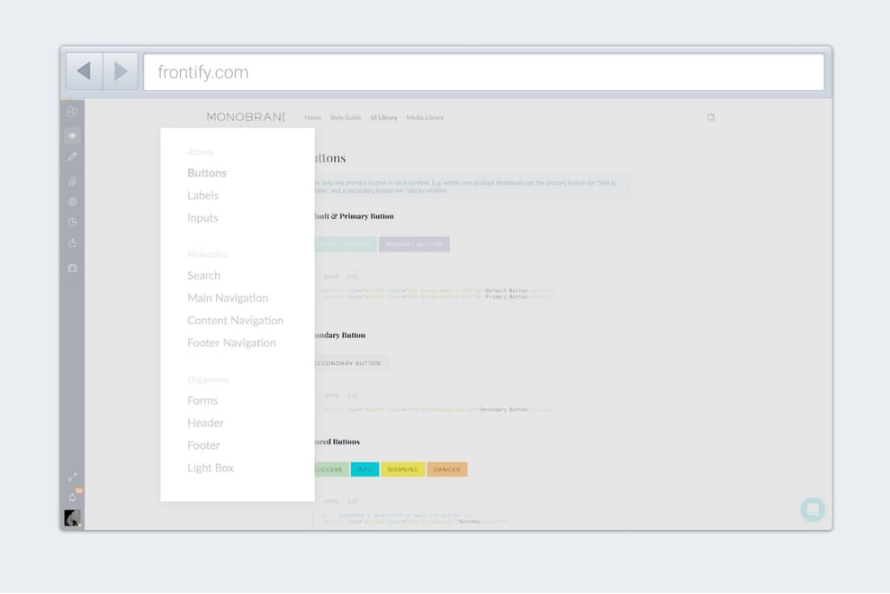
These kinds of practical questions are important when it comes to implementing patterns. Answering these issues also forces the designer and developer to work closely together to come to an agreement, and prevents the designer just throwing a design over the wall.

Ensure your patterns are accessible

In this era of web applications, accessibility is often forgotten. That is why I always include accessibility considerations in my pattern libraries. There will always be a section in a pattern definition where I make notes that a pattern should be keyboard accessible or can be interpreted by a screen reader.

Think about where you keep the code

A lot of pattern libraries display their code inline. However, I don't believe this is a good idea. I believe there should be a single repository for code that is always kept up to date and that this repository should be in source control. By having code in multiple places, it requires more maintenance, and it would be easy for somebody to use the wrong code snippets for a pattern.

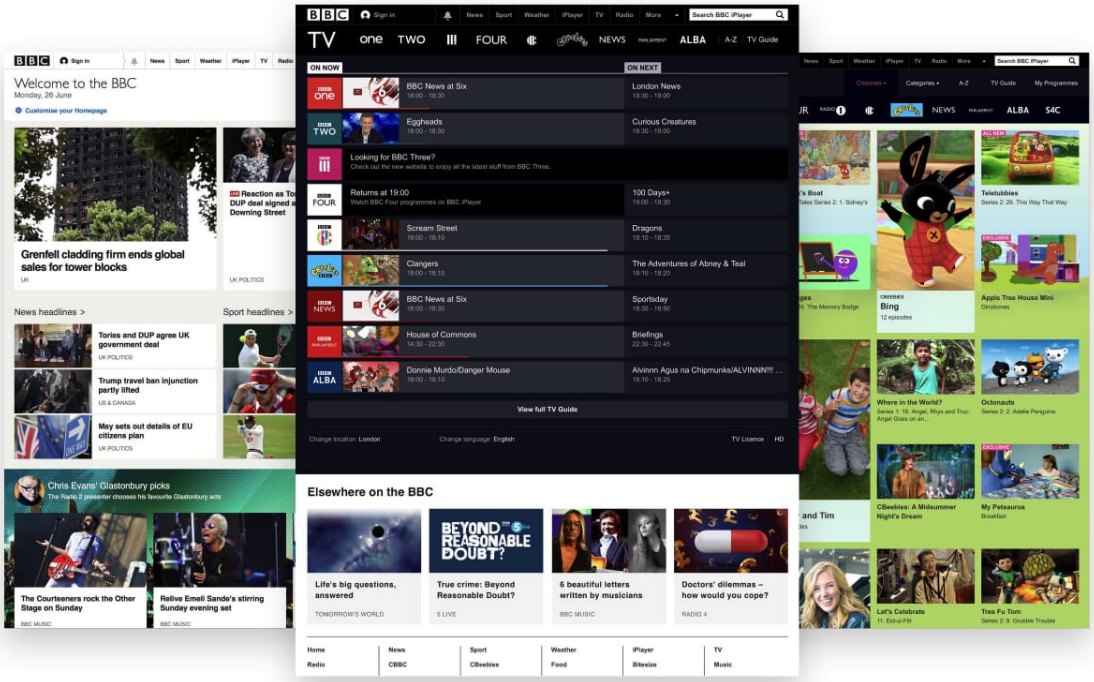


A tool like Frontify allows you to to create attractive pattern libraries, while syncing the code with your source repository via the Frontify API.

That said, there are tools like Astrum that allow you to keep code in source control, but still display it in the pattern library. However, you ultimately decided to solve this problem, make sure there is a single definitive source of code for each pattern and keep it up-to-date.

Consider how customisable your pattern library will be

Finally, think about whether patterns can be customised and to what extent. That will depend on how your brand operates. If you have a single consistent brand, then you probably want to offer very little in the way of customisation. However, if like the BBC you run multiple brands, then it will be important that the appearance of patterns can be customised to match the different aesthetics.



HTML Coding Conventions

Web developers are often uncertain about the coding style and syntax to use in HTML.

Between 2000 and 2010, many web developers converted from HTML to XHTML.

With XHTML, developers were forced to write valid and "well-formed" code.

HTML5 is a bit more sloppy when it comes to code validation.

Be Smart and Future Proof

A consistent use of style makes it easier for others to understand your HTML.

In the future, programs like XML readers may want to read your HTML.

Using a well-formed-"close to XHTML" syntax can be smart.

Always keep your code tidy, clean and well-formed.

Use Correct Document Type

Always declare the document type as the first line in your document:

```
<!DOCTYPE html>
```

If you want consistency with lower case tags, you can use:

```
<!doctype html>
```

Use Lower Case Element Names

HTML5 allows mixing uppercase and lowercase letters in element names.

We recommend using lowercase element names because:

- Mixing uppercase and lowercase names is bad
- Developers normally use lowercase names (as in XHTML)
- Lowercase look cleaner
- Lowercase are easier to write

Bad:

```
<SECTION>
  <p>This is a paragraph.</p>
</SECTION>
```

Very Bad:

```
<Section>
  <p>This is a paragraph.</p>
</SECTION>
```

Good:

```
<section>
  <p>This is a paragraph.</p>
</section>
```

Close All HTML Elements

In HTML5, you don't have to close all elements (for example the <p> element).

We recommend closing all HTML elements.

Bad:

```
<section>
  <p>This is a paragraph.
  <p>This is a paragraph.
</section>
```

Good:

```
<section>
  <p>This is a paragraph.</p>
  <p>This is a paragraph.</p>
</section>
```

Close Empty HTML Elements

In HTML5, it is optional to close empty elements.

Allowed:

```
<meta charset="utf-8">
```

Also Allowed:

```
<meta charset="utf-8" />
```

However, the closing slash (/) is REQUIRED in XHTML and XML.

If you expect XML software to access your page, it is a good idea to keep the closing slash!

Use Lower Case Attribute Names

HTML5 allows mixing uppercase and lowercase letters in attribute names.

We recommend using lowercase attribute names because:

- Mixing uppercase and lowercase names is bad
- Developers normally use lowercase names (as in XHTML)
- Lowercase look cleaner
- Lowercase are easier to write

Bad:

```
<div CLASS="menu">
```

Good:

```
<div class="menu">
```

Quote Attribute Values

HTML5 allows attribute values without quotes.

We recommend quoting attribute values because:

- Developers normally quote attribute values (as in XHTML)
- Quoted values are easier to read
- You MUST use quotes if the value contains spaces

Very bad:

This will not work, because the value contains spaces:

```
<table class=table striped>
```

Bad:

```
<table class=striped>
```

Good:

```
<table class="striped">
```

Image Attributes

Always add the `alt` attribute to images. This attribute is important when the image for some reason cannot be displayed. Also, always define image width and height. It reduces flickering because the browser can reserve space for the image before loading.

Bad:

```
<img src="html5.gif">
```

Good:

```
<img src="html5.gif" alt="HTML5" style="width:128px;height:128px">
```

Spaces and Equal Signs

HTML5 allows spaces around equal signs. But space-less is easier to read and groups entities better together.

Bad:

```
<link rel = "stylesheet" href = "styles.css">
```

Good:

```
<link rel="stylesheet" href="styles.css">
```

Avoid Long Code Lines

When using an HTML editor, it is inconvenient to scroll right and left to read the HTML code.

Try to avoid code lines longer than 80 characters.

Blank Lines and Indentation

Do not add blank lines without a reason.

For readability, add blank lines to separate large or logical code blocks.

For readability, add two spaces of indentation. Do not use the tab key.

Do not use unnecessary blank lines and indentation. It is not necessary to indent every element:

Unnecessary:

```
<body>

 <h1>Famous Cities</h1>

 <h2>Tokyo</h2>

 <p>
  Tokyo is the capital of Japan, the center of the Greater Tokyo Area,
  and the most populous metropolitan area in the world.
```

```
  It is the seat of the Japanese government and the Imperial Palace,
  and the home of the Japanese Imperial Family.
 </p>

</body>
```

Better:

```
<body>

<h1>Famous Cities</h1>

<h2>Tokyo</h2>
<p>Tokyo is the capital of Japan, the center of the Greater Tokyo Area,
and the most populous metropolitan area in the world.
It is the seat of the Japanese government and the Imperial Palace,
and the home of the Japanese Imperial Family.</p>

</body>
```

Table Example:

```
<table>
 <tr>
  <th>Name</th>
  <th>Description</th>
 </tr>
 <tr>
  <td>A</td>
  <td>Description of A</td>
 </tr>
 <tr>
  <td>B</td>
  <td>Description of B</td>
 </tr>
</table>
```

List Example:

```
<ul>
 <li>London</li>
 <li>Paris</li>
 <li>Tokyo</li>
</ul>
```

Omitting <html> and <body>?

In HTML5, the `<html>` tag and the `<body>` tag can be omitted.

The following code will validate as HTML5:

Example

```
<!DOCTYPE html>
<head>
 <title>Page Title</title>
</head>

<h1>This is a heading</h1>
<p>This is a paragraph.</p>
```

However, we do not recommend omitting the `<html>` and the `<body>` tag.

The `<html>` element is the document root. It is the recommended place for specifying the page language:

```
<!DOCTYPE html>
<html lang="en-US">
```

Declaring a language is important for accessibility applications (screen readers) and search engines.

Omitting `<html>` or `<body>` can crash DOM and XML software.

Omitting `<body>` can produce errors in older browsers (IE9).

Omitting <head>?

In HTML5, the <head> tag can also be omitted.

By default, browsers will add all elements before `<body>` to a default `<head>` element.

You can reduce the complexity of HTML by omitting the `<head>` tag:

Example

```
<!DOCTYPE html>
<html>
<title>Page Title</title>

<body>
 <h1>This is a heading</h1>
 <p>This is a paragraph.</p>
```

```
</body>

</html>
```

However, we do not recommend omitting the `<head>` tag.

Omitting tags is unfamiliar to web developers. It needs time to be established as a guideline.

Meta Data

The `<title>` element is required in HTML5. Make the title as meaningful as possible:

```
<title>HTML5 Syntax and Coding Style</title>
```

To ensure proper interpretation and correct search engine indexing, both the language and the character encoding should be defined as early as possible in a document:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
 <meta charset="UTF-8">
 <title>HTML5 Syntax and Coding Style</title>
</head>
```

Setting The Viewport

HTML5 introduced a method to let web designers take control over the viewport, through the `<meta>` tag.

The viewport is the user's visible area of a web page. It varies with the device, and will be smaller on a mobile phone than on a computer screen.

You should include the following `<meta>` viewport element in all your web pages:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

A `<meta>` viewport element gives the browser instructions on how to control the page's dimensions and scaling.

The width=device-width part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).

The initial-scale=1.0 part sets the initial zoom level when the page is first loaded by the browser.

Here is an example of a web page *without* the viewport meta tag, and the same web page *with* the viewport meta tag:

Tip: If you are browsing this page with a phone or a tablet, you can click on the two links below to see the difference.

Without the viewport meta tag

With the viewport meta tag

HTML Comments

Short comments should be written on one line, like this:

```
<!-- This is a comment -->
```

Comments that spans more than one line, should be written like this:

```
<!--
  This is a long comment example. This is a long comment example.
  This is a long comment example. This is a long comment example.
-->
```

Long comments are easier to observe if they are indented two spaces.

Style Sheets

Use simple syntax for linking to style sheets (the type attribute is not necessary):

```
<link rel="stylesheet" href="styles.css">
```

Short rules can be written compressed, like this:

```
p.intro {font-family: Verdana; font-size: 16em;}
```

Long rules should be written over multiple lines:

```
body {
  background-color: lightgrey;
  font-family: "Arial Black", Helvetica, sans-serif;
  font-size: 16em;
  color: black;
}
```

- Place the opening bracket on the same line as the selector
- Use one space before the opening bracket
- Use two spaces of indentation
- Use semicolon after each property-value pair, including the last
- Only use quotes around values if the value contains spaces
- Place the closing bracket on a new line, without leading spaces
- Avoid lines over 80 characters

Loading JavaScript in HTML

Use simple syntax for loading external scripts (the type attribute is not necessary):

```
<script src="myscript.js">
```

Accessing HTML Elements with JavaScript

A consequence of using "untidy" HTML styles can result in JavaScript errors.

These two JavaScript statements will produce different results:

Example

```
var obj = getElementById("Demo")

var obj = getElementById("demo")
```

Visit the JavaScript Style Guide.

Use Lower Case File Names

Some web servers (Apache, Unix) are case sensitive about file names: "london.jpg" cannot be accessed as "London.jpg".

Other web servers (Microsoft, IIS) are not case sensitive: "london.jpg" can be accessed as "London.jpg" or "london.jpg".

If you use a mix of upper and lower case, you have to be extremely consistent.

If you move from a case insensitive to a case sensitive server, even small errors will break your web!

To avoid these problems, always use lower case file names.

File Extensions

HTML files should have a **.html** or **.htm** extension.

CSS files should have a **.css** extension.

JavaScript files should have a **.js** extension.

Differences Between .htm and .html

There is no difference between the .htm and .html extensions. Both will be treated as HTML by any web browser or web server.

The differences are cultural:

.htm "smells" of early DOS systems where the system limited the extensions to 3 characters.

.html "smells" of Unix operating systems that did not have this limitation.

Understanding hard vs soft navigations

In the world of single page apps, there are two distinct types of "navigations":

- Hard navigation: The first time a visitor loads your app
- Soft navigation: Every route change after the hard navigation

These two types of SPA navigations have drastically different performance characteristics, and it's important to be able to analyze them separately. For most SPAs, every URL on your site could be loaded either via a hard navigation or a soft navigation, depending on whether the user started from another site (e.g. search results) or from an internal page.

A hard navigation is the first navigation to your site. During this navigation the browser is doing a full navigation by tearing down any page it was on previously, fetching the new page's HTML and downloading all of its resources. Remember that with single page applications, during a hard navigation the onload event is no longer relevant and is just one milestone on the way to getting all of the page's resources. After the SPA framework is downloaded (i.e. the angular.js file), it will begin to fetch any templates, script, images or XHRs necessary to render the current view. All of this may happen after the onload event.

Thus, the hard navigation is really "expensive" from a performance perspective. Not only do you have to pay the cost of loading the base HTML, SPA framework, CSS and JavaScript, but the framework then needs to fetch all of the resources for the current view.

A soft navigation, on the other hand, is every navigation after the first hard navigation. It's called a soft navigation because it's not a "real" navigation, from the browser's perspective — the page does not reload and the onload event does not fire again. The base framework, CSS, and JavaScripts stay in memory and don't have to be fetched again. Instead, the soft navigation swaps out the old view for the new view, which may often contain a lot of the same content (e.g. the header). A soft navigation only needs to fetch a "delta" from the previous view.

How we updated Boomerang: Because of these factors, a soft navigation will always be faster than the initial hard navigation. Boomerang now logs whether each SPA navigation is a hard or a soft navigation, so you can track the experience for each.

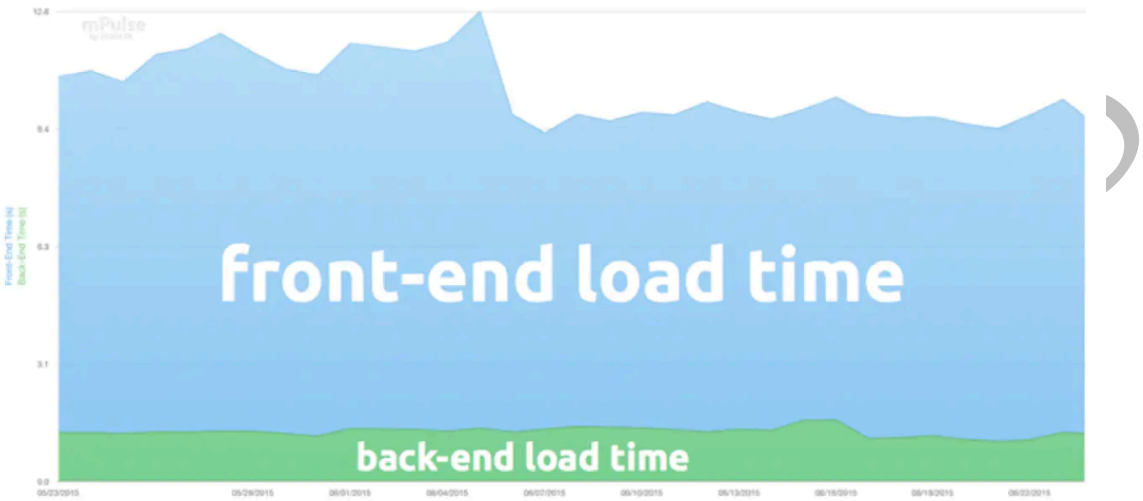
Understanding back-end versus front-end time

In a traditional website, page load performance is often split between back-end and front-end times. Here's how our community defines the two for a traditional (non-SPA) website:

- Back-end: From the beginning of the navigation to the first bytes of the root HTML
- Front-end: From the first bytes of the root HTML to the page's onload event

Splitting the time up in this way is important because it helps you determine which components are contributing to the overall page load experience. It's useful to have this breakdown so you

can know where to focus your investigations. For example, if your back-end time comprises 80% of the overall page load time, you may want to look into speeding up your web servers. But typically, front-end dominates overall load time, as in this graph, which shows front-end and back-end times over a one-month period:



With single page apps, we can't use these same definitions. For one thing, the onload event doesn't matter. And second, hard navigations and soft navigations behave differently and require unique definitions of what back-end time means.

We've decided to define back-end and front-end for SPAs differently than with traditional apps:

- Hard navigations:
  - Back-end: From the beginning of the navigation to the first bytes of the root HTML
  - Front-end: From the first bytes of the root HTML to the end of the last resource fetched for the view (route)
- Soft navigations:
  - Back-end: The union of any time where there was a XHR or JavaScript being fetched on the network
  - Front-end: The total time of the soft navigation minus the back-end time

Note that for soft navigations, we've defined back-end time as essentially any time there's a critical resource — such as XHR or JavaScript — waiting on the network. This is similar to how a traditional page defines back-end time, using just the root HTML (which is a critical-path resource). Soft navigations are often waiting on XHR and JavaScript as the primary "back-end" resources that need to be fetched before the framework can decide what additional resources (such as CSS or images) need to be fetched to render the view.

How we updated Boomerang: Boomerang has been updated to measure the back-end and front-end for SPAs time according to these definitions.

Being flexible

In addition to the monitoring features above, we've found that each customer site is built a little differently than the next. To help our customers measure the exact metrics that are important to them, we've made Boomerang a bit more flexible for single page apps.

For example, we've added the following features:

- Some "routes" (views) aren't as important as others. For example, depending on how you've built your app, the URL may change when a minor widget on the page is clicked, even though the entire view doesn't change. We've given our customers the ability to filter out specific route changes so they can focus on the ones that matter most.
- The way we've defined the end of a SPA navigation may not work in every circumstance, so we've added a feature to Boomerang so the customer can let us know when the navigation is complete.
- In addition to monitoring single page apps, we also support monitoring individual XMLHttpRequests (XHRs). We've expanded XHR support to be able to monitor XHRs before boomerang.js has even loaded.

How we updated Boomerang: You can see these new features (and more) in the open-source Boomerang project.

Universal Router

A simple middleware-style router for isomorphic JavaScript web apps

and single-page applications

API

```
import Router from 'universal-router';

const router = new Router([
  { path: '/one', action: () => 'Page One' },
  { path: '/two', action: () => 'Page Two' }
]);
```

```
router.resolve({ path: '/one' }).then(result => {
  document.body.innerHTML = result || 'Not found';
});
```

router.resolve({ path, ...context }) → Promise<any>

Traverses the list of routes in the same order they were defined, finds the first route matching the provided URL path string and whose action method returns anything other than **null** or **undefined**

..where **action** is just a regular function that may, or may not, return any arbitrary data — a string, a React component, anything!

Child Routes

Each route may have an optional **children: [...]** property containing the list of child routes:

```
const routes = {
  path: '/admin'
  children: [
    {
      path: '/', action: () => {/* ... */}        // www.example.com/admin
    },
    {
      path: '/users',
      children: [
        {
          path: '/', action: () => {/* ... */}    // www.example.com/admin/users
        },
        {
          path: '/:id', action: () => {/* ... */} // www.example.com/admin/users/123
        }
      ]
    }
  ]
};
```

URL Parameters

Named route parameters are captured and added to **context.params**

```
const router = new Router({
  path: '/hello/:username',
  action: (context) => `Welcome, ${context.params.username}!`
});

router.resolve({ path: '/hello/john' })
  .then(result => console.log(result));
  // => Welcome, john!
```

This functionality is powered by path-to-regexp npm module and works the same way as the routing solutions in many popular JavaScript frameworks such as Express and Koa. Also check out online router tester.

Context

In addition to a URL path string, any arbitrary data can be passed to the router's **resolve()** method, that becomes available inside action methods

```
const router = new Router({
  path: '/hello',
  action(context) {
    return `Welcome, ${context.user}!`
  }
});

router.resolve({ path: '/hello', user: 'admin' })
  .then(result => console.log(result));
  // => Welcome, admin!
```

Async Routes

```
const router = new Router({
  path: '/hello/:username',
  async action({ params }) {
    const resp = await fetch(`/api/users/${params.username}`);
    const user = await resp.json();
    if (user) return `Welcome, ${user.displayName}!`;
  }
});

router.resolve({ path: '/hello/john' })
  .then(result => console.log(result));
  // => Welcome, John Brown!
```

The router works great with asynchronous functions out of the box!

Use Babel to transpile your code with **async / await** to normal JavaScript. Alternatively, stick to ES6 Promises:

```
{
  path: '/hello/:username',
  action({ params }) {
    return fetch(`/api/users/${params.username}`)
      .then(resp => resp.json())
      .then(user => user && `Welcome, ${user.displayName}!`);
  }
}
```

Middlewares

Any route action function may act as a **middleware** by calling **context.next()**

```
const router = new Router({
  path: '/',
  async action({ next }) {
    console.log('middleware: start');
    const child = await next();
    console.log('middleware: end');
    return child;
  },
  children: [
    {
      path: '/hello',
```

```
      action() {
        console.log('route: return a result');
        return 'Hello, world!';
      }
    }
  ]
});

router.resolve({ path: '/hello' });

// Prints:
//   middleware: start
//   route: return a result
//   middleware: end
```

Use with React.js

Just create a middleware that's responsible for rendering **React** components

```
import Router from 'univeral-router';

const router = new Router({
  path: '/',
  async action({ next }) {
    const component = await next();
    return component && <Layout>{component}</Layout>;
  },
  children: [
    { path: '/', action: () => <HomePage /> },
    { path: '/about', action: () => <AboutPage /> }
  ]
});


router.resolve({ path: '/hello' }).then(component =>
  ReactDOM.render(component, document.getElementById('root'))
);
```

Use with Node.js / Express

```
import express from 'express';
import Router from 'universal-router';
```

```
const router = new Router({  // Note: The exact same routes can be used
  path: '/',                 // in both client-side and server-side apps
  async action({ next }) {
    const component = await next();
    return component && <Layout>{component}</Layout>;
  },
  children: [
    { path: '/', action: () => <HomePage /> },
    { path: '/about', action: () => <AboutPage /> }
  ]
});

const app = express();

app.use((req, res, next) => {
  router.resolve({ path: req.path }).then((component) => {
    const body = ReactDOM.renderToString(component);
    const html = `<html><head>...</head><body>${body}</body></html>`;
    res.send(html);
  }).catch(next);
});

app.listen(3000);
```

Use with React / Relay

```
import Router from 'universal-router';

const router = new Router({
  path: '/products/:id',
  action({ params }) {
    return {
      component: ProductPage, // <- Relay container
      params,                 // <- query parameters
      queries: {              // <- root queries
        viewer: () => Relay.QL`query { viewer }`,
        product: () => Relay.QL`query { product(id: $id) }`,
      }
    };
  }
});

router.resolve({ path: '/products/1' }).then(props =>
  ReactDOM.render(
    React.createElement(RelayApp, props),
    document.getElementById('app')
  )
);
```

Code Splitting

This routing approach works great with code splitting

```
const router = new Router([
  {
    path: '/',
    async action() {
      const Homepage = await import('./Homepage');
      return <Homepage />;
    }
  },
  {
    path: '/:username',
    async action ({ params }) {
      const [ UserProfile, data ] = await Promise.all([
        import('./UserProfile'),
        fetch(`/api/users/${params.username}`).then(x => x.json()),
      ]);

      if (data) return <UserProfile {...data} />;
    }
  }
]);

router.resolve({ path: '/john' }).then(/* renderer */);
```

Universal Router + History+ React

Create routes

First, I need to create routes for Universal Router, which is just a Javascript object literal.

routes

An “action” method can return any kind of value. It could be a string literal or a React element. A “next” function is given as the part of the argument of the action method. The “next” function is the action method of the corresponding child route. I nest a child React element with a parent one using this mechanism.

Create a router

With the routes, now I create a router.

Calling the “resolve” method of a router with a path string literal or a location object, I get a Promise object that gives me the value that the corresponding action method returns. In my case, the value is a React element.

Integrate a router and ReactDOM

Integrating them is pretty simple. I created a “render” method, which takes a location object and renders a corresponding React element within a DOM element.

Create a Browser History object

I used History library for this project. And, I wanted to prevent a new history state from being added to a history stack when the new state is the same as the previous state. So, I wrapped up the push method of the history object.

Put them all together

Finally, I integrate the history object with the render method.

Link Component

Additionally, I created Link Component. It’s handy to use for location transition. (This is like the Link Component from React Router.)

Conclusion

Universal Router works pretty well with React and History. It’s totally independent from React, so it’d be very flexible. And, since it’s a small library that only takes care of issues related with router, it’d be easy to learn and understand what it does. On the other hand, you’d need to implement some essential features like Link Component or redirect on your own, which other bigger libraries, like React Router, have as built-in features. Personally, I prefer Universal Router to React Router because I can implement a router with a regular Javascript object literal.

OTHER RESOURCE READING

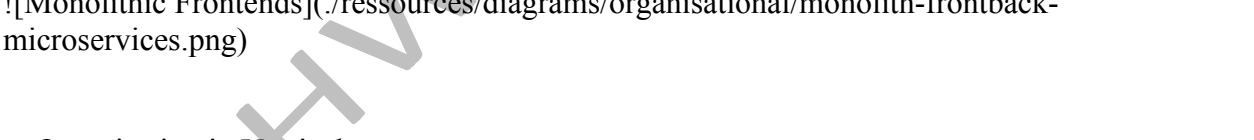
Techniques, strategies and recipes for building a __modern web app__ with __multiple teams__ that can __ship features independently__.

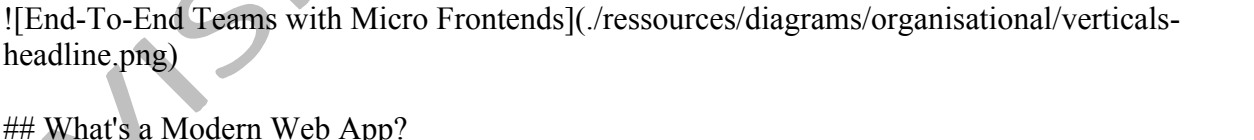
What are Micro Frontends?

The term __Micro Frontends__ first came up in [ThoughtWorks Technology Radar](https://www.thoughtworks.com/radar/techniques/micro-frontends) at the end of 2016. It extends the concepts of micro services to the frontend world. The current trend is to build a feature-rich and powerful browser application, aka single page app, which sits on top of a micro service architecture. Over time the frontend layer, often developed by a separate team, grows and gets more difficult to maintain. That's what we call a [Frontend Monolith](https://www.youtube.com/watch?v=pU1gXA0rfwc).

The idea behind Micro Frontends is to think about a website or web app as __a composition of features__ which are owned by __independent teams__. Each team has a __distinct area of business__ or __mission__ it cares about and specialises in. A team is __cross functional__ and develops its features __end-to-end__, from database to user interface.

However, this idea is not new. It has a lot in common with the [Self-contained Systems](http://scs-architecture.org/) concept. In the past approaches like this went by the name of [Frontend Integration for Verticalised Systems](https://dev.otto.de/2014/07/29/scaling-with-microservices-and-vertical-decomposition/). But Micro Frontends is clearly a more friendly and less bulky term.

__Monolithic Frontends__


__Organisation in Verticals__


What's a Modern Web App?

In the introduction I've used the phrase "building a modern web app". Let's define the assumptions that are connected with this term.

To put this into a broader perspective, [Aral Balkan](https://ar.al/) has written a blog post about what he calls the [Documents-to-Applications Continuum](https://ar.al/notes/the-documents-to-applications-continuum/). He comes up with the concept of a sliding scale where a site, built out

of __static documents__, connected via links, is __on the left__ end and a pure behaviour driven, __contentless application__ like an online photo editor is __on the right__.

If you would position your project on the __left side of this spectrum__, an __integration on webserver level__ is a good fit. With this model a server collects and __concatenates HTML strings__ from all components that make up the page requested by the user. Updates are done by reloading the page from the server or replacing parts of it via ajax. [Gustaf Nilsson Kotte](https://twitter.com/gustaf_nk/) has written a [comprehensive article](https://gustafnk.github.io/microservice-websites/) on this topic.

When your user interface has to provide __instant feedback__, even on unreliable connections, a pure server rendered site is not sufficient anymore. To implement techniques like [Optimistic UI](https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/) or [Skeleton Screens](http://www.lukew.com/ff/entry.asp?1797) you need to be able to also __update__ your UI __on the device itself__. Google's term [Progressive Web Apps](https://developers.google.com/web/progressive-web-apps/) aptly describes the __balancing act__ of being a good citizen of the web (progressive enhancement) while also providing app-like performance. This kind of application is located somewhere __around the middle of the site-app-continuum__. Here a solely server based solution is not sufficient anymore. We have to move the __integration into the browser__, and this is the focus of this article.

Core Ideas behind Micro Frontends

* __Be Technology Agnostic__
Each team should be able to choose and upgrade their stack without having to coordinate with other teams. [Custom Elements](#the-dom-is-the-api) are a great way to hide implementation details while providing a neutral interface to others.
* __Isolate Team Code__
Don’t share a runtime, even if all teams use the same framework. Build independent apps that are self contained. Don't rely on shared state or global variables.
* __Establish Team Prefixes__
Agree on naming conventions where isolation is not possible yet. Namespace CSS, Events, Local Storage and Cookies to avoid collisions and clarify ownership.
* __Favor Native Browser Features over Custom APIs__
Use [Browser Events for communication](#parent-child-communication--dom-modification) instead of building a global PubSub system. If you really have to build a cross team API, try keeping it as simple as possible.
* __Build a Resilient Site__
Your feature should be useful, even if JavaScript failed or hasn't executed yet. Use [Universal Rendering](#serverside-rendering--universal-rendering) and Progressive Enhancement to improve perceived performance.

The DOM is the API

[Custom Elements](https://developers.google.com/web/fundamentals/getting-started/primers/customelements), the interoperability aspect from the Web Components Spec, are a good primitive for integration in the browser. Each team builds their component __using their

web technology of choice__ and __wraps it inside a Custom Element__ (e.g. `<order-minicart></order-minicart>`). The DOM specification of this particular element (tag-name, attributes & events) acts as the contract or public API for other teams. The advantage is that they can use the component and its functionality without having to know the implementation. They just have to be able to interact with the DOM.

But Custom Elements alone are not the solution to all our needs. To address progressive enhancement, universal rendering or routing we need additional pieces of software.

This page is divided into two main areas. First we will discuss [Page Composition](#page-composition) - how to assemble a page out of components owned by different teams. After that we'll show examples for implementing clientside [Page Transition](#page-transition).

Page Composition

Beside the __client-__ and __serverside__ integration of code written in __different frameworks__ itself, there are a lot of side topics that should be discussed: mechanisms to __isolate js__, __avoid css conflicts__, __load resources__ as needed, __share common resources__ between teams, handle __data fetching__ and think about good __loading states__ for the user. We'll go into these topics one step at a time.

The Base Prototype

The product page of this model tractor store will serve as the basis for the following examples.

It features a __variant selector__ to switch between the three different tractor models. On change product image, name, price and recommendations are updated. There is also a __buy button__, which adds the selected variant to the basket and a __mini basket__ at the top that updates accordingly.

[![Example 0 - Product Page - Plain JS](./ressources/video/model-store-0.gif)](./0-model-store/)

[try in browser](./0-model-store/) & [inspect the code](https://github.com/neuland/micro-frontends/tree/master/0-model-store)

All HTML is generated client side using __plain JavaScript__ and ES6 Template Strings with __no dependencies__. The code uses a simple state/markup separation and re-renders the entire HTML client side on every change - no fancy DOM diffing and __no universal rendering__ for now. Also __no team separation__ - [the code](https://github.com/neuland/micro-frontends/tree/master/0-model-store) is written in one js/css file.

Clientside Integration

In this example, the page is split into separate components/fragments owned by three teams. __Team Checkout__ (blue) is now responsible for everything regarding the purchasing process - namely the __buy button__ and __mini basket__. __Team Inspire__ (green) manages the

__product recommendations__ on this page. The page itself is owned by __Team Product__ (red).

[![Example 1 - Product Page - Composition](./ressources/screen/three-teams.png)](./1-composition-client-only/)

[try in browser](./1-composition-client-only/) & [inspect the code](https://github.com/neuland/micro-frontends/tree/master/1-composition-client-only)

__Team Product__ decides which functionality is included and where it is positioned in the layout. The page contains information that can be provided by Team Product itself, like the product name, image and the available variants. But it also includes fragments (Custom Elements) from the other teams.

How to Create a Custom Element?

Lets take the __buy button__ as an example. Team Product includes the button simply adding `<blue-buy sku="t_porsche"></blue-buy>` to the desired position in the markup. For this to work, Team Checkout has to register the element `blue-buy` on the page.

```
class BlueBuy extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button type="button">buy for 66,00 €</button>`;
  }
  disconnectedCallback() { ... }
}
window.customElements.define('blue-buy', BlueBuy);
```

Now every time the browser comes across a new `blue-buy` tag, the constructor is called. `this` is the reference to the root DOM node of the custom element. All properties and methods of a standard DOM element like `innerHTML` or `getAttribute()` can be used.

![Custom Element in Action](./ressources/video/custom-element.gif)

When naming your element the only requirement the spec defines is that the name must __include a dash (-)__ to maintain compatibility with upcoming new HTML tags. In the upcoming examples the naming convention `[team_color]-[feature]` is used. The team namespace guards against collisions and this way the ownership of a feature becomes obvious, simply by looking at the DOM.

Parent-Child Communication / DOM Modification

When the user selects another tractor in the __variant selector__, the __buy button has to be updated__ accordingly. To achieve this Team Product can simply __remove__ the existing element from the DOM __and insert__ a new one.

```
container.innerHTML;
// => <blue-buy sku="t_porsche">...</blue-buy>
container.innerHTML = '<blue-buy sku="t_fendt"></blue-buy>';
```

The `disconnectedCallback` of the old element gets invoked synchronously to provide the element with the chance to clean up things like event listeners. After that the `constructor` of the newly created `t_fendt` element is called.

Another more performant option is to just update the `sku` attribute on the existing element.

```
document.querySelector('blue-buy').setAttribute('sku', 't_fendt');
```

If Team Product used a templating engine that features DOM diffing, like React, this would be done by the algorithm automatically.

![Custom Element Attribute Change](./ressources/video/custom-element-attribute.gif)

To support this the Custom Element can implement the `attributeChangedCallback` and specify a list of `observedAttributes` for which this callback should be triggered.

```
const prices = {
  t_porsche: '66,00 €',
  t_fendt: '54,00 €',
  t_eicher: '58,00 €',
};

class BlueBuy extends HTMLElement {
  static get observedAttributes() {
    return ['sku'];
  }
  constructor() {
    super();
    this.render();
  }
  render() {
    const sku = this.getAttribute('sku');
    const price = prices[sku];
    this.innerHTML = `<button type="button">buy for ${price}</button>`;
  }
  attributeChangedCallback(attr, oldValue, newValue) {
    this.render();
  }
  disconnectedCallback() {...}
}
window.customElements.define('blue-buy', BlueBuy);
```

To avoid duplication a `render()` method is introduced which is called from `constructor` and `attributeChangedCallback`. This method collects needed data and innerHTML's the new markup. When deciding to go with a more sophisticated templating engine or framework inside the Custom Element, this is the place where its initialisation code would go.

Browser Support

The above example uses the Custom Element V1 Spec which is currently [supported in Chrome, Safari and Opera](http://caniuse.com/#feat=custom-elementsv1). But with [document-register-element](https://github.com/WebReflection/document-register-element) a lightweight and battle-tested polyfill is available to make this work in all browsers. Under the hood, it uses the [widely supported](http://caniuse.com/#feat=mutationobserver) Mutation Observer API, so there is no hacky DOM tree watching going on in the background.

Framework Compatibility

Because Custom Elements are a web standard, all major JavaScript frameworks like Angular, React, Preact, Vue or Hyperapp support them. But when you get into the details, there are still a few implementation problems in some frameworks. At [Custom Elements Everywhere](https://custom-elements-everywhere.com/) [Rob Dodson](https://twitter.com/rob_dodson) has put together a compatibility test suite that highlights unresolved issues.

Child-Parent or Siblings Communication / DOM Events

But passing down attributes is not sufficient for all interactions. In our example the __mini basket should refresh__ when the user performs a __click on the buy button__.

Both fragments are owned by Team Checkout (blue), so they could build some kind of internal JavaScript API that lets the mini basket know when the button was pressed. But this would require the component instances to know each other and would also be an isolation violation.

A cleaner way is to use a PubSub mechanism, where a component can publish a message and other components can subscribe to specific topics. Luckily browsers have this feature built-in. This is exactly how browser events like `click`, `select` or `mouseover` work. In addition to native events there is also the possibility to create higher level events with `new CustomEvent(...)`. Events are always tied to the DOM node they were created/dispatched on. Most native events also feature bubbling. This makes it possible to listen for all events on a specific sub-tree of the DOM. If you want to listen to all events on the page, attach the event listener to the window element. Here is how the creation of the `blue:basket:changed`-event looks in the example:

```
class BlueBuy extends HTMLElement {
  [...]
  connectedCallback() {
```

```
    [...]
    this.render();
    this.firstChild.addEventListener('click', this.addToCart);
  }
  addToCart() {
    // maybe talk to an api
    this.dispatchEvent(new CustomEvent('blue:basket:changed', {
      bubbles: true,
    }));
  }
  render() {
    this.innerHTML = `<button type="button">buy</button>`;
  }
  disconnectedCallback() {
    this.firstChild.removeEventListener('click', this.addToCart);
  }
}
```

The mini basket can now subscribe to this event on `window` and get notified when it should refresh its data.

```
class BlueBasket extends HTMLElement {
  connectedCallback() {
    [...]
    window.addEventListener('blue:basket:changed', this.refresh);
  }
  refresh() {
    // fetch new data and render it
  }
  disconnectedCallback() {
    window.removeEventListener('blue:basket:changed', this.refresh);
  }
}
```

With this approach the mini basket fragment adds a listener to a DOM element which is outside its scope (`window`). This should be ok for many applications, but if you are uncomfortable with this you could also implement an approach where the page itself (Team Product) listens to the event and notifies the mini basket by calling `refresh()` on the DOM element.

```
// page.js
const $ = document.getElementsByTagName;

$('blue-buy')[0].addEventListener('blue:basket:changed', function() {
  $('blue-basket')[0].refresh();
});
```

Imperatively calling DOM methods is quite uncommon, but can be found in [video element api](https://developer.mozilla.org/de/docs/Web/HTML/Using_HTML5_audio_and_video#Controlling_media_playback) for example. If possible the use of the declarative approach (attribute change) should be preferred.

Serverside Rendering / Universal Rendering

Custom Elements are great for integrating components inside the browser. But when building a site that is accessible on the web, chances are that initial load performance matters and users will see a white screen until all js frameworks are downloaded and executed. Additionally, it's good to think about what happens to the site if the JavaScript fails or is blocked. [Jeremy Keith](https://adactio.com/) explains the importance in his ebook/podcast [Resilient Web Design](https://resilientwebdesign.com/). Therefore the ability to render the core content on the server is key. Sadly the web component spec does not talk about server rendering at all. No JavaScript, no Custom Elements :(

Custom Elements + Server Side Includes = □□

To make server rendering work, the previous example is refactored. Each team has their own express server and the `render()` method of the Custom Element is also accessible via url.

```
$ curl http://127.0.0.1:3000/blue-buy?sku=t_porsche
<button type="button">buy for 66,00 €</button>
```

The Custom Element tag name is used as the path name - attributes become query parameters. Now there is a way to server-render the content of every component. In combination with the `<blue-buy>`-Custom Elements something that is quite close to a __Universal Web Component__ is achieved:

```
<blue-buy sku="t_porsche">
  <!--#include virtual="/blue-buy?sku=t_porsche" -->
</blue-buy>
```

The `#include` comment is part of [Server Side Includes](https://en.wikipedia.org/wiki/Server_Side_Includes), which is a feature that is available in most web servers. Yes, it's the same technique used back in the days to embed the current date on our web sites. There are also a few alternative techniques like [ESI](https://en.wikipedia.org/wiki/Edge_Side_Includes), [nodesi](https://github.com/Schibsted-Tech-Polska/nodesi), [compoxure](https://github.com/tes/compoxure) and [tailor](https://github.com/zalando/tailor), but for our projects SSI has proven itself as a simple and incredibly stable solution.

The `#include` comment is replaced with the response of `/blue-buy?sku=t_porsche` before the web server sends the complete page to the browser. The configuration in nginx looks like this:

```
upstream team_blue {
```

```
  server team_blue:3001;
}
upstream team_green {
  server team_green:3002;
}
upstream team_red {
  server team_red:3003;
}

server {
  listen 3000;
  ssi on;

  location /blue {
    proxy_pass  http://team_blue;
  }
  location /green {
    proxy_pass  http://team_green;
  }
  location /red {
    proxy_pass  http://team_red;
  }
  location / {
    proxy_pass  http://team_red;
  }
}
```

The directive `ssi: on;` enables the SSI feature and an `upstream` and `location` block is added for every team to ensure that all urls which start with `/blue` will be routed to the correct application (`team_blue:3001`). In addition the `/` route is mapped to team red, which is controlling the homepage / product page.

This animation shows the tractor store in a browser which has __JavaScript disabled__.

[![Serverside Rendering - Disabled JavaScript](./ressources/video/server-render.gif)](./ressources/video/server-render.mp4)

[inspect the code](https://github.com/neuland/micro-frontends/tree/master/2-composition-universal)

The variant selection buttons now are actual links and every click leads to a reload of the page. The terminal on the right illustrates the process of how a request for a page is routed to team red, which controls the product page and after that the markup is supplemented by the fragments from team blue and green.

When switching JavaScript back on, only the server log messages for the first request will be visible. All subsequent tractor changes are handled client side, just like in the first example. In a later example the product data will be extracted from the JavaScript and loaded via a REST api as needed.

You can play with this sample code on your local machine. Only [Docker Compose](https://docs.docker.com/compose/install/) needs to be installed.

```
git clone https://github.com/neuland/micro-frontends.git
cd micro-frontends/2-composition-universal
docker-compose up --build
```

Docker then starts the nginx on port 3000 and builds the node.js image for each team. When you open http://127.0.0.1:3000/ in your browser you should see a red tractor. The combined log of `docker-compose` makes it easy to see what is going on in the network. Sadly there is no way to control the output color, so you have to endure the fact that team blue might be highlighted in green :)

The `src` files are mapped into the individual containers and the node application will restart when you make a code change. Changing the `nginx.conf` requires a restart of `docker-compose` in order to have an effect. So feel free to fiddle around and give feedback.

Data Fetching & Loading States

A downside of the SSI/ESI approach is, that the __slowest fragment determines the response time__ of the whole page.
So it's good when the response of a fragment can be cached.
For fragments that are expensive to produce and hard to cache it's often a good idea to exclude them from the initial render.
They can be loaded asynchronously in the browser.
In our example the `green-recos` fragment, that shows personalized recommendations is a candidate for this.

One possible solution would be that team red just skips the SSI Include.

Before

```
<green-recos sku="t_porsche">
  <!--#include virtual="/green-recos?sku=t_porsche" -->
</green-recos>
```

After

```
<green-recos sku="t_porsche"></green-recos>
```

Important Side-note: Custom Elements [cannot be self-closing](https://developers.google.com/web/fundamentals/architecture/building-components/customelements#jsapi), so writing `<green-recos sku="t_porsche" />` would not work correctly.

The rendering only takes place in the browser.
But, as can be seen in the animation, this change has now introduced a __substantial reflow__ of the page.
The recommendation area is initially blank.
Team greens JavaScript is loaded and executed.
The API call for fetching the personalized recommendation is made.
The recommendation markup is rendered and the associated images are requested.
The fragment now needs more space and pushes the layout of the page.

There are different options to avoid an annoying reflow like this.
Team red, which controls the page, could __fixate the recommendation containers height__.
On a responsive website its often tricky to determine the height, because it could differ for different screen sizes.
But the more important issue is, that __this kind of inter-team agreement creates a tight coupling__ between team red and green.
If team green wants to introduce an additional sub-headline in the reco element, it would have to coordinate with team red on the new height.
Both teams would have to rollout their changes simultaneously to avoid a broken layout.

A better way is to use a technique called [Skeleton Screens](https://blog.prototypr.io/luke-wroblewski-introduced-skeleton-screens-in-2013-through-his-work-on-the-polar-app-later-fd1d32a6a8e7).
Team red leaves the `green-recos` SSI Include in the markup.
In addition team green changes the __server-side render method__ of its fragment so that it produces a __schematic version of the content__.
The __skeleton markup__ can reuse parts of the real content's layout styles.
This way it __reserves the needed space__ and the fill-in of the actual content does not lead to a jump.

Skeleton screens are also __very useful for client rendering__.
When your custom element is inserted into the DOM due to a user action it could __instantly render the skeleton__ until the data it needs from the server has arrived.

Even on an __attribute change__ like for the _variant select_ you can decide to switch to skeleton view until the new data arrives.
This ways the user gets an indication that something is going on in the fragment.

But when your endpoint responds quickly a short __skeleton flicker__ between the old and new data could also be annoying.
Preserving the old data or using intelligent timeouts can help.
So use this technique wisely and try to get user feedback.

Navigating Between Pages

__to be continued soon...__

watch the [Github Repo](https://github.com/neuland/micro-frontends) to get notified

Additional Resources
- [Book: Micro Frontends in Action](https://www.manning.com/books/micro-frontends-in-action?a_aid=mfia&a_bid=5f09fdeb) Written by me. Currently in Mannings Early Access Programm (MEAP)
- [Talk: Micro Frontends - MicroCPH, Copenhagen 2019](https://www.youtube.com/watch?v=wCHYILvM7kU) ([Slides](https://noti.st/naltatis/zQb2m5/micro-frontends-the-nitty-gritty-details-or-frontend-backend-happyend)) The Nitty Gritty Details or Frontend, Backend, Happyend
- [Talk: Micro Frontends - Web Rebels, Oslo 2018](https://www.youtube.com/watch?v=dTW7eJsIHDg) ([Slides](https://noti.st/naltatis/HxcUfZ/micro-frontends-think-smaller-avoid-the-monolith-love-the-backend)) Think Smaller, Avoid the Monolith, □ □the Backend
- [Slides: Micro Frontends - JSUnconf.eu 2017](https://speakerdeck.com/naltatis/micro-frontends-building-a-modern-webapp-with-multiple-teams)
- [Talk: Break Up With Your Frontend Monolith - JS Kongress 2017](https://www.youtube.com/watch?v=W3_8sxUurzA) Elisabeth Engel talks about implementing Micro Frontends at gutefrage.net
- [Article: Micro Frontends](https://martinfowler.com/articles/micro-frontends.html) Article by Cam Jackson on Martin Fowlers Blog
- [Post: Micro frontends - a microservice approach to front-end web development](https://medium.com/@tomsoderlund/micro-frontends-a-microservice-approach-to-front-end-web-development-f325ebdadc16) Tom Söderlund explains the core concept and provides links on this topic
- [Post: Microservices to Micro-Frontends](http://www.agilechamps.com/microservices-to-micro-frontends/) Sandeep Jain summarizes the key principals behind microservices and micro frontends
- [Link Collection: Micro Frontends by Elisabeth Engel](https://micro-frontends.zeef.com/elisabeth.engel?ref=elisabeth.engel&share=ee53d51a914b4951ae5c94ece97642fc) extensive list of posts, talks, tools and other resources on this topic
- [Awesome Micro Frontends](https://github.com/ChristianUlbrich/awesome-microfrontends) a curated list of links by Christian Ulbrich
- [Custom Elements Everywhere](https://custom-elements-everywhere.com/) Making sure frameworks and custom elements can be BFFs

- Tractors are purchasable at [manufactum.com](https://www.manufactum.com/) :)
This store is developed by two teams using the here described techniques.

Related Techniques
- [Posts: Cookie Cutter Scaling](https://paulhammant.com/categories.html#Cookie_Cutter_Scaling) David Hammet wrote a series of blog posts on this topic.
- [Wikipedia: Java Portlet Specification](https://en.wikipedia.org/wiki/Java_Portlet_Specification) Specification that addresses similar topics for building enterprise portals.

Things to come ...

- Use Cases
  - Navigating between pages
    - soft vs. hard navigation
    - universal router
  - ...
- Side Topics
  - Isolated CSS / Coherent User Interface / Style Guides & Pattern Libraries
  - Performance on initial load
  - Performance while using the site
  - Loading CSS
  - Loading JS
  - Integration Testing
  - ...

Contributors
- [Koike Takayuki](https://github.com/koiketakayuki) who translated the site to [Japanese](https://micro-frontends-japanese.org/).
- [Jorge Beltrán](https://github.com/scipion) who translated the site to [Spanish](https://micro-frontends-es.org).

This site is generated by Github Pages. Its source can be found at [neuland/micro-frontends](https://github.com/neuland/micro-frontends/).