SMASHING BOOK 6

New FRONTIERS in WEB DESIGN

Making Design Systems Work In The Real World

by Laura Elizabeth

Many of us have been there before: at some point, somebody in the company suggests we build a design system. It sounds pretty straightforward: the team creates a group of reusable components that everybody can mix and match to build reliable, consistent interfaces faster. In fact, it also sounds like a great tool to improve the workflow: the team stops wasting time reinventing the wheel with every new feature. Instead, everybody can use previously tested code with baked-in accessibility, visual language, functionality, and naming conventions, and we all can focus on the bigger picture.

It's no wonder that according to UXPin's 2017–2018 Enterprise UX Industry Report,[1] 69% of enterprise companies either actively use a design system or are currently working on one.

Why, then, are we still struggling to make a design system stick? What seems to be the main problem? Is it that working with a component-driven workflow is too difficult, or that maintaining the design system feels like too much work? I wanted to find the answers. After speaking with many different companies, large and small, there are a few key reasons that kept cropping up:

1. The team didn't use it.
2. Management didn't support it.
3. After a while, the system stalled and it never recovered.

1 http://smashed.by/uxpinreport

The interesting thing to note here is that none of these are technical issues. Many people I spoke to during my research for this chapter did articulate issues with the tech but as it turned out, the tech wasn't *really* the problem. As we'll find out in this chapter, despite its robotic name, design systems are predominantly about one thing: people.

Roadblock #1: The Team Isn't Using The Design System

If you go to any conference or meet-up about design systems, I guarantee this question will come up at least once: "How do I convince other people in my organization to use the pattern library we created?"

It's an important question because ultimately, no matter how grand your design system is (whether it's a pattern library with sharable components, editorial style guides, and so on), if people don't use it, there's not much point to it being there.

So what stands in the way of people using a design system? I spoke to a developer who has been working on a small system in his spare time to help him and his team members produce website templates with already tested and quality assured components. He felt like he was solving a real pain, not only for himself but for his team. Working quickly with untested code is hard, not to mention risky. So he offered a solution to help but got met with resistance:

> *I thought the team would pick it up instantly. But I found there were a lot of roadblocks stopping people from using it, like needing to switch to different naming conventions. People were used to doing things a certain way, and didn't want to spend time changing their process which, for all intents and purposes, works perfectly fine."*

It turns out he needed a way to make the design system easier for his team members to adopt. As it stands, they don't have much of an incentive to use it. As far as they're concerned, their process is fine, albeit a little inefficient, but fine.

This is something they're still working on to this day but to understand why people don't use your design system, you first need to understand people and their motivations. Una Kravets argues that we need to be more empathetic to what people are being judged on at work.

> *We're all being judged on different things. Most people aren't being judged on clean code. The engineers we're building this for are being judged on their ability to build the product and ship features quickly.*

This can be applied to other disciplines, too. Designers aren't being judged on their articulation of design principles, or the use of symbols or smart objects in their tools. They're being judged on making great looking products that are easy to use and increase conversions.

Ultimately, if your design system doesn't help your team meet *their* goals, you're going to have a hard time convincing them to use it.

Why Are You Doing This?

The first question you need to ask yourself is why do you even want a design system in the first place? Or, more accurately, how will a design system help your company and its team members? You could in fact argue that *every* company would benefit from a design system.

And there's truth to that. If we could flick a switch and suddenly have a design system, it would no doubt be beneficial.

But you don't make a design system by flicking a switch, or pressing a button. A design system often takes an extraordinary amount of work, effort and time.

The benefits, or at least the promise of benefits, needs to be worth the headaches of systemizing your website. It needs to be worth the change in process and workflow that your team is going to go through.

So how can you find out whether a design system is really something your team needs?

Nathan Curtis from EightShapes recommends starting every design system with a phase of discovery:

> *A design system doesn't start with choosing a first color. Instead, ground a system in a strategy that discerns customer needs, sets objectives, explores and converges on a design direction, pitches a strategy, and obtain an organization's commitment.*
>
> *— Nathan Curtis, "Starting a Design System"*[2]

Before you even begin putting together your design system (or even if you've already started) you need to understand how your team currently works and what the biggest struggles are.

You can do this by talking to people directly, or sending out an anonymous survey. I find keeping questions fairly broad and open yield the best results. We're not trying to steer their answers in any way, we just want to find out how they work and what they'd like to change.

2 http://smashed.by/startds

We can ask questions like:

1. What is your favorite part about your job?
2. What is the most frustrating thing about your day-to-day work?
3. If you could get rid of one aspect of your job, what would that be?
4. What are the main inefficiencies in your design/development process?
5. In your opinion, how can your company improve in terms of enabling people to get their work done more effectively?

It's important to note that we haven't mentioned design systems at all. We're not asking "would you find a design system helpful?" or "do you wish you were more efficient?" because most people want to be more efficient. What we want to find out is whether efficiency is *the* problem that's keeping them up at night.

With this exercise, we're looking for any commonalities that people are struggling with. What do people want to fix in their jobs? What will make them happier, more effective people?

Once you've collected the data, whether it's a surveys or a transcripts of conversations, you can start to normalize the data. I usually do this by creating a spreadsheet with different category headings based on feedback. So maybe "lack of efficiency" is one, or "difficulty in collaborating."

Then you can paste the exact verbatim people have used beneath these.

For example, a developers problem might be that they're under pressure to keep building new features, but they're getting complex and inconsistent styling from the designers for each new component. The codebase is becoming messy and they're struggling to meet the deadlines.

Feedback	*Lack of efficiency*	*Need core styling*	*Can't build fast enough*	*Lack of shared vocabulary*	*Lack of collaboration*
Every time we launch a new feature, I have to create a new component. I can't meet the deadlines!	*x*				
We have to wait too long to launch new features or they are sub-par		*x*			
Developers are never happy with what I give them but I don't understand what they want from me			*x*		
I'd be more effective if I could work closely with designers on new features out			*x*		
I never have enough time to get everything done				*x*	
The code base is a mess and it's making me inefficient					*x*
The shared workspace is counter-productive. I wish there was a quite zone I could go when I need to get some focused work done.			*x*		
Inconsistent styling makes it hard for me to know which bits of design is 'our' design.		*x*			

Then you could have a product manager who is frustrated because they feel like they're waiting too long to launch new features. They also think that these features end up nothing like they had envisioned or talked about at the start.

And then you have designers, maybe they're frustrated because developers never seem happy with what they give them but they don't understand how they can be more helpful to them.

These could be problems that are cropping up over and over in your company.

And from these issues we can categorize them: whether it's creating core styles so developers aren't having to code new styles for every new feature. Or saving time by being able to launch faster using previously tested components. Or even just creating a shared vocabulary so people can talk to each other and work together more effectively

This will give you a good overview of the current problems in your company. The idea is that you'll use this to decide whether a design system is really what your company needs *right now*. Or is there something else that you need to fix first.

If the answers do show you could benefit from a design system, it'll also give you your why. For some companies, they might feel pretty efficient in their work, but they struggle to collaborate with each other. Maybe testing is a big roadblock for your company – you're having to spend too long testing every component, it means you can't push features out fast enough.

This will help you build your design system because you'll have a better understanding of *why* you're doing it. You'll be working towards what's best for your team, making the design system work for them and their needs, which will help with adoption.

Getting Your Team To Adopt The Design System

Unfortunately, it's all too easy for us to declare "our process is inefficient, we're solving a *real* problem" and expect that alone will be enough to ensure the survival of our design system. In her book *Thinking in Systems*, Donella Meadows states that "information about the existence of a problem may be necessary but not sufficient to trigger action – information about resources, incentives, and consequences is necessary too."

Let's look at these one by one.

RESOURCES

People are busy. No matter how much technology advances to make our lives more efficient, we always find ways to fill that time. Most people aren't sitting around at work, twiddling their thumbs, looking for projects to work on. We have deadlines, meetings, and the allure of Twitter vying for our immediate attention.

How can you demonstrate to your team members that you have the resources available to put this into action without adding to their already full plates?

The developer I spoke with earlier in the chapter tried to convince his coworkers to stay behind after hours so he could do a workshop on how to use the design system he was working on. But because management hadn't provided any additional resources to do this during the day, he had a very low turnout: "I found it tough getting people to stay after hours. If management allowed a workshop in working hours

I think I'd be more successful in showing people how a design system could help them."

Expecting people to spend their already limited free time learning about design systems sets the wrong precedent. You're already selling the design system as extra work from the get-go. It's important to make sure your team has the resources needed for a design system without putting the burden on your teammates.

INCENTIVES

We've already identified key problems that a design system could solve in your company and hopefully you've got a clear idea of what your coworkers desperately want fixed. In most cases, promising these pains will be solved is enough incentive.

When it doesn't work, the most likely cause is that you're not solving a problem painful enough for them. Sure, your coworkers want to be efficient, but is that enough? After all, they're getting paid the same amount each month no matter how efficient they are.

Remember, people are typically resistant to extra work. As Scott Berkun points out:[3]

> *Designers love to draft design principles, rules for their coworkers to follow to 'do good design' – this rarely works. [...] The principles always create more work, or at least demand more thinking. But humans avoid work. [...] There is little motivation for anyone to change their own behavior.*

3 http://smashed.by/designprinciples

If you're finding team members still aren't motivated, you may need to dig deeper. James Ferguson from Skyscanner knew that different teams had each spent weeks or months creating various date pickers for their website. He knew these were a massive pain for people to create and that it was unnecessary creating so many different variations when one would do.

If James could take away the need for all these teams to create their own date pickers and have a single, tested component that would solve a huge problem for them:

> *"Our job is to be a pain killer, if you like. A lot of stuff we do isn't sexy work. It's a total pain in the ass, but we're doing it so that other people don't have to do that multiple times."*

James wasn't selling being more efficient. He was selling no longer needing to recreate a date picker over and over again. It may sound trivial to some, but for his team this was a huge incentive to get some kind of system in place.

So find out what people are *really* struggling with. The kind of thing they'd pay someone to solve for them, and solve that first.

CONSEQUENCES

The last element mentioned is consequences. What will happen if a team *doesn't* get behind a design system?

Often, this is just a reverse of the incentives plus one additional caveat: The longer you wait before starting to systemize your website, the bigger the job is going to be.

Building a working system around your website isn't something that happens overnight. For many people, it's like organizing your home. It's that thing they know they're going to have to tackle at some point but the longer you leave it, the more you're going to have to clean up.

Getting Your Team To Adopt The System Organically

When we talk about incentivizing your team to get behind a design system, often there's an obvious question in the back of people's minds: Why not just impose a design system top-down?

It's a valid question. But if there's one theme that's cropped up again and again with companies who are successfully utilizing a design system it's: "You can't police a design system". What this means is you can't create a library of reusable components for example, and expect everyone to use and contribute it "because we said so". Any type of design system is only as good as the people using it, so if the people using it are doing it out of obligation, your design system is not going to be as stable.

This is something Brent Hardinge understood and took measures to curb during the creation of the design system for the Seventh-day Adventist Church.

> *We always knew organic growth was going to be better than having forced growth. We spent a lot of time building relationships with people who would benefit from the system. We'd meet them in person, talk to them and try to get to a certain level of trust before moving forward.*

Brent admits that this is a slower process, but he's found it to be far more lasting because through meeting people and talking to them, he's been able to create a deeper understanding of what a design system is and why it will benefit them.

Getting your team to use and contribute to a design system may not happen quickly, but as long as you're making their live easier it *will* happen. Instead of taking a top-down approach and requiring every team member to use the design system, try taking a more collaborative approach – show them the benefits and let them decide whether this is something they want to get involved in.

Roadblock #2: Management Doesn't Support Your Design System

In our industry, we talk a lot about whether designers should learn to code, or whether developers need to learn design principles. One subject which doesn't get spoken about enough is business. One way or another, we're all invested in business. Whether you work in-house at a company, at a web agency, or you're a freelancer. We are all hired for the same reason: to make a company more profitable.

Yes, on the face of it we've been hired to improve the user experience, or to build out new features. But the end goal is still to make a profit. It doesn't have to be in a cutthroat "we will make more money, no matter the cost" way. Increasing revenue means creating more jobs (and thus, supporting more families), better equipment and facilities, higher paychecks, and so on.

Those who understand that immediately make themselves more valuable to their employers. Not only that, they'll know exactly how to ask

for what they want and get a positive response. This is important when talking about design systems because as we mentioned earlier, we rely on getting support and resources for management in order to help us do the best jobs we can without sacrificing our personal time.

Ryan Rumsey wrote a great article about how he managed to get buy-in for his design system at EA. He had struggled with this previously with other projects so understood the value in putting together a case for a design system up front:

> *I've learned first-hand, through my experience, friends, and the design community at large, that asking for up-front buy-in on many projects is difficult. There's a certain amount of research and analysis needed to be done to get a good proposal together. It's made an even more difficult if data to evaluate the potential gains on efficiency is not available.*
>
> *— Ryan Rumsey, "Selling a Design System before asking for buy-in"*[4]

He found that asking for additional time or resources without anything tangible in terms of what the business will see for it meant the conversations often turned into subjective opinions which is something management will struggle to sign off.

So how can we get them on board. From my research, I've seen two approaches:

1. Create a small, working pattern library or style guide — then use the results from that to ask for support later.
2. Ask for permission to start sytemizing your website upfront before you do any work on it.

4 http://smashed.by/askbuyin

Create A Small System First And Ask For Support Later

Many teams have had great success creating a working design system, using that to prove it's value and then asking for additional resources later. Wolf Brüning from the German online retailer OTTO went down this route when his UX lead brought together a small team of three: himself, a front-end developer and an external UX consultant to start researching the best way to put together a design system.

They ended up bringing on one front end developer from each of their product teams and did a tightly scoped workshop which resulted in a small, but working, pattern library. The adoption of the pattern library happened slowly, but organically. By simply referencing the pattern library instead of giving detailed specifications when asked about a component, they encouraged their team members to use it. And as their team members started to see the benefits, they began using it independently. This then had a knock-on effect with management: "As management saw the benefits of saving time, cleaner code, better UI quality an reducing complexity we also got buy-in from there".

Dedicating a set amount of time to try out a small style guide or pattern library and measuring its effects across your team is the simplest way to get buy-in. If you've already created something and it's proving to be effective, you'll be hard pressed to find a manager who isn't willing to help sustain the efforts.

What If You Don't Have the Resources to Create and Measure a Design System without Support from Management?

Creating even a small pattern library can be a time-intensive process. Sometimes you simply don't have the resources at your disposal to be able to assemble a team, put one together *and* stay on top of your workload.

Not only that, involving a small portion of your team can actually impede adoption by alienating other people from the system. This is what Donna Chan and Isaak Hayes from AppDirect found during their early attempts at a design system:

> *Upon completion, many of the style guides were basically put in the trash and not adopted, in the sense that they were unusable for the intended users. We heard comments like "They were a waste of time" and "All these resources were put in and what came of it?"*

They learned that by trying to implement a design system on their own, they weren't helping the very people who were meant to be using it. Without support from management to work with everyone on the team, they were creating a system useful to only a handful of people.

In these cases, it's more effective to spend some up-front time ensuring you get buy-in from management so you get the resources you need to create something useful for your team. Which leads us to...

Asking For Permission Up-Front

The second option is asking for permission upfront, before you do any work on a design system. This has some obvious advantages. Mainly that you can get the time and resources you need to work on the system. You don't need to try to fit it around your current workload.

But getting permission for such a mammoth task is notoriously difficult. Many stakeholders won't necessarily know what a design system is, so there's an educational element. Then we have the problem of a design system *sounding* simple ("We're just making our website reusable – like Lego!"), stakeholders may find it difficult to understand why you need a whole team just to work on the design system.

And when we try to talk about metrics, we usually want to see either an increase in efficiency, saving money, or making a profit. In other words, we want to see a return on our investment (ROI). Is the time and money that we need to put in to create a design system going to be less than what we get out of it? Not necessarily, says Alla Kholmatova in her book *Design Systems*:[5] "Some teams struggle for a while to see a return on investment in a modular system, which makes it hard to justify the investment in the first place."

An ROI assumes that the design system is adopted by the team, used consistently, and enables them to get results faster. None of this can be truly guaranteed but as we'll go into later in the chapter, there are things we can do to dramatically increase its chances of success.

When approaching management, it's best not to go in with a full 5 year roadmap for a design system. Instead, it's easier to get approval if you

5 http://designsystemsbook.com/

start small and low-risk. Remember, at this stage you're still researching whether a design system is the best option for right now. For some teams, a style guide and basic pattern library might be enough. A full-fledged design system might be too much work for too little reward *right now.* And if that's the case, don't force it until your company is ready.

So what exactly do management need to see in order to invest in the project? At a minimum, you'll need to be answering questions like:

- What is a design system and why would your company benefit from one?
- What kinds of results have similar companies been getting from implementing a design system?
- What resources do you need?
- What do you hope to achieve with a design system?
- What's your roadmap for getting one off the ground?
- How will you measure results and stay on track?

The first three points are pretty straightforward to get together so it's the last three I'd like to focus on now.

Setting Goals, Coming Up With A Roadmap, And Measuring Results

From our last exercise, we already have a start when it comes to goals. Those pains we identified at the start are what we are aiming to eliminate. Now we need to go a bit deeper and turn those problems into something we can work toward solving.

One goal-setting solution that's gaining popularity is Google's Objectives and Key Results (OKRs). What I really like about OKRs is that they're used with team members, not just management. They empower every team with tasks that contribute to the higher-level business goals of an organization. In a study by Sears Holding Company in 2013, they found that OKRs have actually been shown to improve staff efficiency and the bottom line with their sales teams[6]

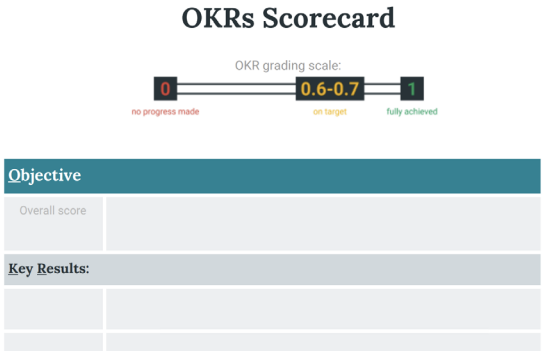
> *For the group who used OKRs we saw an increase in their average sales per hour from $14.44 per hour to $15.67, or an average increase of 8.5%. This increase is not only statistically significant, but practically significant.*
>
> *— Chris Mason, Senior Director, Strategic Talent Solutions at Sears Holding Company*

An OKR is essentially a framework of defining and tracking objectives and their outcomes. Honestly, it seems pretty similar to most goal-setting techniques but I think OKRs just give a really simple, no-fluff framework which is:

1. Where do I want to go? AKA The objective.
2. How will I know if I've got there? AKA the key results.

You can use OKRs in any aspect of an organization (like the sales team example above), including your design system. The reason OKRs are great for design systems is because you're treating the system as a means to an end. In other words, the result isn't having a design system. The result is improving efficiency, or shipping more products.

6 http://smashed.by/sears



This works well for management because instead of saying "This design system will absolutely make us more efficient, which will enable us to release more products and features, which will save us money," you're saying "We think this will help us do the former, and here's how we're going to test that hypothesis. We'll keep you in the loop the whole way and will only ask for more budget **if we can prove to you that this is effective**."

In doing this, you're reducing the risk for management. Instead of asking for half the team to create something which may or may not ever be adopted, you're asking for a few people in a limited time frame to test something that could drastically improve the company.

Let's look at an example of a company who have used OKRs successfully with a design system. Brent Hardinge from the Seventh-day Adventist Church worked with Dan Mall from SuperFriendly on their design system and they lined out exactly what they wanted to get out of it and how they would measure whether it's been successful or not.

Their primary goal was to create a pattern library that a team member could use to envision what she wants to build. It also needed to contain the code that could help their developer quickly bring her vision to life.

That's their why. This will be different for every company. What's important is that they've clearly defined what the goal is.

They then split this into three objectives that will help them achieve this goal and laid out exactly how they'd measure it:[7]

1. "**Create a foundational, deeply-rooted pattern library.**
   - 1,800 Adventist websites (15% of the 12,000 sites) make obvious use ALPS.
   - 57 websites made by the General Conference (50% of the 114 sites produced by the GC) make obvious use of ALPS.
   - The first Adventist websites built for certain languages report 0 issues when building."

2. "**Allow for customization and individuality in the new pattern library.**
   - A trained person can build a site with the new design system in two days.
   - 2,400 Adventist sites (20% of the 12,000 websites) use one of the color preset options."

3. "**Involve the community in the creation and adoption of the pattern library.**
   - 18 unions (30%) register in a feedback program.
   - 3 ideas originating from the community not included in the initial delivery of the design system have been adopted.
   - Design system adopted by 3 customers that weren't part of the initial interviews or any feedback program."

7 http://smashed.by/adventist

They kept track of each goal as they progressed with their design system and it gave them the motivation to keep going and get the results they hoped for. Some they achieved, others they didn't but OKRs are meant to be adapted as you learn more about what's achievable. This means if you set the bar too low or high, or are tracking the wrong metrics you can change them.

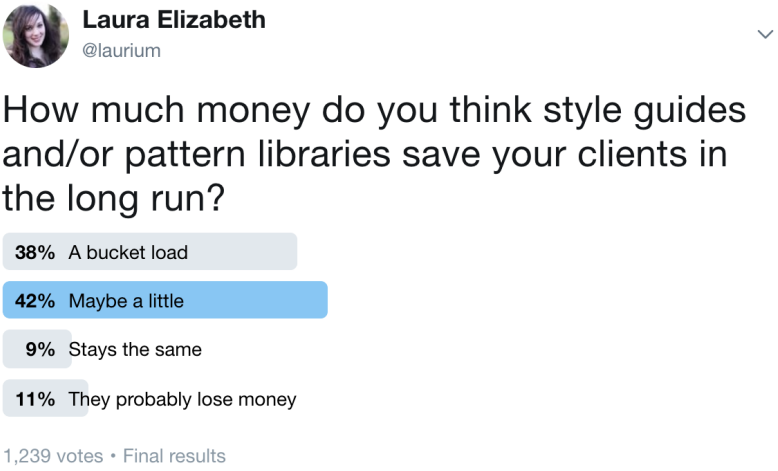
On reflection, Brent found the OKRs essential "The project is so broad, so the OKRs really helped us narrow down on what we were building and doing." When the time came to review their goals, they could celebrate the ones they achieved and, more importantly, review the ones which hadn't before setting new goals.

HOW TO MEASURE WHEN THE DATA IS SUBJECTIVE

A design system's effectiveness is notoriously difficult to measure. When I first started selling a systemized solution (like a pattern library) to clients, I tried to focus on the financial benefits. The reason being, if you can prove something will either make more money or save money, it becomes the easiest thing to sell.

However, just *saying* a pattern library would save money wasn't enough. I always felt a bit icky making a claim that I didn't know was true. So I decided to go out and look for evidence.

In a survey on Twitter of over 1,200 people, a whopping 80% of them said they think style guides and/or pattern libraries save money. 9% said it made no difference and 11% said they probably lose money in the creation of them.



But when I tried to dig deeper I found that nobody seemed to have any proof that money was saved. Their answers were based on intelligent reasoning but no evidence.

The issue is that a design system is always interconnected with other projects and improvements. A company doesn't shut down to build a pattern library. This makes it nearly impossible to define exactly which figures the design system is responsible for and which were from an entirely different change.

But there are other benefits to a design system. And there are benefits that would appeal directly to management. Here are four benefits that will get your managers or stakeholders on board:

1. **Getting new features into production much faster.**
   Management like to see movement. They like to see their company progressing and feel like they're keeping up with their competitors. Getting new features shipped faster would be a great incentive for most managers.

2. **Makes their company more attractive when hiring.**
Managers deeply understand the importance of hiring the right people for a job. They put a lot of stock into making sure their company is attractive for potential employees. The benefits of a more systemized way of working has been gaining traction in the last few years and many of the top designers and developers are specifically looking for roles in companies that prioritize this. If a systemized way of working makes their company more attractive to exceptional employees, they'll more than likely listen.

3. **Saves the cost of a redesign every other year.**
Redesigns or rebrands are always going to happen. But if you have a large website, a redesign is not only going to be incredibly time-consuming, but it'll cost the company an awful lot of money. To make matters worse, giant overhauls can have a negative impact on both SEO and user experience. In my mind, a redesign is far more risky than a design system!

   When you have a design system, you'll still be able to rebrand, optimize for conversions and improve the UX but you'll be doing it in a more sustainable and more effective way. You can work on and test small chunks of the website over time, rather than trying to overhaul the entire thing. Unlike a redesign which decreases in value over time, a design system has the opposite effect: it continuously increases in value.

4. **Everyone else is doing it!**
Yes, managers experience FOMO too. Often, just knowing that this is something a handful of their competitors have bought into is enough to get them to listen up and take your request seriously.

This all goes back to what we talked about earlier. Ask yourself "What is my manager being judged on?" Are they being judged on how well designers and developers work together? Not exactly. They're being judged on how they fare next to their competitors, they're being judged on profit, jobs being created, growth.

So now we know the outcomes we're looking for, we can get a better idea as to how to measure those effects. We can split these into two categories: measuring based on shipping and measuring based on time saved. Let's look at two case studies that have made these work.

Amazon Web Services Increased the Amount of Features They Shipped by 98% in One Year

Thomas Lobinger leads the team responsible for the design system at Amazon Web Services. Thomas noticed that since launching they went from releasing 722 new services and features in 2015 to a whopping 1,430 (which averages out at around 4 per day) in 2017. AWS offers over 100 services today. Each service team is developing their service's UI so having a central system where everybody could use previously tested and approved components and user flows was a huge factor in releasing so much, so quickly.

Not only that, they found that with the design system they were continuously improving the UX, accessibility and conversion rates throughout the suite of Amazon Web Services. When something worked, they could apply it elsewhere and dramatically improve their services in a fraction of the time.

When talking about the success of the system, Thomas said:

> *We were able to sell and grow the team because of the already apparent side effects from the simple UI components that we developed in the first weeks. From the very beginning, there was already customer value created. I would advise teams to start even if they don't feel like all the conditions are met. A small team and system can already provide huge benefits.*

FutureLearn Found That a Component Had to Be Used Just Two or Three Times before It Became a Net Positive

At FutureLearn, Alla Kholmatova and her team did some experiments to measure the effect systemizing their components was having on staff efficiency. She found that on average a reusable component took around 2-3x longer to create than a one-off alternative but the time was made up when they reused a component at least twice. So for FutureLearn, this gave them a good baseline as to when it would make sense for the team to create a reusable component vs. a custom one.

There is one small caveat with measuring a design system based on saving time. That is that you can inadvertently have a negative effect on the end user. As Alla and her team found out, the increase in productivity they enjoyed had led them to proactively try to reuse a component as many times as possible. After all, the more they used it, the more cost-effective it became. However, this didn't always end up with the best results:

> *We then found we were reusing components for the sake of it, to fit it into our pattern library, even if it wasn't the best solution for the user. Now we're spending time redoing components to make them more usable.*

So by all means, encourage and celebrate efficiency with your design system, just make sure you do what FutureLearn did and keep an eye on your end-users. Make sure your reusable components aren't detrimental to their experience.

So now we've managed to get management on board and we're clued up on how to encourage your team to use the design system, what happens after the initial rush wears off? What happens if (or more likely when) you stall?

Roadblock #3: Your Design System Stalls

Pretty much every design system stalls at some point. No matter how motivated you are at the start, at times progress is going to feel slow and frustrating. It's simply a part of creating a design system.

When you first start to work on the system, there will probably be a ton of excitement. People will be motivated and you'll feel like things are going to move very quickly. Then, you'll get hit with roadblock after roadblock. You'll end up in these long, drawn out discussions about naming conventions or trying to work out some problem that you hadn't foreseen.

And before you know it, weeks have passed and your boss is asking to see progress on the design system. You don't have anything and you're essentially just spinning your wheels and the program risks getting axed.

This is a pretty pessimistic outlook but not uncommon. Fortunately, there's a fix. The problem isn't that you're not cut out for systemizing

your website. It's not that your company is too complex to make a design system work. It's usually just a misalignment between strategy and implementation. These are two very different but equally important tasks and it's important to understand and recognize the difference.

What tends to happen is that we mix these two tasks up. We're implementing something and the conversation turns to strategy. We get on tangents, we find new issues that we hadn't considered before and then at the end of the day, week, or month we realize we're no farther along. We've made virtually no measurable progress aside from a lot of conversations.

So what we need to do is give ourselves dedicated time for each task. Save up all the big-picture conversations for a strategy day, where you get together the key decision-makers in your design system team and iron out all the high-level details that have been cropping up. It also gives you a chance to reassess your goals and make any adjustments until the next strategy day.

These questions could include things like:

- Should our components all follow naming conventions based on a particular theme?
- Should we open-source or build a marketing website around our design system?
- How should we accept contributions to the design system? Can anybody contribute or should we have an application process?

These are questions that are probably going to require a lot of discussion. Saving them up for when you have the time to address them is

going to be a huge help for you and your team. It leaves you free to work on making measurable progress with the design system, without being held back by too many roadblocks. And when you're working on your day-to-day stuff, if you find a conversation turns to strategy, you can simply delay talking about it until your next strategy session.

This is simply an exercise in learning to identify the difference between strategy and implementation, noticing it and being able to compartmentalize it. If you do this, you'll find your design system makes progress and you'll be far less likely to give up out of frustration.

If you still find yourself stalling and your team is lacking in motivation, Alan Wilson, a designer at Adobe, finds getting passed a stall is often as simple as taking a break

> *I find that I need occasional breaks – sometimes just an hour or a day, sometimes longer, like a week – to work on other (non-design system) things. For instance, it's nice to design a product using your design system, which means you come back to design systems with more empathy for your customer.*

He finds that switching from working *on* a design system to working *with* a design system can be enough for you to step back, see it with fresh eyes and understand what you need to do next to keep it progressing.

Above all, we need to remember that stalls are natural and they don't necessarily mean your system is broken. But they can also indicate that there's something wrong with the direction you're going in. Some companies find they can't progress until they change how the system is governed. Others find they can't progress until they change how components are organized.

It doesn't mean you've failed, but it does mean that you need to take a step back and try to figure out why your system is stalling. Is it because the effort you're putting in isn't equal to the benefits? (And if so, why?) Or is it just because you haven't seen tangible progress for a few weeks?

A Roadmap For A Successful Design System

Now we've gone over the three main roadblocks with making a design system stick (team members didn't adopt it, management didn't support it and it stalled but never recovered), now let's look at a roadmap for how you can start a design system and give yours the best chance of succeeding.

1. Find Your Purpose

The first thing as we talked about at the beginning of this chapter is to find what your motivation is for having a design system. Why do you want one? What will that do for your fellow team members? How will this serve your users? And how will this help grow your company?

It's important not to gloss over this stage and come up with vague sounding principles. Dig deep into your companies culture and find out whether a design system is what's needed *right now*. (If it's not, don't do it yet.) As you'll come to realize, if you haven't already, a design system is not about technology. It's about people. A big part of this journey is going to be spent talking to people and helping solve their problems.

2. Start Small, With An Experiment

There's a lot of risk in creating a design system. What if nobody uses it? What if we put in all this work and we don't see any benefits? I wouldn't recommend anyone going all in on systemizing all of your flagship projects at once.

It's much safer to test the waters with something small. Maybe there's a new feature that you can test your system on? Perhaps there's a particular flow that (if you did nothing else) would benefit a large amount of people. Even a simple style guide is a good exercise to start seeing how this could benefit your company.

So when starting a design system, look for something small that you can test the system and its adoption on. The goal is to help people do their jobs better and more efficiently. If you do that, you won't get any backlash. If you do find you're getting backlash or people aren't interested, you may need to go back and assess why.

3. Get Past The Stall (And Have Regular Check-Ins)

Setting up regular check-ins to focus on the goals and strategy of a design system is going to help reduce the likelihood of a stall halting work on your design system. Make sure you spend at least one day in regular intervals (say every 4-6 weeks) taking a high-level look at your design system and reassessing your goals. This frees up your day-to-day work and enables you to focus on getting things done, as opposed to being stuck in long conversations about the overall strategy.

4. Don't Focus On What Everyone Else Is Doing

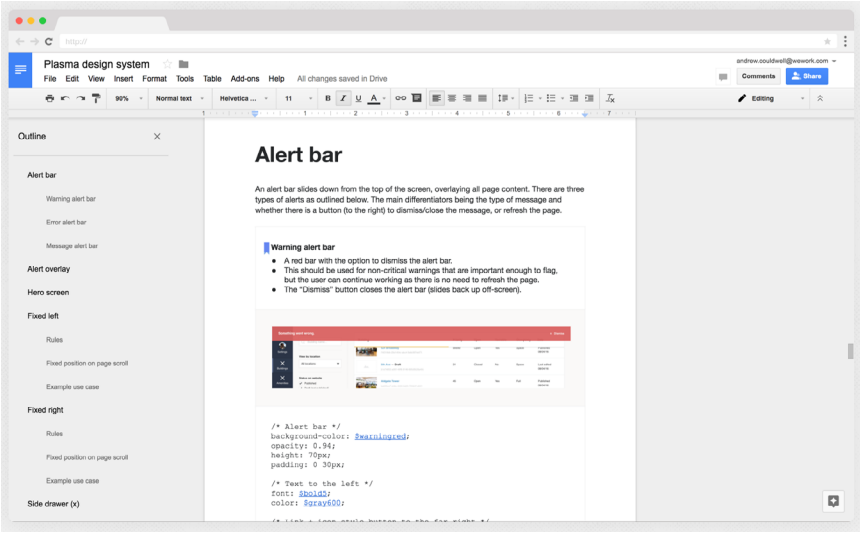
It's easy to get caught up in what other companies are doing with their design systems. Every day, it seems like bigger, better, shinier design systems are being unveiled. Focusing too much on this can end up crippling your efforts to create a useful design system. You'll see things that other companies are doing and think "We need to do that too!" But just remember that a design system's success is measured by how many people it helps, not how many claps you get on Medium.

A great example of a team who did their own thing despite everyone else doing it another way is WeWork. When they built out their design system, Plasma, they assumed they would create a dedicated website to show the system, specs, examples and guidelines because it's the done thing. They started up a Google doc to get all their components documented before pushing it to a live website, but over time they realized the doc had everything they need to document a design system. It has built-in navigation, the ability to add comments and collaborate, it's accessible to everyone on their team and doesn't need any special skills to update it.

Andrew Couldwell, the design lead on Plasma said about the success of the Google doc:

> *As the document grew, I realized it did exactly what we needed it to – the only reason to create a public, branded website for this would be as a "pride project" for WeWork Digital, or as a resource if we open-source the system.*
>
> *– Andrew Couldwell, "Plasma design system"*[8]

8 http://smashed.by/plasma



There are many benefits to creating a public design system (accountability, PR, education) but WeWork's ability to look past what everyone else is doing and make a judgement on whether this was the best solution for them, right now, ended up paying off. So don't worry about what you feel like you *should* be doing, just work on what's going to be most effective for your company.

5. Become An Evangelist...

Having at least one person in charge of keeping the design system front-of-mind is going to be a huge help when working on a design system. When you're first starting out, it's not likely that you're going to have a full-time role for a design systems lead (but as you grow, this becomes a necessity), but you will benefit from somebody who is going to act as an evangelist for the system.

The evangelists job is to be in charge of the design system, acting as a go-between for everybody using it. For that reason, it's important to have someone who can easily speak to designers, developers, marketers, editors etc. You're looking for a people person who's incredibly enthusiastic about systems – not necessarily the most technical.

These people are also in charge of educating the team about design systems and keeping everyone updated with progress. Alan Lindsay from Experian organized an entire conference called One UX for their team members around the world to learn about design systems and get excited about using them.

Even though Experian has incredible design talent and has always been a very innovative company, prior to the conference they operated in disparate design teams that sometimes didn't even know of each other's existence. Alan ended up becoming "chief evangelist" and uncovered pockets of amazing talent all across the globe, organized a community where they discussed challenges and eventually met together at One UX to work out solutions, including creating the global design system they enjoy today. Considering so many people contribute to the design system, they found having at least one person in charge of evangelism a huge help:

Given our current situation, evangelism was absolutely essential – someone had to dedicate time and back this vision to make it real and it was my privilege to start a conversation and gather these incredible people in the same room so we could elevate the company together. Now we're in the process of formalizing several roles that naturally emerged through this journey because the business recognizes how valuable managing this collaboration is. It's not always easy – there are challenges with evangelism like every other area of business – but the payoff is completely worth it!

So to have a successful design system, there needs to be at least one person in charge of keeping it front of mind, making sure everyone is happy with it, collating feedback and educating others.

6. ...But Don't Mandate

Perhaps the most important thing to remember if you want a successful design system is that no matter how much you believe in it, don't force anyone to use it. You shouldn't have to persuade anyone to use a design system. The benefits should be so clear that people will want to use it off their own backs. As Wolf Brüning said, "The most evangelizing is done by our library itself, as it provides a great way to save time and build a better product by not reinventing the wheel again and again."

If getting people to use your design system is like pulling teeth, it's not doing its job.

A Design System Is About People

Design systems are often presented as a technical challenge. What tech stacks should we use? How do we make it "living"? Is this a molecule or an organism? These are certainly valid questions but not the most important ones. Design systems that succeed are not the ones with the best tech, they're the ones that help the most people.

As James Ferguson puts it:

> *"It's not about your design skills, it's not about your engineering skills, it's about your problem and people skills. Trying to find that level of people skills **and** engineering is difficult. You need to get out there and speak to people, help people out and be really proactive. That's the biggest thing I think, that's the hardest problem, is people. It's always people."*

So remember, if your design system improves efficiency, team collaboration, gives your website a better user experience and even impacts the bottom line, you're onto a winner. But at the same time, no design system is perfect. Striving for perfection is admirable but unrealistic. Perhaps we need to learn to be OK with imperfection because the most successful systems are the ones that can embrace those imperfections, adapt quickly, and have a clear purpose that benefits not just businesses, but people.

About The Author

Laura Elizabeth is a designer turned product creator. She predominantly helps developers conquer their fear of design and runs designacademy.io, a training platform where developers can learn how to design.

SMASHING BOOK 6

New FRONTIERS in WEB DESIGN

Production-Ready Grid Layouts

by Rachel Andrew

CSS Grid Layout launched in the majority of browsers during 2017, with Chrome, Firefox and Safari all shipping their implementations of the specification in March, and Edge following up with an updated implementation in October. In this chapter I'm not going to attempt to teach all of the specification; instead I'm going to assume that you have followed some of the existing tutorials referenced at the end and are ready to get started using Grid in production.

I'll cover some of the main things that trip people up when starting to use Grid Layout. I'll also look at some fundamental parts of CSS that become very important when using layout methods such as Grid and Flexible Box Layout (Flexbox). Key to both of these methods are concepts of sizing, space distribution and alignment. My hope is that by reading this chapter, and building the examples, you'll have a better understanding not only of Grid, but of CSS layout as a whole.

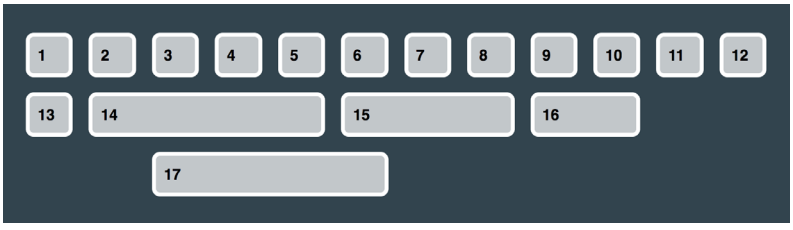
Getting Into a Grid Mindset

To understand Grid Layout, and become confident with the methods it enables, you need to make a shift in thinking. This involves no longer thinking about creating layout by giving sizes to items and lining them up with one another, and instead thinking about creating a grid on the container and placing the items into it.

Consider a float-based layout. The main layout method since responsive design arrived on the scene had us assign a percentage width to

floated items. We carefully calculated our percentages, making sure we didn't end up with more than 100% per "row," and achieved something that looked like a grid, and that maintained the correct ratios as the available width changed.

```
.wrapper {
  max-width: 980px;
  margin: 0 auto;
  padding-right: 2.093333%;
}
.col {
  margin-bottom: 1em;
  margin-left: 2.093333%;
  width: 6.20%;
  float: left;
}
.row::after {
  content: "";
  display: block;
  clear: both;
}
 .col.span2 { width: 14.493333%; } /* total of 2 column tracks plus 1
margin-left */
 .col.span3 { width: 22.786666%; } /* total of 3 column tracks plus 2
margin-left */
 .col.span4 { width: 31.079999%; } /* total of 4 column tracks plus 3
margin-left */
 .col.push2 { margin-left: 18.679999%; } /* total of 2 column tracks
plus margins */
```



We can create something that looks like a grid using floats.

We then gained Flexbox. However this one-dimensional layout method isn't suited to grid-based design either. While you declare `display:flex` on the parent, to make things line up in rows and columns we have to return to adding percentages to our items and their margins to achieve something that looks like a grid.

```
.wrapper {
  max-width: 980px;
  margin: 0 auto;
  padding-right: 2.093333%;
}
.col {
  margin-bottom: 1em;
  margin-left: 2.093333%;
  width: 6.20%;
  flex: 0 0 auto;
}
.row {
  display: flex;
}
.col.span2 { width: 14.493333%; } /* total of 2 column tracks plus 1
margin-left */
.col.span3 { width: 22.786666%; } /* total of 3 column tracks plus 2
margin-left */
.col.span4 { width: 31.079999%; } /* total of 4 column tracks plus 3
margin-left */
.col.push2 { margin-left: 18.679999%; } /* total of 2 column tracks plus
margins */
```

Grid behaves differently. We define our grid on the container and, while doing so, also define *tracks* for columns and rows. We assign the sizing of these tracks from the container, rather than adding sizing to the items. As we shall see, this doesn't mean items cannot affect the track sizes. But sizing *starts* with the grid container. In this way, Grid Layout is unlike previous layout methods. The only layout method that behaves similarly is Multi-column Layout.

Once we have our track sizing, all that needs to be added to rules on the items is a description of how many column or row tracks the item should span. That might start from a fixed position on the grid, such as the item in the code below with a class of `special`, which starts on line 3 and ends on line 7. Or we might use auto-placement, asking items to start where they would be placed with `auto` and span a certain number of tracks.

```
.wrapper {
  max-width: 980px;
  margin: 0 auto;
  display: grid;
  grid-gap: 20px;
  grid-template-columns: repeat(12, minmax(0,1fr));
}
.col.span2 { grid-column: auto / span 2; } /* start where grid would
place the item, span 2 tracks */
.col.span3 { grid-column: auto / span 3; }
.col.span4 { grid-column: auto / span 4; }
.col.special { grid-column: 3 / 7; }
```

With a layout method unlike anything that came before, a number of new ways to define sizing have been created. They allow us to distribute space in proportion, to set minimum and maximum track sizes, to indicate that we want the size of the content to dictate track size, and to mix fixed-size tracks with flexible ones. It is with these new sizing methods that I want to start, as I believe that a lot of the apparent complexity of Grid is because we see all of these new methods, and wonder why things got so complicated.

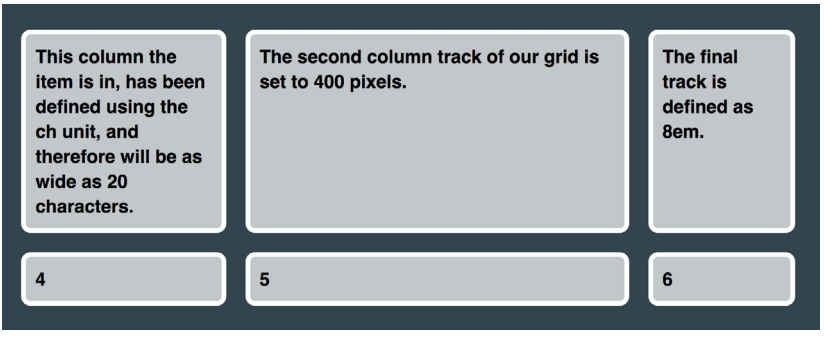
Existing Length Units

We can start very simply with sizing we are familiar with. Grid tracks can be created using existing length units, which are specified in

the CSS Values and Units specification.[1] You can create tracks with pixel-sized widths, ems or even the ch unit, which denotes width by number of characters.

In the following example, I have created a grid with one column set to 20ch; a second column is defined in pixels and a third as ems. The first column will be as wide as 20 characters of the character 0 (zero) in the font size used for the grid container. The second is 400 pixels wide and the third 8em. I have added a second row of grid items to demonstrate that this sizing continues to the next row of the grid. It is vital to remember that Grid is a two-dimensional model. If you set sizing for a track, it takes effect all the way down the columns and along the rows.

```
.grid {
  display: grid;
  grid-template-columns: 20ch 400px 8em;
  grid-gap: 20px;
}
```



Grid tracks defined using length units.

1 http://smashed.by/lengths

You can also use percentages, just as with a float-based or flex "grid." This approach can be useful if you are trying to ensure that a grid component introduced into an existing design lines up with other elements, which likely have percentage-based sizing. As we shall see, however, using percentages for sizing and having to work out the math yourself is something we'll probably need to do less of in the future.

```
.grid {
  display: grid;
  grid-template-columns: 22% 50% 22%;
  grid-column-gap: 3%;
  grid-row-gap: 2em;
}
```

A grid with tracks sized in percentages.

PERCENTAGE SIZING AND GRID-GAP

The `grid-gap` property is a shorthand for `grid-column-gap` and `grid-row-gap`. It enables the creation of gaps between grid tracks. If you use percentage sizing for your gaps, this will work as you would expect for column gaps. The gap essentially acts like any percentage-sized track: you need to make sure you do not end up with a total of more than 100%.

With row gaps, what the percentage resolves against is less obvious. Should it use the width, so that a gap of 10% is the same for rows and columns, or should it use the height? Currently, results across browsers are not the same, so you should avoid using a percentage value for `grid-row-gap`. In the example above I used em instead.

Sharing Space with the fr Unit

The unit that will save us from the need to calculate percentages is the fr unit. The fr unit is described as a *flexible length* in the CSS Grid Specification,[2] and is not part of CSS Values and Units as it is not a length unit. This is helpful to remember as fr cannot be combined with other units in `calc()` because it doesn't represent a length in the way other length units do.

The fr unit allows us to distribute space in the grid container, and to do so in proportion. If you have used Flexbox, and distributed available space by giving flex items a different `flex-grow` value, then you will find the behavior of the fr unit very similar.

In the example below I have a grid container with one `1fr` track, and two `2fr` tracks. This means that the available space in the grid container is divided into five: one part given to the first track, and two parts each to the second and third tracks.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 2fr 2fr;
}
```

2 http://smashed.by/frunit

The first track is `1fr`*, the next two are each* `2fr`*.*

To achieve a similar layout using Flexbox, you would use the CSS below. Using the flex properties I set my first item to a `flex-grow` factor of 1, and the next two items to have a `flex-grow` factor of 2:

```
.flex {
  display: flex;
}
.flex > :nth-child(1) {
  flex: 1 0 0;
}
.flex > :nth-child(2),
.flex > :nth-child(3){
  flex: 2 0 0 ;
}
```

This example also highlights the point made earlier about Grid working on the container and Flexbox on the items. With Flexbox you target only that individual item; we have to go to each item and decide how much it can grow or shrink in proportion to the others. With Grid we size the full track, which means that every item in that column will go into the track created by the fr unit sizing.

It would be possible to get the above Grid layout using percentages. What would be harder when using percentages, though, is when we want to mix fixed-length and flexible tracks.

We can see a very simple example of this if we add 20-pixel gaps to our grid. Before the fr units are calculated, the amount of space required for the gaps is taken away from the total available space.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 2fr 2fr;
  grid-gap: 20px;
}
```

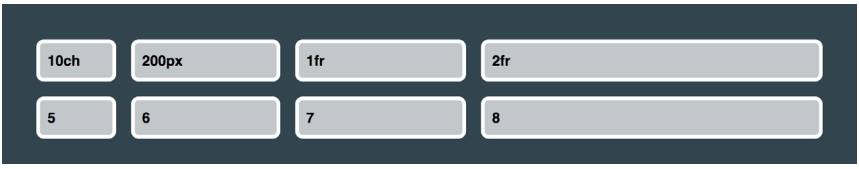
The fr units share out the space after accounting for the gaps.

The fr unit shares out *available space*, which means you can ask it only to share out space left *after* fixed-size tracks have been created. So if your layout requires some fixed-size elements in it, this is not a problem. In the next example, I have a first track of `10ch`, one of 200px, and then two flexible tracks of `1fr` and `2fr`. The available space the flexible length tracks have to play with is whatever is left after the two fixed tracks have been laid out.

The `ch` unit is relative to the size of the character `0` (zero).

This available space is then shared into three parts: one part given to the first track, and two parts to the second track.

```
.grid {
  display: grid;
  grid-template-columns: 10ch 200px 1fr 2fr;
  grid-gap: 20px;
}
```



Fixed and flexible tracks in the grid definition.

Content-Sized Tracks

In addition to tracks using familiar length units, and those using a flexible length, we also have ways to allow the content to dictate the track sizing. When we start to do this, we have to keep in mind the two-dimensional nature of a Grid layout. If an item in a row or a column track is able to change the size of the track, it will change the size of that track all the way down the column or along the row. This is essentially the big difference between Grid and Flexbox. If you do not want items to change the size of entire rows in *both* dimensions, a flex layout is probably what you're looking for.

auto Track Sizes

The simplest way to allow content to change track sizing is to use a value of `auto` for grid tracks. This is the default for grid tracks and something you will be familiar with if you have created a grid and specified only the column sizes, allowing rows to be created using auto-placement. Those rows are `auto`-sized by default. This means that whatever content is placed into them, the row will grow to be the height of that content.

If auto is used for column track sizing and you have not used any of the CSS Box Alignment properties discussed later in this chapter, the column will stretch to take up available space in the grid – so the track may end up larger than is required for the content inside. In the next example, I have set justify-content to start in order that the tracks do not stretch.

```
.grid {
  display: grid;
  grid-template-columns: auto auto auto;
  grid-gap: 20px;
  justify-content: start;
}
```

All of the columns as `auto` sized.

Content Sizing Keywords

Using `auto` for track sizing tends to be what we want for rows, but for columns we often want a little more control. This is where the new keyword values for `width` and `height` defined in the CSS Intrinsic and Extrinsic Sizing specification[3] come in very useful. These new keywords are `min-content`, `max-content` and `fit-content`.

These keywords are not specifically for Grid layout. They can be used anywhere you might use a width or a height (although browser support is currently limited outside Grid). For example, we can look at how `min-content` behaves by giving a `div` which contains a string of text a `width` of `min-content`.

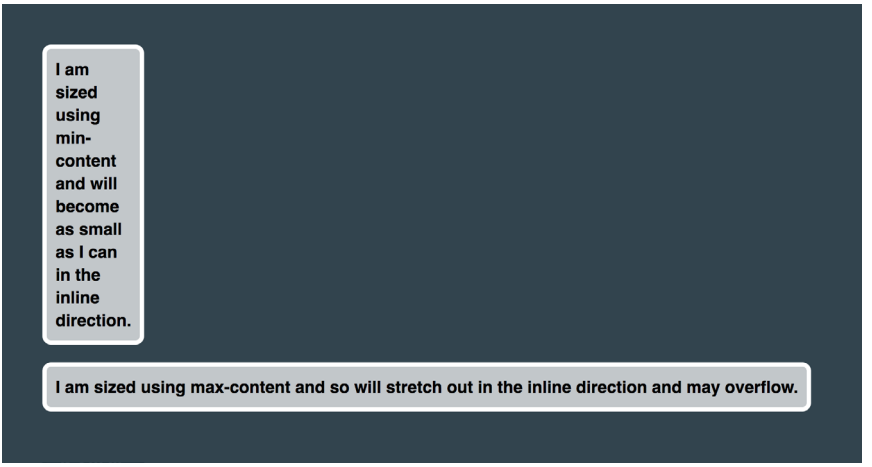
```
.min-content {
 width: min-content;
}
```

The element displays on the page with a width defined by the longest word in the string. The text takes all possible soft-wrapping opportunities, becoming as small as it can be.

If we were instead to use the keyword value `max-content`, the element would become as large as it could possibly be, taking none of the possible soft-wrapping opportunities. This could cause overflow as the item now does not wrap to fit into the container.

```
.max-content {
  width: max-content;
}
```

3 http://smashed.by/minmaxcontent

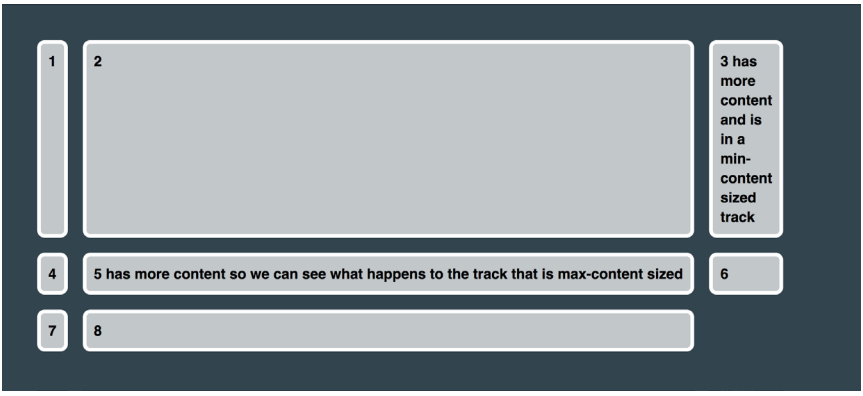


The first box has a width of `min-content`*, and the second a width of* `max-content`.

If we bring these keywords into our `grid-template-columns` and `grid-template-rows` definitions, we can use them to help define track sizing based on the content of the tracks.

In the example below, I am using `min-content` and `max-content` to define column tracks. You can see how the tracks are sized based on the content, and that this sizing happens all the way down the column track.

```
.grid {
  display: grid;
  grid-template-columns: min-content max-content min-content;
  grid-gap: 20px;
}
```



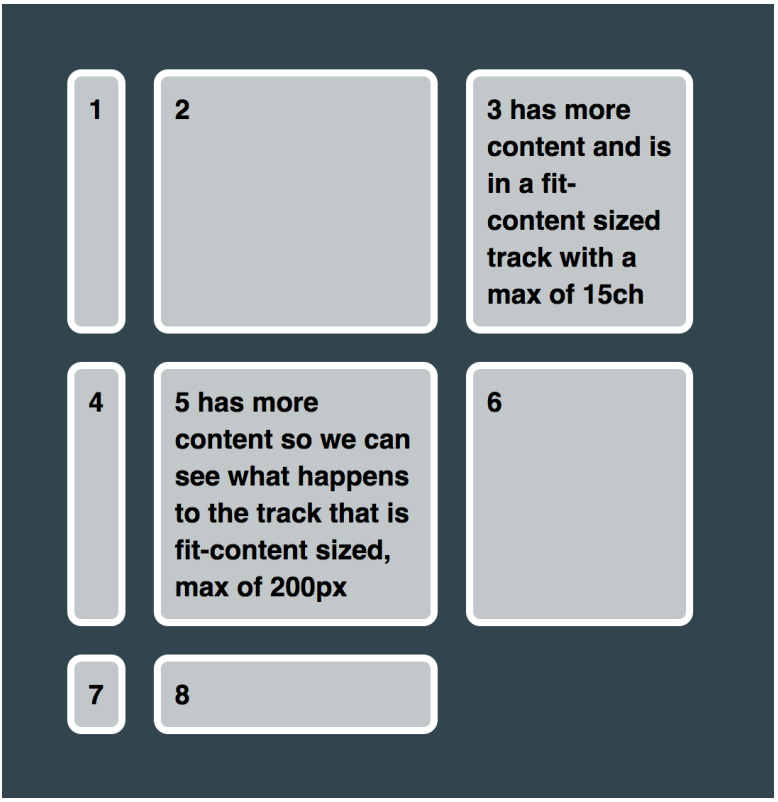
Using `min-content` *and* `max-content` *for grid tracks.*

Using `max-content` to force a track not to wrap at all has the potential to cause overflows. There is a final new keyword of `fit-content` that we can use to get the best of both worlds. It provides a defined limit to the maximum size a track can reach.

The `fit-content` keyword can be passed a value that becomes the maximum size of the track. The track behaves as though sized with `max-content`, becoming as big as it can and not taking advantage of soft-wrapping opportunities. However, once it reaches the value entered it stops, and then text inside wraps. In the next example, I have three tracks: the first two use `fit-content(200px)`, and the last track uses `fit-content(15ch)`.

```
.grid {
  display: grid;
  grid-template-columns: fit-content(200px) fit-content(200px) fit-
content(15ch);
  grid-gap: 20px;
}
```

In the first column track there is nothing that makes the track wider than 200 pixels, so the track uses `max-content` sizing. The second track has a long string of text, so when it hits 200 pixels, the track stops growing and the content wraps. The same is true for the third track sized at `15ch` (15 characters). There is a string in that track longer than `15ch` and so the track maxes out and the content wraps.



Using `fit-content` for track sizes.

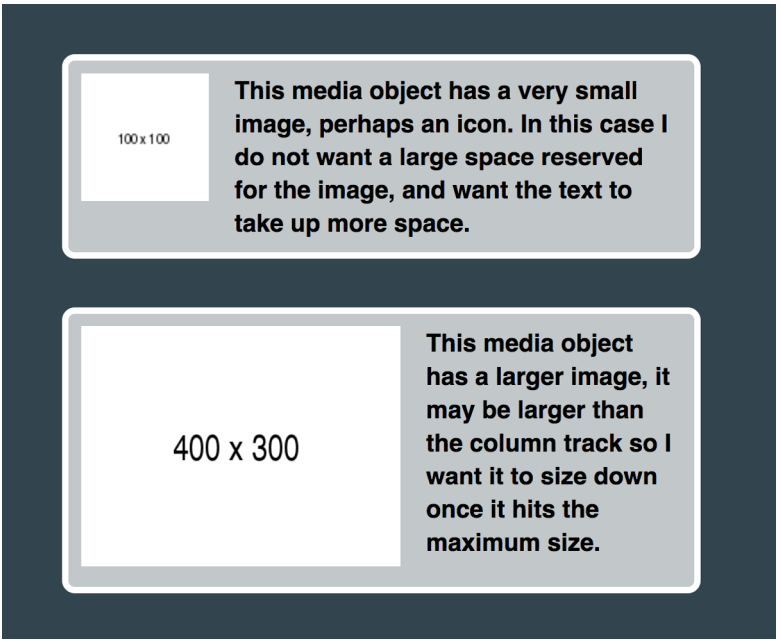
This allows some interesting use cases. For example, I might want to have a media object that sometimes contains an icon and other times a larger image, which might need to be scaled down inside the container depending on the available width.

If I use `fit-content` for the track size of the image, I can show the image at full size if it is smaller than the maximum specified, but clamp the track at a maximum so a large image does not spread out too much, leaving no space for my text. The column for the text is set to `1fr` so takes up whatever space is left.

```
.media {
  width: 500px;
  display: grid;
  grid-gap: 20px;
  grid-template-columns: fit-content(250px) 1fr;
}
```

Don't forget these content-sizing keywords as you think about the grid for your design: they can be incredibly useful. In production, I tend to use these for small components, like our media object above. The keywords mean I can have fewer individual design patterns for components, allowing Grid to do a lot of the work when figuring out how big things need to be.

We should eventually have these keywords available for us in `flex-basis` too. For now, to use them as in our example above, you will need to use Grid Layout for the component, where there is good support. All browsers that support Grid Layout also support `min-content`, `max-content` and `fit-content` for track sizes.



`fit-content` *used to create a flexible media object.*

minmax()

The content-sizing keywords – and `fit-content` in particular – help you define tracks that base their size on their content. We also have another flexible way to define track sizing: using the `minmax()` function. This track-sizing function defines a track with a minimum and maximum size.

I find this particularly useful when defining rows, or when specifying the size of rows created in the implicit grid. As we have already learned, a row track with a size of `auto` will expand to accept whatever content is placed into it. If we use `auto` as the value for

`grid-auto-rows` (the default value) then row tracks will be created and content will not overflow the track.

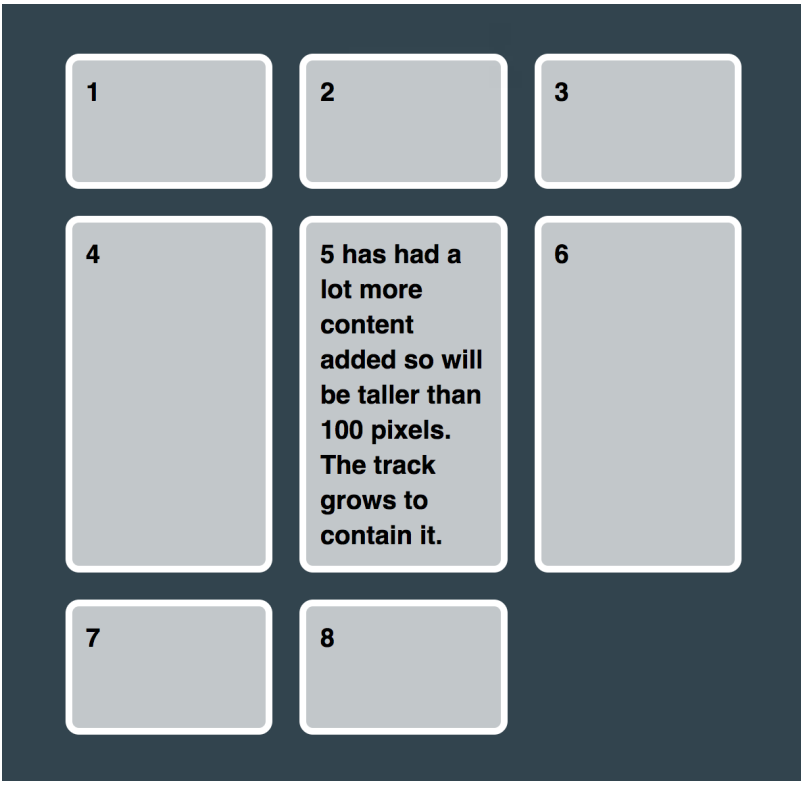
```
.grid {
  display: grid;
  grid-auto-rows: auto; // this is the default value
}
```

In some designs, you might need to create row tracks that are always a minimum height. If you were to use a fixed-length unit for that height (for example, 100 pixels), then content would overflow the fixed-height track if more content than you designed for was added. If we know anything about designing for the web, it is that at some point more content will *always* be added!

This type of situation is one that `minmax()` deals with very well. In the example below, I have set up a grid with rows created in the implicit grid with a minimum of 100 pixels and a maximum of `auto`.

```
.grid {
  display: grid;
  width: 500px;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
  grid-auto-rows: minmax(100px, auto);
}
```

Rows which contain content that is shorter than 100 pixels will be 100 pixels tall. Rows with more content than can fit in 100 pixels will be taller as they go to `auto`. When you design a layout to be built using Grid, you can think about this kind of functionality and be more precise in your layouts, knowing that in the situation where the content might overflow, Grid can help you deal with it.



Rows created using `minmax()` *can be a fixed height, but still grow if there is more content added.*

It is important to remember the two-dimensional nature of grid. One cell's growth to contain content will cause the *entire row to grow*. Content in the other cells needs to be able to cope with this expansion too.

Repeating

There is another use of `minmax()` that means you can specify a track listing that will create as many tracks as will fit, with a specified minimum. To do this, you need to use `repeat` notation when declaring your grid.

By using `repeat` you can duplicate an entire track listing or section of a track listing. To get four `1fr` tracks, for example, you would create the following track listing using `repeat`:

```
.grid {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
}
```

You can replace the number of times to repeat (`4` in the example above) with one of two keywords: `auto-fill` or `auto-fit`. You will then need to use a length unit as the value to repeat, and Grid will create as many column tracks of that length as will fit into the container.

```
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, 200px);
}
```

We can't use `1fr` as the value to get as many columns as will fit, because `1fr` will stretch to take up all available space: the number of `1fr` tracks you can have auto-filled in a container is one. However, we can use `1fr` as a maximum with `minmax`.

```
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px,1fr));
}
```

With this statement as our track definition, the browser will fill as many 200-pixel-wide columns as will fit. It is likely, then, that there will be a little bit left over; with the maximum set to `1fr`, that bit left over will be distributed evenly to all tracks. Therefore, we end up with as many flexible tracks as will fit, with a minimum size of 200 pixels.

The `auto-fill` and `auto-fit` keywords work in the same way as each other *unless* you do not have enough grid items to fill the container. In that case, if you use `auto-fill`, any completely empty tracks will still have space reserved for them – there would be a space at the end. With `auto-fit`, after laying out the items any completely empty tracks are collapsed to zero, and the space redistributed to the tracks that contain grid items.

As you can see, you have a range of solutions to draw on when sizing tracks. When you develop components for your sites, keep in mind these different options. Use them in combination with one another and you will probably find that you need to resort to media queries far less, and are able to create more reusable components.

Box Alignment

We frequently look at grid layout examples where elements on the grid are stretched over the entire grid area they have been placed into. The default behavior of an element without an intrinsic aspect ratio is to stretch in both directions.

```
.grid {
  height: 80vh;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
}
```

```
.grid :first-child {
  grid-row: 1 / 3;
  grid-column: 1 / 3;
}
```

By default, items stretch to fill their grid area.

This behavior matches setting the value of `align-items` and `justify-items` to `stretch`. You can play around with alignment of grid items, and in doing so can create the impression of breaking a strict grid.

Using `align-items` on the grid container will change the alignment of all the grid items in the *block direction*. Block direction is the direction in which blocks (a paragraph, for example) display down the page. The items will be aligned within their defined grid area. Using the same grid as in the example above, I can set `align-items: start` and all of the items now display at the start of the block axis within each area of the grid.

```
.grid {
  height: 80vh;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
  align-items: start;
}
```

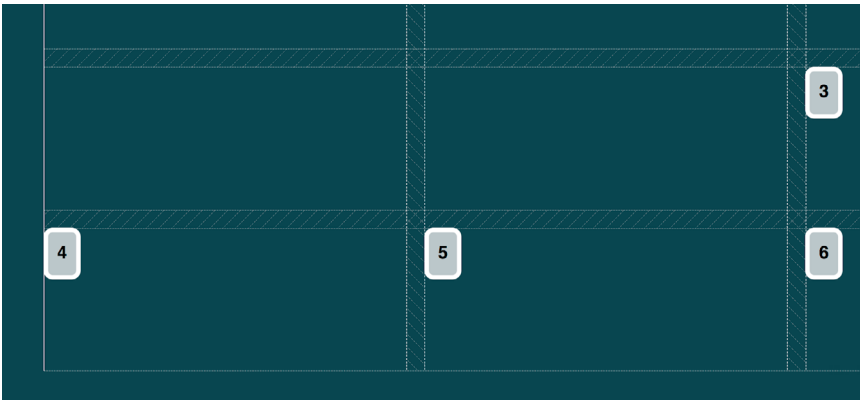
Aligning items to `start` *in the block direction.*

Using `justify-items` changes alignment to the *inline direction*. Inline direction is the direction that text in a sentence runs on your page. Once again, the content is aligned inside each defined grid area.

```
.grid {
  height: 80vh;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
  align-items: start;
  justify-items: start;
}
```

Items are aligned to start on both axes.

It's much easier to see what is happening when you align and justify items if you use Firefox's Grid Inspector. You'll then be able to see the extent of each grid area and the alignment of the item inside it.



Highlighting areas with Firefox's Grid Inspector.

The available values for `align-items` and `justify-items` are as follows:

- `auto`
- `normal`
- `start`
- `end`
- `center`
- `stretch`
- `baseline`
- `first baseline`
- `last baseline`

Specifying `baseline` as the value will align items with the first baseline in the alignment group – this will be the same as specifying `first baseline`. The value `last baseline` aligns the items with the last baseline of the group. All Grid-supporting browsers support `baseline`. At the time of writing, only Firefox supports `first baseline` and `last baseline`. Check the page on MDN for up-to-date support information and examples.[4]

Applying the `align-items` and `justify-items` properties to the grid container will change the alignment of all of the items as a group. However, you can target individual items with `align-self` and `justify-self` in the CSS rules of the relevant grid item.

4 http://smashed.by/alignitems

```
.grid :nth-child(2) {
  align-self: end;
}
.grid :nth-child(3) {
  justify-self: center;
}
```

Aligning individual items with `align-self` *and* `justify-self`.

Note that the default stretching behavior only applies to items without an intrinsic aspect ratio. Anything with an aspect ratio, such as an image, will be aligned to start on both axes. This is to prevent an image being pulled out of shape. (If that is what you actually want, then use `align-self: stretch` and `justify-self: stretch` on the item concerned.)

Aligning the Grid Tracks

The properties we have looked at so far deal with shifting content around inside a grid area. You can also align grid tracks when your grid container has more space than is needed to display the tracks.

In the following code, I'm creating three grid column tracks, each 100px wide, and three row tracks that are 100 pixels tall. The grid container is 500 pixels wide and 500 pixels tall. This means we have more space in both dimensions than is needed to display the tracks.

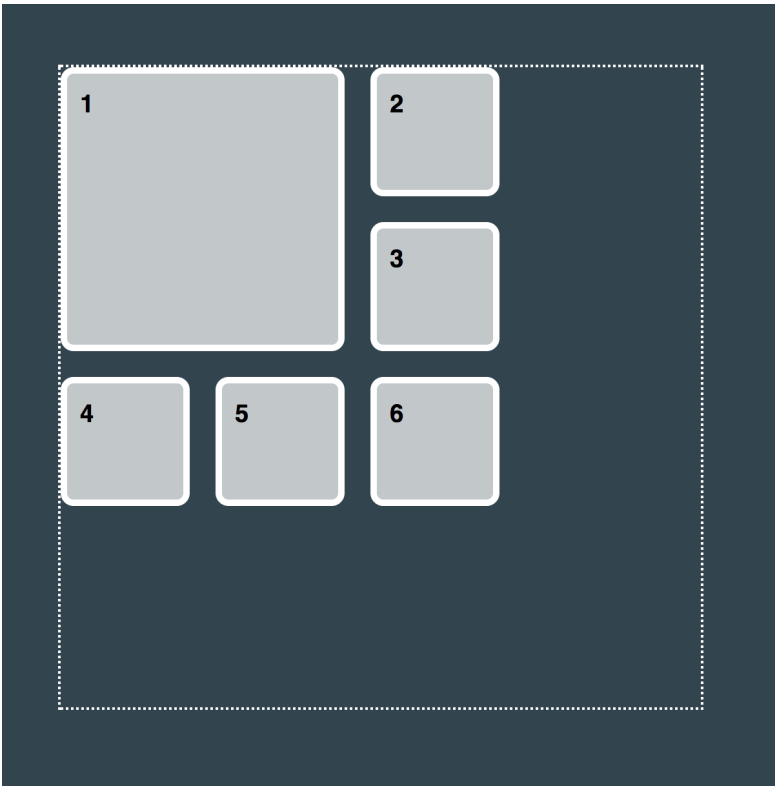
```
.grid {
  height: 500px;
  width: 500px;
  border: 2px dotted #fff;
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 100px 100px 100px;
  grid-gap: 20px;
}
.grid :first-child {
  grid-column: 1 / 3;
  grid-row: 1 / 3;
}
```

The properties used to align and justify tracks are `align-content` and `justify-content`. If you do not provide a value, they default to `start`, which is why our grid tracks line up at the start of the grid container for both rows and columns.

Remember that `start` is only `top-left` because I am working in English, a `left-to-right` and `top-to-bottom` language. If I were working in Arabic, written `right-to-left` then `start` would align the tracks to the top and right of the grid container.

You can use any of the values of `align-content` and `justify-content` to align the tracks.

- `normal`
- `start`

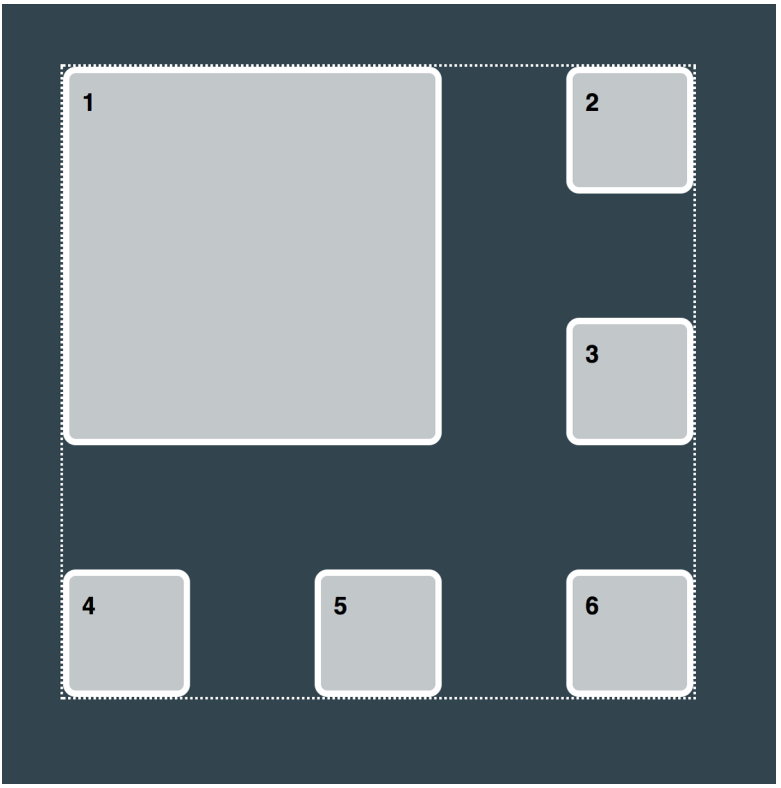


With more space in the container than needed for the tracks, items line up at the start.

- `end`
- `stretch`
- `space-around`
- `space-between`
- `space-evenly`
- `baseline`
- `first baseline`
- `last baseline`

There are no `-self` values of `align-content` and `justify-content` because these properties work by distributing space between the entire group of items.

Note that any items spanning more than one grid cell will need to absorb the extra space created if you use values such as `space-between`, `space-around` or `space-evenly`.



Box 1 has become larger as it absorbs the space that it spans.

```
.grid {
  height: 500px;
  width: 500px;
  border: 2px dotted #fff;
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 100px 100px 100px;
  grid-gap: 20px;
  justify-content: space-between;
  align-content: space-between;
}
```

GRID GUTTERS AND ALIGNMENT

In previous versions of the Grid specification were included the properties `grid-column-gap` and `grid-row-gap`, along with their shorthand `grid-gap`. These properties allowed us to specify a gutter or alley between grid tracks.

In 2017, these properties were moved into the Box Alignment specification, to sit alongside the alignment properties. At this point they were also renamed to make them generic properties that can be used in other specifications, not just Grid. The renamed properties are `column-gap`, `row-gap` and a shorthand of `gap`. Currently you need to use the `grid-*` prefixed properties; browsers will be aliasing these to the new properties, so should maintain support. To be bulletproof, you could add both properties, in the same way we add the real property name after vendor-prefixed properties.

```
.grid {
  display: grid;
  grid-gap: 10px;
  gap: 10px;
}
```

If you combine grid gutters and alignment of tracks with `align-content` and `justify-content`, you should be aware that this can make the gutters larger than expected, as seen in the previous example.

Responsive Design

Grid has been designed to be responsive by default, and because of this we generally find we can use far fewer media queries and breakpoints in our work. For example, we don't need to work out percentages, as the fr unit means we can let the computer do that work. We can display as many columns as will fit, with a minimum size being assigned.

It's definitely worth considering if there is a Grid method to do what you need to without adding a media query – quite often there is. That said, media queries remain very useful, particularly when used for larger components, and for full page design you may need to use media queries to change your layout at different breakpoints. You essentially have two choices when adding a media query, and you can choose either or both in your design:

1. Redefine the grid by changing the `grid-template-columns` and `grid-template-rows` values.
2. Redefine the grid areas by changing how many tracks an element spans.

The second approach is very similar to how existing grid systems work in frameworks like Bootstrap. You always have a twelve-column grid, but at narrower breakpoints items span more tracks of the grid. With a simple grid layout, laying out three boxes, we could have the following code:

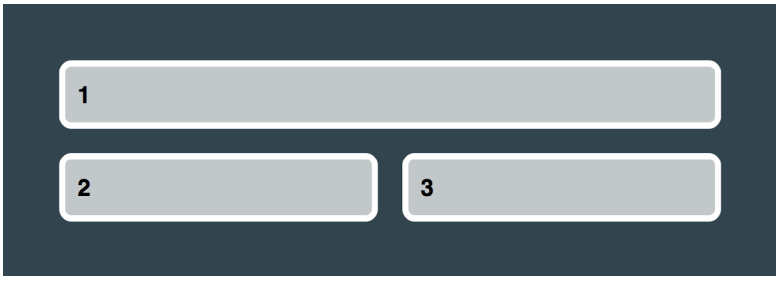
```
.grid {
  display: grid;
  grid-template-columns: repeat(12, minmax(0,1fr));
  grid-gap: 20px;
}
.col1, .col2, .col3 {
  grid-column: 1 / -1;
}
@media (min-width: 500px) {
  .col2 {
    grid-column: 1 / 7;
  }
  .col3 {
    grid-column: 7 / 13;
  }
}
@media (min-width: 700px) {
  .col1 {
    grid-column: 1 / 5;
  }
  .col2 {
    grid-column: 5 / 9;
  }
  .col3 {
    grid-column: 9 / 13;
  }
}
```

This sets up a twelve-column grid. At very narrow breakpoints, items are displayed one below the other. I have spanned the item from column line 1 (the start line) to column line -1 (the end line).

At the narrowest breakpoint, items span all twelve columns.

Once we reach 500 pixels, I change the second and third box to display as a split layout of half the number of columns each.

A



medium breakpoint: the bottom two items each span six columns.

With more screen real estate, I have a one-row layout of three equal-width boxes. In this way, we keep our twelve-column grid; where the columns are narrower, we span more of them.

At the widest breakpoint there are three items each spanning four columns.

An alternate approach to redefining the grid can work well if you are relying on auto-placement to lay out your items.

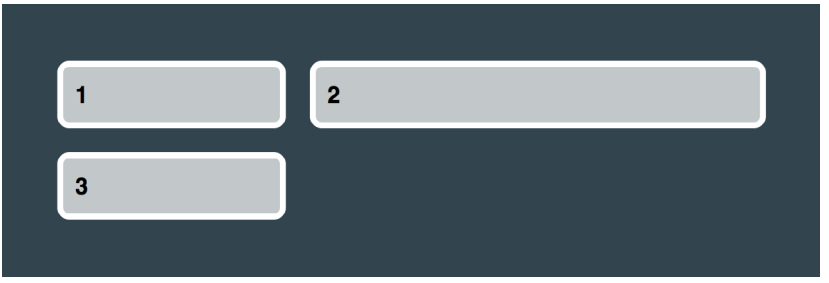
```
.grid {
  display: grid;
  grid-template-columns: 1fr;
  grid-gap: 20px;
}
@media (min-width: 500px) {
  .grid {
    grid-template-columns: 1fr 2fr;
  }
}
@media (min-width: 700px) {
  .grid {
    grid-template-columns: 1fr 2fr 1fr;
  }
}
```

In this example I am using auto-placement to lay out my items, and then using media queries to redefine the grid definition itself.

At the narrowest breakpoint the items display in a single `1fr` column.

The narrow view.

We get slightly wider and have two columns of `1fr` and `2fr`.



The middle breakpoint.

The widest breakpoint has three columns. In this case, then, we have different numbers of column tracks at the different breakpoints. You can, of course, combine both approaches and redefine the grid, as well as where items sit on the grid.

The widest breakpoint.

In general, you should find your use of media queries reduces – but don't be afraid to experiment. Sometimes adding a breakpoint is what the design needs to provide the best experience.

Browser Support and Fallbacks

Before deciding how to support browsers that don't recognize Grid, it is worth working out what you mean by support. Support might mean that the site has to look absolutely identical in all the browsers on your list. It might mean that you are happy for some finishing touches not to be available in all browsers. Or that you are testing these browsers but are happy for them to receive a much simplified experience. This is a discussion you should have with your client or the project owner, to make sure that everyone understands what the aim is.

At the time of writing, Edge, Chrome, Firefox, Opera, Safari, and iOS Safari all support Grid Layout. IE10 and IE11 have support for the original spec with an `-ms` prefix. *Older* browsers you might consider include:

- Internet Explorer 9 (or IE 11 and below if only considering the new spec)
- Edge 15 and below
- Firefox older than version 52

- Safari and iOS Safari older than version 10.1
- Chrome older than version 57
- Samsung Internet older than version 6.2

These popular desktop and mobile browsers are joined by browsers more commonly used in emerging markets, some of which haven't yet adopted Grid. If we take a worldwide view, UC Browser comes in at 8.1% of traffic – the third most popular browser in the world. Yet you may never have heard of it if you live in the US or Europe. Therefore, considering the site's goals and reach is important too.

Can We Polyfill Grid?

Grid does things that are impossible with older layout methods. To replicate Grid in browsers that don't support it, you would need to do a lot of work in JavaScript. Even on a well-resourced computer with a fast rendering engine, you are likely to have something of a janky experience as heights are calculated and items positioned. As we already know, the browsers that don't support grid are older and slower, or most often found on lower-powered devices in emerging markets. Why would you force a bunch of JavaScript onto those devices?

Instead of searching for a polyfill, consider how using Grid Layout can actually provide a better experience to people whose browsers don't support it. Grid allows you to create complex layout experiences for supporting browsers with minimal CSS, while still offering a good experience to those without support. Yes, it will be a little more work than just throwing a polyfill at the problem, but by doing so you are ensuring that support means providing a good experience, rather than making the most important goal getting the site to look the same.

It turns out that CSS itself has evolved to a point where you can provide that good experience just by writing CSS, and you can do so in a way that doesn't involve completely replicating your code.

Browsers Ignore CSS They Don't Understand

The first part of the picture is the fact that browsers ignore CSS they don't understand. If a browser that doesn't support CSS Grid comes across the `grid-template-columns` property, it doesn't know what it is and so discards that line and continues. Nothing breaks, we just don't get any column tracks.

This means that you can use some old CSS, like floats or `display: table-cell` to provide a Grid-type layout for older browsers, just as you would in the past. The browsers that do not support Grid will use this layout and ignore all the grid instructions. Browsers supporting Grid Layout will continue, discover the grid instructions and apply those. So we need to consider what happens if an item using another layout method becomes a grid item.

New Layout Knows About Old Layout

Defined in the specification is exactly how Grid behaves if you have elements in your page positioned by other layout methods. The methods you are likely to want to use are:

- floats
- `display: inline-block`
- `display: table`

- the `vertical-align` property, along with `display: inline-block` or the `table-*` values
- Flexbox
- multi-column layout

Floated items or those that use the `clear` property, and which then become a grid item, no longer exhibit any floating or clearing behavior, as though they were never applied. This means that in the example below, if the browser does not support Grid Layout, the user will see the floated layout. If the browser supports Grid they get the grid layout without any float behavior intruding.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
}
.grid > * {
  float: left;
}
```

The same is true for `inline-block`. The value `inline-block` can be applied to the child item, but as soon as the parent has `display: grid` the `inline-block` behavior will no longer be applied.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
}
.grid > * {
  display: inline-block;
}
```

I often use `display: table-cell` when I need to create a column layout and also align items in non-supporting browsers, since the `vertical-align` property works when you use `display: table-cell`.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
}
.grid > * {
  display: table-cell;
  vertical-align: top;
}
```

When you use `display: table-cell` to create columns, CSS will create what are known as *anonymous boxes*. These are the missing parts of the table: a table cell in a real HTML table will be inside a `<tr>` element, and that will be inside a `<table>` element. The anonymous boxes essentially fix these missing parents. If your table-cell item becomes a grid item, however, this happens *before* the boxes are generated, and so once again the item will act as if the CSS tables display had never happened.

The `vertical-align` property does not apply once in Grid Layout either, so if you use it in a CSS tables layout or with `inline-block` you can safely ignore that and use Box Alignment for Grid Layout.

You can also use Flexbox as a fallback. If you have used the `flex` property or individual `flex-grow`, `flex-shrink` or `flex-basis` properties on the item, these will be ignored once it becomes a grid item.

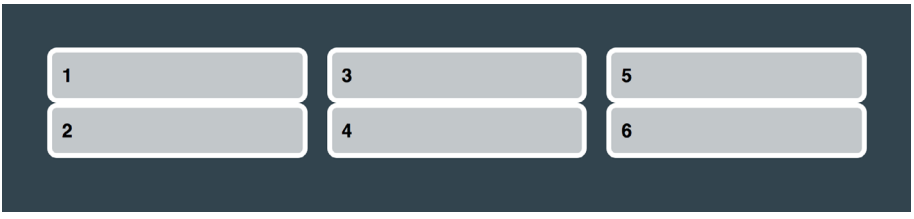
```
.grid {
  display: flex;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
```

```
}
.grid > * {
  flex: 1 1 auto;
}
```

Finally, don't forget that Multi-column Layout can be used in some cases as a fallback; for example, when laying out a list of card components or images. Items will be displayed in columns rather than across the row, but in some circumstances this can be useful. Apply `column-count` or `column-width` on the container to make it a multi-column container. If you then apply `display: grid` the `column-*` behavior will be ignored.

The example below will display a three-column multi-column layout if Grid Layout is not supported, and a three-column grid layout if Grid is supported. The difference is that in multi-column layout, the boxes will display by column, rather than by row.

```
.grid {
  column-count: 3;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  column-gap: 20px;
  grid-gap: 20px;
}
```



In Multi-column Layout, the boxes display down each column.

In the Grid version, the boxes display by row.

Feature Queries

Because you can simply overwrite one layout method with another means that, in many cases, you won't need to fork your code and can provide two different ways of creating the layout: a simple one using old methods enhanced with your more complex design. There are certain situations, however, where things you need to create your layout method for non-Grid browsers will also be interpreted by Grid-supporting browsers. A key situation like this is when you have applied widths to items.

As discussed earlier, older layout methods rely greatly on adding widths to items and then lining the items up so they look like they are in a grid. However, If you have applied a percentage width to a floated item, and then turn it into a grid item, the width will still be applied – even though the float is not. The item will become a percentage of the grid area it has been placed in, which is probably not what you want.

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 20px;
}
.grid > * {
  float: left;
  width: 33%;
}
```

The items are now one-third of the width of each grid area.

In this situation we need a way to tell the browser that if it supports Grid Layout, it should not use the width, as the size of the item will now be controlled by the grid area it is in. This is where CSS feature queries are useful. Feature queries (using `@supports`) act very much like media queries, but instead of querying the size of the viewport or device orientation, we check whether the browser has support for a certain feature. If it does, we can do things like override the width, setting it back to `auto`.

```
.grid > * {
  float: left;
  width: 33%;
}
@supports (display: grid) {
  .grid > * {
    width: auto;
  }
}
```

It is possible to check whether a browser *doesn't* support a certain feature, though this is currently less useful. Should the browser support neither Grid nor media queries, the code will never run. Instead, continue by doing what you need to for non-supporting browsers, then overwriting it for browsers with support – with or without an accompanying feature query.

Grid Layout in IE10 and IE11

While Edge has now updated to modern Grid Layout, IE10 and 11 have support only for the early version first shipped with an `-ms` prefix in those browsers. The Grid specification we know today came originally from Microsoft. Far from being unhappy about this old implementation, we should be glad they kickstarted the process and gave us Grid in the first place. You can read more about the story in the article "The Story of CSS Grid, from Its Creators."[5]

You might decide to offer IE10 and 11 a fallback experience based on a floated or other layout type as described above. This will work well, as will using feature queries, which are not supported in IE10 and 11. As long as you use these to overwrite your older methods, checking for support then creating the version for supporting browsers, IE10 and 11 will continue to use the older method.

You could make use of the `-ms-grid` version to create a fallback method. However, this prefixed version is not the same as modern Grid Layout; it was the first – and experimental – version. Things have changed in the five years or so since it shipped. This means you can't just use Autoprefixer[6] to add the prefixes. That approach will probably leave IE10 and 11 users with a worse experience than if you do nothing at all. Instead, you need to create a layout using this different and more limited spec.

5 http://smashed.by/cssgridstory
6 http://smashed.by/autoprefixer

The key points to note are as follows:

1. There is no auto-placement. You need to place each item on the grid using line-based positioning.
2. The `grid-template-areas` ASCII-art method of positioning is not part of the implementation.
3. There are no grid gap properties.
4. Instead of specifying start and end lines, you specify a start line and the number of tracks to span.

You can find a full breakdown of all of these properties in my blog post, "Should I try to use the IE implementation of Grid Layout?"[7]

If you have a high number of users with these browsers, you may find that this old spec is helpful. It is definitely worth knowing it exists, even if you only use it to solve a couple of small issues that are real showstoppers for you.

How to Add Grid Components to an Existing Site

If you are working on an existing project it is very unlikely that you will throw everything away and replace it with a grid layout. But Grid has been designed to coexist very happily with older layout methods in CSS. You can start to use Grid for new components that are being introduced into your pages, and benefit from being able to take a more creative approach, or simply having far less CSS to deal with.

7 http://smashed.by/iecssgrid

An Image Gallery Panel

Let's say I need to add an image gallery to an existing site that uses Bootstrap. Grid would be ideal as I can avoid using media queries to work out how many columns of images I can display. I can use the `auto-fill` method to create as many flexible columns as will fit in the container for the gallery.

Some larger images span two tracks, and I am using `grid-auto-flow: dense` to backfill the gaps left in the grid. These are benefits of Grid Layout that aren't available with any other layout method, making it worth using Grid for this component – I gain advantages if I do so, and the Grid code to achieve all of this is tiny.

```
<ul class="gallery">
  <li class="landscape"><img src="myimage.jpg" alt="alt text"></li>
  <li class="portrait"><img src="myimage.jpg" alt="alt text"></li>
</ul>
img {
  max-width: 100%;
  display: block;
}
.gallery {
  list-style: none;
  margin: 0;
  display: grid;
  grid-gap: 2px;
  grid-template-columns: repeat(auto-fill,minmax(200px, 1fr));
  grid-auto-flow: dense;
}
.gallery img {
  object-fit: cover;
  height: 100%;
  width: 100%;
}
.landscape {
  grid-column: auto / span 2;
}
```

Gallery component.

In a browser that does not support Grid Layout the images will display in regular block flow, one after another.

I can then make a decision about how to display these images for other browsers. Maybe I simply leave it alone: images displaying one after another isn't a terrible experience. It is likely, though, that you will feel there are enough non-Grid browsers visiting your site to want to do a bit more than that.

Which layout method I choose will depend on the browsers I know are visiting my site. With an existing site you have this knowledge, which can be pretty helpful! If I know that most browsers have Flexbox, then my fallback could be Flexbox.

With Flexbox, I won't have the spanning of tracks and backfilling possibilities, or grid gap. The items won't stay in a strict grid either, but I can get a nice layout. As we've already discovered, once a flex item becomes a grid item, the flex properties no longer apply, so we can safely implement Flexbox and then add Grid.

```
.gallery {
  list-style: none;
  margin: 0;
  display: flex;
  flex-wrap: wrap;
  display: grid;
  grid-gap: 2px;
  grid-template-columns: repeat(auto-fill,minmax(200px, 1fr));
  grid-auto-flow: dense;
}
.gallery > * {
  flex: 1 1 200px;
}
.gallery img {
  object-fit: cover;
  height: 100%;
  width: 100%;
}
.landscape {
  flex: 1 1 400px;
  grid-column: auto / span 2;
}
```

The Flexbox version of the layout. The fallback requires six extra lines of code.

In this case, if a browser does not support Grid or Flexbox they will still see the images displayed one after another in block layout. This isn't a disaster and is better than a completely broken layout.

If I know some very old browsers visit the site, I might decide to add as my fallback a method supported by pretty much everything. So I am picking an `inline-block` layout. This will never be as neat as the Grid,

or even Flexbox, version of the layout, but with a small amount of code we can make a reasonable fallback, and as soon as those items become flex or grid items, all `inline-block` behavior is removed.

For this layout, I also need to add a CSS feature query, with `@supports`, to remove the width applied to items once we are in Grid Layout.

```
.gallery {
  list-style: none;
  margin: 0;
  display: flex;
  flex-wrap: wrap;
  display: grid;
  grid-gap: 2px;
  grid-template-columns: repeat(auto-fill,minmax(200px, 1fr));
  grid-auto-flow: dense;
}
.gallery > * {
  display: inline-block;
  vertical-align: top;
  width: 33%;
  flex: 1 1 200px;
}
@supports (display: flex) or (display: grid) {
  .gallery > * {
    width: auto;
  }
}
.gallery img {
  object-fit: cover;
  height: 100%;
  width: 100%;
}
.landscape {
  flex: 1 1 400px;
  grid-column: auto / span 2;
}
```

A few more lines of code provide an `inline-block` *fallback.*

As a demonstration of how this component can sit happily within an existing layout, I have taken a standard Bootstrap 3, floated two-column layout and dropped my panel into it. The gallery panel works well and is responsive owing to the `auto-fill` columns, and it coexists happily with the Bootstrap layout.

Our component in the Bootstrap layout.

Grid Alongside Other Tools and Techniques

I've shown you how Grid Layout can work as a component of an existing site. But existing sites also are likely to be using tools such as Sass or PostCSS plug-ins.

In general, the developers of commonly used tools have been quick to adopt Grid Layout. If your site was built prior to Grid landing in browsers, then you should check you are using up-to-date versions of any tool you use. If you are using the named lines syntax, you should ensure that Sass

has a version greater than 3.5, as support for bracketed lists was added in this version.[8]

Having learned there is an older version of Grid Layout in IE10 and IE11, with an `-ms` prefix, you might wonder if using that old version of Grid is as simple as running Autoprefixer. The team at Autoprefixer have made efforts to get this to work, and if you have very simple grid layouts it may well do. However, as detailed earlier in this chapter, new Grid Layout is fundamentally different to old Grid Layout. Test the result of using Autoprefixer, if you try it, as it could well be that the experience is worse than it would be if you served IE10 and 11 the same fallbacks as used for other browsers. You always have the option of disabling Grid in Autoprefixer if you find it makes a mess of your layouts, by setting `grid: false`.

Is Grid Really Ready for Us to Use Now?

A final thought to leave you with as I close this chapter: I've heard the thought voiced that Grid is "too new" and will be poorly implemented in browsers with huge numbers of bugs. This isn't the case. Grid was developed in browsers for over five years, but behind an experimental flag so most people didn't know about it. The implementations and the specification are at an advanced stage, and you will be surprised to discover how few issues remain.

If you want to track the bugs we know about, you can keep an eye on my GridBugs site.[9] Everyone wants the Grid implementations to be as bug-free as possible, so things are being fixed very quickly as they are found.

8 http://smashed.by/sassrelease
9 http://smashed.by/gridbugs

Learning Grid Layout: Tutorials and References

With Grid Layout being new, there are a huge number of resources being created and updated all the time. If Grid is completely new to you, I have a set of free video tutorials explaining the various concepts.[10]

There is also a comprehensive set of guides on the MDN website.[11] On Smashing Magazine we have some extensive guides and articles on Grid Layout,[12] and I keep an up-to-date set of resources as they are published.[13]

About The Author

Rachel Andrew is a front and back-end web developer, author, speaker, and Editor-in-Chief of Smashing. She has written several web development books, most recently "CSS Grid Layout." She also the founder of edgeofmyseat.com, the company behind Perch CMS.

10 http://smashed.by/gridbyexample
11 http://smashed.by/mdncssgrid
12 http://smashed.by/smashingcss
13 http://smashed.by/gridbyexample2

SMASHING BOOK 6

New FRONTIERS in WEB DESIGN

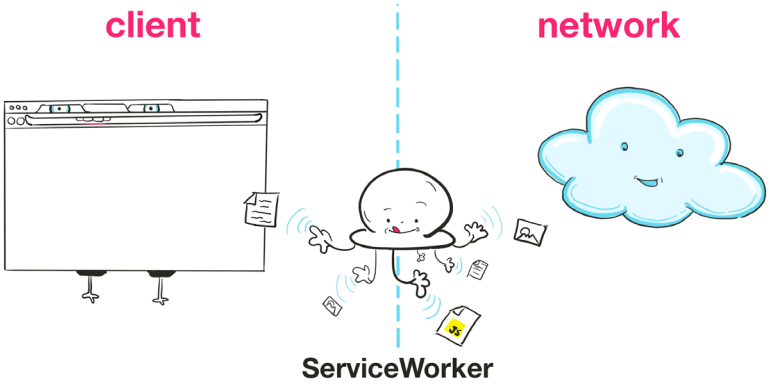
Service Workers

by Lyza D. Gardner

Ah, yes! Service workers! They can make sites and apps work offline. They're an integral part of progressive web apps (PWAs). They're big boons for network performance. You'll need service workers if you want to get at some of the nifty new web APIs like Web Notifications or Web Background Synchronization.

Yep: they enable a wide array of great features and are beginning to feel like a lynchpin of the modern web. Yet even folks who've cooked up a few service workers of their own can be hard-pressed to explain exactly what a service worker *is*.

Let's start here: a service worker consists of a script – its behavior is defined by a JavaScript file. The JavaScript in that file is executed in the background within a browser, on a separate thread from the pages loaded in the browser's windows or tabs. A service worker, then, has a different *context* from a web page, and it is able to access some features and APIs you can't get at otherwise.



A service worker acts as a proxy, sitting between the client – your site or app's web page JavaScript – and the network.

The canonical use case for a service worker is as a *proxy*, intercepting network requests for resources within its scope and deciding how to handle them. That's vague, so let's break it down. A service worker sits between your site or app and the network, and can decide what to do whenever a resource (like an HTML page or an image) is requested by the browser.

For any given request, the service worker can opt to go to the network and fetch fresh content from a server, or it might retrieve a resource from a cache (where the service worker has previously stashed the resource). A service worker is only able to intercept requests that fall within its *scope*: a range of URLs within the service worker's control.

In addition, a service worker has a bunch of special goodies available in its global context: references to the aforementioned Notifications and Background Sync APIs, for example. You can't get to some of these APIs from *web-page-loaded-in-a-browser*[1] JavaScript.

Fetch handling – responding to requests for resources that fall within its scope – is a fundamental functionality of service workers. However, if you read the Service Worker spec[2] (I highly encourage this, by the way; it's fairly short and concise), you'll see it doesn't define how `fetch` works. That's because `fetch`, though critical even to the simplest service worker, is its own separate standard, and is not part of Service Worker itself.

1 That, if you can believe it, is not the scientific term. JavaScript in a web page executes in a browsing context (global object: window), while service workers have a different context, specifically `ServiceWorkerGlobalScope`.

2 W3C Service Workers 1 Working Draft (http://smashed.by/serviceworkersw3)

The Service Worker API makes heavy use of JavaScript Promises. If you're new or rusty with Promises, do a Google or two and read up a bit. The Mozilla Developer Network's (MDN) coverage[3] is clear and readable.

Browser support for service worker functionality continues to strengthen. At the time of writing, some level of support is available in every major browser – Chrome, Firefox, Opera, Safari and Microsoft Edge. Not all Service Worker features are supported in every browser. Google is a major driver of these technologies – unsurprisingly, some of the newer Service-Worker-related APIs and features are supported only in Chrome.[4]

Service workers are not too hard to work with. There are two hard-and-fast rules, however:

- The code in a service worker needs to be responsive and non-blocking. A browser has the right to terminate any service worker that is taking too long to do its thing. That means that web APIs that are inherently slow and block the execution thread (like `LocalStorage` or synchronous XHR) are no-nos in service workers.
- Service workers are only able to mediate communication for pages served over HTTPS. There is a handy exception to this for development purposes: resources served from `localhost` do not have this HTTPS requirement.

3 http://smashed.by/mdnpromise

4 See Jake Archibald's "Is service worker ready?" for the latest details of who supports what: https://jakearchibald.github.io/isserviceworkerready/.

The Life of a Service Worker

A service worker doesn't go away when you close a tab, navigate to another site, or shut down the browser. It has its own life, related to but definitely distinct from the pages it controls. It executes in a different context from a web page and is event-driven, relying on an assortment of relevant events to get its job done.

UNDERSTANDING CONTEXT

To understand a bit more about how service workers tick and how their lives work, let's talk about their API parentage. Service workers are a kind of *web worker*. Service workers inherit from web workers the trait of running in the background, on a different thread from the pages displayed in a browser's tabs or windows.

A service worker executes within a `ServiceWorkerGlobalScope`, which extends `WorkerGlobalScope` (the global scope for any web worker) with a few things that service workers need to be, well, service workers.

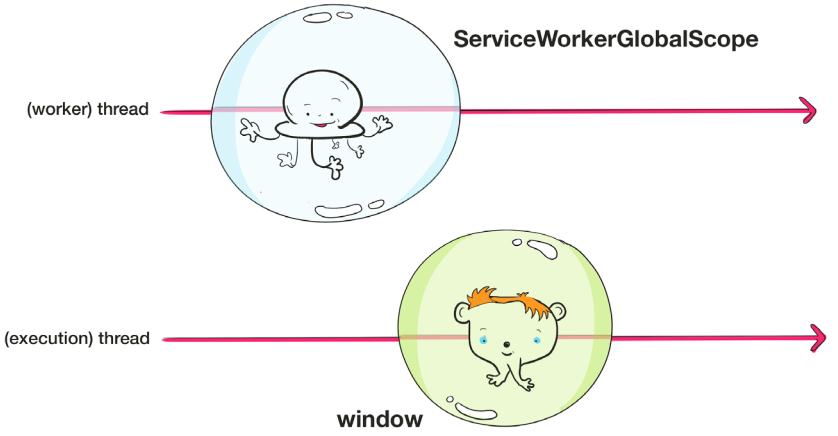
A web page in a browser tab or window, on the other hand, executes in a *browsing context*, represented at a global level by the `window` object. A browsing context is, as the HTML spec explains, "an environment in which `Document` objects are presented to the user."[5] There's some additional nuance, but for the purposes of our explorations we can say that each open window or tab in your browser corresponds to a browsing context.[6]

5 https://html.spec.whatwg.org/multipage/browsers.html#windows

6 `iframe` elements also get their own browsing contexts.

As such, a web page and its controlling service worker are in separate bubbles. They can't manipulate each other directly,[7] and each context has different stuff in its global scope.

A `ServiceWorkerGlobalScope` has no `window` object or DOM – that wouldn't make sense, as the service worker is always invisible and in the background.



Web workers, including service workers, execute on a different thread and in a different context than the clients (web pages) they control.

SERVICE WORKER LIFE CYCLE

From the moment it's created until it's done and retired, a service worker's capabilities change in predictable ways. That is, a service worker has a *life cycle*, and its position in its overall life cycle at any given moment is indicated its by its `state` attribute. `state` has one of the following values:

7 They can, however, absolutely communicate – see the "Channel Messaging API" section later in this chapter.

- parsed
- installing
- installed
- activating
- activated
- redundant

All service workers start out as `parsed`. Subsequently, they move through an installation phase and eventually, if everything goes right, get activated. Though they sound similar, those two life cycle phases – installation and activation – have distinct meanings and purposes.

The installation phase is used to get the service worker ready, typically by stuffing some things into a cache that it'll need to access later (such as core, application shell resources). The activation phase is for housecleaning, providing an opportunity to clean up after old service worker versions and caches (like deleting crufty cache entries).

SERVICE WORKER SCOPE

A service worker applies to a specific *scope*: a range of URLs it can control. A service worker is unable to respond to anything happening that relates to resources at URLs outside of its scope.

A browser that implements the Service Worker API is required to keep track of registered service workers, mapping between service workers and the scope that each covers,[8] so it can figure out which to select for a given URL. There can be only one *active* service worker for a given URL at one time.

8 Technically, browsers map `ServiceWorkerRegistration` objects to scopes, not `ServiceWorker` objects directly.

Just because a service worker registration is maintained in the browser doesn't mean the associated service worker is always *running*. Service workers are event-driven: if a given service worker doesn't have anything to do – there's no event to handle at the moment – a browser can opt to terminate it.[9]

It can be confusing: the word *scope* has two different meanings! There's scope as it pertains to execution context: all JavaScript executing in the browser has a scope; in Service Worker's case this is represented by `ServiceWorkerGlobalScope`. But service workers are also associated with a scope: a range of URLs they can control.

Because of the browser's prerogative to shut down inactive service workers, you should never rely on persisting something in global scope (keeping track of something by incrementing a global variable, for example).

SERVICE WORKER AND EVENTS

Events are what make service workers *go*. A service worker's functionality resides primarily in handling several key events: two *life cycle events* (`install` and `activate`), as well the functional event `fetch`:

- `install` is dispatched after the service worker moves into the installing state.
- `activate` is dispatched after the service worker moves into the activating state.
- `fetch` is dispatched whenever the browser wishes to fetch a resource that falls within the given service worker's scope.

9 Browsers will likely shut down a service worker if it hasn't done anything for about thirty seconds.

`install`, `activate` and `fetch` are the key Service-Worker-related events, but they are not the only ones a service worker might choose to listen for. Other useful events include, but are not limited to: `message` (Channel Messaging API), `push` (Push API) and `sync` (Background Sync API).

Anatomy of a Basic Service Worker

There are two real-world pieces to a service worker: the service worker script file itself and a bit of code that goes in your site or app's web page(s) to *register* that service worker against a scope.

REGISTERING A SERVICE WORKER

Something has to tell the browser to register a service worker and where to find it. Otherwise, how would it know what to do? This job is handled by a bit of JavaScript you can put in your site's web pages. This registration code looks like this:

```
<script>
  navigator.serviceWorker.register('service-worker.js', './');
</script>
```

Browsers that support Service Worker expose `serviceWorker` on the global `navigator` object.[10] In browsers that don't support Service Worker, `navigator.serviceWorker` will be undefined. To keep those browsers from choking,[11] you can wrap the registration call in a simple feature test:

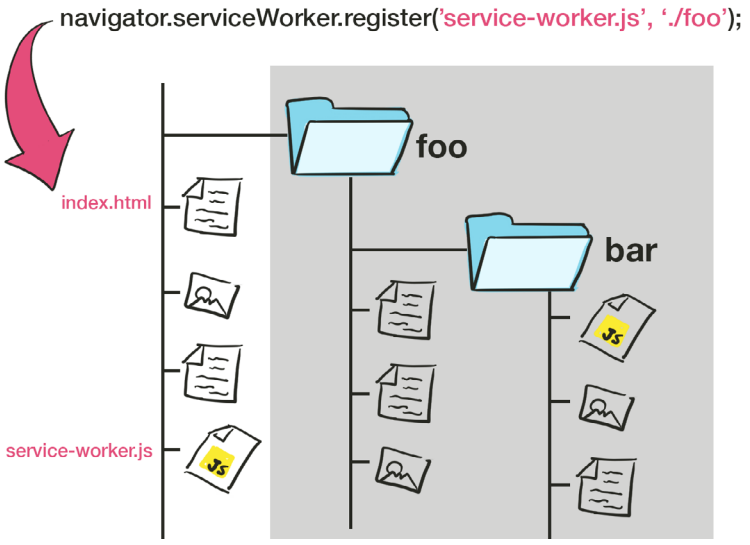
10 Specifically, `navigator.serviceWorker` returns a `ServiceWorkerContainer` object, which contains methods for registering and unregistering service workers, among other things.

11 Trying to invoke `navigator.serviceWorker.register` in a browser with no exposed `navigator.serviceWorker` will throw a `TypeError`.

```
<script>
if ('serviceWorker' in navigator) { // make sure navigator.serviceWorker
exists
  navigator.serviceWorker.register('service-worker.js', './');
}
</script>
```

The register method takes two arguments: a script URL and an optional options object. The following three statements are functionally identical:

```
// Service Workers 1 defines only one recognized option property: scope
navigator.serviceWorker.register('service-worker.js', { scope: './' });
// Shorthand is supported: pass `scope` as a string instead
navigator.serviceWorker.register('service-worker.js', './');
// Default scope is './' when the argument is not present
navigator.serviceWorker.register('service-worker.js');
```



In index.html, a service worker at service-worker.js is registered against the scope './foo'. Once activated and in control, that service worker can respond to fetches for resources in the shaded area – within the foo directory – but could not respond to fetches outside of that range of URLs.

Scope is relative to the service worker script's location, not the registering web page.

A service worker's highest allowed scope is the location of its script file. If this proves problematic, there's an Service-Worker-Allowed HTTP header you can take advantage of.[12]

YOUR FIRST (USELESS) SERVICE WORKER

The other piece of the puzzle is, of course, the service worker script itself. Let's start with something very simple inside of *service-worker.js*:[13]

```
self.addEventListener('fetch', event => {});
```

Recall that a fetch event is dispatched whenever the browser wants to fetch a resource that falls within the service worker's scope.

Within a fetch handler function we can opt to respond to a fetch, satisfying the browser's needs by determining what kind of response to use and where to retrieve it from (a great example of a service worker acting as a proxy between the browser and the network). Of course, our stub of a fetch handler doesn't do anything (yet).

There are several ways to respond to a fetch, but the most common techniques are:

12 Appendix B: Extended HTTP headers, Service Workers 1 Working Draft: http://smashed.by/httpheaders

13 You can name your service worker file anything you like, of course, but I'll use *service-worker.js* throughout this chapter.

- using the fetch API to retrieve a resource from the network
- using the caches API to retrieve a previously-cached resource (avoiding the network)
- generating our own Response object

Let's start by learning about fetch.

In web-page JavaScript, the window object provides a reference to the global object. In Service Worker JavaScript, it is conventional to use the reference self when accessing methods or properties on the global object (e.g. `self.addEventListener`). These references will certainly work without the self notation (i.e. addEventListener will work by itself) but can feel "magical." Being explicit about scope, by using self when applicable, can make your code more readable and easier to maintain.

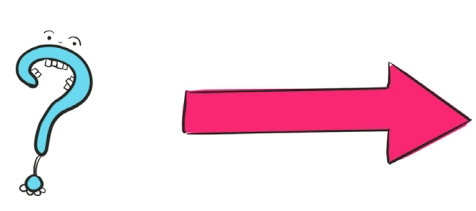
Not all browsers support arrow functions (`() => {}` notation), but every browser that supports Service Worker does. So you can use that syntax with impunity within service workers. The same goes for the const keyword and other more modern-ish JavaScript features used in examples here.

Fetch API

The Fetch API lets you go out and get resources from the network asynchronously. It's not too different from `XMLHttpRequest` but it does more things and is certainly easier to pronounce.

Fetch's bread and butter are Request and Response objects. You feed `fetch()` a `Request` object and it returns a `Promise` which – if everything goes OK and the network is available – resolves to a `Response` object. Requests in, Responses out:

```
var aResponseIHope = fetch(someRequest);
```



Request

method
url
headers
mode
cache

Response

url
type
status
ok
text()
headers

Conceptually, `Request` *objects are fed to* `fetch` *and* `Response` *objects result.*

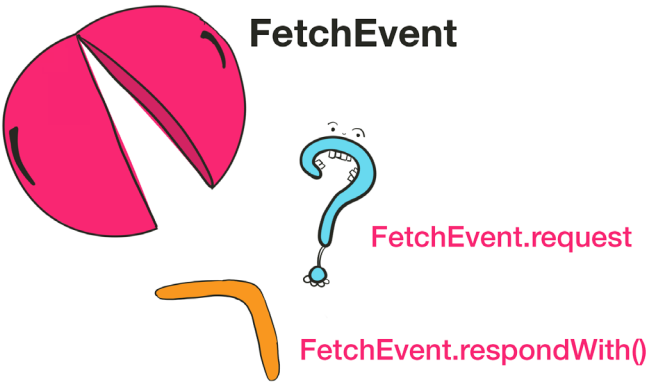
FETCHEVENT OBJECTS

A Request defining what the browser is trying to obtain is available to fetch handlers within the `FetchEvent` passed to the function.

We've already created an (empty) event handler for `fetch` events inside of our service worker:

```
self.addEventListener('fetch', event => { });
```

The argument passed to a fetch event handler – the event parameter in our code snippet – is a `FetchEvent`.



If you crack open a `FetchEvent` object, you'll find, among other things, a relevant Request and a vital method, `respondWith`.

`FetchEvent` objects have a request property – aha! That's the Request object representing what the browser is looking for in this fetch.

If we want the service worker to intercept this particular fetch and supply its own Response, we need to be explicit by using the `FetchEvent`'s `respondWith` method.[14] Like so:

```
self.addEventListener('fetch', event => {
  // FetchEvent.respondWith takes a Response or a Promise
  // that will ultimately resolve to a Response
  event.respondWith(
    fetch(event.request)
  );
});
```

This explicitly fetches the given Request from the network, using the Fetch API, and returns the result (a Promise which we hope will resolve to a Response).

14 If you neglect to invoke `FetchEvent.respondWith`, whatever Response you cook up or return will not get used by the browser, so don't forget to use it!

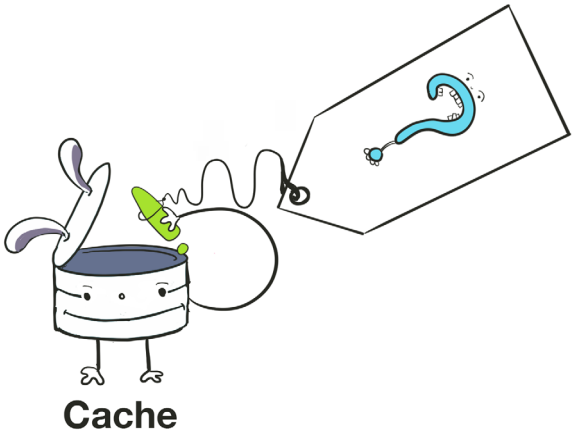
Guess what? Still functionally useless. All we've done is what the browser would do by default anyway: fetch things from the network. Don't worry: it's not always pointless to use the Fetch API. We'll encounter some good reasons to use it within fetch handlers as we continue on our journey.

A fetch handler is by no means expected to *respond* to every fetch it handles. If `FetchEvent.respondWith` is not invoked during a fetch-handler invocation, that's fine: the browser will carry on and go to the network to try to find the resource it needs.

Working with Caches

One of a fetch handler's greatest superpowers is to avoid trips to the network at all, instead responding to certain requests with responses from cache. This can be a great boon to performance.

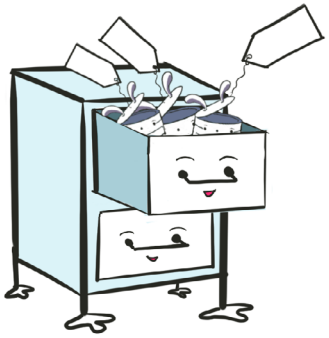
A service worker can create, add items to, and remove items from an arbitrary number of caches. A Cache object is, in a nutshell, a map: it relates Request objects to Response objects.



A Cache object is a collection of Responses keyed by Requests. It behaves similarly to a JavaScript Map object. `Request-Response` *pairs are added to a Cache so they can be available for use later.*

CACHE OBJECTS AND THE CACHESTORAGE INTERFACE

All access to and manipulation of Cache objects is handled through the CacheStorage interface, available globally in service workers as caches. CacheStorage is a kind of directory, a file drawer full of all of the caches available to you.



All of the available Cache objects are accessed through the CacheStorage interface. Cache objects are keyed by the names you give them.

You create and manage your own caches – these Cache objects are completely distinct from the browser's built-in caching mechanism. They are entirely under your control.

Cache access is origin-based. That means you'll have access to the same set of caches, via CacheStorage, from any service worker running on the same origin.

To add to or retrieve items from a Cache, you first need to get a hold of it by asking the CacheStorage interface to open it:

```
const myCacheIHope = caches.open('my-cache-name');
```

Opening a Cache you've previously created and creating a new Cache looks identical: if a Cache with the key given to `CacheStorage.open` doesn't correspond to an existing Cache, a new, empty Cache will be created and returned.

The value of a Cache object's key (name) is up to you: any string will do.

`myCache` is a Promise that will resolve to a Cache object representing the cache you've asked for. Once the Promise resolves, you can work with the cache you've opened:

```
myCacheIHope.then(cache => {
  // Do something with this cache!
});
```

Remember, Cache objects that you manipulate in service workers are completely distinct from a browser's built-in caching.

PUTTING THINGS IN CACHES

To be able to get things out of caches when responding to fetch events, a service worker needs to put things in them in the first place. Remember, caches are entirely controlled by the logic in the service worker – nothing happens automatically. There are three useful methods on Cache objects for putting stuff inside of them. We'll try out each of these in subsequent recipe examples:

- `Cache.put(request, response)`: puts a `Request-Response` pair into the cache.
- `Cache.add(RequestOrURL)`: retrieves the Request or URL and stores the Response in the cache.
- `Cache.addAll([urls])`: Takes an array of URL strings and retrieves each, adding their Responses to the cache.

The behavior of `Cache.put` is a little different from `Cache.add` and `Cache.addAll`. `Cache.put` takes both a Request and a Response, allowing you to pair any Request with any Response – this gives you the most fine-grained control over what goes into a cache. `Cache.add` and `Cache.addAll` go to the network to retrieve the corresponding Responses that get stored.

CAUTION: `Cache.put`, `Cache.add` and `Cache.addAll` all return a Promise (which resolves to undefined). When that Promise resolves is in flux at present: early supporting browsers resolve it when the Response is fully written to storage, but there is a push underway to make the Promise resolve earlier for better performance.

GETTING STUFF OUT OF CACHES

Of course, you'll also need to be able to look for something in cache when you need it (likely in a fetch handler). Once again, there are three ways you can do this (all three return Promises):

- `Cache.match(request)`: look for a match for the given Request inside of a specific Cache.
- `Cache.matchAll([requests])`: look for matches for an array of Requests in a specific Cache (the returned Promise will resolve to an array of all matching entries).
- `caches.match(request)`: look for a match to a given Request across all caches you have access to.

Given that `Caches.match` doesn't require you to first open a specific Cache, it can be convenient; but there are times when the specificity of `Cache.match` is useful (for example, to rule out duplicate requests in other caches).

THE INSTALL LIFE CYCLE PHASE

Looking at Cache object management in the abstract, you may reasonably wonder when exactly you're supposed to stick stuff into caches for later retrieval from within fetch handlers.

The way to do this is to take advantage of the `install` life cycle event, stashing resources you know you'll need later into cache. This is a technique used in a common service worker recipe for pre-caching application shell files (images, styles, scripts, and so on, used on all or most pages of your site or app).

You can add an `install` handler in your service worker:

```
self.addEventListener('install', event => {
  // Take care of install concerns like pre-caching
});
```

EXTENDABLEEVENT OBJECTS

Opening up a cache and putting items in it are asynchronous operations. You can run into problems if you do something like this:

```
self.addEventListener(`install`, event => {
  // psssst: don't do this:
  caches.open('some-cache').then(cache => cache.addAll(arrayOfURLs);
});
```

The reason that this is potentially problematic is that the service worker may keep marching along through its life cycle; installation may complete before this handler has successfully added the items to cache.

Fortunately, ServiceWorker life cycle events – install and activate – are a kind of `ExtendableEvent`. `ExtendableEvent.waitUntil()` gives you the ability to say, "Hey! Wait until I'm done doing some stuff before you move on!", stretching out the lifetime of the event.

```
self.addEventListener('install', event => {
  event.waitUntil(caches.open('some-cache').then(cache => cache.
addAll(arrayOfURLs)));
});
```

Altering the code to read like the above will cause the install phase to hang on, to wait *until* the `Promise` given to `event.waitUntil` resolves before moving on.

Generating Custom Responses

In fetch handlers, we can respond to the request in any way we like, as long as we're satisfying the browser's need for a Response object. Using the fetch API or retrieving a Response from cache are typical ways of do this, but there's nothing stopping us from rolling our own Response objects:

```
self.addEventListener('fetch', event => {
  // Remember, FetchEvent.respondWith is looking for a Response
  event.respondWith(new Response('<p>Service Worker has taken over</p>', {
    headers: { 'Content-Type': 'text/html' }
  }));
});
```

Let's talk about what this does, though. For every fetch that occurs within the service worker's scope, it will respond with a Response that looks like a web page.[15]

15 The first argument to the Response constructor is the body of the Response – here an HTML paragraph. Setting a Content-Type header to text/html in the second, options argument makes the Response look like a web page to browsers.

Intercepting fetches for HTML pages – so-called *navigation requests* – and responding with this bogus little web page is silly enough, but it gets even goofier when you consider that *all* fetches are being intercepted and responded to in this manner – including those for images, CSS, scripts, and so on. Obviously, you wouldn't want to respond with a (lame) web page when the browser is really looking for a PNG or a JavaScript file. But you *could*. That's the power of service workers!

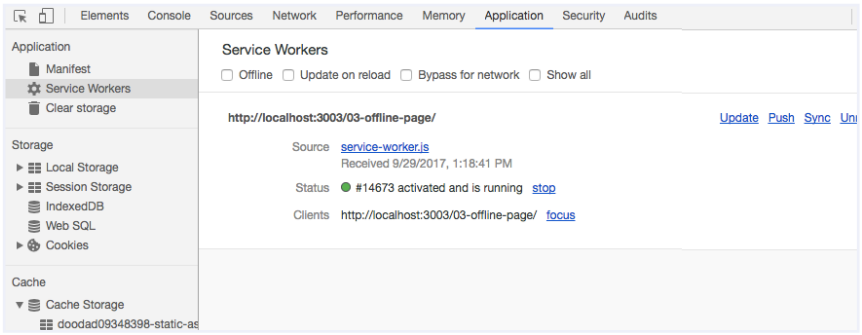
Classic Service Worker Recipes

All right: that's some high-level overview of some key APIs involved in service workers: fetch, caches and their allies. Now let's see how these work together in the real world to do real things.

Inspect and debug tools for developing service workers are getting more mature and fully featured. The service worker features in Chrome's Developer Tools[16] are especially powerful. Some key service-worker-related tips:

- You can simulate offline behavior by toggling the **Offline** checkbox, which can be found both on the **Network** tab and in the **Service Workers** section of the **Application** tab (they function identically).
- The **Application** tab has a **Service Workers** section, which shows details about all service workers for the current origin.
- You'll also find a **Cache Storage** section in the **Application** tab, which allows you to see (and manipulate) all of the entries in available caches.

16 As I write this, I'm using Chrome 64. By the time you're reading this, that will probably seem laughably obsolete.



Application→Service Workers *tab in Chrome Developer Tools. Firefox provides an* `about:serviceworkers` *page that can be handy, too.*

PROVIDING AN OFFLINE MESSAGE

We've seen how we can use the Fetch API and Response objects to respond to fetch events in useless or even ridiculous ways, but let's now look at a simple example that combines both in a way that could potentially be beneficial: providing an offline message. Let's say we want to display a simple web page that lets users know when they are offline. This is what we want to do:

```
self.addEventListener('fetch', event => {
 // 1. Try to satisfy the fetch by using the fetch API and retrieving from
the network
 // 2. if that doesn't work – the network is unavailable – respond with an
offline message
});
```

We can detect that a fetch didn't work out – the network is unavailable – because the Promise returned by fetch will reject instead of resolving. We can provide a handler for that by chaining a `catch` onto the Promise:

```
self.addEventListener('fetch', event => {
  event.respondWith(
    // 1. use the fetch API and try retrieving from the network
    fetch(event.request)
    // 2. on failure, return offline message
      .catch(error => {
        return new Response('<p>Oh, dear! You are currently offline.</p>',
          { headers: { 'Content-Type': 'text/html' } });
      })
  );
});
```

This is close to being sensible, with one wrinkle: it will respond to *all* failed fetches with an HTML page. Let's dial this back so it only does so if the request is a *navigation* request; that is, a request for a full HTML document:

```
self.addEventListener('fetch', event => {
  if (event.request.mode === 'navigate') {
    event.respondWith(
      fetch(event.request).catch(error => {
        return new Response('<p>Oh, dear! You are currently offline!</p>',
          { headers: { 'Content-Type': 'text/html' } });
      })
    );
  }
});
```

`Request.mode` is a new-ish property that will return the string navigate if this is indeed a navigation request.

`Request.mode` support is missing in some browsers, but those browsers also happen to be browsers that don't support Service Worker anyway (at least, not yet). An equivalent fallback is:

```
if (event.request.method === 'GET' &&
     event.request.headers.get('accept').includes('text/html'))
```

PROVIDING AN OFFLINE PAGE

Creating a simple Response object that masquerades as an HTML page to provide an offline message is a starting point, but we can do better. Cobbling together a more complex Response body within the service worker code itself is unwieldy and hard to maintain. Instead, we can designate a real HTML page – in this case, a file called offline.html at the same directory level as the service worker – to use as an offline fallback page when navigation requests can't be satisfied.

It's necessary to pre-cache the offline page during the service worker's installation so that it will be available later, when needed. The following recipe pre-caches the offline page (`install` handler) and responds to failed navigation fetches with it (`fetch` handler):

```
self.addEventListener('install', event => {
  event.waitUntil(caches.open('offline-fallbacks')
    .then(cache => cache.add('offline.html'))
  );
});
self.addEventListener('fetch', event => {
  if (event.request.mode === 'navigate') {
    event.respondWith(fetch(event.request)
      .catch(error => {
        return caches.open('offline-fallbacks')
        .then(cache => cache.match('offline.html'));
      })
    );
  }
});
```

You're not limited to offline (HTML) pages. You can also handle failed requests for other resource types with a similar approach. Images are a good example – the next recipe responds to failed image requests with a custom offline image.

If your designated offline page has any subresources (like CSS or images), those will also need to be cached. See the Pre-Caching Application Shell Resources recipe for how to cache a collection of things at one time and retrieve them from cache in a fetch handler.

PROVIDING AN OFFLINE IMAGE

SVG is a good choice for a fallback image because it will scale to fit wherever it's needed. What's more, an SVG is defined by an XML-based markup language, meaning that its source can be included in the service worker itself. The following defines a simple SVG that displays the text "offline" – it can go at the top of the service worker file:

```
const offlineSVG = `<svg role="img" aria-labelledby="offline-title"
 viewBox="0 0 400 300" xmlns="http://www.w3.org/2000/svg">
<title id="offline-title">Offline</title>
<g fill="none" fill-rule="evenodd">
<path fill="#D8D8D8" d="M0 0h400v300H0z"/>
<text fill="#9B9B9B" font-family="Times New Roman,Times,serif"
 font-size="72" font-weight="bold">
<tspan x="93" y="172">offline</tspan></text></g></svg>
`;
```

Unlike with the *offline.html* page, the SVG here won't need to be pre-cached in the `install handler`, as the meat of the Response – the body – is all contained within that `offlineSVG` string, generating an image that looks something like this:

Next, expand the fetch handler. Detect image requests and respond with the SVG if the fetch `Promise` rejects. But don't do this for *every* request!

To identify image requests, you can look at the Request's `Accept` header – accessible via `Request.headers.get('Accept')`. An image request will have an `Accept` header value that matches the substring 'image' (for example, `'image/jpg'`, `'image/png'`).

```
if (event.request.mode === 'navigate') {
  // ... as before if you want to keep the offline page fallback
} else if (event.request.headers.get('Accept').indexOf('image') !== -1) {
  event.respondWith(fetch(event.request)
    .catch(error => {
      return new Response(offlineSVG,
        { headers: { 'Content-Type': 'image/svg+xml'}});
    })
  );
}
```

This is a good example of when it's useful (and not just a parlor trick) to create your own Response object.

There are many variations on this theme: you could return a different kind of offline image for different types of image request, and you could create more fallbacks for more resource types – videos, for example.

PRE-CACHING APPLICATION SHELL RESOURCES

We've already encountered the basic building blocks needed to pre-cache application shell resources, those static assets that are used on most or all pages and define your site or app's frame or chrome. That corresponds to things like core icons, scripts, and CSS. The recipe needs to:

1. Define a set of URLs corresponding to those static application-shell assets.
2. In an `install` handler, pre-cache those identified assets using `Cache.addAll`.
3. In a `fetch` handler, when a request's URL matches any of the app-shell asset URLs, return it from cache instead of fetching from the network.

This is a double-whammy: your app's shell always shows up (even when offline, as those assets are cached); *and* it has better online performance, as those network requests are avoided even when the network *is* available.

Here's a starting point for this recipe:

```
const appShellURLs = [
  'thing1.jpg',
  'thing2.jpg',
  'thing3.jpg'
];
self.addEventListener('install', event => {
  event.waitUntil(caches.open('static-assets')
    .then(cache => cache.addAll(appShellURLs))
  );
});
self.addEventListener('fetch', event => {
  const url = new URL(event.request.url);
  if (appShellURLs.indexOf(url.pathname) !== -1) {
    // url.pathname is a string relative to the site's root, e.g. '/foo/
img/bar.jpg'
    event.respondWith(caches.match(event.request));
  }
});
```

The above fetch handler behaves just fine if everything works as expected – if the app-shell resource requested is in cache as expected – but doesn't have any safety nets. We really can't ever be *sure* that something is where we left it in cache. A user might manually clear out caches (it's not typical, but it *could* happen), or the browser might decide to ditch stuff. There's no contract that guarantees that things you put in the cache will remain there.

The returned Promise from `caches.match` (and `Cache.match`) doesn't reject if there isn't a match in the cache(s) – instead it resolves to undefined(!). That means the code above could end up responding to the fetch with undefined if the resource is not in cache – that's nasty stuff and will cause an error because, remember, the browser always wants a Response object. Let's fix it!

```
self.addEventListener('fetch', event => {
  const url = new URL(event.request.url);
  if (appShellURLs.indexOf(url.pathname) !== -1) {
    event.respondWith(caches.match(event.request)
      .then(response => {
        if (!response) { // No match in caches: any falsy response is no good
          // This will cause the chained `catch` to be invoked
          // similar in effect to a rejected `Promise`
          throw new Error('${event.request} not found in cache');
        }
        return response; // DO return the response if it's OK
      })
      .catch(error => fetch(event.request)) // fetch from network if not in cache
    );
  }
});
```

When adding assets to cache in an `install handler`, the URLs used (which will be used in the keys for the cache entries) should be relative to the root of your website, even if your service worker's scope is

constrained to a subdirectory. This also makes it easier to move your service around later without having to worry about breaking paths.

Let's say a service worker is located inside of */foo/bar*. The following will appear to work:

```
const appShellURLs = ['thing.jpg']; // actual file location: /foo/bar/
thing.jpg
self.addEventListener('install', event => {
  event.waitUntil(caches.open('static-assets')
    .then(cache => cache.addAll(appShellURLs))
  );
});
```

In the `install handler`, thing.jpg will be successfully retrieved by `Cache.addAll` (relative to the service worker) and cached – this creates a valid cache entry with a key of 'thing.jpg'. But in later fetch handling, the url.pathname will always be the full path: /foo/bar/thing.jpg:

```
self.addEventListener('fetch', event => {
  const url = new URL(event.request.url);
  // uh oh
  // ['thing.jpg'].indexOf('/foo/bar/thing.jpg') = -1
  if (appShellURLs.indexOf(url.pathname) !== -1) { // nope
  }
});
```

Always use URLs relative to web root and you'll protect yourself:

```
const appShellURLs = ['/foo/bar/thing.jpg'];
```

Note: Don't cache the service worker script itself.

DEFINING NETWORK STRATEGIES

As with pre-caching application shell resources, *network strategies* are a service worker trick that can produce that double-whammy boost: improving both offline and online performance.

With network strategies, you can tune which types of requests should get the freshest content possible and which should try to get responses from cache. That helps improve online performance by avoiding unnecessary network round trips for static assets.

Implementing network strategies involves using a technique called *read-through caching*. With read-through caching, as you respond to fetches, you proactively stick responses from the network into cache where they can be retrieved later if needed. This is a great trick for keeping copies of stuff around for potential reuse if the user is subsequently offline.

A network strategy can be thought of as a sequence of priorities inside of a fetch handler: first try to respond to the fetch in this manner; if that doesn't work out, then try this; then this... and so on. You reorder those priorities based on the type of resource being requested or other criteria specific to your site's needs.

For the purpose of this recipe example, let's say we're working with a site that has significant text content (in HTML pages) that changes often, but has images that rarely change. In this case, a reasonable approach can be sketched out as:

```
self.addEventListener('fetch', event => {
  // For navigation requests (HTML/content fetches):
  // 1. Try fetching from the network – we want fresh content if possible
  // 1a. If successful, read-through cache a copy of the Response for later use
  // 2. If fetching rejects, try seeing if there is a match in cache
  // 3. If not in cache, respond with offline fallback page (optional)
  // For image requests
  // 1. Try finding a match in cache – prefer cached copy of images
  // 2. If not in cache, try to fetch from network
  // 2a. If fetch successful, stash image in cache for later use (read-
through)
  // 3. If fetch fails, respond with fallback image (optional)
});
```

The priorities used for navigation requests – getting the freshest content possible – constitute a *network-first* strategy, while the cache-preferring priorities for images represent a *cache-first* strategy.

Because both network strategies defined here share some chunks of logic, let's encapsulate the operations of putting things in cache and retrieving items out of cache into functions for convenience: `addToCache` to add items to cache during read-through caching; `findInCache` to retrieve items out of cache; and `fallbackImage` as a convenience function for generating an offline SVG Response.

```
function addToCache (request, response) {
  if (response.ok) {
    const copy = response.clone();
    caches.open('content')
      .then(cache => cache.put(request, copy));
  }
  return response;
}
function findInCache (request) {
  return caches.match(request).then(response => {
    if (!response) {
      throw new Error(`${request} not found in cache`);
    }
```

```
    return response;
  });
}
function fallbackImage () {
  return new Response(offlineSVG,
    { headers: { 'Content-Type': 'image/svg+xml'}});
}
```

There are two things to note about addToCache:

- `response.ok: fetch` Promises will reject if there is a network error and a Response cannot be obtained, but will still resolve happily if the Response is problematic (a 404, for instance). However, the ok property on a Response will return a false value if the HTTP status is not in the 200 range. Checking that makes sure we're dealing with a Response that we actually *want* to cache.
- `response.clone()`: In read-through caching like this, the Response will be used twice in the same overall `fetch` life cycle: it will be added to a cache here and it will satisfy the current fetch (and get used immediately). A Response's body can only be read once, so we'll make a copy of the Response (`Response.clone`) and store the cloned copy.

We have to check `Response.ok` here because we're using `Cache.put`. Neither `Cache.add` nor `Cache.addAll` will cache a bad (non-200) Response.

And now, the network-strategy-enhanced fetch handler (this code assumes that offline.html has previously been cached):

```
self.addEventListener('fetch', event => {
  const req = event.request;
  if (req.mode === 'navigate') {
    event.respondWith(
      fetch(req)                                  // 1. network-first
        .then(res  => addToCache(req, res))       // 2. read-through
caching
        .catch(err => findInCache(req))           // 3. cache fallback
        .catch(err => findInCache('offline.html')) // 4. offline fallback
    );
  } else if (req.headers.get('Accept').indexOf('image') !== -1) {
    event.respondWith(
      findInCache(req)                            // 1. cache-first
        .catch(err => fetch(req))                 // 2. network fallback
        .then(res  => addToCache(req, res),       // 3. read-through
caching
              err  => fallbackImage())            // 4. offline fallback
    );
  }
});
```

Maintaining and Versioning Service Workers

Over time, your service worker will evolve. You might have new application shell files you want to pre-cache. Network strategies may need tweaking. Mistakes may be found. You'll need to have a plan for versioning and maintaining your service worker.

Once a service worker is installed and active, the browser will, from time to time, check to see if the service worker file from the server is identical to the one it has already installed and activated. If there are any differences between the two files, the browser will download the new, updated service worker and install it.

Put another way, making changes to the service worker script file will prompt the browser to download and install the new version. The (new) service worker will go through its life cycle phases – `install` and `activate` event handlers – with any changes introduced – will get invoked.

USING VERSION STRINGS

A common tactic for managing changes to service workers is the use of a version string, assigned to a constant within the service worker file itself. For example:

```
// Really, this string could have any value; the important thing is that
// you change it every time you update your service worker script.
//Personally, I use the names of lesser-known Greek deities on my own site
// (recent versions include Corus, the spirit of surfeit and disdain; and
// Stheno, an immortal gorgon).
const SWVERSION='39b4893a9f';
```

Doing this may initially seem extraneous or pointless, given that any changes to the service worker file will cause the browser to download and install the new version, but there are a few reasons that having a version string is useful.

For one thing, there are some situations in which you might want browsers to download and install a service worker afresh – kicking off a new install-activate cycle – without making any changes to the service worker script itself. Bumping the version string will make browsers see the service worker as changed. We'll run into this with the "Listing Application Shell Assets in JSON" recipe.

Version strings also give you a handy, version-specific string for naming caches. By prefixing all cache names (keys) with the string, you can

differentiate between caches created by different versions – and delete caches associated with old versions. In fact, there's a life cycle phase targeted at just this kind of task: the activation phase.

CLEANING UP OLD CACHES DURING THE ACTIVATION LIFE CYCLE PHASE

The activation life cycle phase is intended for cleanup of stale service worker assets. It is especially handy for cleaning out crufty caches. Its associated event is activate.

First, make sure that whenever you create or reference a cache in your service worker, you involve the version string:

```
// prefixing cache names with a version key
self.addEventListener('install', event => {
  event.waitUntil(caches.open(`${SWVERSION}-static-assets`)
    .then(cache => cache.addAll(appShellURLs))
  );
});
```

By doing this, you know that any cache name that does not begin with the current version string is old and crufty. It's good practice to clean up – delete – those old caches within an activate event handler. In the handler function:

1. CacheStorage.keys returns a Promise that resolves to an array of cache keys (names) for all of the caches it knows about.
2. Filter this list of keys down to isolate cache keys that do not start with the current version string: these caches should be deleted.
3. Delete each of the old caches using `CacheStorage.delete`.

All of these asynchronous steps should be completed during the activation phase, so take advantage of `ActivateEvent.waitUntil` – the `activate` event is also an `ExtendableEvent`.

```
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys()
      .then(cacheKeys => {
        const oldKeys = cacheKeys.filter(key => key.indexOf(SWVERSION) !==
0);
        const deletePromises = oldKeys.map(oldKey => caches.
delete(oldKey));
        // Don't proceed until ALL delete operations are complete:
        return Promise.all(deletePromises);
      })
    );
});
```

Both `CacheStorage` and Cache objects have a keys method. `CacheStorage.keys` resolves to a list of all cache names, while `Cache.keys` resolves to a list of all keys for entries within that cache.

INSTALL AND ACTIVATE CHRONOLOGY

The activation phase comes after the installation phase, but the timing of when those phases occur depends on a few things.

Once any service worker is downloaded and parsed, it immediately moves into the installation phase, and any registered `install` handlers are invoked.

If the service worker is new new – that is, there is no already active service worker for the current scope – it will move immediately from installation into activation (and any registered `activate` handlers will be invoked).

However, if there is already an installed service worker, activation won't occur until any open client (window or tab) using the old service worker is closed. There's a way around this, however! The `skipWaiting` method available on `ServiceWorkerGlobalContext` will allow activation to occur immediately. It's a useful thing to add to the end of `install` handlers to keep things moving along:

```
self.addEventListener('install', event => {
  event.waitUntil(caches.open(`${SWVERSION}-static-assets`)
    .then(cache => cache.addAll(appShellURLs))
    .then(() => self.skipWaiting()) // Go on to activation!
  );
});
```

After activation, the newly activated service worker won't immediately take control (that is, actually do things like respond to fetches, and so on) until any open clients are reloaded. Once again you can get around this. Invoking the `clients.claim` method at the end of the `activate` handler will cause the activated service worker to take control of its clients immediately:

```
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys()
      .then(cacheKeys => {
        const oldKeys = cacheKeys.filter(key => key.indexOf(SWVERSION) !==
0);
        const deletePromises = oldKeys.map(oldKey => caches.
delete(oldKey));
        return Promise.all(deletePromises);
      })
      .then(() => self.clients.claim())
    );
});
```

LISTING APPLICATION SHELL ASSETS IN A JSON FILE

It's fine to define an array of application shell file URLs to pre-cache within your service worker script, but you may find it more streamlined to maintain that list in an external JSON file. A JSON file could be integrated with automated build tools, updating itself when your site is built or deployed, for example, or even being generated dynamically on the server when it is requested. A build script could even update the version string in the service worker file automatically. Even if you don't do any sophisticated integration like that, a JSON file might simply feel cleaner, like a better separation of concerns.

The JSON file merely contains an array of URLs:

```
[
  "/aFile.html",
  "/anotherFile.html",
  "/images/someImage.png",
  "/css/essentialCss.css"
]
```

In an install handler function, fetch the JSON file (here called *cache-files.json*) and parse the Response body as JSON (`Response.json()`) to generate an array of file URLs to pre-cache:

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(`${SWVERSION}-static-assets`).then(cache => {
      return fetch('cache-files.json')
        .then(response => response.json())
        .then(paths => cache.addAll(paths));
    })
  );
});
```

Any time the contents of the JSON file change, you'll want browsers to update what's in the app-shell cache to the latest list of files – you'll need browsers to redownload and reinstall the service worker so its life cycle begins anew and the `install` handler is invoked again.

Remember: browsers will only download a new service worker and kick off the install-activation cycle if the service worker script file itself has changed – you can't just change the JSON file and be done with it. Updating a version string within the service worker script whenever the JSON contents change is a good tactic for addressing this.

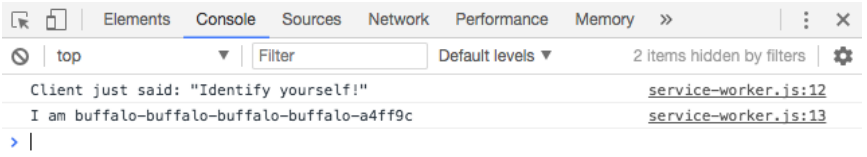
The Channel Messaging API

A service worker can't manipulate a client directly or vice versa – they're running in different contexts – but there are ways that the two can exchange messages and data. Sending a one-way message from a client to a service worker is straightforward, and is part of the Service Worker specification. In web page code:

```
function sendMessage () {
  navigator.serviceWorker.controller.postMessage('Identify yourself!');
}
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('service-worker.js')
    .then(function () {
      if (navigator.serviceWorker.controller) {
        postMessage();
      } else {
        navigator.serviceWorker.addEventListener('controllerchange',
postMessage);
      }
    });
}
```

`navigator.serviceWorker.controller` returns a reference to the serviceWorker currently in control. controller can be null for a few reasons.

The `controllerchange` event will fire when a new service worker takes control of this client – this happens when a freshly activated service worker takes over. Otherwise, if the controller is already extant, we can message with impunity.



*A view of **Developer Tools** → **Console** in Chrome shows the service worker logging from the message event handler.*

In the service worker script, you can listen for message events:

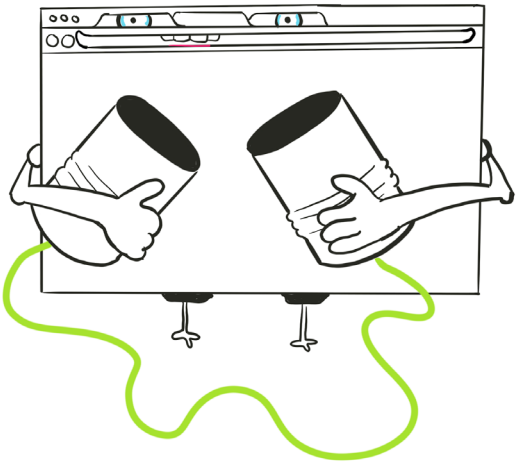
```
self.addEventListener('message', event => {
  console.log(`Client just said: "${event.data}"`);
  console.log(`I am ${SWVERSION}`);
});
```

Logging out to the console is helpful for debugging – you're able to see the service worker's logging in the browser console if you have Developer Tools open – but this really is a one-way system so far; messages go from the client(s) to the service worker, not the other way around.[17] Two-way messaging is possible, however, via the Channel Messaging API.

17 It is technically possible in some cases for the service worker to initiate communication, but it's dependent on being able to derive the ID of the client in question, which isn't always available. It's easier (and more appropriate) to use the Channel Messaging API for exchanging two-way messages.

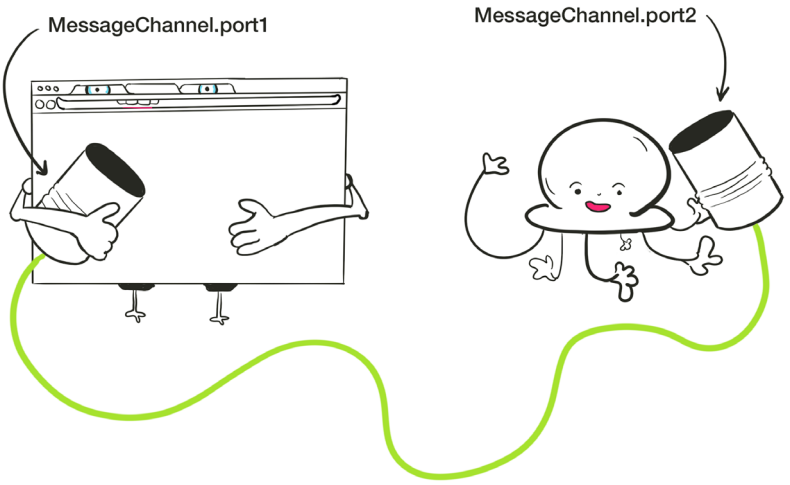
The `postMessage` function in the client can be enhanced to use channel messaging:

```
function postMessage () {
  const messageChannel = new MessageChannel();
  messageChannel.port1.onmessage = event => console.log(event.data);
  navigator.serviceWorker.controller.postMessage(
    {cmd: 'identify'},
    [messageChannel.port2]);
}
```



The `MessageChannel` object created by the client has two ports.

A MessageChannel object has two ports that can be used to communicate: `port1` and `port2`, which are connected to each other; that is, shouting into `port2` will cause stuff to come out of `port1`. The client end will listen to `port1` for a message coming from the service worker, while handing over `port2` for the service worker's use.



In `postMessage`*, the client transfers MessageChannel.port2 ownership to the service worker. The service worker's message handler function has a reference to port2 through events.ports[0].*

The first argument to `postMessage` – the data argument – can contain any value, not just a string. In this example, we're sending an object containing a fake-ish command (cmd) to the service worker, with the intent of making it tell us its current version string.

The second argument – `transferList` – is a sequence (an array, for example) of items whose ownership will get transferred to the recipient. We need to transfer `messageChannel.port2` to the service worker so it has a way to get back in touch with us.

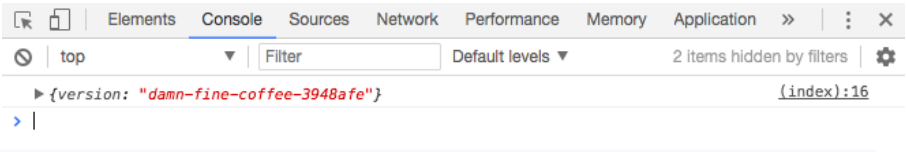
In the service worker message event handler:[18]

18 For brevity, this code assumes the existence of event.data.cmd, but you would want to be more careful, of course, in production code.

```
self.addEventListener('message', event => {
  if (event.data.cmd === 'identify') {
    event.ports[0].postMessage({ version: SWVERSION });
  }
});
```

The transferred `messageChannel.port2` from the client code becomes available at `MessageEvent.ports` – `event.ports[0]` is a reference to `port2` on the messaging channel created by the client. There's only one port available – the array only has one element because the other port (`port1`) is still owned by the client.

And the client is listening to `port1`, in the `postMessage` function:




```
messageChannel.port1.onmessage = event => console.log(event.data);
```

Chrome Developer Tools console showing the client's code logging of the message returned by the service worker. Of course, you could do more useful things with the event's data than logging it!

Web Push and Notifications

Push notifications are one of those killer, native-app-like features. Even though Push and Notifications like to go together in a single phrase, it's helpful to understand that they are two separate web APIs:

- *Web Push*: an API that allows a web worker to receive messages from a server, with an associated event, push, that the worker can handle.
- *Web Notification*: an API that allows a web worker to display a UI notification to the user.

You can have Notifications without Push, but it's rare (albeit possible) to use Push without Notifications. The two do tend to go hand-in-hand.

Setting up Web Push can be somewhat involved, but can be boiled down to these oversimplified steps:

1. Set up a *subscription* in client code via the `ServiceWorkerRegistration` interface.
2. Send that subscription's information to a remote server.
3. Respond to push events from the server in the service worker and show notifications.

STEP 1: SUBSCRIBING TO PUSH

Setting up a Push subscription occurs in client (web page) code. Now, before we get started, we need an *application key* we can give to the `PushManager.subscribe` method. This is intended to be a public key, part of a signing pair that is generated by an application server, and it needs to be a particular kind of `ArrayBuffer`.[19]

Generating a legit key pair is beyond the scope of this example, but you can cook up a fake local public key by cobbling together a `Uint8Array` that has the right kind of structure – this is ferociously hacky, but it will allow you to try out push notifications locally in the browser:[20]

19 More details about applicationServerKey can be found in the Push API spec: http://smashed.by/pushsub

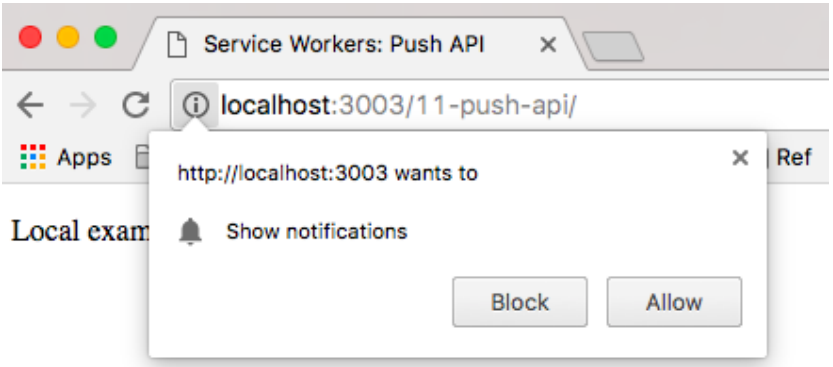
20 If this grosses you out too much, you could instead use a public key generated by Google's Push Companion (https://web-push-codelab.glitch.me/), though you'll still need to convert it to a UInt8Array. You can find a utility function to do so in the Google Web Push Code Lab repository: http://smashed.by/mainjs

```
// ALERT! ONLY USE THIS TO TRY OUT PUSH API LOCALLY IN YOUR BROWSER! ALERT!
function fakestKeyEver () {
  const fakeOutputArray = new Uint8Array(65);
  fakeOutputArray[0] = 0x04; // First octet must be 0x04
  return fakeOutputArray;
}
const appKey = fakestKeyEver(); // The kind of UInt8Array required by Push
```

Now we're ready to set up a subscription:

```
if ('serviceWorker' in navigator && 'PushManager' in window) {
  navigator.serviceWorker.register('service-worker.js')
    .then(subscribeUser);
}
function subscribeUser (swRegistration) {
  return swRegistration.pushManager.getSubscription()
    .then(sub => {
      if (sub === null && Notification.permission !== 'denied') {
        return swRegistration.pushManager.subscribe({
          userVisibleOnly     : true,
          applicationServerKey: appKey
        });
      }
    })
}
```

`navigator.serviceWorker.register` resolves to a `ServiceWorkerRegistration` when successful. That registration gets passed to `subscribeUser`, where it provides access to PushManager. Before invoking `PushManager.subscribe`, the code validates that a subscription doesn't already exist (it's null) and that the user has not denied permission previously.



If you load a web page containing the code above, you'll see that the `PushManager.subscribe` *invocation causes a (notification) permission dialog to appear.*

Even though the Push API is responsible for the subscription here, the user permission is associated with the Notification API. The `userVisibleOnly` options property indicates that the intended purpose of this push subscription is to show notifications to the user for push events. Chrome requires this property to be present and only accepts the value of `true`. Thus, the `subscribe` here will cause a permission prompt for the user.

STEP 2: SEND SUBSCRIPTION INFO TO SERVER

The Promise returned by `PushManager.subscribe` resolves to a `PushSubscription` object when successful, and that object contains vital subscription details, including an endpoint (URL) for a push server. Push messages can be sent to that URL and they will reach the associated service worker. It's not hard to see that security is important here – that endpoint URL needs to be kept secret.

In real-life code, you'd need to send the `PushSubscription` object to an application server, using `XMLHttpRequest` or similar.[21]

STEP 3: HANDLE PUSH EVENTS IN THE SERVICE WORKER

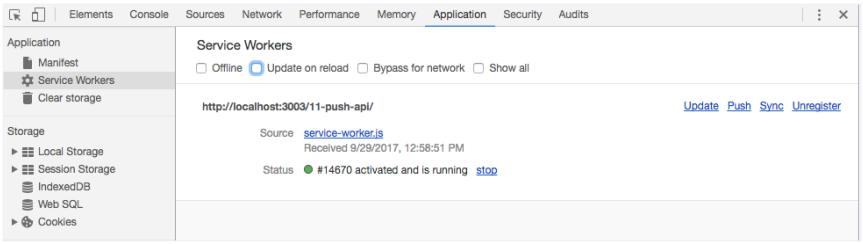
The service worker's job is to respond to push events. In this example, a notification is displayed on every push event (we already have notification permissions via the subscription). Notifications have a `title` – first argument to `showNotification` – and some straightforward attributes:

```
self.addEventListener('push', event => {
  event.waitUntil(self.registration.showNotification('Check it out', {
    body : 'Well, hello there',
    icon : 'smashing.png',
    badge: 'smashing.png'
  }));
});
```

The service worker can also handle the notificationclick event, which is triggered when a user clicks on the notification UI itself:

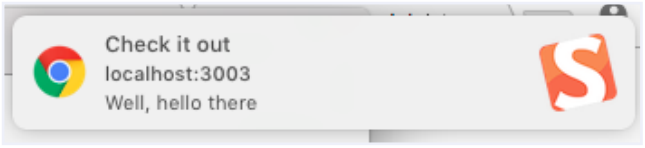
```
self.addEventListener('notificationclick', event => {
  event.notification.close(); // Close the notification UI
  event.waitUntil( // open a window to Smashing Magazine's Site
    self.clients.openWindow('https://www.smashingmagazine.com/')
  );
});
```

21 This very detailed post from JR Conlin at Mozilla digs deeper into generating key pairs and working with server-side pieces of Push: http://smashed.by/mdnpushnotes



*In this view of the Chrome Developer Tools **Application** tab, the **Service Workers** section is active (left side channel), displaying information about service workers at this origin. Note the "Push" link at the far right.*

You can try out this local Push example in Chrome: clicking on the Push link for this service worker in the Developer Tools → Application → Service Workers section will trigger a push event and show the notification.



A notification shown by the service worker's push event handler function.

Background Sync

The Background Sync API (supported in Chrome since spring 2016) allows you to put off doing things until a user's connectivity is assured or restored. Background Sync is not on the standards track at this time and is only supported by Chrome.

Background Sync has an uncomplicated API. You register for a sync in client code

```
if ('serviceWorker' in navigator && 'SyncManager' in window) {
  navigator.serviceWorker.ready.then(function(reg) {
    return reg.sync.register('my-tag');
  });
}
```

`ServiceWorkerRegistration.sync.register` takes a string tag, which you can use to identify your registered syncs. Service workers can listen for `sync` events:

```
self.addEventListener('sync', event => {
  console.log('I heard that!');
  // You might choose to do different things based on the sync's tag...
  console.log(event.tag);
  event.waitUntil( /** ... whatever needs to be done **/);
});
```

As with Push, you can emulate `sync` events from within **Developer Tools → Application → Service Worker**. Try it out!

You may be wondering when `sync` events occur. If there is currently a network connection, the `sync` event will happen immediately; otherwise, sync will happen when the browser next detects a network connection. The idea is that you can register a sync for actions that require a network connection.

You can register a sync with the same tag multiple times; only one `sync` event for that tag will get fired. If you want multiple events, use multiple different tags.

The Future

There's a lot in the web standards pipeline related to service workers. Much targets performance, including changes to data streams in the browser and the navigation preload feature.

There is a potential performance hit when a browser navigates to a site or page whose service worker isn't currently running and needs to be started up. The delay while the service worker gets going ranges from about 50ms to as much as 500ms in problematic situations – enough to be a significant impact to page-load performance. Navigation preload allows the browser to get to work on the network fetch before the service worker is ready; that is, the fetching of the web page – the navigation request – won't be blocked by the service worker start-up process.

Navigation preload is currently behind a flag in Chrome (you have to go to *chrome://flags/* and enable the feature).

Advances in what you can do with streams on the web could make custom-built Responses more performance-friendly. Currently, if you respond to a fetch event with a Response from the network (via `fetch`) or from cache, those responses will stream without any effort on the developer's part. That's a good thing: the whole response body doesn't need to be present before the browser can start doing things with it. However, Response objects you construct yourself require some intervention – currently rather convoluted – to make them stream. API improvements here could make it a lot easier to cobble together streaming Responses.

The future for the service worker landscape seems to coalesce around two main themes: performance and stability improvements to Service Worker itself, as well as the rise of more APIs and features that require a service worker to access – like push notifications. As Service Worker gets more mature, your code can be simpler and more concise, and you can rely on it in a larger share of browsers. And the new APIs add more compelling, app-like functionality to the web platform. The combined advantages of performance *and* features make service workers an increasingly vital piece of the web puzzle.

About The Author

Lyza Danger Gardner is a developer and co-founder of Cloud Four. A 20-year veteran of the web, she is a generalist with an abiding commitment to making the web work everywhere. Lyza is a seasoned and spirited speaker, and is a writer for A List Apart, O'Reilly, net Magazine, Smashing Magazine and others.

Bringing Personality Back to the Web

by Vitaly Friedman

Generic web layouts have become somewhat of a misnomer in conversations circling around web design these days. We're bored and slightly annoyed by how predictable and uninspired most web experiences have become. Not without reason, though. Every single landing page seems to be a twin of pretty much every other web page. In the header, a compelling hero image with a short main heading is followed by a lengthier subheading. Beneath them, uniform blocks of media objects are alternated – an image and a few paragraphs of text. First, text on the left, image on the right; then image on the left, text on the right. Rinse and repeat. Rounded profile photos and a square grid of thumbnails complete the picture, with perfect shapes perfectly aligned along the 12-column grid. The only variations come from sporadic parallax transitions and notorious carousels, positioned at the top or bottom of the page – or perhaps both.

It's not that somebody imposed these rules or limitations on our creative output; usually they originate from good motives and the best intentions. After all, one of the main tenets of web design has always been creating a subtle, almost invisible and functional interface – an interface that doesn't make users think, where less is more, and form follows function, where simplicity prevails – an interface where everything feels just *right*. Yet when everything is structured in a predictable way, nothing really stands out. Given how remarkably similar names, logos, icons, typography, layouts, and even shades of gradients on call-to-action

buttons often are, it's not surprising our users find it difficult to distinguish between brands, products, and services these days.

Very few people miss the golden times of the infamous Flash, with its strikingly experimental layouts and obscure mystery-meat navigation. Admittedly, the focus has shifted from creating an experience to merely providing content in a structured form. Yet unlike in those good ol' days when we talked about how wonderful or horrible websites were, today most experiences are almost invisible, making it exceptionally difficult to connect emotionally with them.

If I asked you to think of a recently visited website that left a lasting, memorable impression on you, or what websites you truly love and admire for their unique design, or what website had a truly remarkable personality, would you be able to answer these questions immediately? Would you be able to provide more than one or two examples? Chances are that you won't.

Not every website has to be unforgettable. It's not that memorable websites automatically perform better, or hit better key performance indicators. However, if you want your product or service to stand out in a highly competitive and challenging environment, you need to be different in *some* way. Many of us would consider this to be the task of the marketing team. After all, they are supposed to place the product in the right light, at the right spot, for the right audience, at the right price. Yet in a world where many digital products are fairly usable and feature-rich, this would be a daunting undertaking that would often require months of extensive research and testing without the guarantee of a successful outcome. And even then, unless you are extremely good at predicting and shaping the next shiny big thing, it might not be good enough.

Customers are used to and expect decent experiences. They aren't always fast or straightforward, but simply because of the sheer number of offerings, there are always decent tools and services out there that would be good enough. We tend to believe we rationalize our decisions to extremes, choosing the best candidates, but it's not necessarily true. According to well-known Herbert A. Simon's *satisficing theory*, we tend to prefer the first option that meets an acceptability threshold, just because we don't know if we can find a better option or how much effort it would take. We rarely study the entire spectrum of options in detail (and sometimes it's nearly impossible), and as a result, we *satisfice* with a candidate that meets our needs or seems to address *most* needs.

To draw an audience's attention, we need to be better than "good enough." Nothing can beat word of mouth, but to get there we need to come up with something that's worth looking at. What if I told you that there was a shortcut to getting there?

It's not just about price. It's not just about features. It's not just about choosing the right placement of buttons, or the right shades of colors in endless A/B tests. And it's not about choosing a cute mascot illustration that shows up in email campaigns. In the end, it's about creating an **experience that people can fall in love with, or connect deeply with** – an experience that, of course, drives the purpose of the site, but also shows the human side of it, like the personality of the people building it, their values and principles, their choices and priorities.

That means designing voice and tone, interface copy, and embracing storytelling, authenticity, inclusivity, and respect; and all of that while establishing a unique visual language supported by original layout compositions and interaction patterns. Together with clear and honest messaging, these create a unique signature, which, used consistently,

makes the product stand out from the rest. This task might sound as daunting as months of marketing research, but it doesn't necessarily require an enormous amount of effort or resources.

In this chapter, we'll look into a few **practical techniques and strategies** that might help you find, form, and surface your personality efficiently. By doing so, we'll explore how doing so consistently could fit into existing design workflows, along with plenty of examples to give you a good start. But before we get there, we need to figure out how omnipresent design patterns and best practices fit into the equation.

Breaking Out by Breaking In

The creative process isn't linear. Every single design decision – from colors and type to layout and interactivity – requires us to consider options and evaluate combinations. While the creative process is often seen as a straightforward, iterative process, in reality it's very rare that we smoothly move from one mock-up to another through a series of enhancements and adjustments. More often than not, we tend to float and diverge, heading from one dead end to another, resolving conflicts and rerouting our creative direction along the way.

Those dead ends happen when we realize we aren't really getting anywhere with the result exposed on our digital canvas. We've been there many times, so we know how to explore uncharted territories and how to maneuver the flanks, and so as we keep sculpting our ideas, we keep making progress, slowly but steadily moving towards a tangible result. Two steps forward, one step back, revisiting what we've done so far and refining those precious pixels – based on… frankly, based on intuition and random experiments. Eventually the back-and-forth brings us to a calm, peaceful, and beautiful place – just where we think we've found a solution – *the* solution.

We know, of course, that it's unlikely it's going to be the *one*, though, don't we?

This journey from *nothing* to *something* isn't just full of conflicting micro-decisions; it's crammed with unknowns, traps, friction, and difficult constraints, be they of a technical nature or time-sensitive. And at every moment of the process, the beautiful, harmless creatures of our imagination can be mercilessly smashed against the harsh reality of user interviews and client revisions. So we swizzle around from one direction to another in a fertile yet remarkably hostile place. As a result, usually we can't afford the luxury of losing time, as we know that the path to *that* deadline, harmlessly floating in the remote future, will be full of surprises and unexpected turnarounds.

To avoid losing time, we rely on things that worked well in our previous projects – the off-canvas navigation, the accordion pattern, rounded profile images, and the holy 12-column grid layout. It's not for lack of knowledge, skill, or enthusiasm that we fall back to all those established practices – it's just infinitely more difficult and time-consuming to come up with something different every single time. And because we lack time, we use all those wonderful, tried-and-tested design patterns – all of them tangible, viable solutions for a particular kind of problem. Obviously, this process might be slightly different for different people, but broken down into its essence, that's what's happening behind the scenes as we make progress in our designs.

When we started working on the redesign of Smashing Magazine a few years ago, one of the first steps we took was listing and exploring components and micro-interactions. We built the article layout and a style guide, responsive tables and forms, and used many of the established best practices to keep them accessible, fast, and responsive. Yet when putting all these perfect components together, we realized that

while they were working well as standalone solutions, they just didn't work together as a whole. The building blocks of the system weren't sufficient to maintain and support the system. We had to redesign what we'd built so far, and we had to introduce overarching connections *between* those components that would be defined through the personality and voice and tone of the new identity.

When we apply design patterns to our interfaces, we essentially bring together a group of loose modules or interactions that lack any connection to everything else. Rather than asking how a particular pattern helps drive the purpose of the experience, we often explore a micro-problem in isolation, putting micro-solutions together.

With design patterns, we run the risk of adding a component just because it's trendy these days – like a parallax-effect, slow and impactful transitions, and fade-ins. By doing so, sometimes we might lose the big picture of what role that component would play at a bigger scale, and how it could be connected to everything else. As a result, we produce soulless, dull, bloated designs with generic compositions and generic visual treatments. That's how we create something that looks like everything else.

It's not that design patterns and best practices are necessarily evil, though. They are merely a double-edged sword helping and troubling the visual output. When applying them, we need to do so carefully and thoughtfully. Whenever you consider resolving a problem with a design pattern, it's a good idea to ask yourself a few questions:

1. What problem exactly are we solving?
2. Is the pattern really the best solution for the problem?
3. How do people experience this interaction, and what pain points do they encounter while doing so?

4. How does this component help us reach the overarching goal of the system?
5. How do we connect that component to the rest of the system – in terms of both aesthetics and interaction design?
6. Is the solution really universally understood, or do we need to provide more clarity to the design (labels, better copy, affordance, replacing icons with words)?
7. Is it a good idea to keep the pattern as is at all times? Or is it better to load or adjust it conditionally, perhaps based on the viewport, or how many times a customer has visited the page?

Essentially, we try to **break down a design pattern** by exploring when and how it's useful or damaging, and how it helps in achieving our goals. In other words, we break out of predictable patterns by breaking into their nature and understanding why we actually use them. First, we examine the component in its bare, abstract form, without the context of where it's typically used and how it's usually designed; for example, rather than thinking of an off-canvas navigation sliding from the left to right, or right to left, we look into the interaction pattern on its own – essentially, progressive disclosure in which content is hidden by default and displayed on click or tap.

Then, for every pattern, we explore its usability issues and problems, resolve them, and then style and design the module in a way that feels connected to everything else. That last step could be something as simple as a consistently used transition, or a geometric pattern, or a non-conventional position in the layout. Finally, once everything is in place, we repackage the design pattern and add it to the library, ready to be served for the rest of the system.

Of course, best practices and design patterns are fantastic shortcuts for getting on the right track faster. They let us tap into predictable interactions and sequential knowledge that most of our users will have. In fact, they are as relevant today as they've always been. The key is in finding a way to apply them *meaningfully* within the context of the visual language used throughout the site, and knowing when to break them *deliberately* to trigger an emotional connection.

Humans Connect to Humans

Do you remember the good ol' days when we used an omnipresent "we" to make our little web shops appear bigger than they actually were? You might have been the only person freelancing from home in slippers and a bathrobe, or one of the very few people in a small design agency, but that profound "we" made the company sound more serious, and hence more trustworthy, didn't it? We've pretended to be somebody else to get projects we wouldn't be entrusted with otherwise – and I'll be the first to admit that I am as guilty of it as everybody else.

These days, when so many things around us are exaggerated and deceptive, authenticity remains one of the few qualities people genuinely connect to. Too often, however, it's not exhibited through a website at all, regrettably creating a vague image of yet another obscure entity covered with corporate stock photos and meaningless jargon. When every brand promises to disrupt or be different, nothing truly feels disruptive or any different, and this causes alienation and skepticism.

Humans can genuinely connect to brands they trust, but brands need to earn that trust first. Obviously, it comes from reliable recommendations and positive experiences. But as designers communicating on

behalf of companies, how do we efficiently elicit trust in people who aren't yet aware of the brand? As it turns out, trust can also come from the appearance of the brand, which can be influenced by its values, beliefs, principles, and activities. It isn't easy to fall in love with a company or organization without knowing somebody who admires it almost contagiously. It's much easier to connect with *people* whose values you support, and with *people* who stand behind their beliefs and principles.

If humans connect best to humans, perhaps if our interfaces reflected the values of the people creating them, we might be one step closer to triggering that desired emotional connection. We've been there before, of course, and so that's why we show the people working in the company on a "Team" page or in the footer of the front page, right? Well, let's look into it from a slightly different perspective.

What if you were asked to describe the personality of your brand? What adjectives would you use? Think about it for a minute, and write them down.

Ready? Chances are high that you've come up with common and predictable answers. Perhaps words such as "simple," "clean," "strong," "dynamic," "flexible," or "well-structured" have come to mind. Or maybe "attentive to details," "focused," "user-centric," and "quality-driven."

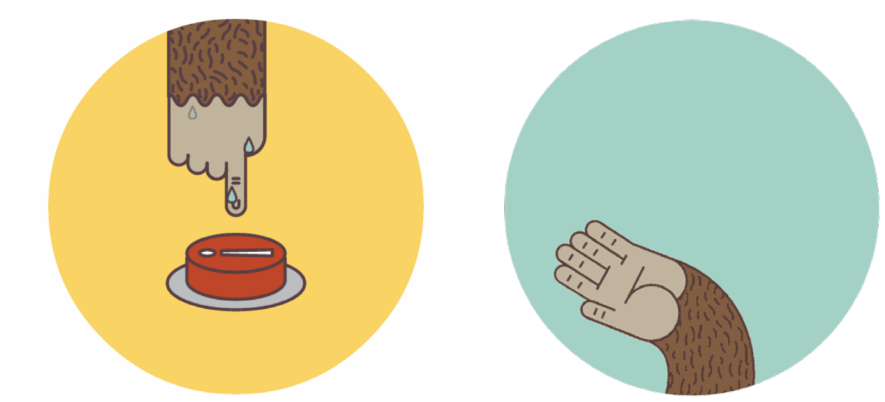
Can you see a problem with these answers? These words describe our *intention* rather than our *personality*. While the former is usually very specific and stable, the latter is usually very fuzzy and ever-changing. The qualities outlined above don't provide a good answer to the question, as they describe *how we want to be perceived*, but not necessarily how we actually *are*. In fact, usually we don't really know who we are or how we are perceived outside of the comfortable company bubble we find ourselves in.

Instead, what if you asked your colleagues and customers a slightly different question: what they care about most in their work, and what they value the most about the company or the product. Maybe they care about the diversity of talented, motivated co-workers who are knowledgeable and experienced, yet also approachable and humble? Maybe it's the fact that the company is actively contributing to pro bono projects for non-profit organizations that make a real difference in the world. Maybe because it supports schools and newcomers to the industry by providing an annual scholarship. Or because it ties in the profits with a fair salary bonus for all employees. Or just because it allows you to play with the latest fancy technologies and crazy experiments, and contribute to open-source in five percent of your working time. The company doesn't need huge ambitions, idealist goals, or a fancy working environment to stand out.

> *Designing humane experiences means being kind and humble, and emphasizing qualities that matter to the company and to users. That means highlighting privacy, respect, ethics, and transparency, but also reflecting the personality of people working on the product.*

Here's an example. Your company could care deeply about diversity, data privacy, accessibility, and transparent pricing. That would mean your interface is accessible and honest, you publicly take a stand against giving away customer data to third parties, and you include features that support pricing comparison without pushing your agenda over the edge. You could highlight those values prominently along with the competitive pricing tiers, and measure the outcome.

Now, can you spot a similar thread among all of the statements above? Because they come from personal experiences, they seem much more human and relatable than more general and abstract terms you might come up with initially.

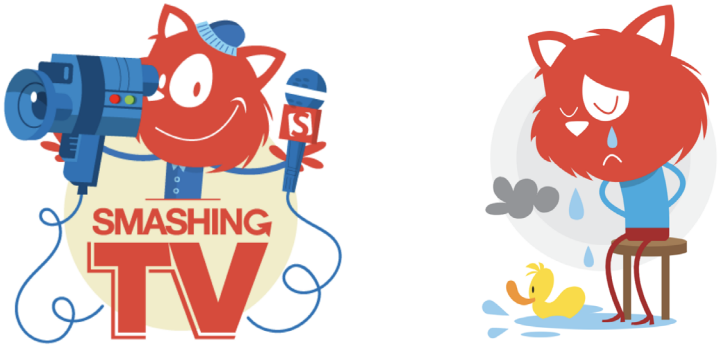


MailChimp's interaction design just before and just after an email campaign is sent out.

Slack's loading messages reflect the personality of the brand and the people working there. That's the power of copywriting at play.

That's why companies like Slack or MailChimp feel so much more tangible than brands like Uber or General Electric. They employ quirky and informal microcopy and illustrations that reflect their human side.

They don't shine through a mission statement or press releases, but through the quirks in the interface and how they communicate publicly, via email, or in social channels. That's the underlying foundation of a character deeply integrated into the user experience.



Cats have become one of the key figures in Smashing Magazine's branding. The Smashing Cat reflects the character and attitude of people working behind the scenes.

To avoid a generic appearance, you need to **define your personality first**. That means asking the right questions and finding accurate answers. When conducting user interviews with our readers, we quickly realized they had a quite different perspective on the Smashing brand than we did. We confidently described the brand by listing all the usual suspects, the qualities you probably came up with initially. The truth was baffling, though: we couldn't have been further away from how the brand was actually perceived.

We always wanted the magazine to be a professional, respectable publication with a strong voice in the industry, highlighting important work done by members of the community. User interviews brought up qualities that didn't really describe that goal in the way we always strived for. Instead, we heard words such as "informal," "quirky," "friendly," "approachable," "supportive," "community," and – most importantly – "cats."

Now, we never wanted our legacy to be cats, but it wasn't really up to us at this point. Back in 2012, our dear illustrator Ricardo Gimenes chose to bring a Smashing cat to life as a mascot for our very first Smashing Conference. There was no conscious decision for or against it. We didn't even properly discuss it, as we didn't know if we'd host more conferences in the future anyway. This small decision put something in motion that we couldn't dismiss years later. Because conferences turned out to become one of our central products, we've been promoting them heavily in our mailings, announcements, release posts, and social media messages.

Over time, every conference had to put up with a cat illustration of its own, and all these cats were facing our customers over and over again for years. Cat illustrations heavily influenced the perception of the brand without us actively fostering or guiding it. So we had to make a decision: either let the cats slowly fade away into oblivion, or integrate them heavily into the new design. You probably have discovered by now what we've settled with. As of this point, we have over 70 quirky and friendly cats freely wandering all over the new Smashing Magazine website.

However, as much as a mascot can help make the brand more approachable, it's rarely enough to convey the full story. Interviews also helped us realize how important the **community aspect** of Smashing Magazine actually was. The words "community" and "people" appeared in user interviews a lot, and not without reason – the magazine wouldn't exist without humble and generous contributions from people behind the scenes. Our design didn't really reflect it, though. So we chose to shift the focus heavily towards highlighting the people behind the scenes – authors, editors, and members of the community. Showing people prominently has become another attribute defining our design signature – and that explains why author

thumbnails take up such a prominent position in the design, and why we highlight authors publishing on their own blogs or other platforms on our front page.

What does it all mean for you? Ask questions to surface humane qualities that lie in the very heart of the company first. This will give you a foundation to build a visual language on – a language that would translate your qualities to the interface design. Every company has a unique signature in *some* way, and often it's reflected through the people working there. Ultimately, it's just about finding the time and courage to explore it – and to embrace the fact that our flaws and quirks are a part of it as much as our big ambitions and good intentions are.

Personality Is Never Perfect

As designers, we often take pride in being perfectionists. Every pixel has to be polished, every angle has to be just right, and all components should be aligned to the grid. Remember that never-ending discussion about the perfect `border-radius` on call-to-action buttons? After an eloquent and long-winded debate, the design team eventually settles on 11px, just to switch over to 13px a few months later, just to move back to 12px by the end of the year. In many companies, these changes are prompted through numerous ongoing A/B tests, in which nothing is left to chance, and everything – from assumptions to design decisions – has to be tested and proved first.

We restlessly strive to reach the most effective, the best performing solution – a solution that's just right. However, aren't we riding our horses to death trying to improve the same tiny component over and over again, just to find a slightly better variant of it, with all those minimal, microscopic changes?

Espen Brunborg, a creative lead for a graphic design agency in Norway, suggests to **never conduct A/B tests alone**.[1] According to Espen, A/B tests help us reach a *local* maximum of the user experience, but often they aren't far-reaching enough to encompass the big picture in its entirety, effectively stopping us from reaching a *global* maximum.[2] That's why in addition to A/B tests (in which microcopy and colors and positions in the layout are tested), they run so-called *A/Z tests*, testing an existing "baseline" design against completely different designs. Their differences lie not only in the shade of a button or copy, but in absolutely different layouts and visual treatments. The branding and the core principles remain the same, but pretty much everything else keeps evolving. This allows Espen and his team to reach new absolute maxima in terms of conversion and KPIs every few months.

In one of our conversations years back, Elliot Jay Stocks, who was involved in the 2012 redesign of Smashing Magazine, briefly mentioned one fine detail of his design process that stayed with me for quite some time. He said that a good design possesses one of two qualities: it's either *absolutely perfect* in every way, with perfect alignment, sizing, and hierarchy (which is usually quite hard to achieve), or it's *deliberately imperfect* in a few consistent ways (which is much easier to achieve). According to Elliot, in a good design there shouldn't be anything in between. In other words, buttons should either be perfectly aligned to the grid, or not aligned at all – offset by 20–30 pixels and more. Being off just by a few pixels would feel wrong, while being off by 20–30px looks deliberate, and hence less broken.

1 In his article "Putting A/B Testing in Its Place" (http://smashed.by/abtests), Jakob Nielsen highlights the limitations of A/B testing. E.g. it should never be the only testing method. Observations of user behavior often generate deeper insights.

2 Bill Buxton was probably the first to discuss this problem in his book *Sketching User Experiences* back in 2007. According to him, designers often end up with a local hill-climbing problem when the design gets plateaued on a local maximum.

So what if, instead of chasing the perfect solution for every single component, we ran and tested various expressions of our personalities? In interface design, it would mean entirely different creative directions. Perhaps a multicolumn layout with bold typography, against a geometric layout with a single accent color? What if, rather than seeking the perfect roundness of a button, you deliberately introduced slight inconsistencies? A custom animation on one of the call-to-action buttons, or a dynamic placement of an image outside of the box in which it's usually supposed to be placed? Or perhaps rotating a subheading by 90 degrees? The personality can be expressed in many entirely different ways, so the task is to discover variations that are promising enough for testing.

A personality is never perfect, and so perhaps our websites shouldn't be perfect either. What if you set up a publicly visible art board in your company, with magnets representing the qualities on one side, and magnets representing components or visual treatments on the other side, and then randomly clashed one against the other to produce a visual direction for the next A/Z test? Apply perfectionism to the level of detail required to produce deliberately imperfect designs.

This approach won't always win, but complemented with A/B tests, it might bring you to new heights you wouldn't be able to achieve otherwise. Ultimately, we want customers to fall in love with their experience and consequently the brand, to form a lasting bond. A deliberately imperfect yet humane interface can help us get there. It requires you to find just one distinguishable quality that nobody else has, and boost it up.

Choose One Thing and Boost It Up

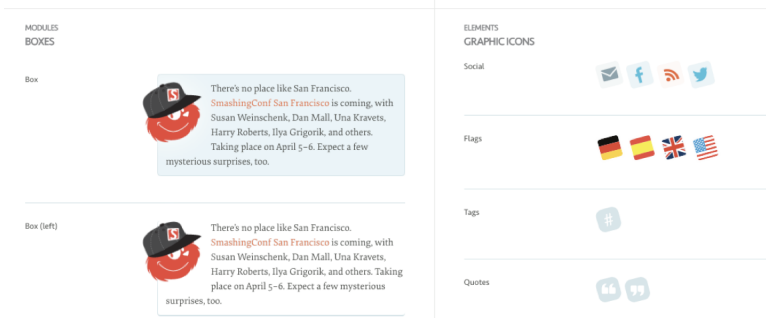
Personality can be expressed through a *design signature* – a recurring visual treatment, the voice and tone of the copy, or an interaction pattern used consistently from one page to another. It might be tempting to explore a diverse mix of sophisticated, non-conventional treatments that would be seen in the interface miles away from the mouse cursor. However, that's a recipe for a disastrous experience that prioritizes a designer's expression over users' intentions. However bold the personality is, its design signature should remain subtle.

When working with Dan Mall on the Smashing redesign, one interesting detail Dan brought up at the very start of the project was the role of the signature in the final outcome. According to Dan, choosing a few distinct, competing expressions of the personality is often too much: it's enough to choose just one little detail and boost it up all the way. In more practical terms, that means picking one pattern and using it consistently, *everywhere*: on every page, and in every user interaction. How do you find that sacred detail? You go back to the roots of the company.

In the very early days of Smashing Magazine, we didn't have any branding at all. We chose a pretty random WordPress theme, placed the name in Arial, and that was it. Eventually, in early 2007 Ryan Denzel from South Africa designed Smashing Magazine's logo, which included a letter S tilted by 11.6 degrees. Despite minor alterations in the shade and colors of the logo, we stayed true to the design for over a decade, and with the recent redesign, we weren't considering changing it. However, when seeking a design signature that would be deeply connected with the brand, we actually took the tilting of the logo very close to our heart – from the very start.

Early design explorations with Andy Clarke used the tilting consistently for every single visual element on the site. This signature carried over to the final design as well. Today, all icons, author images, conference flags, job board logos, illustrations on product panels, and book covers on product pages are all consistently tilted. That tiny detail doesn't break the experience, yet it lends a unique visual treatment to the design that's clearly distinguishable from everything else as a result.

Admittedly, we did redesign the tilting through the process, moving away from 11.6 degrees to 11 degrees, and adding 11px roundness to all components. It was months later that the bold colors and typography and layout came into play, supporting the quirkiness and informal style of the tilted elements – all slowly crawling up into the design mockups eventually.



Early design explorations used the tilting element consistently for every visual element on the site. This signature carried over to the final design as well.

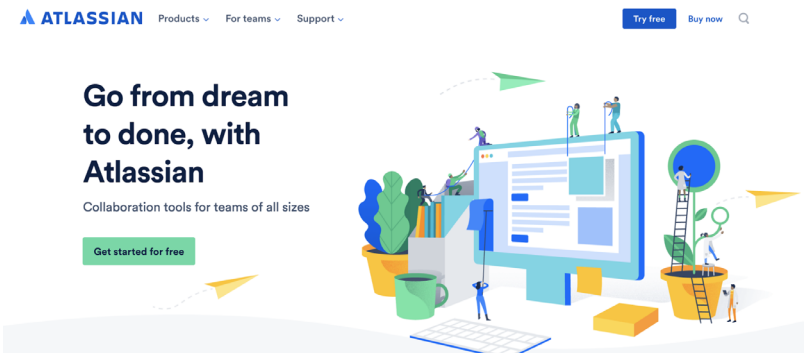
At this point you might be slightly worried that you don't really have any distinctive element that could be promoted to become your signature. You might not have the tilting or a particular color palette that stands out. As it turns out, *anything* can become a design signature. In the next sections, we'll explore some examples and ideas you could use for you own particular situation.

Always Prefer Custom Illustrations

Once the qualities of a personality have been identified, the next step is to translate these qualities into a distinct visual language. Initially it happens via color and typography, so when defining the visual style, look out for these qualities in initial color combinations and type families.

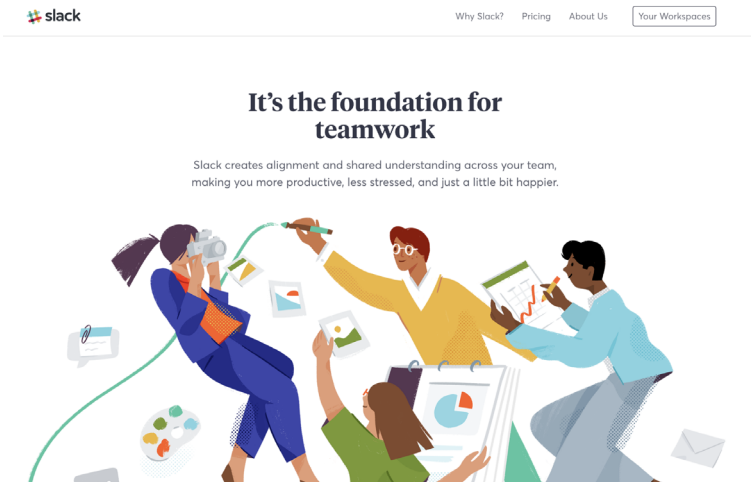
Probably the easiest way to come up with your own design signature is by using custom illustrations designed specifically for the brand. Every artist has their own unique style, and unlike stock images or stock photos that often almost enforce generic appearance into layouts, custom illustrations give a brand a unique voice and tone. You don't need to go overboard and create dozens of illustrations; just a few would probably do. Think about replacing all the stock photos you've purchased with custom illustrations – this should give you a good enough baseline to start with.

Atlassian is a wonderful example of an illustrative style applied thoroughly and beautifully at every touchpoint of the experience. The illustrations are more approachable than stock photos. Notice, however, that they rarely appear on a plain background – they are supported by the color palette and typographic choices that complement the illustration style.



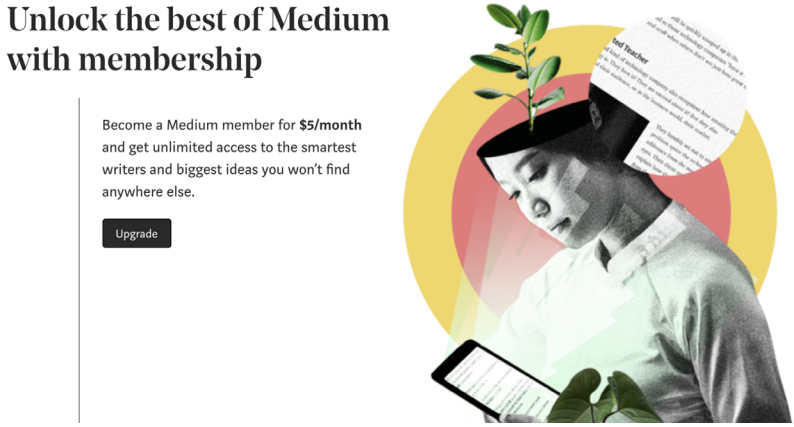
Atlassian uses a friendly illustrative style with color consistently used for buttons and headings at every touchpoint of the experience.

Why are custom illustrations not enough to stand out? Because just like many other attributes on the web, illustrative style also follows trends. Compare Atlassian's style to Slack's visual language. Yes, the fine details are different, but the pastel color combinations are similar. The illustrations from these different projects could happily coexist within one single website, and many customers wouldn't notice a difference.



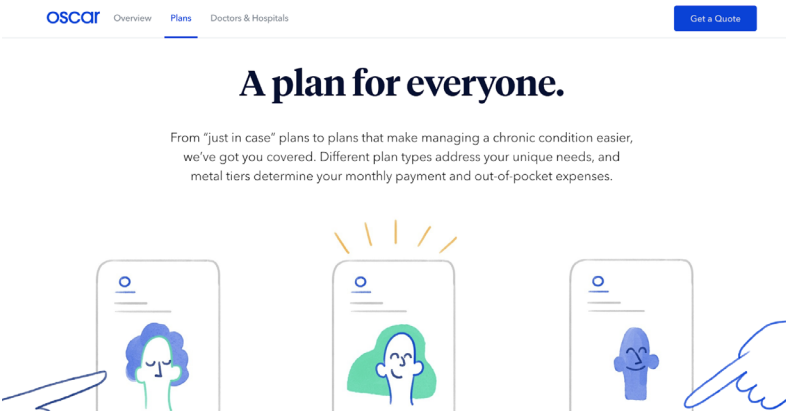
The style of the illustrations on both the Atlassian and the Slack sites look fairly similar. Both illustrations could happily coexist within one single website.

A distinct visual style requires further attention to other elements on the page, primarily backgrounds, typography, shapes, and layout. Notice how all these **elements play together** on *Bond*.[3] They interplay with the background, text colors, and the layout.



Medium uses a collage-like style for all its illustrations on landing pages and help pages. The key is that illustrations are used consistently across pages. They might not make sense to every visitor, but they contribute to the unique visual appearance of the brand.

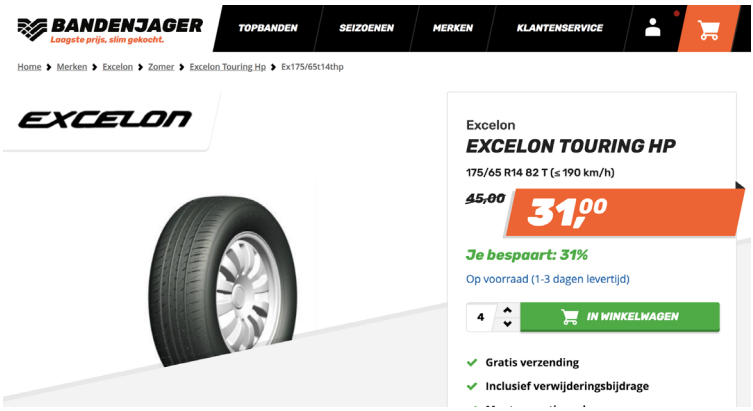
3 https://bond.backerkit.com/



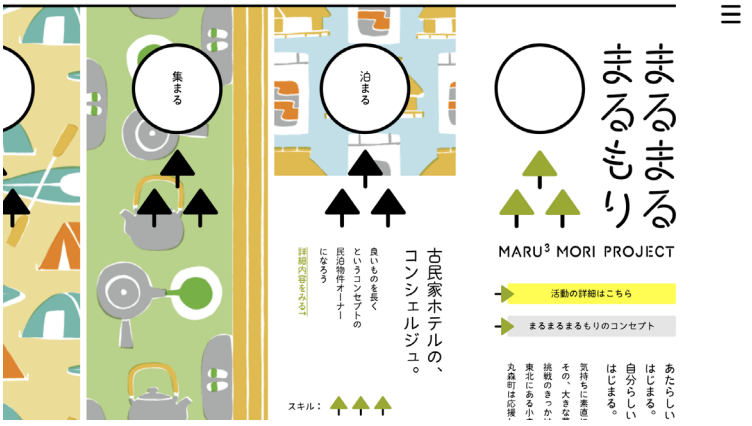
Health insurance is a very competitive and not particularly friendly nor transparent environment for citizens and business. With custom illustrations, subtle animated GIFs, and straightforward copywriting, *Oscar*,[4] a newcomer to the industry, appears more approachable and relatable.

WebConf.asia is a conference website with vivid color combinations and background, and boxy components designed as if they were **three-dimensional**. This is enough to set the design apart. The visual treatment produces depth, which is used for speakers, talks, and main navigation.

4 https://www.hioscar.com/

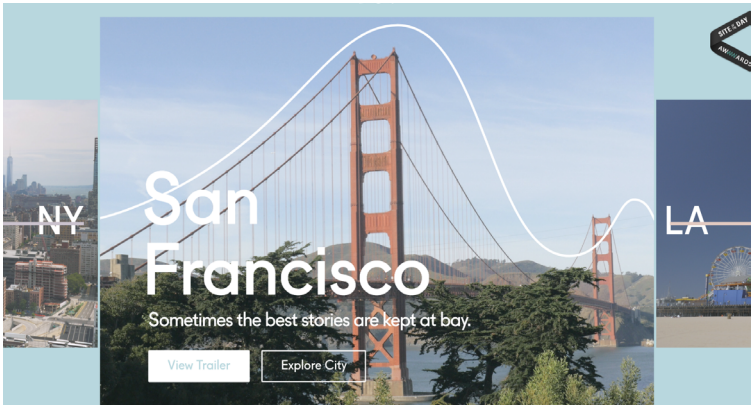


Bandenjager uses **slanted shapes** and compositions consistently on call-to-action buttons, in the navigation, and even in the quantity selector on the product page. That's their design signature. Notice how even micro-components such as product labels use the same pattern.

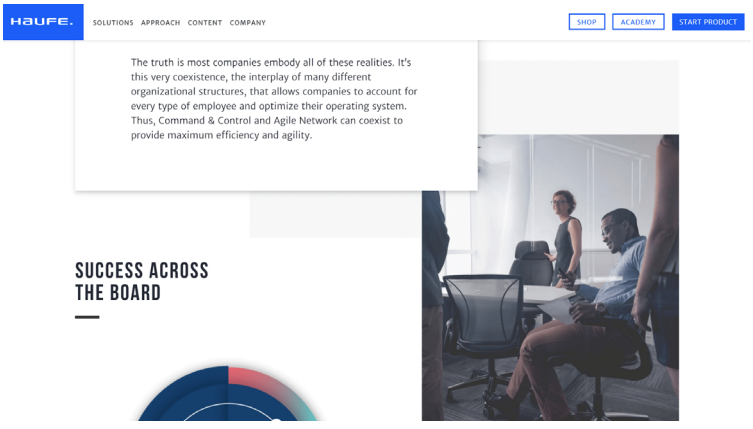


Maru Mori Project[5] is an economic development project initiated by the municipal of Marumori, Japan. The site uses circles and tree shapes everywhere — "Maru" means circle, "Mori" (森) means forest.

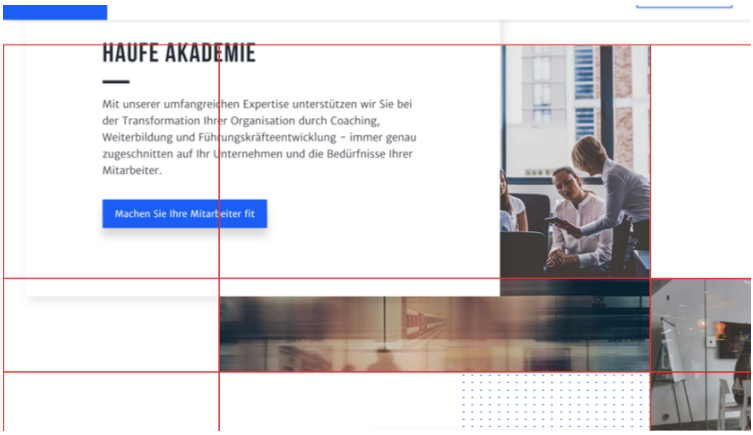
5 https://marumarumarumori.jp/



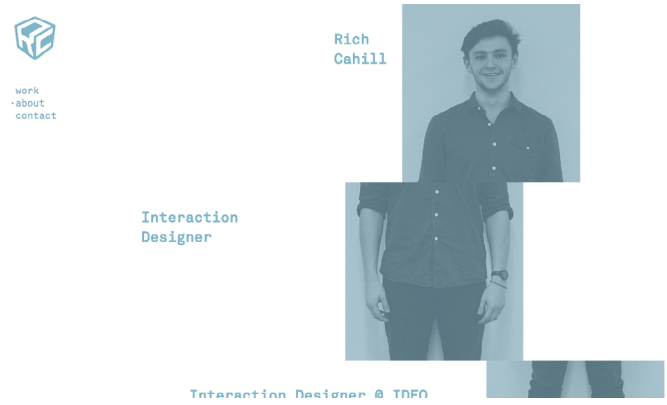
Storytrail is a city guide with interactive maps and videos. Every city has its own signature which is a wavy horizontal line outlining a city's most important landmark. The design uses this signature line for various animations, transitions, and arrangement of items in the layout.



Haufe uses overlapping backgrounds to add more dynamics to the design. The main structure of the grid is derived from the letter H, which is the main character of the company's identity. All components are laid out on the grid to support that personality trait. A nice play of



photos, original compositions, and a variety of geometric backgrounds at once. *Haufe*'s design system[6] beautifully describes the underlying principle of Haufe's dynamic grid.



Another way of drawing attention is by adding randomness to your composition. Rich Cahill's site[7] splits its photos into three vertical parts, randomly offset horizontally and colored with a set of pre-defined colors. It's a nice example of combining predictable parts of the system in seemingly random, unpredictable ways.

6 http://smashed.by/haufeds
7 http://rc3.me/

Lynn Fisher[8] also adds some randomness to her portfolio. The layout changes completely between different breakpoints, creating a totally different experience on mobile and desktop devices. Even the favicon changes dynamically as well.

When considering the visual direction of a site, it's a good idea to consider custom illustration style, backgrounds, typography, and shapes. Establish strong connections between all of these attributes by reusing design decisions, such as the choice of colors and spacing. While doing so, of course, it wouldn't hurt avoiding predictable options used widely everywhere else. One of the effective ways to achieve this is by keeping tabs on ongoing design trends, then pick the most prevalent one and… *smash* it to pieces.

8 https://lynnandtonic.com/

Pick a Trend and Smash It to Pieces

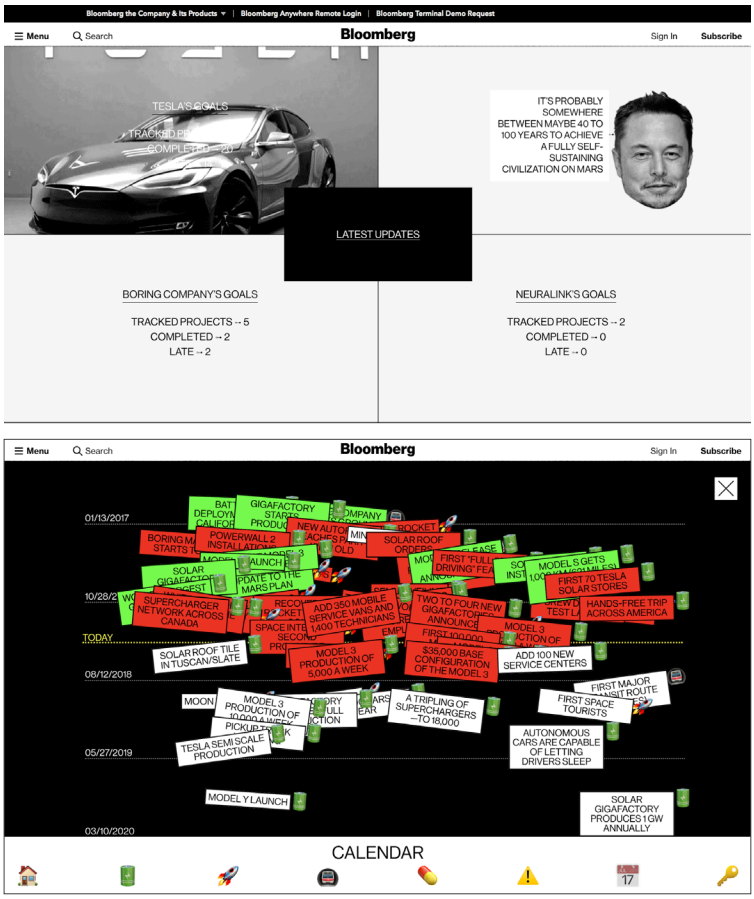
When talking about great design, Yves Saint-Laurent, a well-known French fashion designer, once noted that "Fashions fade; style is eternal." According to Saint-Laurent, to create timeless designs it's important to take note of trends, yet serve an *interpretation* of trends through the lens of your own personal style. That's not what we usually see on the web.

It's almost ironic that it has become trendy to dislike trends these days, and for good reason: usually their primary purpose is visual embellishment, rather than driving a designer's intent, and often they don't add much to the experience beyond friction, confusion, and fancy whistles and bells. No wonder then that designers have started to fight back with "*brutalist designs*"[9] – websites that aim to exhibit the essence of a website in its unstructured form, exposing the website's functions to extremes.[10]

While doing so, designers often deliberately break design patterns, usability practices, and design trends. At first glance they might appear as designs created with the sole purpose of being different, but because they have a striking personality, they draw attention to themselves. Admittedly, sometimes they seem to be too far-fetched in how they deliberately turn their back on well-established design principles. Not everybody can afford it, and not everybody would feel comfortable connecting such non-conventional aesthetics to their brand.

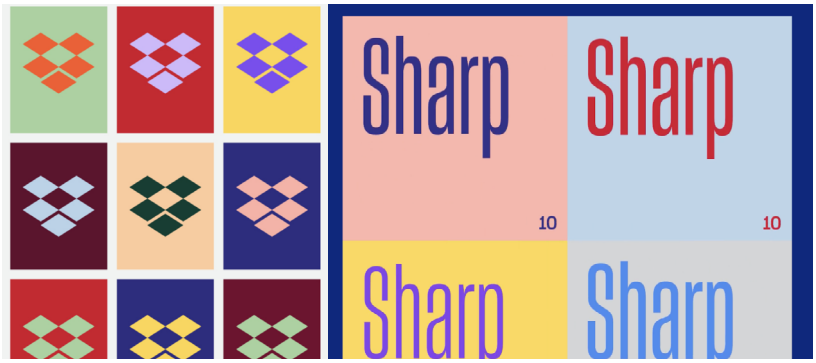
9 http://brutalistwebsites.com/

10 Brutalism in architecture is characterized by unconcerned but not intentionally broken aesthetics. When applied to web design, it often goes along with deliberately broken design conventions and guiding principles.



A brutalist design showing up in its glory. A feature on Elon Musk, published by Bloomberg.com, breaks most conventions we've got used to over the years.

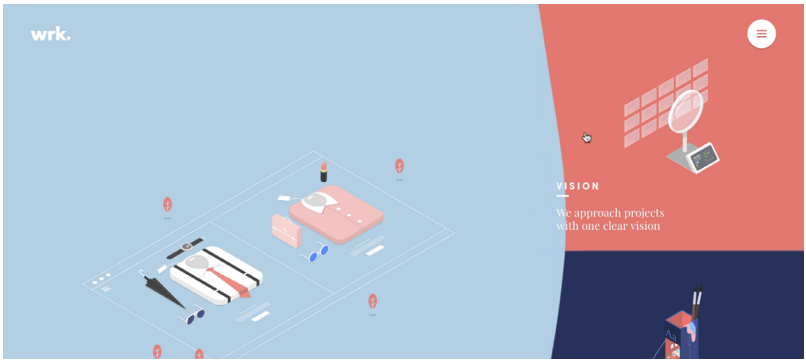
A slightly more pragmatic strategy lives somewhere between generic designs and brutalist designs. To get there, you could pick a trend, find a unique perspective and apply your personality to it. For example, if you see many websites using smooth and silky animations, think about how they would fit into your story, and find the twist that would enrich it, and make it more personal. Break down the trend into pieces to understand its mechanics and what's happening behind the scenes, then twist some parts of it, repackage, and integrate into your design.



Dropbox rebranding wasn't received too well. The design is very bold, and very noticeable. As such, Dropbox achieved its goal of being talked about in their redesign.

Instead of bouncy animations, you could use an artificial delay which slows down the appearance of items on a page. If most profile images have a perfect circular shape, try to come up with a different shape that would work well for showing avatars. If most photos are rectangular, think of another shape that might do the job well. Lo-Flo Records[11] uses an artificial delay with geometric placeholders. The website uses geometric placeholders; smooth, well-orchestrated transitions come to life when content is being populated.

11 http://loflorecords.com/



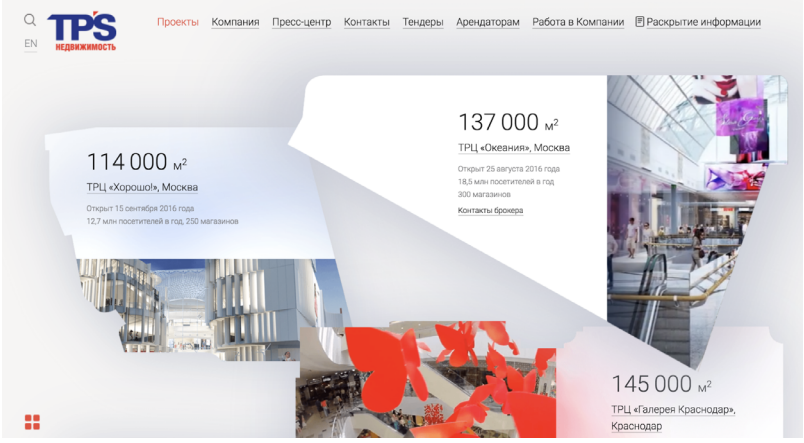
Instead of using off-canvas transitions, think about a particular transition or animation that would reflect your brand best. For more corporate entities, a fast-paced transition might work best; for creative projects, a slightly more playful and slow transition might be a better fit. *Waaark*[12] is a wonderful example of the latter. If all transitions were removed, the portfolio website would look like pretty much any other portfolio site.

Implement Consulting Group uses a short and subtle geometric animation to highlight the featured article on the site. Foreground and background images are a bit offset and animated, with a geometric shape in

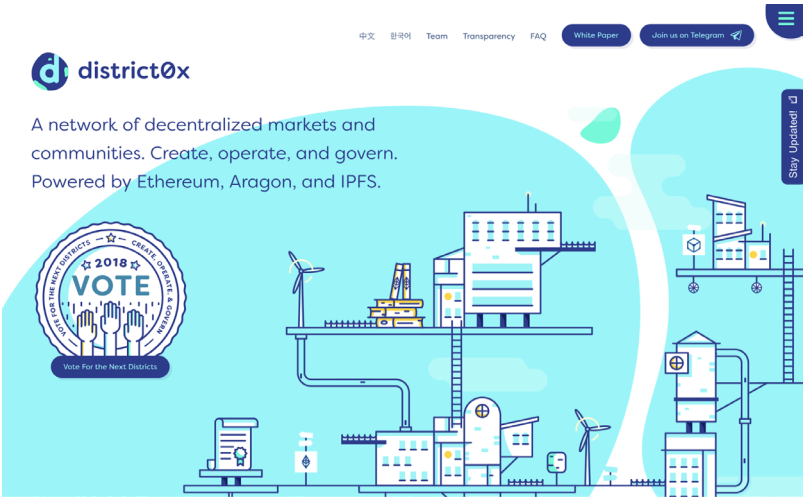
12 https://waaark.com

the background and a story preview in the foreground. That's enough to give the experience some personality.

Imagine for a second that you have to redesign your ongoing project, but can't use any basic shapes such as circles, rectangles, or triangles. What would you choose? We all know there is an infinite amount of options, but why is it then that so often we are constrained by highly predictable and heavily used choices? What is neither a circle nor a rectangle nor a triangle? Well, slanted or tilted elements aren't. Neither are letters and large typography. Nor are custom responsive illustrations or iconography. Nor whitespace, audio, and video. Nor transitions and animations. Nor pretty much any other shape created via a polygon, with content embedded via SVG masks. TPS,[13] a Russian real estate agency, uses the shape of its properties for thumbnails, rather than generic squares or circles. Every single property has its own shape. The idea is used consistently for every single property.



13 http://www.tpsre.ru/



Districtox is a network of decentralized markets and communities. The site uses custom shapes, smooth transitions, and animations to provide a distinct experience. No rectangles or circles. And notice how well the colors, background images, and typography work together on the site.

It's not that all basic shapes should be ignored and dismissed from now on, of course. Avoiding basic shapes deliberately is one of the very first exercises we do when we try to come up with a slightly more original art direction. Once you've come up with a decent idea without basic shapes, you can start bringing them back sparingly when necessary. Chances are high, however, that you might be able to get away without them altogether.

Do Make People Think

Why is it that when we are puzzling our way around a foreign city and a souvenir shop owner is desperately trying to catch our attention on the street and push a sale, we pass by in haste; yet we slowly walk into a beautifully designed souvenir store that is silent and humble just around the corner? Perhaps it's because we seek authentic, honest, and respectful experiences, and tend to ignore everything that doesn't fit the bill. In his fantastic book *Blink*, Malcolm Gladwell outlined an interesting phenomenon related to how humans value their experiences.

According to Gladwell, we tend to be more satisfied with our experiences when we feel valued, listened to, and understood. Doctors who take a disproportionate amount of time listening, asking questions, and taking notes with their patients tend to get significantly better reviews and higher ratings despite the fact that other doctors might be as proficient and knowledgeable. They might jump to correct conclusions faster, yet their efficiency doesn't elicit trust and connection in their patients. Of course, primarily we want the problem to be solved, but we also love falling in love with a charming personality, wisdom, expertise, and human kindness.

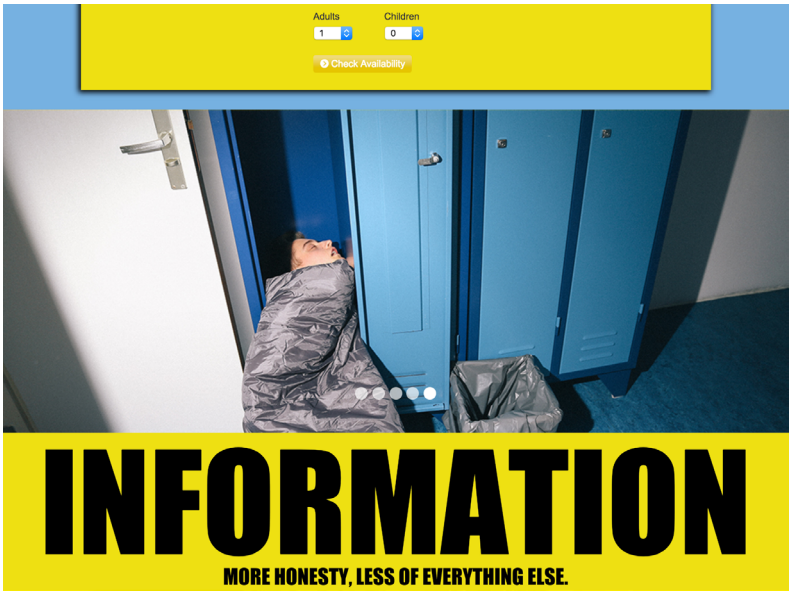
We know by now that we can enable human connections by embedding compassion into our interfaces. However, these connections don't just happen overnight – they take time. But where does it leave us in the age of instant gratification and invisible interfaces, when it has become the essence of our job to avoid interruptions and distractions, and create a clear path for customers to follow seamlessly? If we aren't supposed to make people think, how do we even get a chance to establish an emotional connection in the first place?

We do so by slowing down. By making people think. Not much. Just a little bit. Just enough to make them feel valued, or smile, or get curious. We do so by adding friction. A few bumps here and there, enough to offer a chance of being directly confronted with the personality infused in our interfaces. It might even mean confusing the customer every now and again just to enable a speedy recovery from that confusion with a dash of positive emotion in their eyes. That's how memorable experiences emerge.

Everything is a *little* off on *BAO London*[14] – the spacing, the color combinations, the form layout, the hierarchy, the buttons, the cursor, the lightboxes, the illustrations. This consistent breaking of predictable patterns makes the website appear interesting and appealing. Breaking things slowly and deliberately, one component at a time. That's not a regular restaurant website.

Everything is *way* off on the Hans Brinker Budget Hostel website[15] and it's done deliberately as well. The hostel was struggling to sell rooms as

14 https://baolondon.com/
15 http://hansbrinker.eu/



If you can't make it better, make it worse. Even if you don't have a wonderful product to sell, it's always possible to wrap a story around it and make it more interesting.

the competition is quite tough in Amsterdam. Rather than improving the design, they made it worse to fit well within their story. Pretty much every element on the page actively makes people confused – from color combination to typography to interactions. And it worked – they are expanding to Lisbon now. Not everybody will like it, and some people will find it annoying, confusing, misleading, childish, or just over the top. Very much like we find it difficult to connect to some people, we might experience the same issue with an interface that attempts to show its human side. But isn't it worth it? Perhaps in times when everything is remarkably similar and doesn't really stand for anything, it's worth striving for our product to be genuinely loved by many people for the price of being genuinely disliked by some people, rather than eliciting no feelings at all.

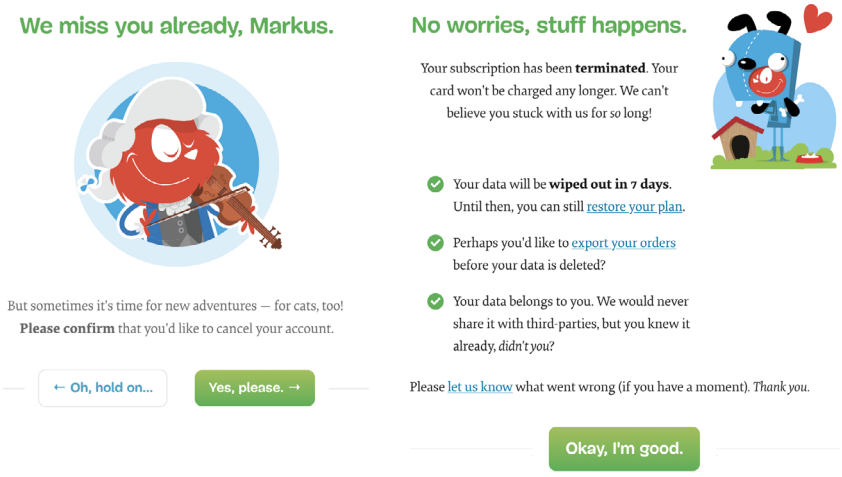
In his *How I Built This* interview on NPR,[16] Mike Krieger, the co-founder and creative mind behind Instagram, mentioned that rather than spending a significant amount of time trying to understand why people abandon the service, one of the fundamental principles that drives growth is focusing on customers who deeply love your product and stay around for years. By prioritizing existing customers and what they truly love about your product, you might not only attach them to your product, but also boost the word-of-mouth marketing that's infinitely more effective than traditional landing pages.

It doesn't mean, though, that we shouldn't take good care of experiences customers have when abandoning the product, or – even worse – that we should make it harder for them to leave. The humane qualities of the interface should shine through all the touchpoints of the experience – and it holds true for onboarding as much as offboarding. The latter is mostly deemed as being not *that* important – after all, when the customer will face it, they have almost abandoned the product.

16 http://smashed.by/npr

Offboarding Matters

Just like human relationships sometimes end abruptly and badly, leaving a lasting negative aftermath, so can our relationships with digital products. It's highly unlikely that a person abandoning a product with a mysteriously long-winded journey through cancellation redirects would praise the product to friends and colleagues. People leave for very different reasons, and sometimes it has literally nothing to do with the service or the experience. They might have moved on, or just want to save money for something more important, or perhaps they just found an alternative that better fits their needs.



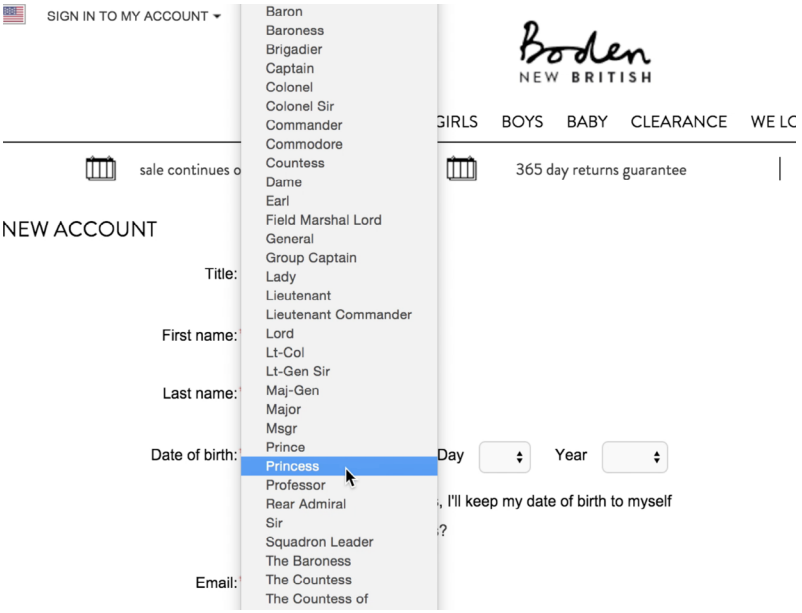
What if at the moment of departure we make them feel deeply valued and understood? Admittedly, with Smashing Magazine's redesign, we didn't spend too much time designing the offboarding UX, but it was important for us that the experience fitted well within the overall personality of the interface. When our customers cancel their membership subscription, we greet them with a respectful and even encouraging notice, providing a little gift for sticking around with us for so long, and explaining what happens to their personal data.

The result was surprising: we discovered that customers who cancel the subscription and go through the offboarding UX, sometimes tend to be even more eager to recommend us to their friends and total strangers than some loyal members who stick around for a long time. They just admire how respectfully and thoughtfully we deal with their decision, and that we don't pull all the shady tricks from the trenches to make it difficult for them to leave.

Make Boring Interesting

It's difficult to introduce playful elements into an experience which is otherwise very much corporate and formal. However, whenever you are designing a particular interaction, be it as simple as hovering a button, or moving from one section to another, or filling in a form, there is always some room to make the experience slightly more interesting.

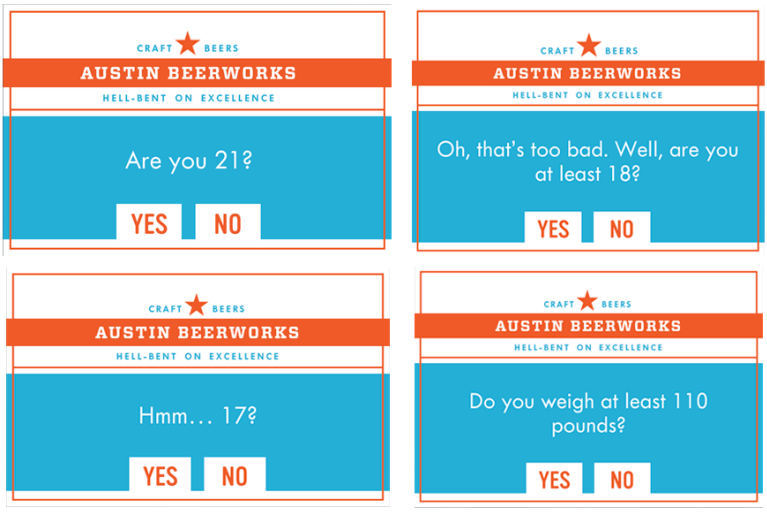
For example, out of all the wonderful form elements on a given page, what could be less exciting than a "Title" input? Sometimes appearing alongside radio button counterparts or a dropdown, rigorously asking customers for very personal information about their marital status, without any obvious reason whatsoever. And that's exactly the moment when we can make it shine beautifully. A great way of creating memorable experiences is adding a bit of surprise at the point where **it's most unexpected**. Pick the most boring, unnoticeable part of the experience and try to make it interesting. Now, is there a way to make this interaction more interesting?



When creating a new account on *Boden*,[17] customers are dazzled with a quite unusual selection of options, ranging from Admiral to Squadron Leader and Baroness. Who hasn't wanted to be a Squadron Leader at some point in their life? This little design decision elicits smiles, and prompts customers to share this gem with their friends and colleagues. By the way, the list of options is quite lengthy.

Austin Beerworks (see the example on the next page) is just one of many local breweries in the US. When customers enter the site, as always they are prompted with an age check that's supposed to ensure they are over a certain age limit. Most people – honestly or dishonestly – would click on "Yes" and move on, but if the customer chooses to click on "No," they embark on a "choose-your-own-adventure" trip to be guided to a video that best describes their personality.

17 http://bodenusa.com/



Who doesn't love disliking a pop-up? However, pop-ups can be made interesting too. *Volkshotel*[18] uses the most annoyingly delightful pop-up out there, beautifully illustrated as a person holding a sign in front of the website. As the visitors hover over it to close it, the pop-up sneakily moves away a little bit, making it just a tad more difficult to close it. Personally, I wish every single website had a pop-up like that.

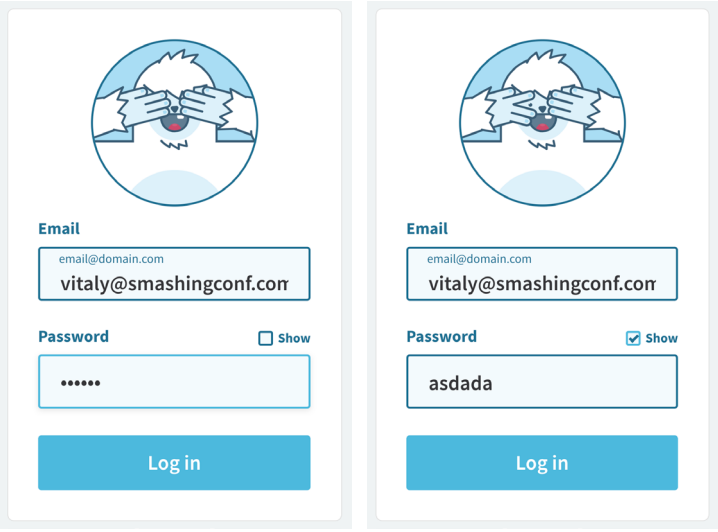
18 https://volkshotel.nl

Tympanus' 3D Room Exhibition[19] doesn't look particularly exceptional until the visitor chooses to interact with it. When moving from one exhibition detail page to another, rather than just loading another page, the user is moved from one room to another within a 3D space.

What's a common interaction on the web? Forms, in all their different flavors and appearances. In fact, chances are high that you have some sort of a login and password input on your site, and, of course, that's a pretty boring interaction. Adding a character responding to a user's input might spice things up a little. As a result, people might spend more time interacting with the form before signing in. That's better engagement at hand. Darin Senneff's Yeti character[20] (see image on the next page) does just that.

The strategy is simple: choose one predictable, boring pattern, study user expectations and... break them mercilessly by adding something unexpected and uplifting to it. Please note that it doesn't mean breaking usability just for the sake of breaking it; rather, it's about making a handful of boring elements more interesting by adding some unconventional treatments to their design.

19 http://smashed.by/tympart
20 http://smashed.by/yetiform



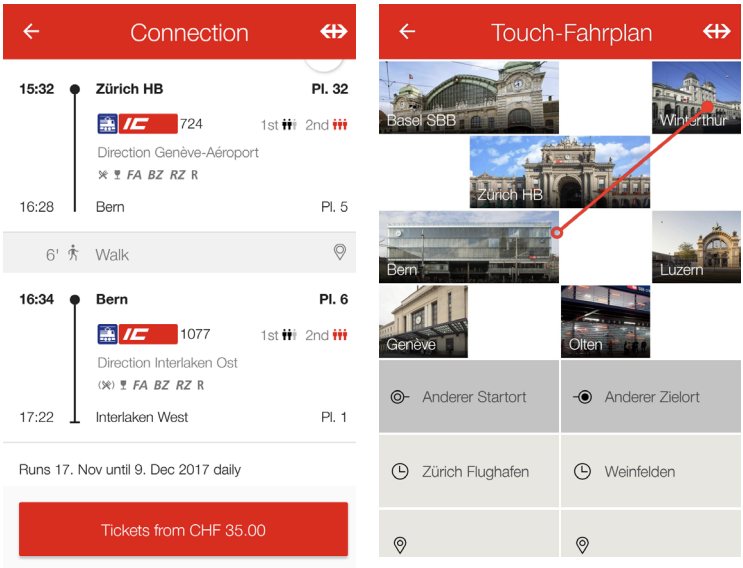
Darin Senneff's Yeti form (http://smashed.by/yetiform) responds to input actions such as revealing a password or typing in an email.

Find a Pain Point and Solve It Well

Can you hear restless voices of skepticism whispering from the corner of the room? Not every corporate setting will sustain a funky custom illustration, a quirky animation, or an unconventional interaction. A striking visual identity might not really fit into your digital presence, custom illustrations might not be within the budget, and you might not want to break customer's expectations anyway. In these cases, you might want to explore a slightly different route. If you can't convey your personality through unconventional aesthetics or interaction, an alternative is to convey it through superior problem solving. It means you need to uncover painful moments of interaction – when customers are annoyed or disappointed or confused – on similar sites, and sweep through experimental and seemingly far-fetched solutions to try to trump the experience that your competitors provide. Take on a problem, and tackle it meticulously, head on.

Surprisingly, most of the time these pain points aren't particular features; it's the perceived complexity of the interaction and the lack of transparency. Too many form fields; too much time investment; too slow an interaction; too many repeated questions; too many unnecessary requirements. The angle is to find a way to make seemingly complex interactions deceptively easy, hence surpassing expectations.

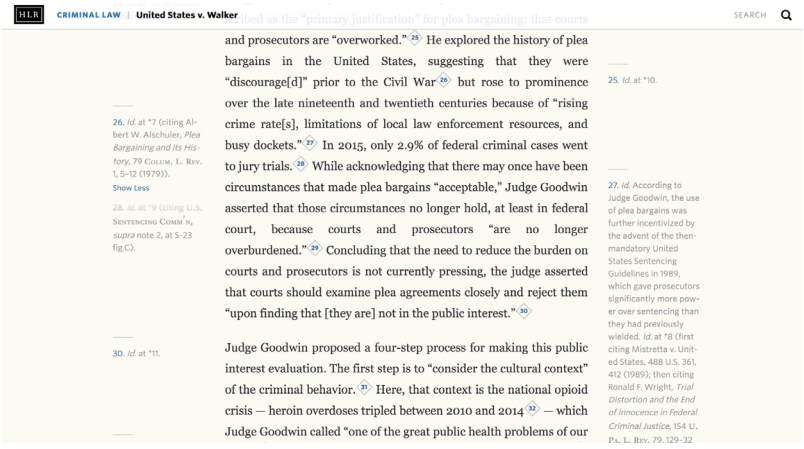
SBB Mobile is a Swiss trip planner app that allows customers to check the schedule of trains and purchase train tickets. On its own, it's a trip planner like every single similar app out there, except for one thing. The app provides a "touch timetable." Customers can define their common destinations and arrange them on a grid. To purchase a ticket from Zurich to Lausanne, for example, it's enough to draw a line on the grid connecting Zurich and Lausanne and then confirm the selection. Booking tickets has never been that fast and easy. That's a great example of making a conventionally complex interaction straightforward, especially for people who commute frequently. Also, it's a unique design signature that nobody else has (yet).



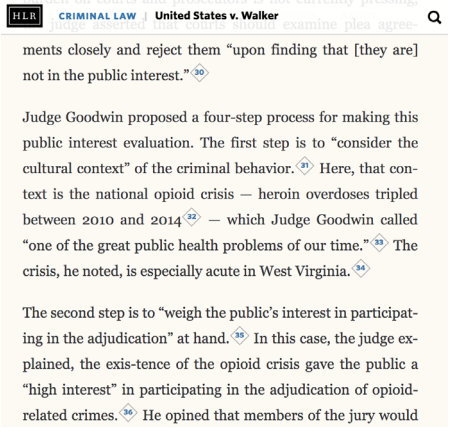
What would it take to provide a remarkable video-playing experience? It might sound as simple as designing a track and a thumb with a few ticks on the track for quick jumps. However, if you study common problems users experience frequently, you'll find one particular issue that stands out. People tend to pause videos and then continue watching later, yet restoring the state of where things were left off is unnecessarily complex in many video player UIs. In fact, you might encounter people writing down the exact time stamp when they paused the video, just to return to it later on another device – but then again, in most UIs it's impossible to jump precisely to a particular second, and most of the time you have to guess and tap the position of a thumb on the track correctly. In the same way, jumping back and forward by 30 seconds or even by a few minutes can be remarkably challenging, especially on mobile, as most interfaces aren't designed around that particular case.

Not only does YouTube provide fully accessible controls for navigation, it also contains a keyframes preview with thumbnails appearing on hover, and navigation via keyboard – and it stores the current state of the video, allowing customers to save a particular time stamp with a unique URL to continue watching later. YouTube also contains many

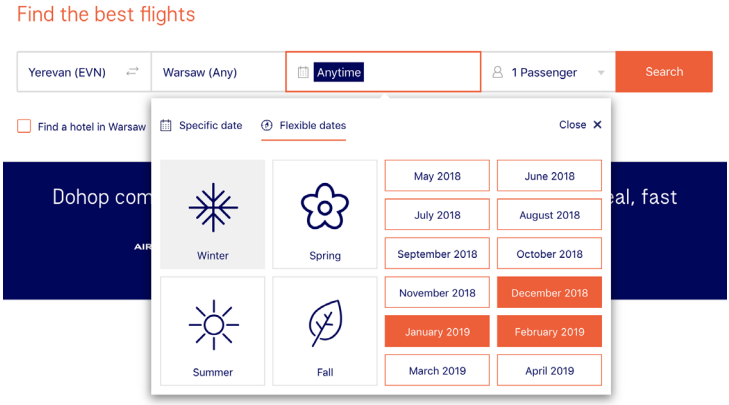
lengthy videos, like documentaries or tutorials, so users can slide up the thumb vertically to adjust the scale of the track and hence jump to the point of interest more precisely. Unfortunately, only a few users know of the feature, and the interaction isn't particularly self-explanatory, but those who *do* know of it, use it frequently. One pain point, solved well.



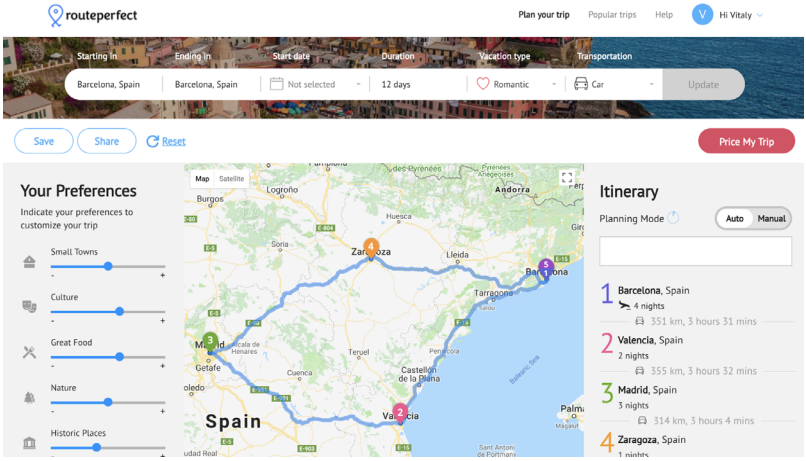
Most academic publications contain dozens of endnotes, footnotes, and references, listed in the order of appearance. If a reader is interested in a particular footnote, they have to jump to the end of the article to read it, and then jump back to continue reading the article. This experience might be a bit too tedious for frequent use, yet it's the default experience we all are accustomed to.



The *Harvard Law Review* solves this problem in a different way. References are always located right next to the point where they are mentioned. Every side note and footnote either appears on the sides on larger screens, or displayed inline via an accordion. Once a user has tapped on the footnote, it expands in its entirety, while the footnote turns into a "close" button. A simple problem solved well.



Imagine you want to book a wonderful vacation with your family, but you haven't picked your dates yet. You have an idea of when you'd like to go, and you have some flexibility regarding the dates for your next vacation. *DoHop* allows its users to choose a flexible date for traveling; for example, particular months of the year, or a particular season, (winter or fall, perhaps). It then suggests dates and a time span with the best price. And if you have a public holiday weekend coming up in a few weeks, and you'd love to make a plan, *RoutePerfect* (see the screenshot on the next page) suggests a route based on your preferences. That's a real problem case solved well. Most traveling websites ask for specific dates for inbound and outbound flights.



Good solutions require time and focus. They require us to *really* understand what pain points users experience first. Users might not be very good at articulating those paint points, so we developed a simple technique that helps us get to the root of the problem.

We ask testers to complete a particular task on a competitor's website, and record their session on video, along with a webcam, using the device that they are used to. It could be as easy as finding an item in a catalog, or checking out in a retail store, or finding a particular section in the navigation. Of course, we observe their behavior and ask questions if something appears to be unusual, but too often many things that happen during the sessions go unnoticed – they are just too difficult to spot right away. That's why we **rewatch recorded user sessions in slow motion**, often slowing down the playback five or six times.

We look for repeated movements and imprecise hits, as well as negative facial expressions and gestures. More specifically, we search for *little moments of despair* – fleeting moments of confusion when movements or gestures don't make any sense: circling around a button or a

link over and over again; focusing on a particular interactive element for far too long; selecting the same text a few times in a row and then continuing to navigate without acting on it. The playback sessions usually happen right after the test, so we still have an opportunity to ask questions and to check our assumptions with the participant. Even a few recordings can provide actionable insights — and they don't require a lot of effort or resources. Should you not have the time to run these tests, it's also a good idea to ask the support team about common complaints.

Once we've identified some issues, we explore solutions that would provide more clarity and ease the interaction, sometimes by designing without any particular visual language in mind. The point is to find an interaction pattern that would be way more superior to the experience customers had on the competitor's sites. We then produce a digital mock-up and invite the same customers to try to solve the same tasks, along with a new group of testers who haven't seen both interfaces yet. We measure the time needed to complete an interaction and ask them to choose which interaction they find more straightforward and useful, and why. Surprisingly, faster interactions aren't necessarily perceived as being faster, and slower interactions aren't necessarily perceived as being slower. Based on that, we iterate and evolve those prototypes. In many ways, those pain points become the heart of our experience that we tackle first and radiate the entire experience out from. That's why sometimes, instead of running a test on a competitor's website, we test our own solutions in the same way.

Good solutions trigger an emotional attachment with or without non-conventional aesthetics or interaction. The more pain points you can address well within your interface, the more likely the difference in experience is to be noticed. Only a few websites make it to customers' browser toolbars, so think about that one pain point and the one solution that would make them do just that.

Exceeding Expectations by Default

Here's another question for you: of all the hotel experiences you ever had, which ones are the most memorable? Think about it for a moment. Think about what made them so special and why they stayed with you all the time. It might have been an extraordinary natural backdrop, or remarkably attentive personnel, or a lavish breakfast buffet. Or something entirely different. In fact, for many of us it could have been a pretty average dormitory as much as an exquisite 5-star chalet in the Swiss alps. The environment matters, but it's not the environment *alone* that matters.

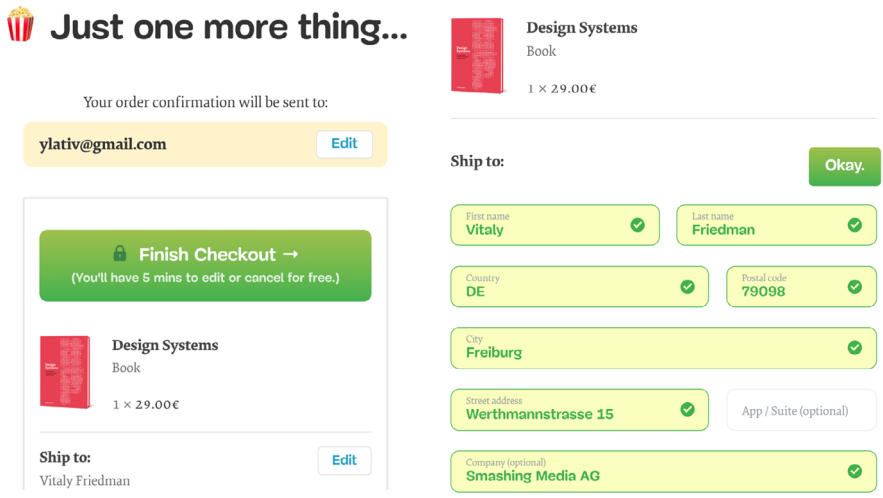
The reason why these experiences are memorable is because they aren't average.[21] In fact, they are the very opposite of average in *some* way, as *something* was exceptional about them. It's not necessarily the hotel itself – it's the timing and the people we happen to spend these experiences with. A good hotel provides a setting that enables wonderful experiences, and so does a good website interface. A *memorable* hotel adds a fine detail to the experience that exceeds our expectations, and it does so without telling us up front. And so do *memorable* websites.

As Brené Brown, a research professor at the University of Houston, so beautifully expressed in her books on empathy, "good design is a function of empathy, while non-empathic design is self-indulgent and self-centered." The key, then, is to empathize with customers both in their negative and positive experiences, rather than pushing your own agenda. To our customers, that extra fine attention to a few little

21 According to Daniel Kahneman's peak-end rule (http://smashed.by/peakend), we judge experiences based on how we felt at its peak (the most intense point) and at its end, rather than the total sum of experiences and whether it was pleasant or unpleasant. That means we can tap into very negative and very positive parts of the experience, and tweak them to create an emotional connection.

details can make all the difference in the world. So we could sprinkle a little bit of human kindness here and there, adding extra value silently, without even mentioning it. That fine detail could be as simple as a custom-designed profile illustration, based on the photo the customer has uploaded. It could be a handwritten thank-you note, signed by the entire team and sent via good ol' snail mail. It could also be an unexpectedly straightforward resolution of a problem after a mistake has been made.

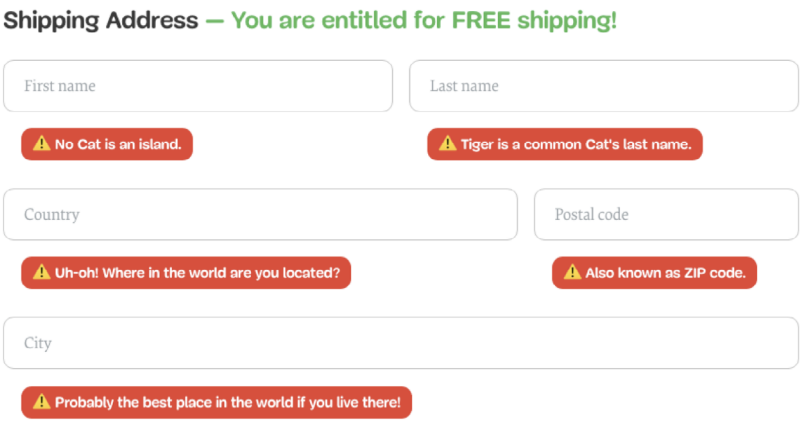
In an eCommerce setting, it could mean the ability to modify or cancel a finished order within five mins after the successful checkout. It would both help a customer avoid a time-consuming interaction with the support team as they can edit an order without any extra charges, even if they realize after checking out that something was wrong with the order details.



A mock-up we're currently exploring in Smashing Magazine's checkout to allow inline editing of data on the review step and editing of the order within 5 minutes after the purchase.

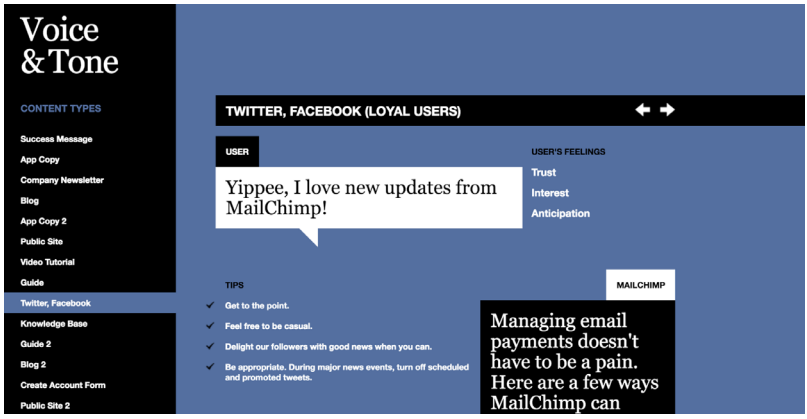
In the same way, an interface could suggest to a signed-in customer to use a use a coupon code saved in their dashboard as it's about to expire soon, or inform them about a similar – and hence potentially duplicate – booking made a while back. The personality of the brand shines best in those little moments when it helps customers prevent mistakes. By acting on behalf of the experience rather than business every now and again, the interface makes the customer feel genuinely valued, respected, and helped, and that works much better than any ingenious interface copy ever would.

One way of preventing mistakes is by writing adaptive and helpful error messages. That's one of the most obvious points of frustration for customers, and it's remarkable how little effort is put into recovery experience, often falling back to generic and abstract messages. Admittedly, these messages don't necessarily cost a sale but they can damage the long-term perception of the brand. People who experience unrecoverable issues during one of the key interactions on a site tend to not use it in the future at all as they expect the issue to creep out in other interactions too.



Error messages deserve slightly more credit than they are usually given. They appear at the point where the customer's progress is blocked, and when they have to slow down and pay full attention to resolve a problem. We can use the situation to our advantage to infuse a bit of personality into the experience. Every single time an interface fails to meet expectations, it's an opportunity to create a memorable impact in the process of speedy recovery. If we manage to turn an annoyance into the feeling of being valued or understood, we might be heading off on the right track.

One of the very first things I started working on when we embarked on the redesign was filling elaborate spreadsheets with alternate wordings for our checkout error messages. It wasn't done with the intention to A/B test the "best performing" error message; it was done primarily to discover better expressions of the personality through the interface. On their own, error messages don't really make sense, but they fit well within the story being told throughout the site. Once an error has occurred, we try to use both adaptive and playful copywriting to address the issue while also raise the occasional smile.



Voice and tone are the main pillars of a personality. MailChimp has built a dedicated voice and tone style guide (http://smashed.by/voicetone) to align designers, customer support, and everybody else in the way they communicate to customers.

Seek critical pain points that customers often experience on the site by looking into customer support logs and user interviews, and design these experiences with extra care and attention. It goes without saying that a quirky personality won't help much if the customer isn't able to solve a problem, so take care of the basics first. Ultimately, it doesn't take that much effort to turn negative experiences into positive ones – it's just a matter of having it on your to-do list when designing an interface.

The Two Sides of Personality

As much as we love sharing our experiences and showing our better side to people around us, we can't stand that one person spending the entire evening talking about themselves. In our interfaces, every time we add yet another parallax transition or a slow bouncy animation to people who have seen it a dozen times already, we are essentially letting the interface highlight its fanciness without helping the user along the way. Eventually, after a few visits, all those whistles and bells that achieve a strong first impact become annoying as they add too much friction.

Nobody loves self-centered characters, and so a website shouldn't be self-centered either. The design signature should never take the leading role in the user experience as it's never the main reason why people access the website. It should be humble and remain in the shadows, noticeable but not obstructing the smooth flow frequent visitors have got used to.

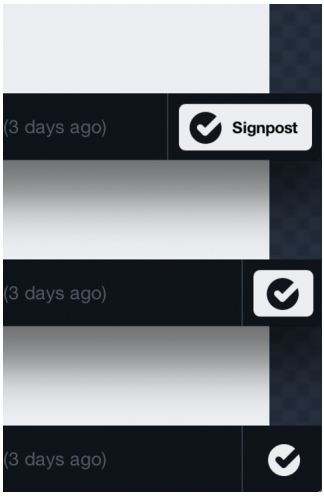
In her talk on *Designing Meaningful Animations*,[22] Val Head, a fantastic designer from Pittsburgh, suggested using prominent animations

22 http://smashed.by/designanimations

very sparingly, as they should be reserved for very special occasions, while subtle micro-animations could accompany the user all along the way. Val suggests using animation only for key compositions of your story, like sending a marketing campaign, or favoriting an item, or seeing a successful purchasing page, while everything else should remain calm and normal. With this idea in mind we could think of designing our interfaces with two kinds of interactions: the prominent "showroom" ones, used rarely; and subtle "workhorse" ones, used frequently.

Reserve special visual treatments and interactions for special occasions, but also embed subtle treatments used consistently across the entire site. Twitter, for example, uses an elaborate animation when a user "hearts" a tweet. Facebook displays a confetti animation when you congratulate a friend on their birthday or a wedding. In Smashing's case, we use vibrant colors and cat illustrations as our showroom signature, while tilting, hover-animations, and shadows beneath them make up our workhorse signature.

We are used to the idea of our designs adjusting to the viewport or network conditions, but it might be worth looking into adjusting the design based on the frequency of usage, too. This technique is called *progressive reduction*,[23] a dynamic simplification of an interface as users become familiar with it. You identify the main features of an interface, and assign levels to them. Then, track your user's usage by monitoring the frequency of use within a certain time period and create proficiency profiles for the user. Based on the current level, you cam adjust these UI components to reduce hand-holding.



23 http://smashed.by/progreduct

As Allan Grinshtein pointed out,[24] a user's proficiency in a given product decays over time without usage (also known as *experience decay*), so if a user's frequency of use and usage volume have gone down, then *their* interface should regress a level or two, depending on how far down their numbers have dropped. This automatic regression is necessary to balance progression; without it, you lose the ability to fully respond to dynamic changes in user behavior.

The more often customers visit the site, the less likely they want to be confronted with anything that would slow them down. Therefore, it might be a good idea to slowly fade out showroom signatures with growing frequency of use, perhaps removing parallax-effects or speeding up transitions for returning users. In the end, it's all about the choreography: don't be that person at a dinner party filling the room with an extensive story of their life.

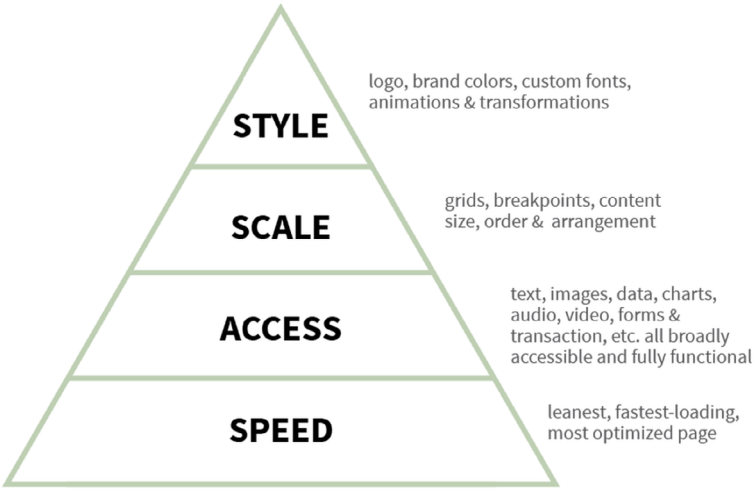
The Signature at the Heart of the Design

The design process is a mythical creature. Everybody somehow manages to come up with their own workflow, tooling, and processes, yet it's very rare that anybody is *really* satisfied with it. When it comes to infusing personality into the design, when and where would be the right point to include it in the design process?

In one of her talks from 2014,[25] Patty Toland, a senior UX designer from Filament Group in Boston mentioned the hierarchy of priorities the team uses when designing and building responsive experiences. The main goal of the process is to create the "leanest, fastest-loading, most optimized page." The main foundation is and has always been a

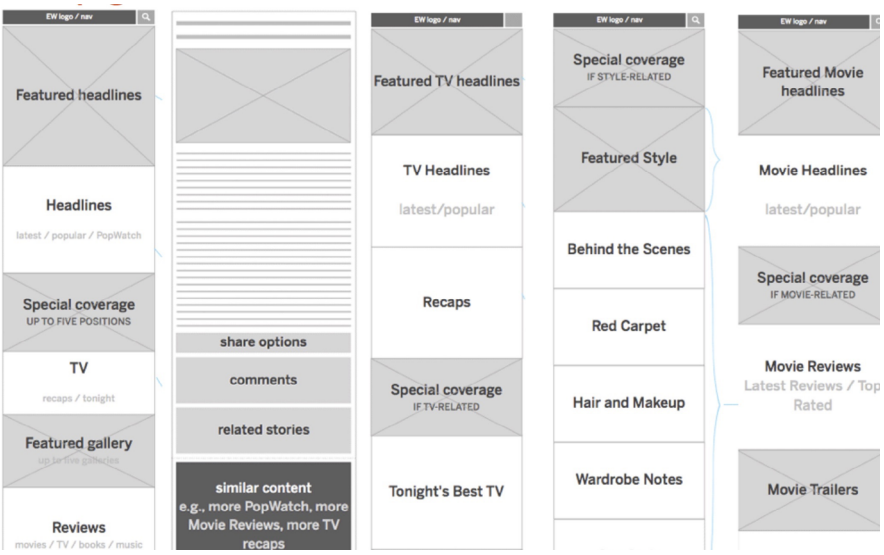
24 http://smashed.by/designanimations
25 http://smashed.by/pattystalk

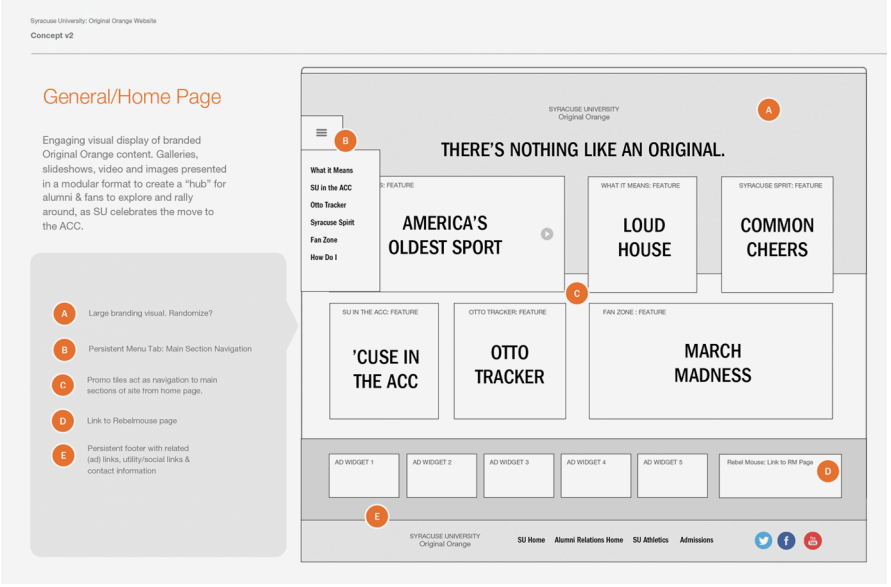
fully accessible experience, in which text, images, data, charts, audio, video, forms and so on are all broadly accessible and function fully in their default form. Applied to the context of the design process, it means meaningful markup and properly defined relationships between components.



Patty Toland, Filament Group, "Design Consistency for the Responsive Web." (*http://smashed.by/pattystalk*)

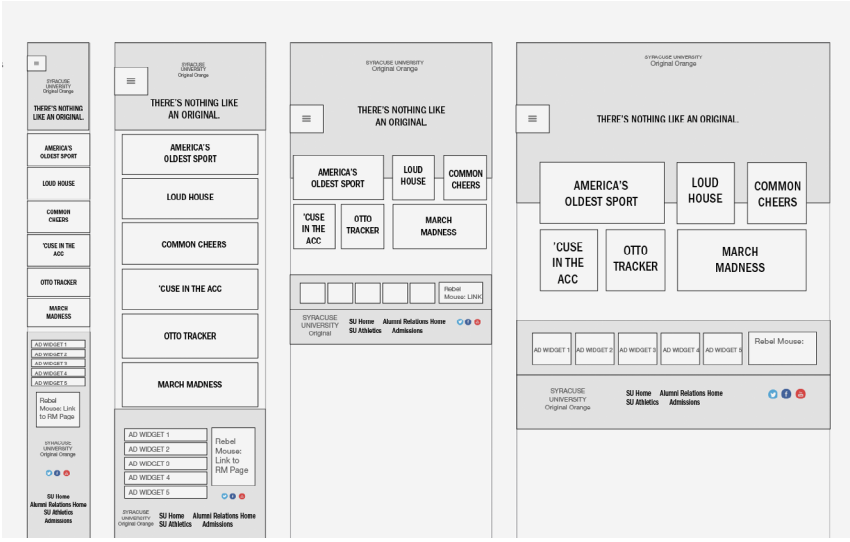
With accessible components ready to be served, the next step is taking care of the scale of the design. That's where the decisions about grid, content size, order, and arrangement, as well as breakpoints, come into play. Often the proportions will be defined using *content wireframes*: low-fidelity mock-ups with gray boxes; the height of each box in proportion to others defines its weight in the layout. Sometimes we add notes about the personality across the content blocks, and then reflect them when it comes to visual design.





Content wireframes in action. At the top the wireframe of Techcrunch redesign (http://smashed.by/techcrunch) and below the wireframe of the Original Orange redesign by Adjacent (http://smashed.by/adjacent). You can follow the progress of the latter in the screenshots on the next pages, too.

With low-fidelity prototypes in place, the next step for the design is to gain style, with logo, brand colors, custom fonts, transitions, and animations added to the mix. Sometimes this hierarchy will be perfectly mapped in the order we write React components and CSS properties with Sass. Even *BEM naming* for classes will happen in that order as well. The prototypes will gain abstract utility classes first, and more elaborate relationships will be reflected through more specific class names throughout the process. The process establishes a clear separation of responsibilities for modules.



This process seems plausible but it raises a very critical question: *what pages to design and prototype first*? When we start designing, we design the *heart* of the experience first: the most critical and impactful part of the experience. More specifically, we try to capture the very essence of the experience by exploring key interactions, then break it down into reusable components, and then radiate outwards from that essence. For an online magazine, it would be reading experience and typography first. For a landing page, it would be the pricing plans and a feature comparison first.

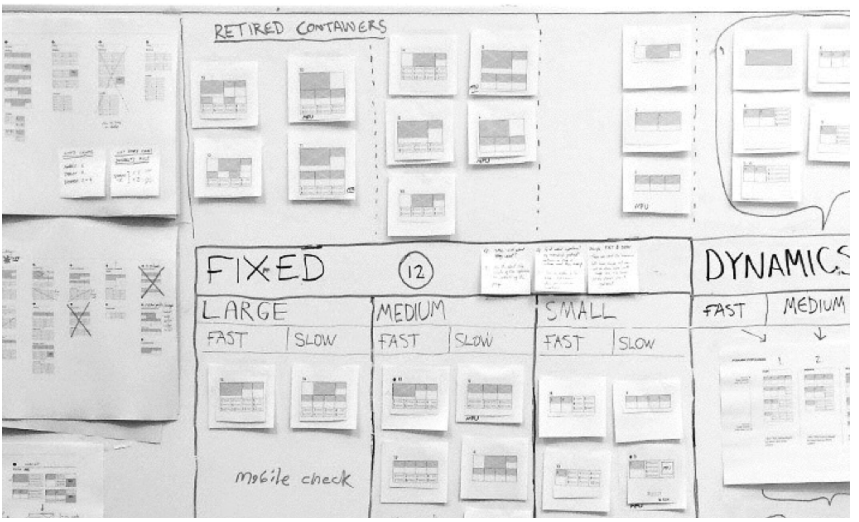
For an ecommerce site it means looking into the components that would make up an extraordinary relevant and useful product page first. That means large image thumbnails, concise copywriting, transparent pricing, exposed ratings and testimonials, psychological anchors, and call-to-action buttons. The visual design decisions made there are then translated to other parts of the interface, specifically forms and labels and error messages in the checkout.

Only then, eventually, we reach the category pages and the FAQ pages living on the far outer edges of the experience spectrum. Somewhere in between we explore the front page, but usually we design it late rather than early in the process – at the point when we've established a strong identity already, so we use the front page to amplify and explore it prominently, potentially with a bold design that would exhibit the main qualities of the personality.

Remember overarching connections mentioned earlier in the chapter? A critical part of the design process is to connect modules, so they don't appear as standalone solutions when put together in the interface. When we have enough modules to build the first prototype, we jump into the browser and build mobile-first. It's in this process that we finally decide on the grid and the layout and the structure, and implement the connections between modules. In fact, for us, the signature is that magical bond that ties things together.

That's why we start thinking about the signature of the design when we start designing the heart of the experience, and sometimes even before that. Spreadsheets exploring error messages, visual experiments around shapes and colors and type, as well as user interviews help us get there. Eventually, decisions made for the first prototype can be reused for other pages, yet sometimes we need to run the process from the start again – as some pages clearly will be one-offs, such as the landing page or a front page. They will still exhibit relationships to everything else because they are designed and built using the personality traits that have been solidified by this point.

It's these relationships that would then lay the main foundation of a design system, along with components and examples of the interface in use. Too often style guides show a component in isolation, along with Sass class names and a code snippet, without including how that component should appear and behave in relation to other modules on the page. Contextual examples matter both for designers and developers, and they give a good reason to both visit and keep the design system up to date in the long-term.



A storyboard with components. Each component also has a speed and level of dynamics attached to them. Image courtesy: Andrew Clarke.

We often create user journey maps to understand the flow users go through to accomplish their tasks, and with personality traits in mind, we could even complement them with storyboards, adding some personality highlights at different points of user experience. Besides, in the context of design systems, we could explore not only components in isolation, but also how the design language can use components to slow down or speed up the experience, or provide greater or lesser impact, as well as dynamic and static layout compositions – very much like we do with showroom and workhorse interactions.

You could even print them out and put them as magnets on a storyboard, just like Sipgate did in the earlier example, so designers can freely move them around and thus explore ways of combining predictable components in unpredictable ways. That's what Andrew Clarke does when embedding art direction and storytelling in his designs – very much like comic strip designers arrange the frames according to narrative dynamics and impact when laying out a comic story.

The design signature lies at the very heart of the design. It's a strand that connects the components in the interface, and it's what helps designers stay on track when maintaining or evolving a design language. Deciding on the personality traits first helps drive the direction of the design, and can be just a good enough constraint to dissolve initial intentions and goals into tangible, distinguishable attributes that could eventually become the heart of the experience.

Wrapping Up

As much as we could get seduced by the charm of a website, in the end, the main purpose of it shouldn't be self-indulgence. Expressions of the personality of the site enable emotional connections with customers and visitors, and because they are human by their nature, they outline a path to authentic, honest, and respectful interfaces. It's up to us to figure out how to shape that path and the outcome ahead of us.

Now, it might not be for everybody, and perhaps not every site needs a personality in the first place, or perhaps it should be subtle and express itself in little nuances here and there. I hope that in either of these cases, once flipping over the last page of this book, you'll have a good enough tool belt of ideas and techniques to create unique and humane experiences – experiences people could fall in love with.

I'd like to express my gratitude to Jen Simmons, Rachel Andrew, Andrew Clarke, Dan Mall, Espen Brunborg, and Elliot Jay Stocks for inspiring work, contributions, and help in discussing the idea of art direction on the web, and making the web more diverse and experimental. I'd also like to thank Marko Dugonjic, Alberta Soranzo, Sashka Maximova, Lilia Zinchenko, Stefan Bucher, Benoit Henry, Nils Mielke, Thord D. Hedengren, and Bertrand Lirette for reviewing the chapter, as well as our fantastic community, which has shared techniques and lessons learned from its work for everybody to use.

About The Author

Vitaly loves beautiful content and complex challenges, and does not give up easily. He co-founded *Smashing Magazine* back in September 2006 and since then spends pretty much every day trying to make it better, faster and more useful. He runs responsive design training and workshops and loves solving complex UX, performance and front-end problems in large and small companies.