Form Design Patterns

A practical guide to designing and coding simple and inclusive forms for the web

By Adam Silver

In memory of my
beautiful and amazing
mum, Libby Silver.

Published 2018 by Smashing Media AG, Freiburg, Germany.
All rights reserved.
ISBN: 978-3-945749-73-9

Cover design: Espen Brunborg
Copyediting: Owen Gregory
Interior layout: Markus Seyfferth
eBook production: Cosima Mielke
Typefaces: Elena by Nicole Dotin, Mija by Miguel Hernández.

Form Design Patterns was written by Adam Silver
and reviewed by Heydon Pickering.

Please send errors to: errata@smashingmagazine.com

Table Of Contents

1 A Registration Form 18

2 A Checkout Form 68

3 A Flight Booking Form 123

4 A Login Form 200

5 An Inbox 220

6 A Search Form 262

7 A Filter Form 278

8 An Upload Form 314

9 An Expense Form 350

10 A Really Long and Complicated Form 372

About The Author

Adam Silver is an interaction designer with over 15 years experience working on the web for a range of companies including Tesco, BBC, Just Eat, Financial Times, the Department for Work and Pensions and many others.

He's particularly interested in inclusive design and design systems and writes about this on his blog and popular design publications such as A List Apart. This isn't his first book either: he previously wrote 'Maintainable CSS' which is about crafting maintainable interfaces with CSS.

About The Reviewer

Heydon Pickering is a freelance web accessibility consultant, interface designer and writer living in the UK. He is an author and editor for Smashing Magazine, and he also works with leading web accessibility specialists, The Paciello Group, focusing on inclusive design systems.

Heydon has written multiple books on the subject of accessibility and inclusive design, including "Apps For All," "Inclusive Design Patterns," and "Inclusive Components."

Acknowledgements

I'd like to thank a number of people who helped me write this book:

Graham Veal for helping to set up the foundations of the design system that accompanies the book. His Node.js and Heroku expertise saved me many hours of pain. *John Oates* for reviewing some of my early drafts, which stopped some of my bad writing habits early in their tracks. *Steven Proctor* for reviewing the section on error message design. If you want to know how to write a good error message, Steven is your man. *Mark Jenkins*, friend and unofficial mentor, for encouraging me to start writing. *Owen Gregory* for editing my words to be simpler, plainer, more coherent, and consistent. *Espen Brunborg* for the book cover that complements the content and message of the book so well. *Markus Seyfferth* for letting me write this for Smashing and producing such a beautifully crafted book. *Heydon Pickering* for many things: championing the book to Smashing; inspiring the book's approach (by problem, not principle); being my technical editor; and if that weren't enough, for writing the brilliant foreword. This book would simply not exist without him.

Finally, my wife Jen, for her patience in letting me spend so much time writing the book while she looked after our two little ones.

Foreword

Every so often, someone will point out that I use blackish text on whitish backgrounds for almost all my page layouts. And the only comeback I can think of is that the same approach has worked for hundreds of billions of publications over the course of hundreds of years. You know, that old chestnut.

Making a habit of flouting convention will garner you attention, spark controversy and earnest debate — even earn you awards. But it will also confound and alienate your readers and users — the people your work is really meant for. That is abject failure.

Paradoxically, in a world saturated with rule breaking and reinvention, a reverence for the straightforward, familiar, and simple becomes radical. And it's a welcome revolution, because interfaces that are obvious are also inclusive. It's not a bad thing to be on the nose.

Let me give you an example. Imagine my excitement when reading this book, to find Adam recommending that form labels should appear above their respective inputs. Not off to the side at an angle, not inside the input where the actual user input should go, and certainly not as some

absurd animated combination of different positions and orientations at different times.

That's actually radical, and really refreshing to read. Because most designers will do anything but the expected. Then I have to tell them off on behalf of the users they're forcing to decipher their interface. Nobody has time for that.

Don't get me wrong: I'm not saying there's nothing new in this book. I learned plenty. It's just that my reaction was never "OK, I guess that's one way of doing it LOL," and always "Damn, that's it — I should have been doing this all along." And it turns out that when you combine standard elements and simple concepts, even daunting components like the airplane seat chooser can be accessible, logical, and lightweight.

To me, this book is about simple solutions to would-be complex problems. As such, it's not just about forms. But if you can make forms easy and pleasurable to use (forms!), then most everything else will be a cinch.

— *Heydon Pickering*

Introduction

I remember my first foray into forms. At the turn of the century, web design was one of the modules on the information communication and technology course I took at sixth form college. My learning mostly consisted of cutting and pasting snippets of HTML, CSS, and Javascript. Yes, I came from the view-source school of web design and development.

My obsession with forms started when — like with any other HTML element — I tried to cut and paste it. Despite rendering OK, when I submitted it nothing happened. Fast forward seventeen years and here I am writing a book about form design patterns.

Why Forms?

Every meaningful interaction that happens on the web is achieved by a form of some sort. Without forms, the web merely becomes a passive experience — just a way to consume content.

Forms allow users to create, update and delete things. Whether it's communicating through email, buying a product, online banking, or working on a fully-fledged administrative digital service, forms are always front and center.
At first glance, forms are rather easy to grasp. In less than an hour, you'll have text boxes, radio buttons and select boxes

on the page. But their low barrier to entry turns them into what Heydon Pickering refers to as a "10,000-volt electro-magnet for attracting usability problems."[1]

This is a big part of why I'm writing this book. Typically, these usability problems come up again and again.

Why Patterns?

Design patterns serve as guidance and solutions to people solving similar problems over and over. The reason for design patterns is twofold.

First, instead of solving the same problem from scratch every time, we can instead use previously designed, available, recognized, and well-researched solutions. This saves a lot of time. And we can use that time to solve newer and perhaps bigger problems.

Second, by solving the same problem in the same way, users have a consistent and more coherent experience. The service, app, or whatever it is, becomes familiar. Familiar interfaces require less effort to operate. Think about it: every time you encounter a door, you just know that it can be opened, closed, and sometimes locked.

1 http://smashed.by/idp

Using design patterns for digital experiences or, more specifically, forms, makes sense too. By the end of the book, you'll have many patterns you can use in your own interface immediately.

Why These Forms?

I first based this book on 50 principles. Originally, each principle would become a short chapter. So there was a chapter called "Always Use a Label" and another called "Placeholders Are Problematic."

There are a few problems with this approach to design. First, rules can be broken — occasionally. Second, evaluating problems by principle is constraining. Many go together: when talking about screen readers, for example, it often makes sense to discuss keyboard users. And sometimes you have to make trade-offs.

Instead of centering the book on principles, I decided to revolve it around real problems. That way we can solve them as we do at work. The result is ten specific problems to solve, each represented as a chapter. The chapters are specific, but most of the patterns are reusable and transferable to many other forms you might be designing. After all, a pattern should be unique, but reusable across projects and organizations.

Here's a chapter rundown.

1. A REGISTRATION FORM

We'll start with a basic registration form and take a look at the foundational qualities of a well-designed form and how to think about them. By applying something called a *question protocol*, we'll look at how to reduce friction without even touching the interface. Then we'll look at some crucial patterns, including validation, that we'll want to use for every form.

2. A CHECKOUT FORM

The *one thing per page* design pattern is a cornerstone of creating well-designed forms. We'll look at why that is before applying it to a checkout flow. After that, we'll consider flow and order with a view to breaking down each step of the checkout flow. Then we'll look at several input types and how they affect the user experience on mobile and desktop browsers, all the while looking at ways to help both first-time and returning customers order quickly and simply.

3. A FLIGHT BOOKING FORM

We'll dive into the world of progressively enhanced, custom form components using ARIA. We'll do this by exploring the best way to let users select destinations, pick dates, add passengers, and choose seats. We'll analyze native form con-

trols at length, and look at breaking away from convention when it becomes necessary.

4. A LOGIN FORM

We'll look at the ubiquitous login form. Despite its simple appearance, there's a bunch of usability failures that so many sites suffer from. Social media login hasn't necessarily helped matters so we'll cover that too.

5. AN INBOX

We'll design ways to manage and action email in bulk, our first look at administrative interfaces. As such, this comes with its own set of challenges and patterns, including a responsive ARIA-described action menu, multiple selection, and same-page messaging.

6. A SEARCH FORM

We'll create a responsive search form that is readily available to users on all pages, and we'll also consider the importance of the search mechanism that powers it. Together, they can make search discoverable, simple, and useful.

7. A FILTER FORM

Users often need to filter a large set of unwieldy search results. Without a well-designed filter, users are bound to

give up. Filters pose a number of interesting and unique design problems that may force us to challenge best practice to give users a better experience.

8. AN UPLOAD FORM

Many services, like photo sharing, messaging, and many back-office applications, let users upload images and documents. We'll study the file input and how we can use it to upload multiple files at once. Then we'll look at the intricacies of a drag-and-drop, Ajax-enhanced interface that is inclusive of keyboard and screen reader users.

9. AN EXPENSE FORM

We'll investigate the special problem of needing to create and add lots of expenses (or anything else) into a system. This is really an excuse to cover the *add another* pattern, which is often useful in administrative interfaces.

10. A REALLY LONG AND COMPLICATED FORM

Some forms are very long and take hours to complete. We'll look at some of the patterns we can use to make long forms easier to manage.

What About Principles?

While I've moved away from principle-oriented chapters, there is still an important place for principles in this book. Without principles, it's hard to know whether what we've designed is objectively good.

But where should our principles come from? We can either steal other people's, or we can define our own. But before we get to that, let's see how we get to certain principles in the first place.

Our principles normally stem from a belief system. We believe something should be a certain way, typically for good reason. A banal example, perhaps, would be showing up on time for meetings: being late (at least, deliberately) reveals a lack of respect for the other attendees, and the meeting's purpose. Without a good reason for our belief system, principles crumble under scrutiny.

This book is about designing forms for the web. It would be remiss of me, then, to ignore the essence of the web itself. The power of the web is one of reach and accessibility. Anyone with a browser and an internet connection gets to use it. The principles in this book need to align with this notion — to uphold its inherent qualities.

Frank Chimero talks about this at length in "The Web's Grain," one of my favorite articles on design.[2] The main point of the article encourages us not to aim to tackle complexity, but do our very best to avoid it in the first place, mostly by "going with the grain" and embracing the web's constraints.

It turns out that not only is this the easiest and cheapest way to design something, but also that users have a better time operating these simpler interfaces in the end. It's the content and functionality users want anyway.

Whatever we build, in the end, is about users. I don't want to leave a single person behind if I can help it. The web is for everyone. I can't think of a better set of principles than the inclusive design principles from the Paciello Group.[3]

These principles are about good design — and good design is inclusive.

1. **Provide a comparable experience**. Ensure your interface provides a comparable experience for all so people can accomplish tasks in a way that suits their needs without undermining the quality of the content.

2 http://smashed.by/websgrain
3 http://smashed.by/idprinciples

2. **Consider situation**. People use your interface in different situations. Make sure your interface delivers a valuable experience to people regardless of their circumstances.
3. **Be consistent**. Use familiar conventions and apply them consistently.
4. **Give control**. Ensure people are in control. People should be able to access and interact with content in their preferred way.
5. **Offer choice**. Consider providing different ways for people to complete tasks, especially those that are complex or non-standard.
6. **Prioritize content**. Help users focus on core tasks, features, and information by prioritising them within the content and layout.
7. **Add value**. Consider the value of features and how they improve the experience for different users.

We'll refer back to these principles throughout the book, pointing out where something works or not. These principles should, at least indirectly, hold us to account throughout the design process.

By looking at common form patterns through the lens of inclusivity, this book will help you learn how to apply and reuse conventions that help users complete the task, regardless of how they choose or need to use your service.

A Registration Form

Let's start with a registration form. Most companies want long-term relationships with their users. To do that they need users to sign up. And to do *that*, they need to give users value in return. Nobody wants to sign up to your service — they just want to access whatever it is you offer, or the promise of a faster experience next time they visit.

Despite the registration form's basic appearance, there are many things to consider: the primitive elements that make up a form (labels, buttons, and inputs), ways to reduce effort (even on small forms like this), all the way through to form validation.

In choosing such a simple form, we can zoom in on the foundational qualities found in well-designed forms.

How It Might Look

The form is made up of four fields and a submit button. Each field is made up of a control (the input) and its associated label.

Register

First name

Last name

Email address

Password

Must be at least 8 characters

Register

A registration form with four fields: first name, last name, email address, and password.

Here’s the HTML:

```
<form>
  <label for="firstName">First name</label>
  <input type="text" id="firstName" name="firstName">
  <label for="lastName">Last name</label>
  <input type="text" id="lastName" name="lastName">
  <label for="email">Email address</label>
  <input type="email" id="email" name="email">
  <label for="password">Create password</label>
  <input type="password" id="password" name="password"
         placeholder="Must be at least 8 characters">
  <input type="submit" value="Register">
</form>
```

Labels are where our discussion begins.

Labels

In *Accessibility For Everyone*, Laura Kalbag sets out four broad parameters that improve the user experience for everyone:[1]

- Visual: make it easy to see.
- Auditory: make it easy to hear.
- Motor: make it easy to interact with.
- Cognitive: make it easy to understand.

By looking at labels from each of these standpoints, we can see just how important labels are. Sighted users can read them, visually-impaired users can hear them by using a screen reader, and motor-impaired users can more easily set focus to the field thanks to the larger hit area. That's because clicking a label sets focus to the associated form element.

First name

The label increases the hit area of the field.

1 http://smashed.by/a11y4all

For these reasons, every control that accepts input should have an auxiliary `<label>`. Submit buttons don't accept input, so they don't need an auxiliary label — the `value` attribute, which renders the text inside the button, acts as the accessible label.

To connect an input to a label, the input's `id` and label's `for` attribute should match and be unique to the page. In the case of the email field, the value is "email":

```
<label for="email">Email address</label>
<input id="email">
```

Failing to include a label means ignoring the needs of many users, including those with physical and cognitive impairments. By focusing on the recognized barriers to people with disabilities, we can make our forms easier and more robust for everyone.

For example, a larger hit area is crucial for motor-impaired users, but is easier to hit for those without impairments too.

Placeholders

The `placeholder` attribute is intended to store a hint. It gives users extra guidance when filling out a field — particularly useful for fields that have complex rules such as a password field.

As placeholder text is not a real value, it's grayed out so that it can be differentiated from user-entered values.

The placeholder's low-contrast, gray text is hard to read.

Unlike labels, hints are optional and shouldn't be used as a matter of course. Just because the `placeholder` attribute exists doesn't mean we have to use it. You don't need a placeholder of "Enter your first name" when the label is "First name" — that's needless duplication.

The label and placeholder text have similar content, making the placeholder unnecessary.

Placeholders are appealing because of their minimal, space-saving aesthetic. This is because placeholder text is placed *inside* the field. But this is a problematic way to give users a hint.

First, they disappear when the user types. Disappearing text is hard to remember, which can cause errors if, for example, the user forgets to satisfy one of the password rules. Users often mistake placeholder text for a value, causing the field to be skipped, which again would cause errors later on.[2] Gray-on-white text lacks sufficient contrast, making it generally hard-to-read.[3] And to top it off, some browsers don't support placeholders, some screen readers don't announce them, and long hint text may get cut off.

Password

Must contain 8+ characters with at le

The placeholder text is cut off as it's wider than the text box.

That's a lot of problems for what is essentially just text. All content, especially a form hint, shouldn't be considered as nice to have. So instead of using placeholders, it's better to position hint text above the control like this:

Password

Must contain 8+ characters with at least 1 number and 1 uppercase letter

Hint text placed above the text box instead of placeholder text inside it.

2 http://smashed.by/nohints
3 http://smashed.by/unreadableweb

```
<div class="field">
  <label for="password">
    <span class="field-label">Password</span>
    <span class="field-hint">Must contain 8+ characters
    with at least 1 number and 1 uppercase letter.</span>
  </label>
  <input type="password" id="password" name="password">
</div>
```

The hint is placed within the label and inside a `<span>` so it can be styled differently. By placing it inside the label it will be read out by screen readers, and further enlarges the hit area.

As with most things in design, this isn't the only way to achieve this functionality. We could use ARIA attributes to associate the hint with the input:

```
<div class="field">
  <label for="password">Password</label>
  <p class="field-hint" id="passwordhint">Must contain 8+
  characters with at least 1 number and 1 uppercase letter.</p>
  <input type="password" id="password" name="password"
  aria-describedby="passwordhint">
</div>
```

The `aria-describedby` attribute is used to connect the hint by its `id` — just like the `for` attribute for labels, but in reverse. It's appended to the control's label and read out after a short pause. In this example, "password [pause] must

contain eight plus characters with at least one number and one uppercase letter.”

There are other differences too. First, clicking the hint (a `<p>` in this case) won’t focus the control, which reduces the hit area. Second, despite ARIA’s growing support, it’s never going to be as well supported as native elements. In this particular case, Internet Explorer 11 doesn’t support `aria-describedby`.[4] This is why the first rule of ARIA is not to use ARIA:[5]

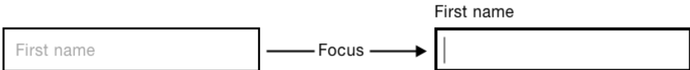
*If you can use a native HTML element or attribute with the semantics and behaviour you require **already built in**, instead of re-purposing an element and adding an ARIA role, state or property to make it accessible, **then do so**.*

Float Labels

The float label pattern by Matt Smith is a technique that uses the label as a placeholder.[6] The label starts *inside* the control, but floats above the control as the user types, hence the name. This technique is often lauded for its quirky, minimalist, and space-saving qualities.

---

4 http://smashed.by/arialabelinput
5 http://smashed.by/firstrule
6 http://smashed.by/floatlabel



The float label pattern. On the left, an unfocused text field shows the label inside; on the right, when the text field receives focus, the label moves above the field.

Unfortunately, there are several problems with this approach. First, there is no space for a hint because the label and hint are one and the same. Second, they're hard to read, due to their poor contrast and small text, as they're typically designed. (Lower contrast is necessary so that users have a chance to differentiate between a real value and a placeholder.) Third, like placeholders, they may be mistaken for a value and could get cropped.

And float labels don't actually save space. The label needs space to move into in the first place. Even if they did save space, that's hardly a good reason to diminish the usability of forms.

Seems like a lot of effort when you could simply put labels above inputs & get all the benefits/none of the issues.[7]

– Luke Wroblewski on float labels

7 http://smashed.by/luketweet

Quirky and minimalist interfaces don't make users feel awesome — obvious, inclusive, and robust interfaces do. Artificially reducing the height of forms like this is both uncompelling and problematic.

Instead, you should prioritize making room for an ever-present, readily available label (and hint if necessary) at the start of the design process. This way you won't have to squeeze content into a small space.

We'll be discussing several, less artificial techniques to reduce the size of forms shortly.

The Question Protocol

One powerful and *natural* way to reduce the size of a form is to use a question protocol.[8] It helps ensure you know why you are asking every question or including a form field.

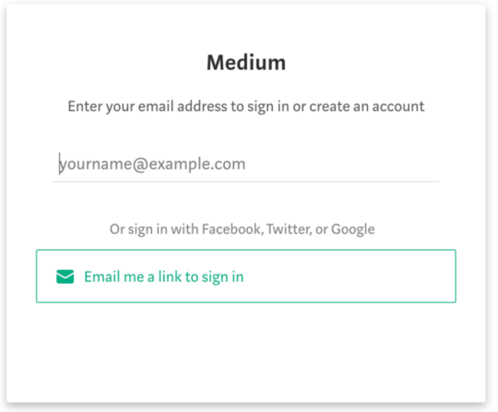
Does the registration form need to collect first name, last name, email address and password? Are there better or alternative ways to ask for this information that simplify the experience?

8 http://smashed.by/questionprotocol

In all likelihood, you don't need to ask for the user's first and last name for them to register. If you need that information later, for whatever reason, ask for it then. By removing these fields, we can halve the size of the form. All without resorting to novel and problematic patterns.

NO PASSWORD SIGN-IN

One way to avoid asking users for a password is to use the *no password sign-in* pattern. It works by making use of the security of email (which already needs a password). Users enter only their email address, and the service sends a special link to their inbox. Following it logs the user into the service immediately.



Medium's passwordless sign-in screen.

Not only does this reduce the size of the form to just one field, but it also saves users having to remember another password. While this simplifies the form in isolation, in other ways it adds some extra complexity for the user.

First, users might be less familiar with this approach, and many people are worried about online security. Second, having to move away from the app to your email account is long-winded, especially for users who know their password, or use a password manager.

It's not that one technique is always better than the other. It's that a question protocol urges us to think about this as part of the design process. Otherwise, you'd mindlessly add a password field on the form and be done with it.

PASSPHRASES

Passwords are generally short, hard to remember, and easy to crack. Users often have to create a password of more than eight characters, made up of at least one uppercase and one lowercase letter, and a number. This micro-interaction is hardly ideal.

Sorry but your password must contain an uppercase letter, a number, a haiku, a gang sign, a hieroglyph, and the blood of a virgin.

— Anonymous internet meme

Instead of a password, we could ask users for a passphrase.[9] A passphrase is a series of words such as "monkeysinmygarden" (sorry, that's the first thing that comes to mind). They are generally easier to remember than passwords, and they are more secure owing to their length — passphrases must be at least 16 characters long.

The downside is that passphrases are less commonly used and, therefore, unfamiliar. This may cause anxiety for users who are already worried about online security.

Whether it's the no password sign-in pattern or passphrases, we should only move away from convention once we've conducted thorough and diverse user research. You don't want to exchange one set of problems for another unknowingly.

Field Styling

The way you style your form components will, at least in part, be determined by your product or company's brand. Still, label position and focus styles are important considerations.

9 http://smashed.by/userfriendlypw

LABEL POSITION

Matteo Penzo's eye-tracking tests showed that positioning the label above (as opposed to beside) the form control works best.[10]

Placing a label right over its input field permitted users to capture both elements with a single eye movement.

But there are other reasons to put the label above the field. On small viewports there's no room beside the control. And on large viewports, zooming in increases the chance of the text disappearing off screen.[11]

Also, some labels contain a lot of text, which causes it to wrap onto multiple lines, which would disrupt the visual rhythm if placed next to the control. While you should aim to keep labels terse, it's not always possible. Using a pattern that accommodates varying content — by positioning labels above the control — is a good strategy.

LOOK, SIZE, AND SPACE

Form fields should look like form fields. But what does that mean exactly?

10 http://smashed.by/labelplacement
11 http://smashed.by/nofloatreasons

It means that a text box should look like a text box. Empty boxes signify "fill me in" by virtue of being empty, like a coloring-in book. This happens to be part of the reason placeholders are unhelpful. They remove the perceived affordance an empty text box would otherwise provide.

This also means that the empty space should be boxed in (bordered). Removing the border, or having only a bottom border, for example, removes the perceived affordances. A bottom border might at first appear to be a separator. Even if you know you have to fill something in, does the value go above the line or below it?

Spatially, the label should be closest to its form control, not the previous field's control. Things that appear close together suggest they belong together.[12] Having equal spacing might improve aesthetics, but it would be at the cost of usability.

Finally, the label and the text box itself should be large enough to read and tap. This probably means a font size of at least 16 pixels, and ideally an overall tap target of at least 44px.[13]

12 http://smashed.by/lawofproximity
13 http://smashed.by/touchtargetsizes

FOCUS STYLES

Focus styles are a simpler prospect. By default, browsers put an outline around the element in focus so users, especially those who use a keyboard, know where they are. The problem with the default styling is that it is often faint and hard to see, and somewhat ugly.

While this is the case, don't be tempted to remove it, because doing so will diminish the user experience greatly for those traversing the screen by keyboard. We can override the default styling to make it clearer and more aesthetically pleasing.

```
input:focus {
  outline: 4px solid #ffbf47;
}
```

The Email Field

Despite its simple appearance there are some important details that have gone into the field's construction which affect the experience.

Email address

The email field.

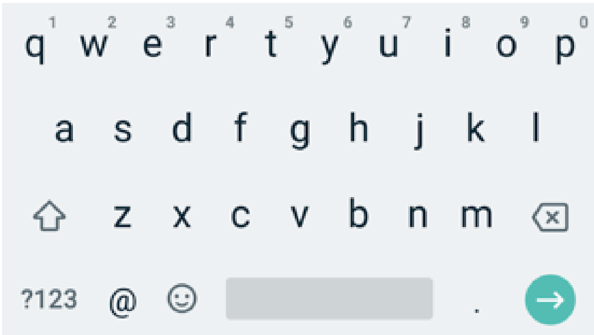
As noted earlier, some fields have a hint in addition to the label, which is why the label is inside a child span. The `field-label` class lets us style it through CSS.

```
<div class="field">
  <label for="email">
  <span class="field-label">Email address</span>
  </label>
  <input type="email" id="email" name="email">
</div>
```

The label itself is "Email address" and uses sentence case. In *Making A Case For Letter Case*, John Saito explains that sentence case (as opposed to title case) is generally easier to read, friendlier, and makes it easier to spot nouns.[14] Whether you heed this advice is up to you, but whatever style you choose, be sure to use it consistently.

The input's `type` attribute is set to `email`, which triggers an email-specific onscreen keyboard on mobile devices. This gives users easy access to the **@** and **.** (dot) symbols which every email address must contain.

14 http://smashed.by/lettercase



Android's onscreen keyboard for the email field.

People using a non-supporting browser will see a standard text input (`<input type="text">`). This is a form of progressive enhancement, which is a cornerstone of designing inclusive experiences.

PROGRESSIVE ENHANCEMENT

Progressive enhancement is about users. It just happens to make our lives as designers and developers easier too. Instead of keeping up with a set of browsers and devices (which is impossible!) we can just focus on features.

First and foremost, progressive enhancement is about always giving users a reasonable experience, no matter their browser, device, or quality of connection. When things go wrong — and they will — users won't suffer in that they can still get things done.

There are a lot of ways an experience can go wrong. Perhaps the style sheet or script fails to load. Maybe everything loads, but the user's browser doesn't recognize some HTML, CSS, or JavaScript. Whatever happens, using progressive enhancement when designing experiences stops users having an especially bad time.

It starts with HTML for structure and content. If CSS or JavaScript don't load, it's fine because the content is there.

If everything loads OK, perhaps various HTML elements aren't recognized. For example, some browsers don't understand `<input type="email">`. That's fine, though, because users will get a text box (`<input type="text">`) instead. Users can still enter an email address; they just don't get an email-specific keyboard on mobile.

Maybe the browser doesn't understand some fancy CSS, and it will just ignore it. In most cases, this isn't a problem. Let's say you have a button with `border-radius: 10px`. Browsers that don't recognize this rule will show a button with angled corners. Arguably, the button's perceived affordance is reduced, but users are left unharmed. In other cases it might be helpful to use feature queries.[15]

15 http://smashed.by/featurequeries

Then there is JavaScript, which is more complicated. When the browser tries to parse methods it doesn't recognize, it will throw a hissy fit. This can cause your other (valid and supported) scripts to fail. If your script doesn't first check that the methods exist (feature detection) and work (feature testing) before using them, then users may get a broken interface. For example, if a button's click handler calls a method that's not recognized, the button won't work. That's bad.

That's how you enhance. But what's better is not needing an enhancement at all. HTML with a little CSS can give users an excellent experience. It's the content that counts and you don't need JavaScript for that. The more you can rely on content (HTML) and style (CSS), the better. I can't emphasize this enough: so often, the basic experience is the best and most performant one.[16] There's no point in enhancing something if it doesn't *add value* (see *inclusive design principle* 7).

Of course, there are times when the basic experience isn't as good as it could be — that's when it's time to enhance. But if we follow the approach above, when a piece of CSS or JavaScript isn't recognized or executed, things will still work.

16 http://smashed.by/designperf

Progressive enhancement makes us think about what happens when things fail. It allows us to build experiences with resilience baked in. But equally, it makes us think about whether an enhancement is needed at all; and if it is, how best to go about it.

The Password Field

We're using the same markup as the email field discussed earlier. If you're using a template language, you'll be able to create a component that accommodates both types of field. This helps to enforce *inclusive design principle 3, be consistent*.

Password
Must contain 8+ characters with at least 1 number and 1 uppercase letter

The password field using the hint text pattern.

```
<div class="field">
  <label for="password">
    <span class="field-label">Choose password</span>
    <span class="field-hint">Must contain 8+ characters
with at least 1 number and 1 uppercase letter.</span>
  </label>
  <input type="password" id="password" name="password">
</div>
```

The password field contains a hint. Without one, users won't understand the requirements, which is likely to cause an error once they try to proceed.

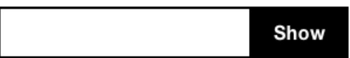
The `type="password"` attribute masks the input's value by replacing what the user types with small black dots. This is a security measure that stops people seeing what you typed if they happen to be close by.

A PASSWORD REVEAL

Obscuring the value as the user types makes it hard to fix typos. So when one is made, it's often easier to delete the whole entry and start again. This is frustrating as most users aren't using a computer with a person looking over their shoulder.

Owing to the increased risk of typos, some registration forms include an additional "Confirm password" field. This is a precautionary measure that requires the user to type the same password twice, doubling the effort and degrading the user experience.

Instead, it's better to let users reveal their password, which speaks to principles 4 and 5, *give control* and *offer choice* respectively. This way users can choose to reveal their password when they know nobody is looking, reducing the risk of typos.



The password field with a "Show password" button beside it.

Recent versions of Internet Explorer and Microsoft Edge provide this behavior natively. As we'll be creating our own solution, we should suppress this feature using CSS like this:

```
input[type=password]::-ms-reveal {
  display: none;
}
```

Now we're ready to enhance the interface with our own version.

First, we need to inject a button next to the input. The `<button>` element should be your go-to element for changing anything with JavaScript — except, that is, for changing location, which is what links are for. When clicked, it should toggle the `type` attribute between `password` and `text`; and the button's label between "Show" and "Hide."

```
function PasswordReveal(input) {
  // store input as a property of the instance
  // so that it can be referenced in methods
  // on the prototype
  this.input = input;
  this.createButton();
};

PasswordReveal.prototype.createButton = function() {
  // create a button
  this.button = $('<button type="button">Show password</
button>');
  // inject button
  $(this.input).parent().append(this.button);
  // listen to the button's click event
  this.button.on('click', $.proxy(this, 'onButtonClick'));
};

PasswordReveal.prototype.onButtonClick = function(e) {
  // Toggle input type and button text
  if(this.input.type === 'password') {
    this.input.type = 'text';
    this.button.text('Hide password');
  } else {
    this.input.type = 'password';
    this.button.text('Show password');
  }
};
```

JavaScript Syntax and Architectural Notes

As there are many flavors of JavaScript, and different ways in which to architect components, we're going to walk through the choices used to construct the password reveal component, and all the upcoming components in the book.

First, we're using a constructor. A constructor is a function conventionally written in upper camel case — `PasswordReveal`, not `passwordReveal`. It's initialized using the `new` keyword, which lets us use the same code to create several instances of the component:

```
var passwordReveal1 = new PasswordReveal(document.
getElementById('input1'));
var passwordReveal2 = new PasswordReveal(document
getElementById('input2'));
```

Second, the component's methods are defined on the prototype — `PasswordReveal.prototype.onButtonClick` for example. The `prototype` is the most performant way to share methods across multiple instances of the same component.

Third, jQuery is being used to create and retrieve elements, and listen to events. While jQuery may not be necessary or preferred, using it means that this book can focus on forms and not on the complexities of cross-browser components.

If you're a designer who codes a little bit, then jQuery's ubiquity and low-barrier to entry should be helpful. By the same token, if you prefer not to use jQuery, you'll have no trouble refactoring the components to suit your preference.

You may have also noticed the use of the `$.proxy` function. This is jQuery's implementation of `Function.prototype.bind`. If we didn't use this function to listen to events, then the event handler would be called in the element's context (`this`). In the example above, `this.button` would be undefined. But we want `this` to be the password reveal object instead, so that we can access its properties and methods.

Alternative Interface Options

The password reveal interface we constructed above toggles the button's label between "Show password" and "Hide password." Some screen reader users can get confused when the button's label is changed; once a user encounters a button, they expect that button to persist. Even though the button *is* persistent, changing the label makes it appear not to be.

If your research shows this to be a problem, you could try two alternative approaches.

First, use a checkbox with a persistent label of "Show password." The state will be signaled by the `checked` attribute. Screen reader users will hear "Show password, checkbox, checked" (or similar). Sighted users will see the checkbox tick mark. The problem with this approach is that checkboxes are for inputting data, not controlling the interface. Some users might think their password will be revealed to the system.

Or, second, change the button's *state* — not the label. To convey the state to screen reader users, you can switch the `aria-pressed` attribute between `true` (pressed) and `false` (unpressed).

```
<button type="button" aria-pressed="true">
  Show password
</button>
```

When focusing the button, screen readers will announce, "Show password, toggle button, pressed" (or similar). For sighted users, you can style the button to look pressed or unpressed accordingly using the attribute selector like this:

```
[aria-pressed="true"] {
  box-shadow: inset 0 0 0 0.15rem #000, inset 0.25em 0.25em
0 #fff;
}
```

Just be sure that the unpressed and pressed styles are obvious and differentiated, otherwise sighted users may struggle to tell the difference between them.

MICROCOPY

The label is set to "Choose password" rather than "Password." The latter is somewhat confusing and could prompt the user to type a password they already possess, which could be a security issue. More subtly, it might suggest the user is already registered, causing users with cognitive impairments to think they are logging in instead.

Where "Password" is ambiguous, "Choose password" provides clarity.

Button Styles

What's a button? We refer to many different types of components on a web page as a button. In fact, I've already covered two different types of button without calling them out. Let's do that now.

Buttons that submit forms are "submit buttons" and they are coded typically as either `<input type="submit">` or `<button type="submit">`. The `<button>` element is more malleable in that you can nest other elements inside it.

But there's rarely a need for that. Most submit buttons contain just text.

> *Note:* In older versions of Internet Explorer, if you have multiple `<button type="submit">`s, the form will submit the value of all the buttons to the server, regardless of which was clicked.[17] You'll need to know which button was clicked so you can determine the right course of action to take, which is why this element should be avoided.

Other buttons are injected into the interface to enhance the experience with JavaScript — much like we did with the password reveal component discussed earlier. That was also a `<button>` but its `type` was set to `button` (not `submit`).

In both cases, the first thing to know about buttons is that they aren't links. Links are typically underlined (by user agent styles) or specially positioned (in a navigation bar) so they are distinguishable among regular text.

When hovering over a link, the cursor will change to a pointer. This is because, unlike buttons, links have weak perceived affordance.[18]

17 http://smashed.by/submitbuttons
18 http://smashed.by/perceivedaffordance

In *Resilient Web Design*, Jeremy Keith discusses the idea of material honesty.[19] He says: “One material should not be used as a substitute for another. Otherwise the end result is deceptive.” Making a link look like a button is materially dishonest. It tells users that links and buttons are the same when they’re not.

Links can do things buttons can’t do. Links can be opened in a new tab or bookmarked for later, for example. Therefore, buttons shouldn’t look like links, nor should they have a pointer cursor. Instead, we should make buttons look like buttons, which have naturally strong perceived affordance. Whether they have rounded corners, drop shadows, and borders is up to you, but they should look like buttons regardless.

Buttons can still give feedback on hover (and on focus) by changing the background colour, for example.

PLACEMENT

Submit buttons are typically placed at the bottom of the form: with most forms, users fill out the fields from top to bottom, and then submit. But should the button be aligned left, right or center? To answer this question, we need to think about where users will naturally look for it.

19 https://resilientwebdesign.com/

Field labels and form controls are aligned left (in left-to-right reading languages) and run from top to bottom. Users are going to look for the next field below the last one. Naturally, then, the submit button should also be positioned in that location: to the left and directly below the last field. This also helps users who zoom in, as a right-aligned button could more easily disappear off-screen.

TEXT

The button's text is just as important as its styling. The text should explicitly describe the action being taken. And because it's an action, it should be a verb. We should aim to use as few words as possible because it's quicker to read. But we shouldn't remove words at the cost of clarity.

The exact words can match your brand's tone of voice, but don't exchange clarity for quirkiness.

Simple and plain language is easy for everyone to understand. The exact words will depend on the type of service. For our registration form "Register" is fine, but depending on your service "Join" or "Sign up" might be more appropriate.

Validation

Despite our efforts to create an inclusive, simple, and friction-free registration experience, we can't eliminate human error. People make mistakes and when they do, we should make fixing them as easy as possible.

When it comes to form validation, there are a number of important details to consider. From choosing when to give feedback, through to how to display that feedback, down to the formulation of a good error message — all of these things need to be taken into account.

HTML5 VALIDATION

HTML5 validation has been around for a while now. By adding just a few HTML attributes, supporting browsers will mark erroneous fields when the form is submitted. Non-supporting browsers fall back to server-side validation.

Normally I would recommend using functionality that the browser provides for free because it's often more performant, robust, and accessible. Not to mention, it becomes more familiar to users as more sites start to use the standard functionality.

While HTML5 validation support is quite good, it's not implemented uniformly.[20] For example, the `required` attribute can mark fields as invalid from the outset, which isn't desirable. Some browsers, such as Firefox 45.7, will show an error of "Please enter an email address" even if the user entered something in the box, whereas Chrome, for example, says "Please include an '@' in the email address," which is more helpful.

We also want to give users the same interface whether errors are caught on the server or the client. For these reasons we'll design our own solution. The first thing to do is turn off HTML5 validation:

```
<form novalidate>
```

HANDLING SUBMISSION

When the user submits the form, we need to check if there are errors. If there are, we need to prevent the form from submitting the details to the server.

20 http://smashed.by/formvalidation

```
function FormValidator(form) {
  form.on('submit', $.proxy(this, 'onSubmit'));
}
FormValidator.prototype.onSubmit = function(e) {
  if(!this.validate()) {
    e.preventDefault();
    // show errors
  }
};
```

Note that we are listening to the form's submit event, not the button's click event. The latter will stop users being able to submit the form by pressing **Enter** when focus is within one of the fields. This is also known as *implicit form submission*.[21]

DISPLAYING FEEDBACK

It's all very well detecting the presence of errors, but at this point users are none the wiser. There are three disparate parts of the interface that need to be updated. We'll talk about each of those now.

Document Title

The document's `<title>` is the first part of a web page to be read out by screen readers. As such, we can use it to quickly inform users that something has gone wrong with their submission.

21 http://smashed.by/implicitsubmission

This is especially useful when the page reloads after a server request.

Even though we're enhancing the user experience by catching errors on the client with JavaScript, not all errors can be caught this way. For example, checking that an email address hasn't already been taken can only be checked on the server. And in any case, JavaScript is prone to failure so we can't solely rely on its availability.[22]

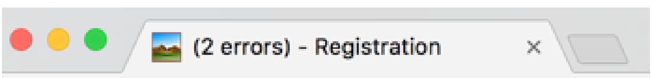
Where the original page title might read "Register for [service]," on error it should read "(2 errors) Register for [service]" (or similar). The exact wording is somewhat down to opinion.

The following JavaScript updates the title:

```
document.title = "(" + this.errors.length + ")"
+ document.title;
```

As noted above, this is primarily for screen reader users, but as is often the case with inclusive design, what helps one set of users helps everyone else too. This time, the updated title acts as a notification in the tab.

22 http://smashed.by/everyonehasjs



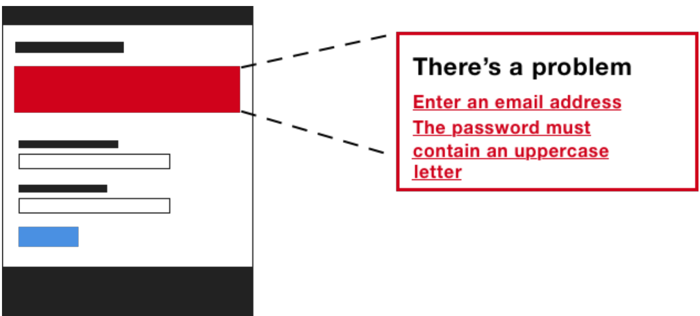
The browser tab title prefixed with "(2 errors)" acting as a quasi notification.

Error Summary

In comparison with the title element, the error summary is more prominent, which tells sighted users that something has gone wrong. But it's also responsible for letting users understand what's gone wrong and how to fix it.

It's positioned at the top of the page so users don't have to scroll down to see it after a page refresh (should an error get caught on the server). Conventionally, errors are colored red. However, relying on color alone could exclude colorblind users. To draw attention to the summary, consider also using position, size, text, and iconography.

The panel includes a heading, "There's a problem," to indicate the issue. Notice it doesn't say the word "Error," which isn't very friendly. Imagine you were filling out your details to purchase a car in a showroom and made a mistake. The salesperson wouldn't say "Error" — in fact it would be odd if they did say that.



Error summary panel positioned toward the top of the screen.

```
<div class="errorSummary" role="group" tabindex="-1"
aria-labelledby="errorSummary-heading">
  <h2 id="errorSummary-heading">There's a problem</h2>
  <ul>
    <li><a href="#emailaddress">Enter an email address
    </a></li>
    <li><a href="#password">The password must contain an
    uppercase letter</a></li>
  </ul>
</div>
```

The container has a `role` of `group`, which is used to group a set of interface elements: in this case, the heading and the error links. The `tabindex` attribute is set to `-1`, so it can be focused programmatically with JavaScript (when the form is submitted with mistakes). This ensures the error summary panel is scrolled into view. Otherwise, the interface would appear unresponsive and broken when submitted.

Note: Using `tabindex="0"` means it will be permanently focusable by way of the **Tab** key, which is a 2.4.3 Focus Order WCAG fail. If users can tab to something, they expect it will actually do something.

```
FormValidator.prototype.showSummary = function () {
  // ...

  this.summary.focus();
};
```

Underneath, there's a list of error links. Clicking a link will set focus to the erroneous field, which lets users jump into the form quickly. The link's `href` attribute is set to the control's `id`, which in some browsers is enough to set focus to it. However, in other browsers, clicking the link will just scroll the input into view, without focusing it. To fix this we can focus the input explicitly.

```
FormValidator.prototype.onErrorClick = function(e) {
  e.preventDefault();
  var href = e.target.href;
  var id = href.substring(href.indexOf("#"), href.length);
  $(id).focus();
};
```

When there aren't any errors, the summary panel should be hidden. This ensures that there is only ever one summary panel on the page, and that it appears consistently in the same location whether errors are rendered by the client or the server. To hide the panel we need to add a class of `hidden`.

```
<div class="errorSummary hidden" ...></div>
```

```
.hidden {
  display: none;
}
```

You could use the `hidden` attribute/property to toggle an element's visibility, but there's less support for it. Inclusive design is about making decisions that you know are unlikely to exclude people. Using a class aligns with this philosophy.

Inline Errors

We need to put the relevant error message just above the field. This saves users scrolling up and down the page to check the error message, and keeps them moving down the form. If the message was placed below the field, we'd increase the chance of it being obscured by the browser autocomplete panel or by the onscreen keyboard.[23]

23 http://smashed.by/errormessages

Inline error pattern with red error text and warning icon just above the field.

```
<div class="field">
  <label for="blah">
    <span class="field-error">
      <svg width="1.5em" height="1.5em">
      <use xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="#warning-icon"></use></svg>
      Enter your email address.
    </span>
    <span class="field-error">Enter an email address</span>
  </label>
</div>
```

Like the hint pattern mentioned earlier, the error message is injected inside the label. When the field is focused, screen reader users will hear the message in context, so they can freely move through the form without having to refer to the summary.

The error message is red and uses an SVG warning icon to draw users' attention. If we'd used only a color change to denote an error, this would exclude color-blind users. So this works really well for sighted users — but what about screen reader users?

To give both sighted and non-sighted users an equivalent experience, we can use the well-supported `aria-invalid` attribute. When the user focuses the input, it will now announce "Invalid" (or similar) in screen readers.

```
<input aria-invalid="false">
```

> *Note:* The registration form only consists of text inputs. In chapter 3, "A Flight Booking Form," we'll look at how to inject errors accessibly for groups of fields such as radio buttons.

SUBMITTING THE FORM AGAIN

When submitting the form for a second time, we need to clear the existing errors from view. Otherwise, users may see duplicate errors.

```
FormValidator.prototype.onSubmit = function(e) {
  this.resetPageTitle();
  this.resetSummaryPanel();
  this.removeInlineErrors();
  if(!this.validate()) {
    e.preventDefault();
    this.updatePageTitle();
    this.showSummaryPanel();
    this.showInlineErrors();
  }
};
```

INITIALIZATION

Having finished defining the `FormValidator` component, we're now ready to initialize it. To create an instance of `FormValidator`, you need to pass the form element as the first parameter.

```
var validator = new FormValidator(document.
getElementById('registration'));
To validate the email field, for example, call the
addValidator() method:
validator.addValidator('email', [{
  method: function(field) {
    return field.value.trim().length > 0;
  },
  message: 'Enter your email address.'
},{
  method: function(field) {
    return (field.value.indexOf('@') > -1);
  },
  message: 'Enter the ‘at’ symbol in the email address.'
}]);
```

The first parameter is the control's `name`, and the second is an array of rule objects. Each rule contains two properties: `method` and `message`. The `method` is a function that tests various conditions to return either `true` or `false`. False puts the field into an error state, which is used to populate the interface with errors as discussed earlier.

Forgiving Trivial Mistakes

In *The Design of Everyday Things*, Don Norman talks about designing for error. He talks about the way people converse:

> *If a person says something that we believe to be false, we question and debate. We don't issue a warning signal. We don't beep. We don't give error messages. [...] In normal conversations between two friends, misstatements are taken as normal, as approximations to what was really meant.*

Unlike humans, machines are not intelligent enough to determine the meaning of most actions, but they are often far less forgiving of mistakes than they need to be. Jared Spool makes a joke about this in "Is Design Metrically Opposed?" (about 42 minutes in):[24]

> *It takes one line of code to take a phone number and strip out all the dashes and parentheses and spaces, and it takes ten lines of code to write an error message that you left them in.*

The `addValidator` method (shown above) demonstrates how to design validation rules so they forgive trivial mistakes. The first rule, for example, trims the value before checking its length, reducing the burden on the user.

24 http://smashed.by/onelineofcode

LIVE INLINE VALIDATION

Live inline validation gives users feedback as they type or when they leave the field (`onblur`). There's some evidence to show that live inline validation improves accuracy and decreases completion times in long forms.[25] This is partially to do with giving users feedback when the field's requirements are fresh in users' minds. But live inline validation (or live validation for short) poses several problems.

For entries that require a certain number of characters, the first keystroke will always constitute an invalid entry. This means users will be interrupted early, which can cause them to switch mental contexts, from entering information to fixing it.

Alternatively, we could wait until the user enters enough characters before showing an error. But this means users only get feedback after they have entered a correct value, which is somewhat pointless.

We could wait until the user leaves the field (`onblur`), but this is too late as the user has mentally prepared for (and often started to type in) the next field. Moreover, some users switch windows or use a password manager when using a form. Doing so will trigger the blur event, causing an error to show before the user is finished. All very frustrating.

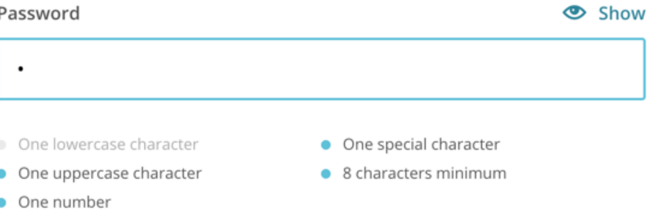
25 http://smashed.by/inlinevalidation

Remember, there's no problem with giving users feedback without a page refresh. Nor is there a problem with putting the error messages inline (next to fields) — we've done this already. The problem with live feedback is that it interrupts users either too early or too late, which often results in a jarring experience.

If users are seeing errors often, there's probably something wrong elsewhere. Focus on shortening your form and providing better guidance (good labeling and hint text). This way users shouldn't see more than the odd error. We'll look at longer forms in the next chapter.

CHECKLIST AFFIRMATION PATTERN

A variation of live validation involves ticking off rules (marking them as complete) as the user types. This is less invasive than live validation but isn't suited to every type of field. Here's an example of MailChimp's sign-up form, which employs this technique for the password field.



MailChimp's password field with instructions that get marked as the user meets the requirements.

You should put the rules above the field. Otherwise the onscreen keyboard could obscure the feedback. As a result, users may stop typing and hide the keyboard to then check the feedback.

A NOTE ON DISABLING SUBMIT BUTTONS

Some forms are designed to disable the submit button until all the fields become valid. There are several problems with this.

First, users are left wondering what's actually wrong with their entries. Second, disabled buttons are not focusable, which makes it hard for the button to be discovered by blind users navigating using the **Tab** key. Third, disabled buttons are hard to read as they are grayed out.

As we're providing users with clear feedback, when the user expects it, there's no good reason to take control away from the user by disabling the button anyway.

CRAFTING A GOOD ERROR MESSAGE

There's nothing more important than content. Users don't come to your website to enjoy the design. They come to enjoy the content or the outcome of using a service.

Even the most thought-out, inclusive and beautifully designed experience counts for nothing if we ignore the words used to craft error messages. One study showed that showing custom error messages increased conversions by 0.5% which equated to more than £250,000 in yearly revenue.[26]

> *Content is the user experience.*
>
> *– Ginny Redish*

Like labels, hints, and any other content, a good error message provides clarity in as few words as possible. Normally, we should drive the design of an interface based on the content — not the other way around. But in this case, understanding how and why you show error messages influences the design of the words. This is why Jared Spool says "content and design are inseparable work partners."[27]

We're showing messages in the summary at the top of the screen and next to the fields. Maintaining two versions of the same message is a hard sell for an unconvincing gain. Instead, design an error message that works in both places. "Enter an 'at' symbol" needs context from the field label to make sense. "Your email address needs an 'at' symbol" works well in both places.

26 http://smashed.by/errormessagesroi
27 http://smashed.by/contentdesign

Avoid pleasantries, like starting each error message with "Please." On the one hand, this seems polite; on the other, it gets in the way and implies a choice.

Whatever approach you take, there's going to be some repetition due to the nature of the content. And testing usually involves submitting the form without entering any information at all. This makes the repetition glaringly obvious, which may cause us to flip out. But how often is this the case? Most users aren't trying to break the interface.

There's a problem

Enter your first name
Enter your last name
Enter your email address
Enter your password

An error summary containing a wall of error messages makes the beginning of the words seem too repetitive.

Different errors require different formatting. Instructions like "Enter your first name" are natural. But "Enter a first name that is 35 characters or less" is longer, wordier, and less natural than a description like "First name must be 35 characters or less."

Here’s a checklist:

- **Be concise.** Don’t use more words than are necessary, but don’t omit words at the cost of clarity.
- **Be consistent.** Use the same tone, the same words, and the same punctuation throughout.
- **Be specific.** If you know why something has gone wrong, say so. “The email is invalid.” is ambiguous and puts the burden on the user. “The email needs an ‘at’ symbol” is clear.
- **Be human, avoid jargon.** Don’t use words like *invalid*, *forbidden*, and *mandatory*.
- **Use plain language.** Error messages are not an opportunity to promote your brand’s humorous tone of voice.
- **Use the active voice.** When an error is an instruction and you tell the user what to do. For example, “Enter your name,” not “First name must be entered.”
- **Don’t blame the user.** Let them know what’s gone wrong and how to fix it.

Summary

In this chapter we solved several fundamental form design challenges that are applicable well beyond a simple registration form. In many respects, this chapter has been as much about what not to do, as it has about what we should. By avoiding novel and artificial space-saving patterns to focus on reducing the number of fields we include, we avoid several usability failures while simultaneously making forms more pleasant.

THINGS TO AVOID

- Using the `placeholder` attribute as a mechanism for storing label and hint text.
- Using incorrect input types.
- Styling buttons and links the same.
- Validating fields as users type.
- Disabling submit buttons.
- Using complex jargon and brand-influenced microcopy.

DEMOS

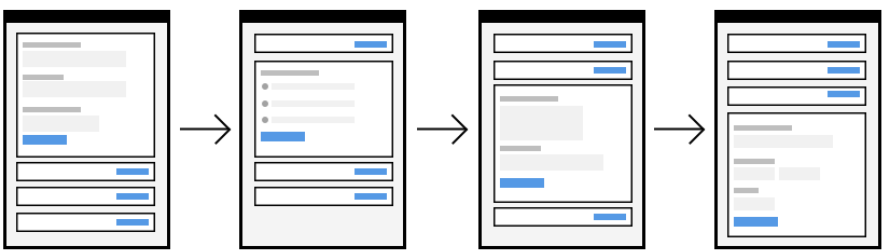
- Registration form: http://smashed.by/regformdemo

A Checkout Form

Here's what I bring to the table: a valid credit card, 90 seconds of my time, and my right thumb. The rest is up to you.

— *Melanie Jones, "8 things parenting taught me about accessibility"*[1]

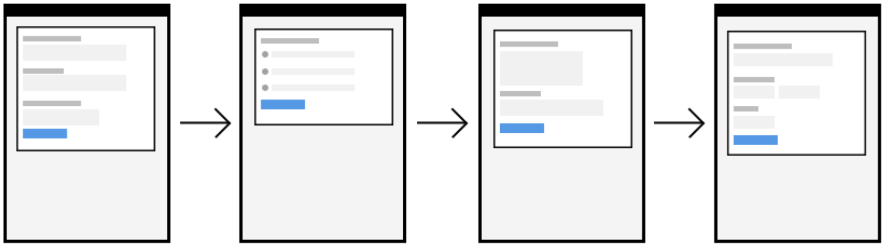
In 2008 I worked on Boots.com, where we designed a single-page checkout flow. This involved the trendiest of techniques from that era, including accordions, Ajax and client-side validation. Each step of the flow (delivery address, delivery options, payment) was an accordion panel, submitted via Ajax. On successful submission, the panel collapsed and the next one opened.



The Boots accordion single page checkout flow.

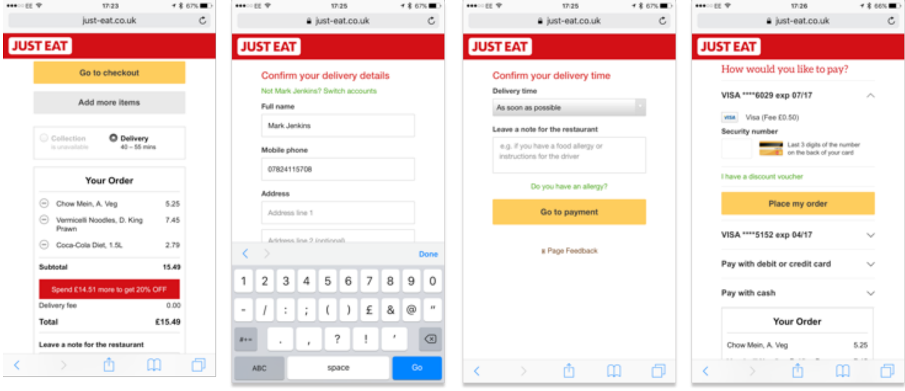
1 http://smashed.by/parenting

Users struggled to complete their orders. Errors were hard to fix as users had to scroll up and down. And the accordion was a distraction. Inevitably, Boots asked us to make changes. We redesigned it so each panel became a page removing the need for an accordion and Ajax. (We kept the client-side validation to avoid an unnecessary trip to the server.)



The Boots one-thing-per-page checkout flow.

This converted a lot better. Although I can't remember the exact numbers (it was back in 2008, remember!), the client was happy with the results. Six years later, in 2014, at Just Eat, the same thing happened. We redesigned the single-page checkout flow so each section became a page. This time I noted the numbers. The result was a whopping 5% increase in conversion. This equated to 2 million orders a year. That's *orders*, not revenue.



The Just Eat one-thing-per-page checkout mobile screens.

Two years later, in 2016, Robin Whittleton from the UK's Government Digital Service (GDS), told me that putting each thing on a page of its own was a design pattern called "one thing per page."[2] Behind the improved numbers, there are many reasons why it drastically improves the user experience.

One Thing Per Page

One thing per page is about splitting up a complicated process into small chunks and placing them on screens of their own. For example, instead of putting delivery address, delivery options, and payment forms on one page, we put them on separate pages.

2 http://smashed.by/1thingperpage

It's not necessarily about having one element or component on a page (although it could be). In all likelihood, you'll still have a header. Similarly, it's not about having a single form field on each page (although it absolutely could). And it doesn't mean you'll always end up with one question per page either.

Forms expert Caroline Jarrett, who first wrote about the pattern in 2015, explains that user research "will quickly show you that some questions will be best grouped into a longer page."[3]

However, she also explains that "questions that naturally 'go together' from the point of view of designers [...] don't need to be on the same page to work for users." She provides an enlightening example when, for GOV.UK Verify, they tested putting "Create a username" on one page, and "Create a password" on the next.

Like most designers, Caroline thought that putting them on separate pages would be overkill. In reality, users weren't bothered. So start with one thing (field or question) per page, then, through research, find out if grouping fields improves the experience.

3 http://smashed.by/nomoreaccordions

While this pattern often bears wonderful and delicious fruit (or orders and conversions, if you hate my analogies), it's useful to understand why it works so well.

- *Inclusive design principle 6* says we should "*design interfaces that help users focus on core tasks by prioritizing them*." It even goes on to say that "*people should be able to focus on one thing at a time*." One thing per page simply follows this principle to the letter, and in doing so drastically reduces the cognitive burden on users.
- When users fill in a small form, "*errors are caught and shown early*" and often. If there's one thing to fix, it's easy to fix, which reduces the chance of users giving up on the task.
- If pages have little on them, they'll load quickly. Faster pages reduce the risk of users leaving and they build trust.
- By submitting information frequently, we can save user information in a more granular fashion. If a user drops out, we can, for example, send them an email prompting them to complete their order.
- Conversely, a long form increases the chance of a page timing out, or the computer freezing. This is what happens to Daniel, the lead character in Ken Loach's film, *I, Daniel Blake*.[4] With declining health and having never

4 http://smashed.by/danielblake

used a computer, it freezes and he loses his data. In the end, he gives up.

- It adds a sense of progression and increases momentum because the user is constantly moving forwards step by step.
- It lets you design interfaces that capitalize on maximal screen space — interfaces that wouldn't work so well if part of a larger form. We'll see an example of this in the next chapter, when we design an airplane seat chooser.

Flow and Order

In *Forms That Work*, Caroline Jarrett and Gerry Gaffney explain the importance of asking questions in a sensible order:[5]

> *Asking for information at the wrong time can alienate a user. The same question put at the right moment can be entirely acceptable. Think about buying a car. You're just browsing, getting a sense of what is available. A salesperson comes along and starts to ask you how you'll pay. Would you answer? Or would you think, "If that person doesn't stop annoying me, I'm out of here"?*

Now think about the point where you've told the salesperson which car you want to buy. Now it's appropriate to start negotiating about payment. It would be quite odd if the salesperson did *not* do so.

5 http://smashed.by/formsthatwork

Just like the car salesperson, we'll ask for the right information at the right time. For example, payment happens toward the end. Users will be given a chance to check their order before submitting it. Finally, the confirmation page acts as a sales receipt for administrative purposes. Here's the complete flow:

1. Email address
2. Mobile phone (optional)
3. Delivery address
4. Delivery options
5. Delivery notes
6. Payment
7. Check your answers
8. Confirmation

Guest Checkout

Inclusive design principle 2, "Consider situation," says:

People use your interface in different situations. Make sure your interface delivers a valuable experience to people regardless of their circumstances.

We'll first design the checkout journey for anonymous users. Not letting users check out as a guest is just about the worst thing we can do, as Jared Spool attests to beautifully in "The $300 Million Button."[6]

6 http://smashed.by/3mbutton

The article tells a story of one company losing 300 million dollars because they thought forcing users to register first would help speed up subsequent purchases. While this is true, it also assumes users want to sign up in the first place.

Bearing in mind what we've learned about flow and order, and the question protocol, that story is hardly a surprise. What value does a first-time user get in return for the added effort of signing up? Nothing. And that's all you really need to know.

We'll look at a more appropriate time to ask users to sign up, as well as looking at how to optimize the journey for second-time users later on in the chapter.

1. Email Address

In chapter 1, "A Registration Form," we had to ask users for an email address (see page 33). We can reuse that pattern here, saving us the effort of solving the same problem again from scratch.

There is, however, an opportunity to adapt the content to fit this context better. By that, I mean users may wonder why they're being asked for an email address just to purchase something. One of the main takeaways from chapter 1 was the need to justify the existence of each and every form field.

The email address field with hint text explaining why users are being asked for this.

Here, it's because we can send users a receipt, which is particularly important if checking out anonymously. Additionally, the email may provide details about how to return the item, or cancel or change the order. We can tell users this transparently via the hint text.

> *Note:* The button's positioning and styling is the same as set out in "A Registration Form." But the label is set to "Continue," which implies progress, and is better suited to the linear checkout flow.

2. Mobile Phone

Like the email field, we should be asking ourselves why we're requesting a phone number. We know the courier offers real-time text messages on the day of delivery —

but the customer doesn't. So we tell them via the hint. Remember, the hint is not just for formatting rules: it's for anything that will help users fill out the field. This transparency builds trust, reduces effort, and promotes the feature all at the same time.

Mobile number (optional)
So we can notify you about delivery

Continue

Mobile phone field with "(optional)" text at the end of the label.

```
<div class="field">
  <label for="mobile">
    <span class="field-label">Mobile number (optional)</span>
    <span class="field-hint">So we can notify you about
delivery</span>
  </label>
  <input type="tel" id="mobile" name="mobile">
</div>
```

The input's `type="tel"` attribute will spawn a telephone-specific keyboard on mobile devices. This makes it easier to enter a phone number thanks to the larger keypad.

The iOS on-screen keyboard for the telephone input.

MARKING OPTIONAL OR REQUIRED FIELDS

While real-time notifications *add value*, we shouldn't assume everyone wants to receive them, nor that everyone has a mobile phone. So we let users choose to skip this field by marking the field as optional. This way, users can opt in if they like.

By convention, required fields are marked with an asterisk. A legend is usually placed above the form to denote its meaning, but as Luke Wroblewski says:[7]

> *[...] including the phrase "optional" after a label is much clearer than any visual symbol you could use to mean the same thing. Someone may always wonder 'what does this asterisk mean?' and have to go hunting for a legend that explains things.*

---

7 http://smashed.by/requiredfields

You might also be wondering why we're marking optional fields, instead of required ones. In "Required Versus Optional Fields," Jessica Enders says:[8]

[…] think about what we are doing when we mark something in an interface. We are trying […] to ***indicate that it's different****.*

Thanks to the question protocol, most fields should be required, so we mark optional fields instead.

> *Note:* Putting the `aria-required="true"` attribute on an input will tell screen reader users that the field is required. As we're only marking optional fields, this attribute isn't necessary. And the optional text from within the label will be announced by screen readers too, giving sighted and non-sighted users an equivalent experience.

3. Delivery Address

The delivery address contains five fields that together make up an address. Visually there is a slight difference between the fields: field width.

8 http://smashed.by/optfields

Delivery address fields: address line 1, line 2, city, and postcode.

```
<div class="field">
  <label for="address1">
    <span class="field-label">Address line 1</span>
  </label>
  <input type="text" id="address1" name="address1">
</div>
<div class="field">
  <label for="address2">
    <span class="field-label">Address line 2 (optional)</span>
  </label>
  <input type="text" id="address2" name="address2">
</div>
```

```
<div class="field">
  <label for="city">
    <span class="field-label">City</span>
  </label>
  <input type="text" id="city" name="city">
</div>
<div class="field">
  <label for="postcode">
    <span class="field-label">Postcode</span>
  </label>
  <input type="text" id="postcode" name="postcode">
</div>
```

FIELD WIDTH

In “Write Less Damn Code”, Heydon Pickering jokingly points out that the reason some people used to add XHTML 1.1 compliant banners to their website was to ensure the height of the menu matches the height of the content.[9] Similarly, you might be tempted to give every address field the same width.

But giving the postcode field the same width as every other field increases the cognitive effort needed to fill it out. This is because the width gives users a clue as to the length of the content it requires.

9 http://smashed.by/heydontalk

Baymard Institute's study[10] found that:

If a field was too long or too short, [users] started to wonder if they had misunderstood the label. […] This was especially true for fields with uncommon data or a technical label like CVV [card verification value].

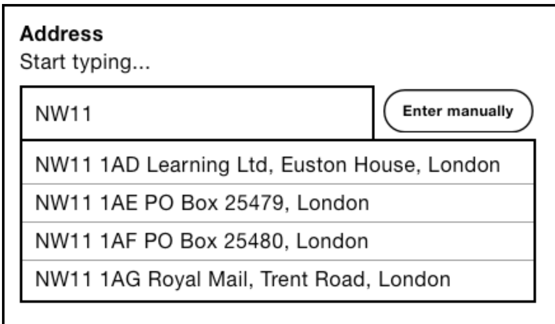
As postcodes consist of six to eight characters, the field's width should be smaller than the other fields. You should apply this rule to every field where the length of the content is known.

CAPTURE+ ENHANCEMENT

Capture+ is a third-party plugin that lets users search for their address quickly and accurately.[11] Instead of manually typing each part of the address in five separate boxes, users type into just one.

As the user types the first line of their address, suggestions will appear from which they can select. This reduces the number of keystrokes and therefore the chance of typos.

10 http://smashed.by/formfieldux
11 http://smashed.by/addresscapture



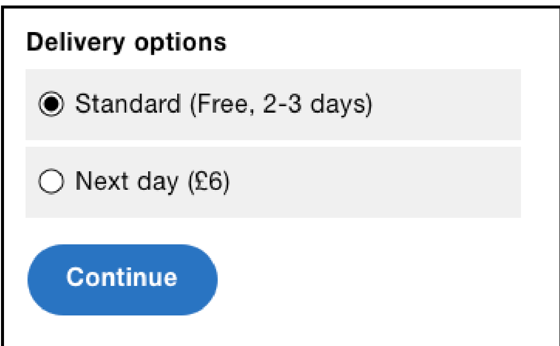
A text box using the Capture+ plugin showing options as users type their postcode.

If no address is found, users can change the interface back to the original address form. In doing so, we conform to *inclusive design principle 5*, "Offer choice."

Capture+ has a third-party script which you can include on your page. But most third-party scripts don't account for the broad range of interaction preferences, usability, and accessibility considerations. We'll look at all of this in the next chapter, when we build our own accessible autocomplete component from scratch.

4. Delivery Options

This is the first field that consists of multiple controls; in this case, radio buttons.



Delivery option radio buttons with two options: free delivery, and premium.

```
<fieldset class="field">
  <legend>
    <span class="field-legend">Delivery options</span>
  </legend>
  <div class="field-radioButton">
    <input type="radio" name="option" id="option"
value="Standard" checked>
    <label for="option">Standard (Free, 2-3 days)</label>
  </div>
  <div class="field-radioButton">
    <input type="radio" name="option" id="option2"
value="Premium">
    <label for="option2">Premium (£6, Next day)</label>
  </div>
</fieldset>
```

GROUPING

To group multiple controls, we must wrap them in a `fieldset`. The `legend` describes the group like a `label` describes the individual control.

Some screen readers, such as NVDA, will read the `legend` out, along with the first individual radio button's `label` when entering the field (in either direction). In this example, "Delivery options, Standard (Free two to three days)" is announced. In other screen readers, such as Voiceover with Safari, the legend is announced for every field.

If we omitted the `fieldset` and `legend` elements, screen reader users would only hear "Standard (Free, two to three days)," which is less clear.

You may be tempted to group all fields this way. For example, the address form from earlier could be wrapped inside a `fieldset` with a `legend` set to "Address." While this is technically valid, it's unnecessary and verbose, as the field labels make sense without a `legend`. Put another way, users don't need to hear "Address: Address Line 1" as it doesn't *add value*.

SMART DEFAULTS

As most users will want free delivery, that option comes first. It's also selected by default thanks to the `checked` attribute. This stops users from ever seeing an error and gives users less to do.

Design for common circumstances first.

– Caroline Jarrett[12]

STYLING

By default, radio buttons (and checkboxes) are rendered quite small. This makes them hard to click or tap, especially for people with motor impairments.

We can increase the size using CSS, but this isn't as simple as it sounds. In "Making radio buttons and checkboxes easier to use," Robin Whittleton explains that the way browsers respond to CSS differs.[13]

Some browsers, such as Internet Explorer 8, won't apply size changes, but undesirable space will be created around the radio button.

12 http://smashed.by/designmantra
13 http://smashed.by/govukissues

Other browsers, such as Firefox on OS X (10.10), will increase the size but the radio buttons will appear blurry. Finally, there are browsers that will apply the changes without making them blurry.

Fortunately, a radio button's label acts as a proxy for the radio button itself. That is, when clicked, the radio button will become checked (or unchecked, depending on state). Unfortunately, many users don't realize they can do this.[14] This is hardly surprising because labels have very little to signify that clicking them would do anything different to regular copy.

To give users a better chance, we can color them gray and make them respond to the mouse on hover. However, even with these enhancements, some users may still be unaware. GDS's research showed this to be the case, which is why they embarked on developing custom radio button controls.

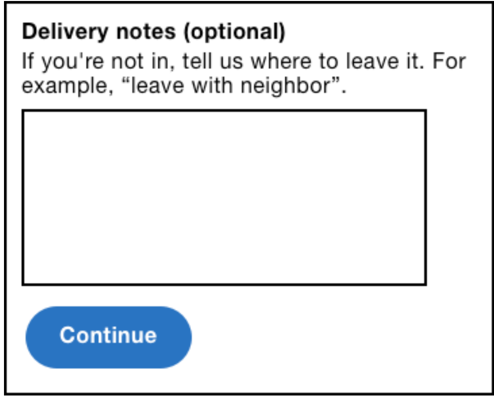
The problem with creating custom controls is that you have to reimplement all the behavior that is provided natively for free. This is very involved, and despite GDS's in-depth attempts, they aren't without their problems.[15]

14 http://smashed.by/govukcheckboxes
15 http://smashed.by/govukissues

5. Delivery Notes

Imagine you're at work. You receive a notification to say your item is being delivered. When you arrive home, instead of seeing the package, you find a card saying it couldn't be delivered because it was too big to fit through the letterbox. Frustrating.

A delivery note, which you can provide at your discretion, stops this from happening. The delivery note tells the delivery person what to do if you're not home. Perhaps you'd prefer it to be left with a neighbor, or inside your recycling bin, which Amazon refers to as a "safe place." This, by the way, works surprisingly well.



Delivery notes field with hint explaining what to enter

```
<div class="field">
  <label for="notes">
  <span class="field-label">Delivery notes (optional)</span>
  <span class="field-hint">Tell us where to leave your
package in case you're not in. For example,
"Leave it with my neighbor".</span>
  </label>
  <textarea id="notes" name="notes"></textarea>
</div>
```

The `textarea` is similar to a text box except it allows users to enter multiple lines of text, which is particularly appropriate for a delivery note. (Remember: the size of the field gives users a clue as to the length of content needed.)

While this question *adds value*, we need to understand how it will be used by the delivery person as this may influence the design. In this case, the viewport on the device is small and can't be scrolled, so we need to limit the amount of text that can be entered.

LIMITING TEXT

Limiting the amount of text a user can type can and should be handled by validation, as set out in chapter 1, "A Registration Form." But there are some additional considerations. The `maxlength` attribute (which takes a number value) limits the amount of a text a user can type. As soon as the limit is reached, the browser will ignore the input.

Support for this attribute on a `textarea` is both lacking and buggy.[16] But even if it was well supported, it's not recommended because some users don't look at the screen as they type — they are focused solely on the keyboard. When a user enters a lot of text, they'll look up to find half their entry has been truncated. Not good.

CHARACTER COUNTDOWN

Instead, we should let users type freely, and tell users how many characters they have left. This way, users can see the feedback when they finally look up at the screen and can edit their entry in response. If they don't notice the feedback, an error will be shown when they submit the form, thanks to the validation routine (set out in chapter 1, "A Registration Form").

Delivery notes (optional)
If you're not in, tell us where to leave it. For example, "leave with neighbour".

You have 150 characters remaining.

Continue

The character countdown telling users how many characters they have left.

16 http://smashed.by/maxlength

To create this component, we need to use JavaScript to inject a status box below the field.

```
<div>You have 100 characters remaining.</div>
```

Then we need to listen to the textarea's `keydown` event, which is the event that fires as the user types:

```
function CharacterCountdown(input, options) {
  this.input = $(input);
  this.input.on("keydown", $.proxy(this, 'onKeydown'));
  // ...
};
```

The event listener will then check the `length` of the typed value against the configurable `maxLength` to calculate how many characters are remaining. This is then injected into the status box:

```
CharacterCountdown.prototype.onFieldChange = function(e) {
  var remaining = this.options.maxLength-this.field.val().
length;
  this.status.html(this.options.message.replace(/%count%/,
remaining));
};
```

Live Regions

The trouble is, this status is only determinable by sighted users. To give screen reader users a comparable experience (*inclusive design principle 1*), we should make sure this information is communicated to them too.

Screen readers will normally only announce content when it is focused, but live regions announce their content when it changes. This means we can communicate to screen reader users without asking them to leave their current location. In this case, it means users can continue to type into the textarea.

```
<div role="status" aria-live="polite">You have 100
characters remaining.</div>
```

Notes:

- The `aria-live="polite"` property[17] and the `status` role[18] are equivalent. Both are provided to maximize compatibility across platforms and screen readers (in some setups, only one or the other is recognized).

17 http://smashed.by/aria
18 http://smashed.by/ariastatus

- The equivalent `alert` and `assertive` values mean the current readout of the screen reader will be interrupted to announce the live region's new content. In this case, interrupting the user as they're typing is aggressive. So we can keep `status` and `polite` values, which means the contents are announced *after* the user stops typing for a moment.

Announcing Only When It's Critical

The character countdown we've designed so far has several provisions that carefully help the user as they type their entry. First, it doesn't stop the user typing when they exceed the maximum amount of characters. And second, it will only inform users when they stop typing.

Even so, if the user is able to type a large amount of characters, or if the maximum is rarely exceeded, it seems a bit overbearing to interrupt users at all.

A more considerate approach might be to give feedback when users get close to the limit. To do this we could add a critical percentage option. Setting this to 10%, for example, when the maximum is 100 would mean users start being informed when they have typed 90 characters.

6. Payment

It's hardly surprising that most transactions are abandoned at the payment page. Not only is this screen shown toward the end of the journey (when users have had the most time to reconsider their decision and used up a lot of energy), but they may have to stop and find their credit card.

Fortunately, there are some usability provisions we can apply here. By using autofill, removing unnecessary fields, using the right input types and crafting label (and hint) text, we can drastically reduce friction and keep users on-task.

REMOVING FIELDS

There are a number of details on a credit or debit card: name on card, card number, valid from date, expiry date, issue number, security number; all of these are commonly found on payment forms. But not all of these details are needed to process a payment.

When we designed Kidly's checkout flow, chief technology officer Øyvind Valland carefully picked Stripe[19] as the payment provider. This way, we didn't have to worry about PCI compliance[20] and the cost of developing a solution from scratch. Here's the payment form we ended up with:

19 http://smashed.by/stripe
20 http://smashed.by/whatispci

You'll notice the valid from date, which is often provided on a payment form is missing, so I spoke with Øyvind to find out why. Here's what he said:

We don't need to ask for "Valid From." Only a handful of debit cards show those, and it provides more hassle for the customer to enter than benefit to us in verifying card details. That is, if the card is stolen, having to enter a valid from date isn't going to stop the thief.

Payment

Card number
The long number on the front of your card

Expiry date
MMYY

Security number
The last 3 digits on the back of the card

Is your billing address the same as delivery?

☑ Yes, my billing address is the same

Continue

Payment form with four fields: card number, expiry date, security number, and same as billing address checkbox (checked by default).

He went on to talk about the billing address, which is the address to which the card is registered:

Only the numerics contained in card details are used for verification. That is, the house number is used, but not street name. We ask for it for our records. Being able to eyeball this stuff is handy in any situation where you have to query what's happened. Besides, some people expect that they'll have to provide an address.

Øyvind is not a designer per se, but his input into the design process was crucial. Many of us assume that back-end developers don't care about the user experience, but tapping into their knowledge is immensely valuable.

Design is a team sport, and so we should treat it as one. By designing (and researching) with a diverse set of people, we'll frequently end up producing a far better experience. We should also question the presence of form fields. If you look at other people's designs and assume something has to be a certain way, we'll never improve micropatterns such as these. Proving assumptions are correct or otherwise is an essential weapon in a designer's arsenal.

When researching this section for the book, I decided to speak to Stripe to see if we could reduce the amount of fields even more. Here's what they said:

The bare minimum information needed to attempt a valid payment is card number, CVV, and expiration date. Additional information will allow the card-issuing bank to make a more informed decision about accepting or declining the payment, so while more information isn't required, it will improve your chances of the payment succeeding. It will also provide you more information for verifying the payment is valid and authorized, and therefore can help reduce the likelihood of a dispute and help you with contesting the dispute if it does occur.

AUTOFILL

Most modern browsers can automatically fill in form fields, by way of the `autocomplete` attribute. When the user focuses a particular field, the browser checks if it has that information stored — if it does, the user can select it without having to type.

Chrome autofill: used 9 billion times/month; saves an average of 12 seconds; 1.25 million days saved/month

— Luke Wroblewski, October 24, 2017[21]

Since iOS 8, Safari lets users scan their card using the iPhone's camera — it uses the same mechanism to automatically fill out those fields.

Not only does this drastically reduce the amount of effort to complete the form, but it also eliminates the chance of typos: two very helpful improvements to a form that has the highest drop-off rates in ecommerce.

As mentioned earlier, autofill is enabled with the `autocomplete` attribute. Most modern browsers support it, but some older browsers offer similar functionality by using the `name` attribute instead. For the widest support, you should specify the correct values for both attributes as shown below.[22]

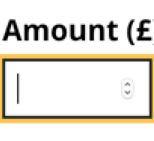
21 http://smashed.by/chromeautofill
22 You can refer to the full list of available values in the HTML specification: http://smashed.by/autocompletespec

```
<div class="field">
  <label for="ccname">
    <span class="field-label">Name on card</span>
  </label>
  <input type="text" id="ccname" name="ccname" autocomplete="cc-name">
</div>
<div class="field">
  <label for="cardnumber">
    <span class="field-label">Card number</span>
  </label>
  <input type="text" id="cardnumber" name="cardnumber"
autocomplete="cc-number">
</div>
<div class="field">
  <label for="expdate">
    <span class="field-label">Expiry date</span>
  </label>
  <input type="text" id="expdate" name="expdate"
autocomplete="cc-exp">
</div>
<div class="field">
  <label for="cvc">
    <span class="field-label">Security code</span>
  </label>
  <input type="number" id="cvc" name="cvc" autocomplete="cc-csc">
</div>
<fieldset class="field">
  <legend>
    <span class="field-legend">Is your billing address the
same as delivery?</span>
  </legend>
  <div class="field-checkbox">
    <label for="things">
      <input type="checkbox" name="things" value="" id="things"
checked>Yes, it's the same
    </label>
  </div>
</fieldset>
```

NUMBER INPUT

The number input (`<input type="number">`) lets mobile users more quickly type a number via a numeric keypad. On desktop, the input will contain increment and decrement buttons called spinners, which make it easy to make small adjustments without having to select and type.

You might think the number input is appropriate for the card number, expiry date, and CVC number — after all, they all consist of numbers. But it's a lot more complicated than that. By looking at what the spec says, what browsers do, and what users want, we can more easily determine when the number input is appropriate or not.



Number input with tiny spinner buttons inside.

Let's start with some definitions. Wikipedia says that:[23]

A number is a mathematical object used to count, measure, and label. [...] numerals are often used for labels (as with telephone numbers), for ordering (as with serial numbers), and for codes (as with ISBNs).

23 http://smashed.by/number

Most of us think of numbers this way. We use them to count and measure, but equally we use them in dates and codes. However, the HTML specification only agrees in part with this definition. It says that:[24]

The `type=number` state is not appropriate for input that happens to only consist of numbers but isn't strictly speaking a number. [...] When a spinbox interface is not appropriate, `type=text` is probably the right choice (possibly with a pattern attribute).

In other words, numbers and numerals are different. Numbers represent an amount of something such as:

- my age (announced "thirty-four years old")
- the price of an apple (announced "forty-five pence")
- the time it took me to cook breakfast (announced "ten minutes")

Conversely, numerals might be used for dates and codes such as:

- birth date (announced "nineteenth of June, nineteen eighty-three")
- pin code (announced "eight, double five, three, two, six")

24 http://smashed.by/typenumber

There's a difference between the way these values are announced. Understanding this helps us see that while the way browsers implement the number input may seem buggy at first — it isn't.

For example, IE11 and Chrome will ignore non-numeric input such as a letter or a slash. Some older versions of iOS will automatically convert "1000" to "1,000." Safari 6 strips out leading zeros. Each example seems undesirable, but none of them stop users from entering true numbers.

Some numbers contain a decimal point such as a price; other numbers are negative and need a minus sign. Unfortunately, some browsers don't provide buttons for these symbols on the keypad. If that wasn't enough, some desktop versions of Firefox will round up huge numbers.

In these cases, it's safer to use a regular text box to avoid excluding users unnecessarily. Remember, users are still able to type numbers this way — it's just that the buttons on the keypad are smaller. To further soften the blow, the numeric keyboard can be triggered for iOS users by using the `pattern` attribute.[25]

```
<input type="text" pattern="[0-9]*">
```

25 http://smashed.by/govuka11y

In short, only use a number input if:

- incrementing and decrementing makes sense
- the number doesn't have a leading zero
- the value doesn't contain letters, slashes, minus signs, and decimal points.
- the number isn't very large

Let's apply these rules to the expiry date. Incrementing it doesn't make sense, the number could start with a zero, and credit cards put a slash in the middle of the expiry date which users should be able to copy. Using a number input is not only inappropriate, but it creates a jarring experience as the user types a slash which would be ignored.

We'll look at appropriate use cases of the number input in the next chapter.

A Note about the Telephone Input

The telephone input (`<input type="tel">`) is sometimes used as a makeshift number input because it gives users the benefits of a number-specific keypad on mobile without some of the pitfalls discussed earlier.

Apart from semantic incorrectness, I've not come across practical reasons not to use it. However, ignoring the standards specification is generally not recommended because we don't know how browsers and devices might handle it in the future. For example, perhaps a browser's autocomplete routine will try to fill in the user's telephone number instead of the credit card number. Or a device may choose to load the user's contact address book, from which they can choose an appropriate number.

Both of these features would be suited to a legitimate telephone field, but not for anything else that happens to consist of numbers. Updating your implementation in response to a new browser exposing new behavior is not an ideal strategy.

FORGIVING BAD INPUT

In "A Registration Form" we briefly talked about forgiving little input mistakes. In fact, the success of the internet is largely down to its robustness, thanks to what's known as Postel's law:

Be conservative in what you send; be liberal in what you accept.

We can apply this principle to the fields in the payment form. For example, a card number typically appears as sixteen digits split into four parts by spaces. Some users may type the space; others might not.

Similarly, for the expiry date some users might type a slash, others may leave it out. Whether it's a slash or space, or a card number, or an expiry date, we should be forgiving by stripping whitespace and normalizing the format where possible.

CARD VERIFICATION CODE (CVC) FIELD

Every payment provider needs the user's CVC number, usually the last three digits found on the back of the card.

The first problem is that sites don't always refer to this field as the CVC number. Sometimes it's referred to as a security code number or card verification value (CVV). Being specified as an acronym doesn't help either. And to top it off, on the card the number is never accompanied by a description, making it hard to reconcile the requirements.

To fix this, we should employ the hint text pattern to tell users exactly what it is and where to find it. For example: "This is the last three digits on the back of the card."

BILLING ADDRESS TOGGLER

The billing address is the address to which the card is registered and is needed to process a payment. For most users, their billing address is the same as the delivery address. As the user has already provided this information, we can use it to improve the experience.

First, we need to add a checkbox field which asks the user if their billing address is the same as their delivery address. This way, users only have to fill out the billing address on the rare occasion that it's different. As it's the most common scenario, it's checked by default.

We can enhance the experience by hiding the billing address until the user unchecks the checkbox. This is a form of progressive disclosure, which means showing information only when it becomes relevant.

Here's the complete script with notes to follow.

```
function CheckboxCollapser(checkbox, toggleElement) {
  this.checkbox = checkbox;
  this.toggleElement = toggleElement;
  this.check();
  this.checkbox.on('click', $.proxy(this,
'onCheckboxClick'));
};
```

```
CheckboxCollapser.prototype.onCheckboxClick = function(e) {
  this.check();
};

CheckboxCollapser.prototype.check = function() {
  if(this.checkbox.prop('checked')) {
    this.toggleElement.addClass('hidden');
  } else {
    this.toggleElement.removeClass('hidden');
  }
};
```

Notes

- The constructor function takes the checkbox element and toggle element (the billing address container) as parameters and assigns them to `this` so they can be referenced in the other methods.
- The state of a checkbox is persisted on the server. As it could be checked or unchecked the state is checked on initialization and hides or shows the billing address fields accordingly.
- The same routine is performed each time the checkbox is clicked. The `check()` method checks to see if the checkbox is checked or not. If so, the billing address container is hidden, otherwise it's shown, by adding and removing the `hidden` class.

7. Check Your Answers Page

Even though we've collected all the information required to process the order, we should first let users review their order. As counterintuitive as this may sound, adding an extra step in the flow actually reduces effort and likely increases conversion.

Take Jack (I made him up), a father of two infants. It's the middle of the night, and his newborn baby is crying inconsolably. Naturally, Jack's tired and stressed. To make things worse, there are no more nappies.

He grabs the phone, adds nappies to the basket, fills out all the details and submits the order. Great — except it isn't. He ordered the wrong size nappies and paid with the wrong card. What Jack entered was valid, but still a mistake.

We can save Jack a lot of frustration by giving him the chance to review the order on a separate page. That way he can focus on the order details. Remember, filling out forms and checking information are two different mental contexts.

This also saves your (client's) business time and money. If Jack wants to cancel the order, then handling calls and processing would be costly — especially if the business offers free returns.

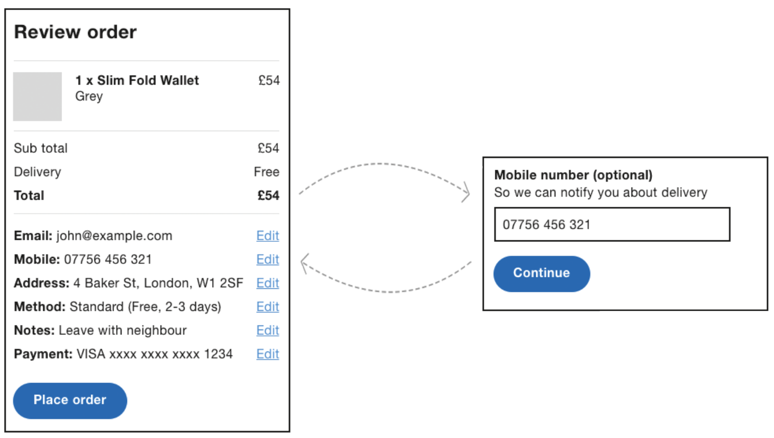
This shows that relying solely on completion time as a metric for success is dangerous. You should also look at how accurate people's orders are by checking how often items are returned.

As this is the final step in the flow, the button's text should be set to "Place order" or similar. Leaving it as "Continue" would mislead the user into thinking there is another step to complete, which is likely to result in more cancelled orders and strain on the customer service department. This would increase operating costs and shows that good design is also good for business.

MAKING CHANGES

Every piece of information gathered during checkout should be represented on the review page. Users shouldn't *have* to go back to check information — that would defeat the purpose of the page. Users should only need to go back if they spot a mistake.

Users can click Edit to make amendments, which is another advantage of using the one thing per page pattern. As pages are small they will load fast; as each page has just one thing on it, making a change is simple.



Left: the review page with edit links. Right: the user editing their mobile number, having spotted a mistake.

When the user makes a change, they are taken back to the 'check your answers' page again for another review, which puts users firmly in control and reduces stress and anxiety.

8. Confirmation Page

Confirmation pages are so much more than just confirming the order. Neglecting the user experience here is a great way to lose out on future business.

We've all probably experienced neglect after purchasing goods. For example, if you want to take out insurance, you

call the free sales number and are quickly put through to a helpful agent. Parting with money is usually made easy. But when you need to make a claim, it's more painful: the number isn't free and calls take a long time be answered. All very stressful.

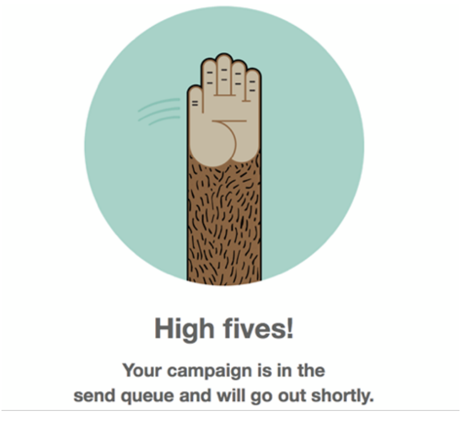
A confirmation page is the first opportunity to start forging a long-term relationship. And this is done in two ways: by looking after the user, and giving them an incentive to come back.

You should tell users what happens next, such as when delivery will take place, and what to do if something goes wrong.

It's also the best time to ask users to sign up (if they checked out anonymously). As we have most of the information to hand, we only need to ask for a password, making this step both optional and easy. Users should have had a good experience up to now, which should naturally encourage sign-up.

By giving users value, they'll hopefully want to sign up. By value, I mean something as simple as offering a faster checkout next time, or offering them a discount on their next order. Depending on the service, you might even ask users to tweet or Instagram their purchase in return for a voucher. Whatever it is, now's as good a time as any to mention it.

Up to now, we've also been sure to use plain and simple language for labels, hints, and errors. On the confirmation page, there is an opportunity to let your brand's personality shine through because the important stuff has already been done. MailChimp's confirmation page shows their mascot Freddie giving you a virtual high five. Nice.



MailChimp's confirmation screen has Freddie high-fiving.

Here's a checklist of things to consider including on the confirmation page:

- a reference number
- contact details
- what happens next and when
- what to do if something goes wrong
- ask users to sign up in return for something

- ask users to spread the word to get a voucher
- links to further information, if they might be useful
- a link to your feedback page

The Second-Time Experience

I've made so many purchases with Amazon that I can't even remember Amazon's first-time experience. That is to say, I've made hundreds of purchases as a second-time user, and only one as a first-time user.

Once everything is good with the first-time experience, it's time to focus on the second-time experience. As we have collected and stored all the user's information, we shouldn't have to ask for it again.

We can bypass most of the steps by sending users straight to the 'check your answers' page. This way, users get a reminder of their default preferences with the chance to make amends should they wish to.

All in all, this significantly reduces the effort and improves conversion. Remember Jack from earlier: the next time he's out of nappies, buying them should be a breeze, something I'm sure he'd appreciate.

Layout

Up to now, we've focused on the design of the form within each page, but we haven't considered the interface holistically. In fact, this is one of the dangers of composing interfaces out of predefined smaller components: the overall design can end up neglected.

Usually, checkout pages are given a special and more streamlined layout that helps reduce noise and keep users on-task. For example, the header usually contains a logo, security note, and accepted cards.

By omitting navigation and search, users can focus on checking out, which speaks to *inclusive design principle 6*, "Prioritize content."

INDICATING PROGRESS

Progress bars or indicators are often used within checkout because — at least in theory — they give users an idea of where they are and how long's left. Despite the sound reasoning, there isn't much evidence to show that progress bars are all that useful or even noticed. For example, the UK government's Carer's Allowance team removed a 12-step progress bar with no effect on completion rates or times.[26]

26 http://smashed.by/sharedspaces

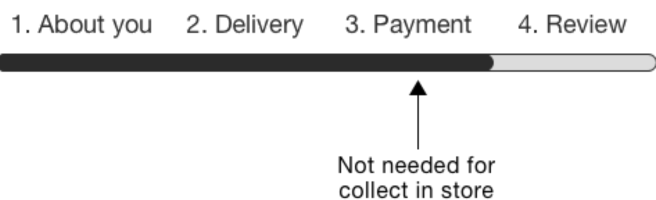
1. About you 2. Delivery 3. Payment 4. Review

A progress bar showing the user they are at step 4.

Progress bars pose some practical design challenges too. First, they take up a lot of space at the top of the page, which is particularly important on mobile where they can push the main content down. Second, fitting an accessible progress bar with clear labeling into a small viewport is nigh on impossible.

If that weren't enough, they're even trickier to design when the journey consists of conditional steps. Imagine a checkout that offers collection or delivery. If the user chooses to collect, they're taken down a different path where they won't need to give their payment details (as they'll pay in-store).

The problem is that a progress bar should tell the user what steps exist in advance. But the steps are based on users' answers. Either you show every possible step, which is misleading, or you update the progress bar (by removing or adding steps) as you go, which somewhat defeats the purpose of having one.



Misleading progress bar because the payment step isn't applicable when collecting in-store.

Not including a progress bar prioritizes the main content by moving it further up the page, which speaks to *inclusive design principle 6* (again), "Prioritize content." In any case, having meticulously designed the journey to be as simple as possible, users should get to the end quickly, which reduces the need for the progress bar.

For these reasons, consider starting without a progress bar. Then test your journey to see if users struggle. Remember, it's far easier (and cheaper) to add features than it is to remove them later on.

> *Note:* Some forms are especially long — a lot longer than a checkout flow. In this case, you might need some indication of progress, which is something we'll look at in chapter 10, "A Really Long and Complicated Form.")

Progress Step Text

If you decide to give users an indication of progress, first try adding the step number inside the heading:

```
<h1>Payment (Step 3 of 4)</h1>
```

Visual Progress Bar

If research shows that a more prominent progress bar is useful, then you can include one — but there are a few things to consider.

First, keep the text inside the `<h1>` so screen readers get a comparable experience (*inclusive design principle 1*). Second, you'll need to hide it from sighted users like this:

```
<h1>Payment <span class="visually-hidden">Step 3 of 4
</span></h1>
```

The `visually-hidden` class contains a special set of properties that hide the element visually, while still ensuring it's perceivable to screen reader users when the `<h1>` is announced.

```
.visually-hidden {
  border: 0!important;
  clip: rect(0 0 0 0)!important;
  height: 1px!important;
  margin: -1px!important;
```

```
  overflow: hidden!important;
  padding: 0!important;
  position: absolute!important;
  width: 1px!important;
}
```

As the visual progress bar is redundant for screen reader users, we can use the `aria-hidden="true"` attribute, which stops it being announced:

```
<!-- progress bar container -->
<div aria-hidden="true">
Visual progress bar here
</div>
```

Order Summary

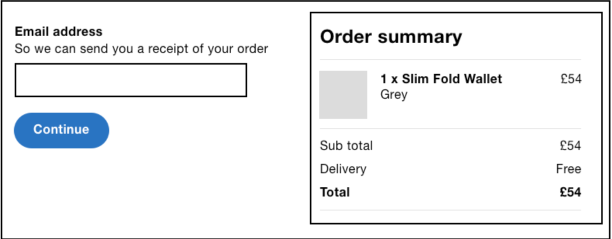
When you're shopping in a physical shop, you pick up items and place them in your shopping basket. Eventually, you checkout at the till. All the while, you can see what you're buying. Sometimes, at the last minute, you change your mind and take an item off the conveyor belt. Or you realize you forgot something and dash off to get it.

The system should always keep users informed about what is going on, through appropriate feedback.

— *Jakob Nielsen, "10 Usability Heuristics for User Interface Design"*[27]

27 https://smashed.by/usabilityheuristics

Giving users a comparable experience digitally is important. We can place an order summary panel on every page to keep users informed, without them having to remember what they're buying, freeing up their mental energy to focus on checking out.



The order summary panel gets populated as the user completes each screen. The email screen containing just the basket.

The mobile number screen, now containing the email address previously populated.

As the user completes each step, the order summary will be populated with more information. For example, on the email address screen (step 1), the summary only shows what they're buying. On the mobile number screen (step 2), the summary will also be populated with their email address, and so forth. If the user spots a mistake, they can jump back to any previous step by clicking the Edit link — just like the 'check your answers' page.

While the summary panel is important, it's less important than the form. So it should be placed beside the form on desktop, where there's enough space to do so, and below the form on mobile, where there's not enough room.

BACK LINKS

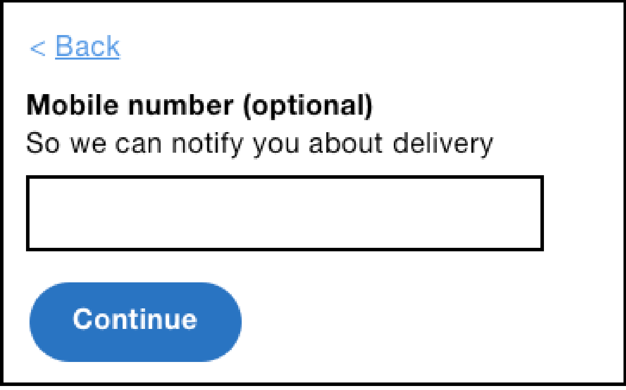
As the user is following a linear flow, we need to consider the need to step back. The browser's back button provides this functionality for free, but some people mistrust it because of bad past experiences when their data was lost.

Ajax-driven and modal-heavy sites haven't helped matters, because clicking the browser's back button often goes against user's expectations.[28] Thankfully, the checkout doesn't need Ajax or modal dialogs so this isn't a problem, but we still need to ensure the browser's back button works

28 http://smashed.by/backbutton

as expected. Users expect it to take them to the previous page in the state they left it.

Research might show you that it's useful to include a back link within the interface itself, and that users will be more inclined to trust it. In this case, position the link at the top left of the page. By placing it at the top of the page, users can see that they can go back if they need to. And they're less likely to fill out the form before hitting back and losing their data.



The back link positioned at top-left of the form.

Summary

In this chapter, we started out by looking at the one thing per page pattern, which helps to break down large forms into small chunks, making it easy for users to fill out and make amendments.

We then looked at capturing optional information, making choices with radio buttons, entering long-form content via the `textarea` and several ways to improve the payment form experience.

After that, we looked at other issues: giving users the ability to review their order, improving the experience for second-time users, and considering the overall design.

Demos

- **Optional telephone field**: http://smashed.by/telinputdemo
- **Delivery radio buttons**: http://smashed.by/deliveryoptionsdemo
- **Delivery notes with countdown**: http://smashed.by/charcountdowndemo
- **Payment form**: http://smashed.by/paymentformdemo

CHECKLIST

- Ask questions in a sensible order.
- Let users checkout anonymously.
- The width of the field should match the required input when the length is known.
- Use a `fieldset` and `legend` to give radio button and checkbox groups an accessible label.
- Add extra questions if they add value. Remember, completion time is not the only useful metric for success.
- Let users check their answers before submission.
- The confirmation page is the end of the transaction but the start of the relationship.
- Store people's information to improve the second-time experience and increase conversion.
- Don't break the back button.

A Flight Booking Form

In this chapter, we'll design a flight booking service. At first glance this may seem a bit niche, especially when compared to "A Registration Form" and "A Checkout Form." However, we're going to explore several complex problems that, in the end, will result in **reusable patterns** — patterns that are very much transferable to other problem domains, such as booking a cinema ticket, or even a hotel room.

Booking a flight consists of many discrete steps. The first few steps simply collect the user's preferences: where to fly, when to fly, and who's flying. Once we have that, we can give users a choice of available flights, followed by choosing where to sit. Finally, users will have to make payment (we covered payment patterns in the previous chapter).

1. Where to Fly

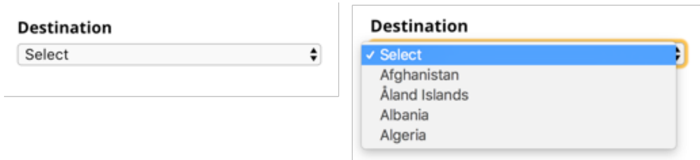
First, users have to choose an origin and destination; that is, places to fly from and to. Without this information, the service can't offer any flights. What's the best way of asking users for this information?

We should try to use the features that are native to the browser. They are familiar (by convention) and fully accessible out of the box. They also require far less work to implement.

You'd be forgiven for thinking you were spoiled for choice when it comes to form controls: select boxes, radio buttons, text boxes and, more recently, datalists. The choice is yours — except it isn't. Not all of these options are suitable. Let's look at some of the pros and cons of each.

SELECT BOX

Also known as dropdown menus, select boxes hide options behind a menu. Clicking the select box reveals the options. Once an option is selected, the menu collapses. Select boxes are often used for their space-saving qualities. What's most interesting, though, is why we need to save space in the first place.



Destination field as a select box. Left: collapsed. Right: expanded with options showing.

Often an interface is crammed with features, usually to please stakeholders, not users. It's understandable, then, that learning ways to hide discrete pieces of an interface has become part of a designer's skill set. But design is about so much more than saving space. After all, if an interface really is crammed, then our first job as designers is to declutter it.

In her talk "Burn Your Select Tags," Alice Bartlett shares the user research she undertook at the UK's Government Digital Service (GDS).[1]

In short, select boxes are hard to use. Besides hiding options behind an unnecessary extra click, users generally don't understand how they work. Some users try to type into them, some confuse focused options with selected ones. And, if that weren't enough, users can't pinch-zoom the options on certain devices.

Usability expert, Luke Wroblewksi wrote an article called "Dropdowns Should be the UI of Last Resort."[2] In it, he suggests some better alternatives, some of which we'll discuss later in this chapter.

1 http://smashed.by/burnselecttags
2 http://smashed.by/dropdownlastresort

Destination

- Afghanistan
- Åland Islands
- Albania

Destination field using radio buttons.

RADIO BUTTONS

Radio buttons, unlike select boxes, are generally well understood and easy to use, not least because they don't hide options. They are exposed, making them easy to compare, scan, and select. They're also malleable; that is, they let us use whatever content, in whatever format we want, inside the related label (more on that shortly).

The problem with radio buttons is that they're less suitable when there are many options. An airline could fly to hundreds of destinations, making the page long and hard to scan. This in turn means users have to scroll a lot more.

Don't get me wrong, users are more than happy to scroll, and we shouldn't use this as a crutch for changing course.[3] But if we can eliminate the need to scroll without introducing new problems, we should.

3 http://smashed.by/ppldontscrollmyth

SEARCH INPUT

A search box (`<input type="search">`) is similar to a regular text box (`<input type="text">`). A search box, however, lets users clear the field by tapping a delete icon, or pressing **Escape** when focused. With a text box, you have to select the text and press **Delete**, which takes a little longer.

Destination field as a search input.

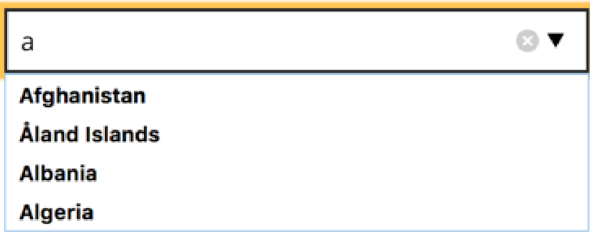
Using a search box is useful when searching a large amount of dynamic data, such as searching Amazon's product catalog. Airlines, however, fly to a finite set of destinations known in advance of the user searching. Letting users search unassisted like this could easily result in a page with no results, due to typos or a data mismatch.

DATALIST

Users need a control that lets them filter a long list of destinations. A control that marries the flexibility of a text box with the assurance of a select box. This type of control goes by many different names, including: type ahead, predictive search, and combo box; but we'll refer to it as an *autocomplete control*.

Autocomplete controls work by filtering options (destinations in this case) as the user types. As suggestions appear, users can select one quickly, automatically completing the field. This saves users having to scroll (unless they want to), while also being able to forgive small typos.

HTML5's `<datalist>` combines with a text box (`<input type="text">`) to create a native autocomplete control, which is unfortunately too buggy for use on the open web.[4] However, if your project is locked down to a few known browsers that don't have these bugs, then a native solution might work for you.



Destination field as a datalist.

But, having already defined our design principles in the introduction, we know we want to design an inclusive experience — one that works for as many people as possible, no matter their choice of browser or mobile device.

4 http://smashed.by/datalistcaniuse

AN AUTOCOMPLETE CONTROL

By creating a custom autocomplete component from scratch, there's an opportunity to create a powerful experience that also allows for common typos and endonyms (discussed later). A word of warning though: we're going to break new ground. Designing a robust and fully inclusive autocomplete control is hard work — but that's what our job is all about.

Do the hard work to make it simple.

— *GDS Design Principle 4*

Accessibility expert Steve Faulkner has what he calls a *punch list*, which is a set of rules to make sure that any custom JavaScript component is designed and built to a good standard.[5] The rules state that a component should:

- work without JavaScript
- be focusable with the keyboard
- be operable with the keyboard
- work with assistive devices

5 http://smashed.by/webcomponents

The Basic Markup

To satisfy the first rule we need to make sure the interface works in the absence of JavaScript. This means starting with a native form control that browsers provide for free.

Having already discussed the options above, we know that there are too many options for radio buttons; a search box requires an unnecessary round-trip to the server, and can lead to zero results; and the datalist is too buggy. By process of elimination, we're left with a select box.

```
<div class="field">
  <label for="destination">
    <span class="field-label">Destination</span>
  </label>
  <select name="destination" id="destination">
    <option value="">Select</option>
    <option value="1">France</option>
    <option value="2">Germany</option>
    <option value="3">Spain</option>
  </select>
</div>
```

The Enhanced Markup

When JavaScript is available, the `Autocomplete()` constructor function will enhance the basic HTML to look like this:

```
<div class="field">
  <label for="destination">
    <span class="field-label">Destination</span>
  </label>
  <select name="destination" aria-hidden="true"
tabindex="-1" class="visually-hidden">
    <!-- options here -->
  </select>
  <div class="autocomplete">
    <input aria-owns="autocomplete-options--destination"
autocapitalize="none" type="text" autocomplete="off"
aria-autocomplete="list" role="combobox" id="destination"
aria-expanded="false">
    <svg focusable="false" version="1.1" xmlns="http://www.
w3.org/2000/svg">
      <!-- rest of SVG here -->
    </svg>
    <ul id="autocomplete-options--destination"
role="listbox" class="hidden">
      <li role="option" tabindex="-1" aria-selected="false"
data-option-value="1" id="autocomplete_1">
        France
      </li>
      <li role="option" tabindex="-1" aria-selected="true"
data-option-value="2" id="autocomplete_2">
        Germany
      </li>
      <!-- more options here -->
    </ul>
    <div aria-live="polite" role="status" class="visually-hidden">
      13 results available.
    </div>
  </div>
</div>
```

Select Box and Text Box Notes

Even though users will no longer interact with the select box, we can't remove it completely. If we were to remove the select box from the document (or hide it with `display: none;`) then its value wouldn't be sent to the server on submission. This is important because the text box value differs from the select box value that will be submitted.

Hiding the select box while still having its value submitted involves a number of techniques in combination. The `visually-hidden` class and `aria-hidden="true"` attribute (as first set out in "A Checkout Form") hide the select box from sighted and screen reader users respectively. The `tabindex="-1"` attribute stops keyboard users from being able to focus it.

Note that the select box `id` attribute is transferred over to the text box because the label must be associated to it so it's read out in screen readers, and to increase the hit area of the control (as explained in chapter 1). The select box, however, no longer needs an `id` — it has effectively become a hidden input. Conversely, the `name` attribute isn't needed on the text box because its value isn't sent to the server — it's purely for interaction purposes and is used as a proxy to set the select box value.

The `role="combobox"` attribute means the control is announced as a combo box instead. A combo box, according to MDN, is "an edit control with an associated list box that provides a set of predefined choices."[6] The `aria-autocomplete="list"` attribute tells users that a list of options will appear. The `aria-expanded` attribute tells users whether the menu is expanded or collapsed by toggling its value between `true` and `false`.

The `autocomplete="off"` attribute stops browsers from showing their own suggestions, which would interfere with those offered by the component itself. Finally, the `autocapitalize="none"` attribute stops browsers from automatically capitalizing the first letter, something we'll look at in detail in the next chapter.

The SVG icon is layered on top of the text box using CSS. Note the `focusable="false"` attribute, which fixes the issue that in Internet Explorer SVG elements are focusable by default.

Menu Notes

The `role="list"` attribute is used to communicate the menu as a list, because it will be populated with a list of options. Each option has a `role="option"` attribute.

6 http://smashed.by/combobox

The `aria-selected="true"` attribute tells users which option within the list is selected or not by toggling the value between `true` and `false`.

The `tabindex="-1"` attribute means focus can be set to the option programmatically when users press certain keys. We'll look at keyboard interaction later.

Finally, the `data-option-value` attribute stores the select box option value. When the user clicks an autocomplete option, the select box value is updated accordingly to keep them in sync. This is what ties the enhanced interface (what the user sees) with the select box (what the user can't see) that's used to communicate to the server when the form is submitted.

Live Region

Sighted users will see the suggestions appear in the menu as they type, but the act of populating the menu isn't determinable to screen reader users without leaving the text box to explore the menu.

To provide a comparable experience (*inclusive design principle 1*), we can use a live region as first laid out in "A Checkout Form" (see page 92). As the menu is populated, we'll also populate the live region with how many results are available; for example, "13 results available." With this information at

hand, users can continue typing to narrow the results further, or they can select a suggestion from the menu.

As the feedback is only useful to screen reader users, it's hidden with the `visually-hidden` class again.

Typing into the Text Box

When the user types into the text box, we need to listen out for certain keys by using JavaScript.

```
Autocomplete.prototype.createTextBox = function() {
  // ...
  this.textBox.on('keyup', $.proxy(this, 'onTextBoxKeyUp'));
  // ...
};
Autocomplete.prototype.onTextBoxKeyUp = function(e) {
  switch (e.keyCode) {
    case this.keys.esc:
    case this.keys.up:
    case this.keys.left:
    case this.keys.right:
    case this.keys.space:
    case this.keys.enter:
    case this.keys.tab:
    case this.keys.shift:
    // ignore these keys otherwise the menu will show briefly
      break;
    case this.keys.down:
      this.onTextBoxDownPressed(e);
      break;
    default:
      this.onTextBoxType(e);
  }
};
```

The `this.keys` object is a collection of key codes (numbers) that correspond to particular keys by their names. This is to avoid magic numbers, which makes the code easy to understand at a glance.[7]

The switch statement filters out the **Escape**, **Up**, **Left**, **Right**, **Space**, **Enter**, **Tab**, and **Shift** keys. If we didn't, the default case would run and incorrectly show the menu. Instead of filtering out the keys we aren't interested in responding to, we could have specified the keys that we *are* interested in. But this would mean specifying a huge range of keys, which would increase the chance of one being missed, creating a broken experience.

We're mainly interested in the last two statements; that is, when the user presses **Down**, or the default case above, which means everything else (a character, number, symbol, and so on). In this case the `onTextBoxType()` function will be called.

```
Autocomplete.prototype.onTextBoxType = function(e) {
  // only show options if user typed something
  if(this.textBox.val().trim().length > 0) {
    // get options based on value
    var options = this.getOptions(this.textBox.val().
trim().toLowerCase());
    // build the menu based on the options
    this.buildMenu(options);
```

7 http://smashed.by/magicnumber

```
      // show the menu
      this.showMenu();
      // update the live region
      this.updateStatus(options.length);
    }
    // update the select box value which
    // the server uses to process the data
    this.updateSelectBox();
  };
```

The `getOptions()` method filters the options based on what the user typed. We'll look at the the filter function later.

Controls Should Have a Single Tab Stop

The autocomplete control is what's known as a composite. That just means it's made up of several different interactive and focusable parts — in this case, the text box and menu. What's important is that composite components should have one tab stop. Here's what the WAI-ARIA Authoring Practices 1.1 specification has to say on the subject:[8]

A primary keyboard navigation convention common across all platforms is that the tab and shift+tab keys move focus from one UI component to another while other keys, primarily the arrow keys, move focus inside of components that include multiple focusable elements. The path that the focus follows when pressing the tab key is known as the tab sequence or tab ring.

8 http://smashed.by/generalnav

A radio button group, for example, is a composite control that has one tab stop. Once the first radio button is focused, users can use the arrow keys to move between the options. Pressing **Tab** at anytime from within the group moves focus to the next focusable control in the tab sequence.

The text box within our autocomplete control is naturally focusable by the **Tab** key. Once focused, the user will be able to press the arrow keys to traverse the menu, which we'll look at shortly. Pressing **Tab** when the text box or menu option is focused should hide the menu to stop it from obscuring the content beneath when not in use. We'll look at how to do this next.

> *Note*: One way to make a composite control have just a single tab stop is to use the `aria-activedescendant` attribute.[9] It works by keeping focus on the component's container at all times, but referencing the currently active element. This doesn't work for the autocomplete component because the text box is a sibling (not a parent) of the menu.

9 http://smashed.by/activedescendant

Hiding the Menu on Blur Is Problematic

The `onblur` event is triggered when the user leaves an in-focus element. In the case of the autocomplete, we could listen to this event on the text box. The virtue of using the `onblur` event is that it will be triggered when the user leaves the field by pressing **Tab** or by clicking or tapping outside the element.

```
this.textBox.on('blur', function(e) {
  // hide menu
});
```

The problem with this approach is that the act of moving focus to the menu (even programmatically like we will) triggers the blur event, which subsequently hides the menu. This would make the menu inaccessible with the keyboard.

One workaround involves using the `setTimeout()` function, which allows us to put a delay on the event. In turn, the delay gives us time to cancel the event (using `clearTimeout()`) should the user move focus to the menu within that time. This would stop the menu being hidden, making it accessible again.

```
this.textBox.on('blur', $.proxy(function(e) {
  // set a delay before hiding the menu
  this.timeout = window.setTimeout(function() {
    // hide menu
  }, 100);
}, this));
this.menu.on('focus', $.proxy(function(e) {
  // cancel the hiding of the menu
  window.clearTimeout(this.timeout);
}, this));
```

Unfortunately, there's a problem with the blur event in iOS 10. It incorrectly triggers the blur event on the text box when the user hides the on-screen keyboard. This stops users from accessing the menu altogether. There's another solution which we'll look at next.

Listening to the Tab Key

Instead of hiding the menu on blur, we can use the `keydown` event to listen out for when the user presses the **Tab** key.

```
this.textBox.on('keydown', $.proxy(function(e) {
  switch (e.keyCode) {
    case this.keys.tab:
      // hide menu
      break;
  }
}, this));
```

Unlike the `blur` event, this approach doesn't cover the case where users blur the control by clicking outside of it. We have to handle this case manually by listening to the document's `click` event, but being careful to work out what's clicked — we don't want to hide the menu if the user clicks *within* the control.

```
$(document).on('click', $.proxy(function(e) {
  if(!$.contains(this.container[0], e.target)) {
    // hide the menu
  }
}, this));
```

The event handler is using jQuery's `contains()` method, which checks to see if what the user clicked (`e.target`) falls outside the container (`this.container[0]`). If it's outside, the menu is hidden. The `[0]` is used because the `contains()` method takes element nodes, not jQuery objects.

Moving to the Menu (Pressing Down)

When the text box is focused, pressing the **Down** key triggers `onTextBoxDownPressed()` like this:

```
Autocomplete.prototype.onTextBoxDownPressed = function(e) {
  var option;
  var options;
  var value = this.textBox.val().trim();
```

```
  /*
    When the value is empty or if it exactly
    matches an option show the entire menu
  */
  if(value.length === 0 || this.isExactMatch(value)) {
    // get options based on the value
    options = this.getAllOptions();
    // build the menu based on the options
    this.buildMenu(options);
    // show the menu
    this.showMenu();
    // retrieve the first option in the menu
    option = this.getFirstOption();
    // highlight the first option
    this.highlightOption(option);
  /*
    When there's a value that doesn't have
    an exact match show the matching options
  */
  } else {
    // get options based on the value
    options = this.getOptions(value);
    // if there are options
    if(options.length > 0) {
      // build the menu based on the options
      this.buildMenu(options);
      // show the menu
      this.showMenu();
      // retrieve the first option in the menu
      option = this.getFirstOption();
      // highlight the first option
      this.highlightOption(option);
    }
  }
};
```

If the user presses **Down** without having typed anything, the menu will populate with every available option, and focus to the first menu option. The same thing will happen if the user types an exact match; this should be rare because most users who notice the suggestion will select it — it's quicker that way.

The `else` condition will populate the menu with options that match (if any), and again will focus the first menu option. At the end of both scenarios the `highlightOption()` method is called, which we'll look at later.

Scrolling the Menu

As mentioned earlier, the menu may contain hundreds of options. To ensure the menu stays visible within the viewport, we need to use CSS.

```
.autocomplete [role=listbox] {
  max-height: 12em;
  overflow-y: scroll;
  -webkit-overflow-scrolling: touch;
}
```

The `max-height` property works by letting the menu grow up to a maximum height of `12em`. Once the content inside the menu surpasses that height, users can scroll the menu thanks to the `overflow-y: scroll` property.

The last property is non-standard and is used to enable momentum scrolling on iOS. This ensures the autocomplete control scrolls the same way as it would everywhere else.

Clicking an Option

Clicking or tapping a menu option should perform a number of discrete tasks, but before we get to them, let's discuss how we might listen to the click event.

The most basic approach involves adding a click event to each of the options individually. But this is problematic for two reasons.

First, each added event must be stored in memory. As there are hundreds of option, they'll use a lot of memory which may impact performance. Second, the menu options are constantly being updated as the user types. This means events need to be constantly added and removed, which is computationally intensive and bothersome to manage with code.

Instead, we can use event delegation, which is made possible by the concept of event bubbling.[10] Events originating from lower down the document tree propagate (bubble up) to the parent container, all the way up to the document root.

10 http://smashed.by/eventdelegation

In our particular case, we can add a single event listener to the menu's container and filter out all events that don't match the option elements we're interested in. To do this, we can use jQuery's `on()` method, which has event delegation built in.

```
Autocomplete.prototype.createMenu = function() {
  //...
  this.menu.on('click', '[role=option]', $.proxy(this,
'onOptionClick'));
  //...
};
```

The click event is bound to the container (`this.menu`), but will only trigger the event handler (`onOptionClick()`) when the event originated on an element with `role="option"`.

```
Autocomplete.prototype.onOptionClick = function(e) {
  var option = $(e.currentTarget);
  this.selectOption(option);
};
```

The event handler retrieves the option (`e.currentTarget`) and hands it off to the `selectOption()` method. Normally, we'd reference `e.target`, but as we're using event delegation, `e.target` would return the delegate (`this.menu`) which

isn't helpful. Whenever you're using event delegation, you'll almost definitely be interested in the originating element (`e.currentTarget`).

```
Autocomplete.prototype.selectOption = function(option) {
  var value = option.attr('data-option-value');
  this.setValue(value);
  this.hideMenu();
  this.focusTextBox();
};
```

The `selectOption()` function takes the option to be selected and extracts the `data-option-value` attribute. That value is passed to the `setValue()` method which populates the text box and hidden select box. Finally, the menu is hidden and the text is focused.

This same routine is performed when the user selects an option with the **Space** or **Enter** keys. We'll look at the menu interactions next.

Menu Keyboard Interaction

Once focus is within the menu (by pressing **Down** while the text box is focused), we need to let users traverse the menu with the keyboard. To do this, we'll listen to the `keydown` event.

```
Autocomplete.prototype.createMenu = function() {
  this.menu.on('keydown', $.proxy(this, 'onMenuKeyDown'));
};

Autocomplete.prototype.onMenuKeyDown = function(e) {
  switch (e.keyCode) {
    case this.keys.up:
     // Do stuff
      break;
    case this.keys.down:
     // Do stuff
      break;
    case this.keys.enter:
     // Do stuff
      break;
    case this.keys.space:
     // Do stuff
      break;
    case this.keys.esc:
     // Do stuff
      break;
    case this.keys.tab:
     // Do stuff
      break;
    default:
     this.textBox.focus();
  }
};
```

Key	Action
Up	If the first option is focused, set focus to the text box; otherwise set focus to the previous option.
Down	Focus the next menu option. If it's the last menu option, do nothing.
Tab	Hide the menu.
Enter or **Space**	Select the currently selected option and focus the text box.
Escape	Hide the menu and set focus to the text box.
Any other character	Focus the text box so users can continue typing.

The Highlight Function

As noted above, the user can focus an option by pressing the **Up** or **Down** keys. When this happens, the `highlightOption()` method is called.

```
Autocomplete.prototype.highlightOption = function(option) {
  // if there's a currently selected option
  if(this.activeOptionId) {
    // get the option
    var activeOption = this.getOptionById(this.
activeOptionId);
    // unselect the option
    activeOption.attr('aria-selected', 'false');
  }
  // set new option to selected
  option.attr('aria-selected', 'true');
```

```
  // If the option isn't visible within the menu
  if(!this.isElementVisible(option.parent(), option)) {
    // make it visible by setting its position inside the
menu
    option.parent().scrollTop(option.parent().scrollTop() +
option.position().top);
  }
  // store new option for next time
  this.activeOptionId = option[0].id;
  // focus the option
  option.focus();
};
```

The method performs a number of discrete steps. First, it checks to see if there's a previously active option. If so, the `aria-selected` attribute is set to `false`, which ensures the state is communicated to screen reader users. Second, the new option's `aria-selected` attribute is set to `true`.

As the menu has a fixed height, there's a chance that the newly focused option is out of the menu's visible area. So we check whether this is the case using the `isElementVisible()` method. If it's not visible, the menu's scroll position is adjusted using jQuery's `scrollTop()` method, which makes sure it's in view.

Next, the new option is stored so that it can be referenced later when the method is called again for a different option. And finally, the option is focused programmatically to ensure its value is announced in screen readers.

To provide feedback to sighted users we can use the same `[aria-selected=true]` CSS attribute selector like this:

```
.autocomplete [role=option][aria-selected="true"] {
  background-color: #005EA5;
  border-color: #005EA5;
  color: #ffffff;
}
```

Tying state and its visual representation together is a good thing because it ensures that state changes are communicated interoperably. Form should follow function, and doing so directly keeps them in-sync.

The Basic Filter Function

Having looked at the main interaction flows and the routines that run off the back of them, we can look more closely at the filtering mechanism. A good filter is designed to forgive small typos and letter casing. It's worth reminding ourselves again that the data driving the suggestions resides in the select box `<option>` elements.

```
<select>
  <option value="">Select</option>
  <option value="1">France</option>
  <option value="2">Germany</option>
  <option value="3">Spain</option>
</select>
```

As noted above, the `getOptions()` method is called when we need to populate the menu with matching options.

```
Autocomplete.prototype.getOptions = function(value) {
  var matches = [];
  // Loop through each of the option elements
  this.select.find('option').each(function(i, el) {
    // if the option has a value and the option's text node
matches the user-typed value
    if($(el).val().trim().length > 0 && $(el).text().
toLowerCase().indexOf(value.toLowerCase()) > -1) {
      // push an object representation to the matches array
      matches.push({
        text: $(el).text(),
        value: $(el).val()
      });
    }
  });
  return matches;
};
```

The method takes the user-entered value as a parameter. It then loops through each of the `<option>`s and compares the value to the option's text content (the bit inside the element). It does so by using `indexOf()` which checks to see if the string contains an occurrence of the specified value. This means users can type incomplete parts of countries and still have relevant suggestions presented to them.

The value is trimmed and converted to lowercase, which means options will still be shown if the user has, for example, turned on caps lock on their keyboard. Users shouldn't have to fix problems we can fix for them automatically.

Each matched option is added to the `matches` array, which will be used by the calling function to populate the menu accordingly.

Supporting Endonyms and Common Typos

An endonym is a name used by the people from a particular area of that area (or themselves or their language). For example, Germany in German is "Deutschland." We can follow *inclusive design principle* 5, "Offer choice," by letting users type an endonym.

To do this, we first need store it somewhere. We can put the endonym inside a data attribute on the `<option>` element.

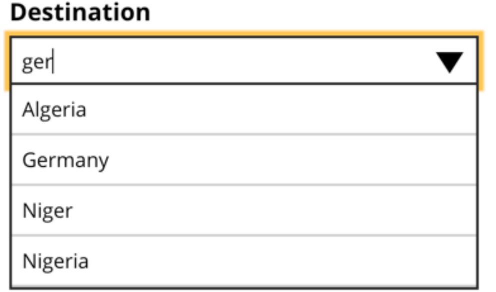
```
<select>
  <!-- options -->
  <option value="2" data-alt="Deutschland">Germany</option>
  <!-- options -->
</select>
<select>
```

With the select box ready, we can change the filter function to check the alternative name like this:

```
Autocomplete.prototype.getOptions = function(value) {
  var matches = [];
  // Loop through each of the option elements
  this.select.find('option').each(function(i, el) {
  // if the option has a value and the option's text node
  matches the user-typed value or the option's data-alt
  attribute matches the user-typed value
    if( $(el).val().trim().length > 0
      && $(el).text().toLowerCase().indexOf(value.
toLowerCase()) > -1
      || $(el).attr('data-alt')
      && $(el).attr('data-alt').toLowerCase().indexOf(value.
toLowerCase()) > -1 ) {
      // push an object representation to the matches array
      matches.push({
        text: $(el).text(),
        value: $(el).val()
      });
    }
  });
  return matches;
};
```

The attribute isn't reserved for endonyms — it can be used to store common typos too.

How It Might Look



The autocomplete component showing suggestions as the user types.

2. When to Fly

Dates are notoriously hard: different time zones, formats, delimiters, days in the month, length of a year, daylight savings, and on and on.[11] It's hard work designing all this complexity *out* of an interface.

Often, three select boxes are used: one for day, month, and year. Admittedly, we've just discussed the cons of select boxes, but it must be said that one of their redeeming qualities is that they help users enter the right information. But in the case of dates, even *this* quality doesn't hold up because you can select an invalid date, such as 31 February 2017.

11 http://smashed.by/falsehoodstime

A date of birth field made up of three select boxes: day, month and year.

Select boxes are also used to avoid locale and formatting differences. Some dates start with month, others with day. Some delimit dates with slashes, others with dashes or dots. We can't reliably determine users' intention based on what they enter. It's just one of those things.

"What date is this?"

A text box populated with "10/09/12" which could be one of several dates depending on the format.

Many of us assume that using a calendar widget is always better than letting users type freely into a text box. But this is not always the case. The GDS Service Manual states:[12]

12 http://smashed.by/dateselector

The way you should ask users for dates depends on the types of date you're asking for.

Let's walk through some of the main types of dates to see what interface is best for users. Then we can see if any of those are suited to the context of our problem: choosing a date to fly on.

DATES FROM DOCUMENTS

Here's what GDS says about asking for dates found on documents and other physical items users may need to reference:

If you ask for a date exactly as it's shown on a passport, credit card or similar item, make the fields match the format of the original. This will make it easier for users to copy it across accurately.

The expiry date from chapter 2, "Checkout," falls under this category. As the expiry date is just four characters with an optional slash, we gave users a single text box that matches the expected format. Essentially, users just copy what they see. Easy.

MEMORABLE DATES

A memorable date is one that you remember easily such as your date of birth. Typing six digits unassisted into a text box is much quicker than scrolling, swiping, and clicking

through multiple years, months, and days within a calendar. Memorable dates are best represented by three text boxes: one for day, month, and year. Why three? Because it solves the locale and formatting issues mentioned earlier.

Date of birth

DD MM YYYY

Day Month Year

Date of birth field made up of three text boxes: day, month, and year.

```
<fieldset class="field">
  <legend>
    <span class="field-legend">Date of birth</span>
    <span class="field-hint">DD MM YYYY</span>
  </legend>
  <div class="field-dayWrapper">
    <label for="day">Day</label>
    <input class="field-dayBox" type="text"
pattern="[0-9]*" name="day" id="day">
  </div>
  <div class="field-monthWrapper">
    <label for="month">Month</label>
    <input class="field-monthBox" type="text"
pattern="[0-9]*" name="month" id="month">
  </div>
  <div class="field-yearWrapper">
    <label for="year">Year</label>
    <input class="field-yearBox" type="text"
pattern="[0-9]*" name="year" id="year">
  </div>
</fieldset>
```

The three fields are wrapped in a `fieldset`. The `legend` (“Date of birth”) gives each text box context and would be read out as “Date of birth, day” (or similar) in screen readers.

> *Note*: The `pattern` attribute is used to trigger the numeric keyboard — a little enhancement for iOS users. If you’re wondering about why we haven’t used the number input, you can refer back to the number input in “A Checkout Form.”

A DATE PICKER

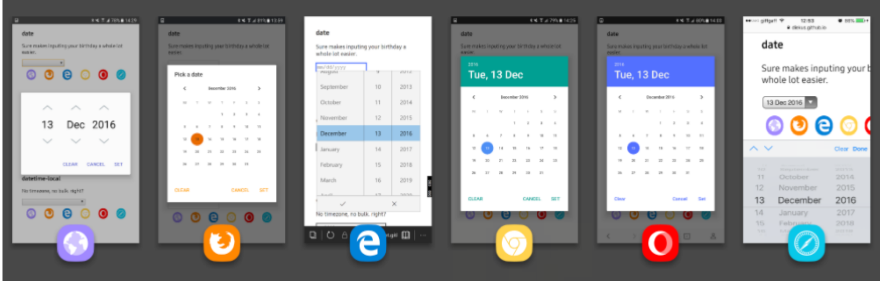
When choosing a date to fly on, users are neither entering a memorable date nor one found in a document. They are searching for a date sometime in the future and usually within the next few months.

We tend to think of time in structured chunks: days, weeks, and months, and so on. And we plan our time using calendars which align with that notion. It’s sensible then, to let users find and pick a date through a familiar and intuitive calendar interface, or what’s commonly known as a date picker.

As usual, our first port of call is to see if there’s a date picker control that browsers provide natively for free. Good news: there is. The date input (`<input type="date">`) offers a

special and convenient interface for picking dates while also enforcing a standard format for the value that's sent to the server on submission.

Mobile browser support is really good and includes Samsung's browser, Firefox, Edge, Chrome, Opera, and Safari. Desktop support is patchier: Chrome and Edge support it, but Internet Explorer and Safari don't (at time of writing). We'll look at how to support them later.



A selection of date pickers on different browsers.

As the date picker is provided by the browser, you'll notice how it looks a lot like the system date picker that's used for setting dates and times on your phone. That's by design so that mobile browsers can outsource the problem to native components. This is good because users will be familiar with it, which speaks to *inclusive design principle 3*, "Be consistent."

You might be concerned that they look different in different browser vendors. Don't be. Most users don't notice the difference and the rest don't care. Remember, most people use the same browser every day. They only see their platform's implementation. Unlike us, they're not agonizing over subtle differences during cross-browser testing.

Nobody cares about your website as much as you do.

— Goran Peuc, "Nobody Wants To Use Your Product"[13]

If you're not able to conduct your own user research, watch "Progressive Enhancement 2.0" (at about 29 minutes in).[14] Nicholas Zakas shows the audience a slide with a photo on it. He moves to the next slide which contains the same photo. He then asks the audience if they noticed any differences. Even though the second photo had a border and a drop shadow, not one person noticed.

Ironically, the audience was made up of designers and developers — people who are trained to notice these things. But they didn't notice them, because like any user, they were focused on the content, not the finer points of the visual aesthetic.

13 http://smashed.by/nobodywantsyourproduct
14 http://smashed.by/progenhance

The Basic Markup

This markup has been used many times already in the book. The only difference is that the input's `type` attribute is set to `date`.

```
<div class="field">
  <label for="departure">
    <span class="field-label">Departure date</span>
  </label>
  <input type="date" id="departure" name="departure">
</div>
```

Browsers that support the date input give users a standard date picker that's familiar, accessible, and performant by default. So that's good. But what happens in other browsers?

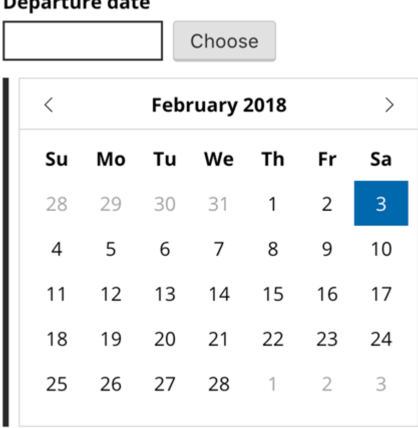
The date input will change into a basic text input — this is known as graceful degradation. While users won't get the convenience of a date picker, they'll still be able to enter a date. This is an example of HTML's inherently resilient nature. When things fails, they don't break.

Depending on your situation, this level of support may be fine. Perhaps entering a date is of low priority, or happens too infrequently to worry about. Or perhaps none of your users will ever use an unsupported browser. But finding a date is integral to the flight booking experience.

We ought to provide a better experience for people using an unsupported browser.

How It Might Look

The date picker consists of a toggle button that reveals the calendar. From there, users will be able to traverse the calendar and ultimately select a date to populate the text box.



The enhanced date picker interface shown in its expanded state, with a button to the right of the text field, and the calendar inline below it.

Notes about the Design

Many date pickers are designed as overlays, but they obscure the rest of the page, and are prone to being cropped by the viewport when positioned absolutely on top of the

interface. Instead, our calendar will be positioned underneath the input and inline, avoiding such issues.

There's an inset left border which visually connects the calendar to the field above. And the interactive elements within the calendar have large tap targets which are easier to tap and click.[15]

You might be tempted to try to squeeze additional information — such as price and availability — into each of the cells. This may be possible in very large viewports, but it's not practical from a responsive design perspective: there's simply not enough room to denote this information in small viewports. This is why it's important to design mobile-first. In any case, the primary user need at this stage of the journey is to select a date. Trying to squeeze in additional information is going to result in a slower, busier and overwhelming experience that slows users down.

Instead, we'll let users focus on choosing a date unencumbered, and later we'll give users more information when it's both useful and practical to do so.

15 http://smashed.by/sizetaptargets

Feature Detection

As we only want to give unsupported browsers the custom date picker component, the first thing we need to do is detect when and when not to initialize the component. Without a provision in place, users might see two opposing date pickers: the native one, and our own, which would be confusing and unnecessary.

We can check for support before enhancing the interface using a little feature detection script.

```
function dateInputSupported() {
  var el = document.createElement('input');
  try {
    el.type = "date";
  } catch(e) {}
  return el.type == "date";
}
```

The function works by trying to create a date input and then checking to see if its `type` attribute is correctly reported as a date input. In browsers that lack support, it will be reported as a text input instead. We can use this function to determine whether the `DatePicker()` component should be defined or not.

```
if(!dateInputSupported()) {
  var DatePicker = function() {
    // code here
  };
}
```

As the `DatePicker()` is only defined when there's no support for the native date input, we can check to see if it's defined before initializing it. This is known as a dynamic JavaScript API, because it changes based on support and is a crucial aspect of designing progressively enhanced interfaces.[16]

```
if(typeof DatePicker !== "undefined") {
 new DatePicker();
}
```

The Enhanced Markup

After the date picker has been initialized, the markup will have been changed to include the date picker controls: toggle button, next and previous month buttons, and the calendar grid.

16 http://smashed.by/featuredetection

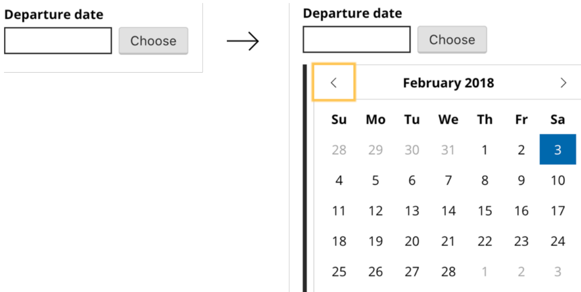
```
<div class="field">
  <label for="when">
    <span class="field-label">Date</span>
  </label>
  <div class="datepicker">
    <input type="text" id="when" name="when">
    <button type="button" aria-expanded="true" aria-
haspopup="true">Choose</button>
    <div class="datepicker-wrapper hidden">
      <!-- Calendar widget goes here -->
    </div>
  </div>
</div>
```

Notes

- The `type="button"` attribute stops the button from submitting the form. If the type was set to `submit` (or omitted altogether) when pressed, it would incorrectly submit the form.
- The `aria-haspopup="true"` attribute indicates that the button reveals a calendar. It acts as a warning that, when pressed, the focus will be moved to the calendar. Its value is always set to `true`.
- The `aria-expanded` attribute indicates whether the calendar is currently in an open (expanded) or closed (collapsed) state by toggling between `true` and `false` values.

Revealing the Calendar

The calendar starts hidden. When the toggle button is clicked, the calendar is revealed by removing the `hidden` class on the wrapper.



The date picker in its original state (left), and after revealing the calendar (right).

Immediately after, the focus is set to the Previous Month button, which is the first focusable element inside the calendar.

```
DatePicker.prototype.onToggleButtonClick = function() {
  // showing
  if(this.toggleButton.attr('aria-expanded') == 'true') {
    this.hide();
  // hiding
  } else {
    this.show();
    this.calendar.find('button:first-child').focus();
  }
};
```

```
DatePicker.prototype.hide = function() {
  this.calendar.addClass('hidden');
  this.toggleButton.attr('aria-expanded', 'false');
};
DatePicker.prototype.show = function() {
  this.calendar.removeClass('hidden');
  this.toggleButton.attr('aria-expanded', 'true');
};
```

The Calendar HTML

The container has two important attributes: the `role="group"` attribute groups the related calendar controls together. When the calendar is revealed, screen reader users will hear the button's label in combination with the group's label: "date picker, previous month, button."

```
<div class="datepicker-calendar" aria-label="date picker"
role="group">
  <div class="datepicker-actions">
    <button type="button" aria-label="previous month">
      <svg focusable="false" version="1.1"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 0 17
17" width="1em" height="1em">...</svg>
    </button>
    <div role="status" aria-live="polite">February 2018</div>
    <button type="button" aria-label="next month">
      <svg focusable="false" version="1.1"
xmlns="http://www.w3.org/2000/svg"
```

```
xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 0 17
17" width="1em" height="1em">...</svg>
    </button>
  </div>
  <!-- grid here -->
</div>
```

The month's title and year are placed within a live region (as first discussed in chapter 2). This means its content will be announced by screen readers when the calendar is revealed. This same information will be continually announced as users move between different months.

> *Note*: As mentioned earlier, the `focusable="false"` attribute on the SVG icon fixes the issue that in Internet Explorer SVG elements are focusable.

Previous and Next Month Buttons

With the calendar now revealed, the user can move between the Previous Month and Next Month buttons by using the **Tab** key. This is because we've used `<button>` elements, which are naturally focusable and part of the tab sequence.

Buttons are interoperable meaning they can be activated by clicking, tapping, or pressing **Space** or **Enter** with the keyboard. All we need to do is listen for the click event.

```
DatePicker.prototype.addEventListeners = function() {
  this.calendar.on('click', 'button:first-child',
$.proxy(this, 'onPreviousClick'));
  this.calendar.on('click', 'button:last-child',
$.proxy(this, 'onNextClick'));
};
DatePicker.prototype.onPreviousClick = function(e) {
  this.showPreviousMonth();
};
DatePicker.prototype.onNextClick = function(e) {
  this.showNextMonth();
};
```

The `showPreviousMonth()` and `showNextMonth()` methods (not shown) work out which month to show, and then update the title and grid HTML accordingly.

The Grid

The days of the month are presented in a grid format of which the `<table>` element is perfectly suited, so that's easy. But it's the careful arrangement of attributes that is crucial in not only enabling interaction but also understanding.

```
<table role="grid">
  <thead>
    <tr>
      <th aria-label="Sunday">Su</th>
      <th aria-label="Monday">Mo</th>
      <th aria-label="Tuesday">Tu</th>
      <th aria-label="Wednesday">We</th>
```

```
      <th aria-label="Thursday">Th</th>
      <th aria-label="Friday">Fr</th>
      <th aria-label="Saturday">Sa</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td tabindex="-1" aria-selected="false"
aria-label="4 February, 2018" role="gridcell"
data-date="Sun Feb 04 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">4</span>
      </td>
      <td tabindex="-1" aria-selected="false"
aria-label="5 February, 2018" role="gridcell"
data-date="Mon Feb 05 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">5</span>
      </td>
      <td tabindex="-1" aria-selected="false"
aria-label="6 February, 2018" role="gridcell"
data-date="Tue Feb 06 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">6</span>
      </td>
      <td tabindex="-1" aria-selected="false"
aria-label="7 February, 2018" role="gridcell"
data-date="Wed Feb 07 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">7</span>
      </td>
      <td tabindex="-1" aria-selected="false"
aria-label="8 February, 2018" role="gridcell"
data-date="Thu Feb 08 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">8</span>
      </td>
      <td tabindex="-1" aria-selected="false"
aria-label="9 February, 2018" role="gridcell"
data-date="Fri Feb 09 2018 00:00:00 GMT+0000 (GMT)">
```

```
        <span aria-hidden="true">9</span>
      </td>
      <td tabindex="-1" aria-selected="false" aria-
label="10 February, 2018" role="gridcell"
data-date="Sat Feb 10 2018 00:00:00 GMT+0000 (GMT)">
        <span aria-hidden="true">10</span>
      </td>
    </tr>
    <tr>...</tr>
    <tr>...</tr>
    <tr>...</tr>
  </tbody>
</table>
```

The `role="grid"` attribute (and each cell's `role="gridcell"` attribute) tells screen readers to treat the table as a grid. Without this, JAWS, for example, won't let users operate the calendar with the arrow keys using JavaScript. This is because the arrow keys are reserved for operating a standard table in a special way.

The `<thead>` contains the names of the days. Note that they're abbreviated, which should be avoided in most cases. But the calendar needs to fit on small viewports too. The unabbreviated heading is placed inside the `aria-label` attribute. Support is a little patchy, but it's a useful enhancement for browser/screen reader combinations that will pick this up, such as NVDA with Firefox.

Each cell contains a number (the date), which is perfectly adequate for sighted users as they can see the entire calendar. But screen reader users would only hear "Seventeen," which is ambiguous unless they carefully remember the previously announced month and year. To provide a comparable experience (*inclusive design principle 1*), we put the full date inside the `aria-label` attribute.

The `<span>` has an `aria-hidden="true"` attribute, which stops the number being read out twice by some screen readers, without hiding it from sighted users.

The `tabindex`, `aria-selected` and `data-date` attributes will be discussed shortly.

Clicking a Day

When the user clicks a day, a number of actions must be performed. The event handler looks like this:

```
DatePicker.prototype.onCellClick = function(e) {
  var d = new Date($(e.currentTarget).attr('data-date'));
  this.updateTextBoxDate(d);
  this.hide();
  this.input.focus();
  this.selectDate(d);
  this.selectedDate = d;
};
```

First, the date string (stored inside the cell's `data-date` attribute) is converted into a JavaScript Date object which is then used to populate the text box. Then the date picker is hidden and the text box is focused.

The `selectDate()` method will mark the selected cell by setting the `aria-selected` attribute to true, and by setting the previously selected cell to false.

Finally, the selected date is stored so we can show the calendar in the correct state if the user decides to pick another date.

Keyboard Interaction

Like the autocomplete component we designed earlier, the grid is a composite control made up of many interactive elements — as many as 31, depending on the month. As discussed earlier, composite controls should have just one tab stop: having to tab through 31 days is tiresome and inefficient.

To solve this problem, we're going to use the concept of *roving tabs*.[17] The way it works is that only one cell in the grid is focusable at any one time — the selected cell. The selected cell will have a `tabindex="0"` attribute, which means users can tab to it from the Next Month button.

17 http://smashed.by/tabindex

The rest of the cells have a `tabindex="-1"` attribute, which means focus can be set programmatically with JavaScript, but users won't be able to get to it with the **Tab** key.

Once the selected cell is focused, users can traverse the calendar with the arrow keys. As the user moves between the cells, the `tabindex` values will be updated to ensure that only the selected cell has a `tabindex` value of `0`.

```
DatePicker.prototype.addEventListeners = function() {
  // ...
  this.calendar.on('keydown', '[role=gridcell]',
$.proxy(this, 'onCellKeyDown'));
  // ...
};
DatePicker.prototype.onCellKeyDown = function(e) {
  switch(e.keyCode) {
    case this.keys.down:
      this.onDayDownPressed(e);
      break;
    case this.keys.up:
      this.onDayUpPressed(e);
      break;
    case this.keys.left:
      this.onDayLeftPressed(e);
      break;
    case this.keys.right:
      this.onDayRightPressed(e);
      break;
    case this.keys.space:
    case this.keys.enter:
      this.onDaySpacePressed(e);
      break;
  }
};
```

Note: As described earlier with the autocomplete component, we could have used the `aria-activedescendant` technique for the grid. However, the benefits of using roving tab indexes is that it's better supported and it ensures that the newly focused element is scrolled into view.

Key	Action
Down	Focus the same day in the subsequent week. If it's the last week, switch to the next month first.
Up	Focus the same day in the previous week. If it's the first week, switch to the previous month first.
Left	Focus the previous day. If it's the first day of the month, switch to the previous month first.
Right	Focus the next day. If it's the last day of the month, switch to the next month first.
Enter or **Space**	Performs the same actions as clicking the day: populate and focus the text box, and hide the menu.
Escape	Hide the calendar and focus the toggle button.

Note: While screen reader users can operate the calendar like this, it's probably easier and quicker for them to type a date directly into the text box. But inclusive design is about not making such assumptions. Instead we let the individual user decide (*inclusive design principle* 5).

Doing Our Best

Despite our efforts to support as many users as possible, there's a rare situation whereby users still won't get a date picker.

In chapter 1 we discussed the importance of progressive enhancement, because we can't be sure that JavaScript is always available. Users experiencing a network or JavaScript failure while also using a browser that doesn't support the native date will just see a text box, without any hint text regarding the format of the date.

The date picker without a hint, as seen when JavaScript is unavailable.

We can't use the hint pattern (from chapter 1) because browsers that support the date input use a different format. Of course, we should be as forgiving as possible, by letting users type slashes, periods, or spaces, but typing a two-digit year first, for example, will still cause an error. A well-written error message may have to suffice.

Another option would be to provide a hint via the `placeholder` attribute (and remove it when the date picker is initialized). Despite the many problems with placeholders (as discussed in "A Registration Form"), this might be the lesser of two evils.

Design is often a question of priorities. What is a good experience for most may create a less-than-ideal experience for some, which is especially the case on the web. Inclusive design is about making decisions that are unlikely to exclude people.

People ignore design that ignores people.

– Frank Chimero

In this rare situation, users are still able to enter a date which makes this pattern an accessible one. In the end, it's about doing our best and we've done that here.

Future Support

The web is constantly changing. Browsers and devices are released at a rapid rate, each with varying features and capabilities. This is why Jeremy Keith refers to the "*web as a continuum, not a platform.*"[18]

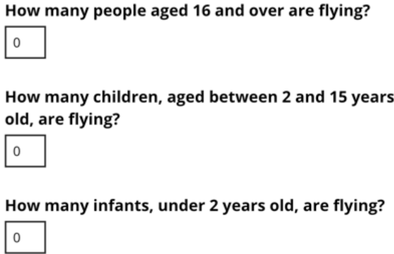
18 http://smashed.by/webcontinuum

We need to think about what level of support makes sense for our users depending on the feature at hand. Earlier, we decided to enhance the experience for browsers that don't support the native date input, which makes sense today.

As browser support improves, the number of people who would experience the degraded version will diminish; at which time we can remove our custom date picker code. Not only does this give us less to maintain, but users will get a faster experience as they don't need to download the code. Lovely.

3. Choosing Passengers

Next we need to know how many people are travelling. The age of the passengers affects the price of the ticket, so we'll arrange the interface according to these age groups.



Passenger count form with three fields: one for adults, children, and infants.

```
<div class="field">
  <label for="adults">
    <span class="field-label">How many people aged 16 years
and over are flying?</span>
  </label>
  <input type="number" id="adults" name="adults" min="0"
max="9">
</div>
<div class="field">
  <label for="children">
    <span class="field-label">How many children, aged
between 2 and 15 years old, are flying?</span>
  </label>
  <input type="number" id="children" name="children"
min="0" max="9">
</div>
<div class="field">
  <label for="infants">
    <span class="field-label">How many infants, under 2
years old, are flying?</span>
  </label>
  <input type="number" id="infants" name="infants" min="0"
max="9">
</div>
```

Each age group is a separate field. As we're asking users for an *amount* of something — passengers — the number input makes sense. (We discussed when to use the number input in "A Checkout Form.")

Number inputs have little spinner buttons (also called steppers), which let users increase or decrease the input's value by a constant amount. Luke Wroblewski's usability testing shows that users prefer them to drop down menus:

When testing mobile flight booking forms, we found people preferred steppers for selecting the number of passengers. No dropdown menu required, especially since there's a maximum of 8 travelers allowed and the vast majority select 1–2 travelers.

The only downside is that the browser-provided spinners are tiny, which make them difficult to use. And some browsers don't show them at all. We can solve this problem by creating our own custom stepper component.

A STEPPER COMPONENT

To supply all browsers with bigger, more ergonomic buttons, we can create a custom stepper component using JavaScript. On mobile, they'll save users from triggering the on-screen keyboard, which reduces the time and effort to complete the task.

How It Might Look

How many people aged 16 and over are flying?

How many children, aged between 2 and 15 years old, are flying?

How many infants, under 2 years old, are flying?

Enhanced passenger form with stepper buttons, making it simple to add or remove passengers.

Hiding the Native Spinners

But first, we need to turn off the native, browser-provided spinners. In WebKit browsers we can hide them like this:

```
input::-webkit-outer-spin-button,
input::-webkit-inner-spin-button {
  -webkit-appearance: none;
  appearance: none;
  margin: 0;
}
```

The Enhanced Markup

When the `Stepper()` component initializes, the markup will be changed to this:

```
<div class="field">
  <label for="adults" id="adults-label">How many people
aged 16 and over are flying?</label>
  <div class="stepper">
    <button type="button" aria-label="Add" aria-
describedby="adults-label">&minus;</button>
    <input type="number" id="adults" name="adults"
value="1">
    <button type="button" aria-label="Remove" aria-
describedby="adults-label">&plus;</button>
    <div class="visually-hidden" role="status" aria-
live="polite">1</div>
  </div>
</div>
```

Notes

- The buttons and number input are wrapped in a `<div>` so they can be styled as a group underneath the label.
- The button's `aria-label` attribute ensures that screen readers announce "Remove" instead of "minus symbol." Same goes for "Add" instead of "plus symbol."

- The button's `aria-describedby` attribute references the label's `id`, which means it combines with the label text to give screen reader users extra context. As there are three fields on the page, this stops users thinking "Remove — remove what, exactly?"
- Each button has a `type="button"` attribute to stop the form submitting when clicked.
- Clicking the Add or Remove buttons updates the live region so screen reader users will hear the change without having to move away from the button (see note).

> *Note:* When the Add (or Remove) button is clicked, the input's value is updated, but screen readers don't announce this change. At first, I put the live region attributes on the input. This didn't work in some screen readers, but worse was that it changed the input's semantics into a status box.

A Note on Using Iconography

You'll notice that we're using icons for the buttons. Icons are often the source of heated debates amongst designers, mostly because they have their pros and cons.

The pros are that they save space, and internationally recognized icons overcome language barriers, which is why they're often used in airports.

The cons are that icons can be misunderstood, and they are a poor replacement for just using text. In "The best icon is a text label," Thomas Byttebier goes as far to say:[19]

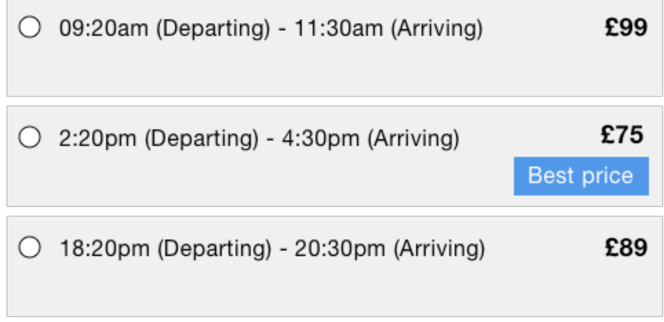
What good has a beautiful interface if it's unclear? Hence it's simple: only use an icon if its message is a 100% clear to everyone. Never give in.

In the case of the stepper buttons, plus and minus icons keep the options equally weighted and are widely understood. Moreover, users can type a number if they want, ignoring the buttons altogether.

19 http://smashed.by/texticons

4. Choosing a Flight

Now all the relevant information has been collected, we can give users a list of flights from which to choose one.



Flight chooser form made up of radio buttons containing departure time, arrival time, and price. The cheapest price is marked with a "Best price" label, and there are pagination controls at the bottom to reveal other days' flights.

The system shows flights that match the date the user specified earlier. Additionally, the interface lets users move back and forth between days. The group's label is set as normal via the `<legend>` and reads "*Available flights on 18 August 2018.*"

The flights are represented as radio buttons — the user can select only one. Each label contains the departure time, arrival time, and ticket price: all useful information. One

advantage of using radio buttons is that you can add any information inside the label and style it as you like, something you couldn't do if you were using a select box.

```
<div class="field-radioButton">
  <label for="flight1">
    <input type="radio" name="flight" value="1"
id="flight">
    <span>Departing at 18:20pm</span>
    <span>Arriving at 20:30pm</span>
    <span>£99</span>
  </label>
</div>
```

GROUP VALIDATION ERRORS

In chapter 1 we looked at how to design an inclusive form validation experience. But because the registration form only consisted of two simple text fields, we never looked at how to handle errors for a field consisting of multiple form controls.

A radio button group is made up of multiple controls. Take a look at the markup below. The fieldset contains the group of controls, and the legend is the group's label. We can effectively use the same error pattern by injecting the error `<span>` inside the legend. Not only will sighted users see the error, but screen reader users will hear the error too.

```
<fieldset aria-invalid="true">
  <legend>
    <span class="field-legend">
      Available flights on 18 August 2018
    </span>
    <span class="field-error">
      <svg width="1.5em" height="1.5em">
<use xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="#warning-icon"></use></svg>
      Choose a flight.
    </span>
  </legend>
  <div class="field-radioButton">
    <label for="flight1">
      <input type="radio" name="flight" value="1" id="flight">
      ...
    </label>
  </div>
  <div class="field-radioButton">
    <label for="flight2">
      <input type="radio" name="flight" value="2" id="flight2">
      ...
    </label>
  </div>
  <div class="field-radioButton">
    <label for="flight3">
      <input type="radio" name="flight" value="3" id="flight3">
      ...
    </label>
  </div>
</fieldset>
```

The `aria-invalid="true"` attribute is placed on the `fieldset`. Putting it directly on the radio button would be incorrect here, because it's not the individual input that's invalid — it's the group. The error `<span>` is exactly the same as the one used for standard text fields, which ensures that errors look and behave the same across all types of form fields, which speaks to *inclusive design principle 3*, "Be consistent."

Available flights on 18 August 2018

Select a flight

09:20am (Departing) - 11:30am (Arriving) £99

2:20pm (Departing) - 4:30pm (Arriving) £75 Best price

18:20pm (Departing) - 20:30pm (Arriving) £89

Previous day Next day

Flight radio buttons with error message.

The error summary needs to contain a link to the first radio button within the group. That is, the link's `href` attribute needs to match the first radio button's `id` attribute. This is why the first radio button in the group has matching `id` and `name` attributes: "flight."

```
<div class="errorSummary" role="group" tabindex="-1" aria-
labelledby="errorSummary-heading">
  <h2 id="errorSummary-heading">There's a problem</h2>
  <ul>
    <li><a href="#flight">Choose a flight</a></li>
  </ul>
</div>
```

5. Choosing A Seat

Choosing a seat isn't the most complicated part of the journey, yet the combination of perceived affordance, layout, and interaction design can make or break this part of the journey if we're not careful.

LAYOUT

Up to now, any field that uses radio buttons has them stacked beneath one another, which is good for most situations. For the seat chooser, however, this makes the page especially long, and — more importantly — harder to scan, as there's a lack of structure.

We can provide that structure by laying out the seats in rows, just like they are on a plane. This will help users map their location, which is useful because users might be looking for aisle or window seats, for example.

Left: seat checkboxes stacked beneath each other making the page long. Right: seat checkboxes laid out in rows making the page shorter and seats easier to find.

To demarcate window seats and aisle seats for screen reader users, we can put hidden text inside the seat's label.

```
<label for="S1A">
  <input type="checkbox" name="seat" value="1A" id="S1A">
  <span class="plane-seatNumber">1A <span
class="vh">Window</span></span>
</label>
```

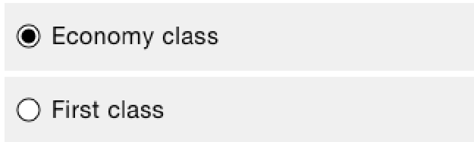
NESTED FIELDSETS

The radio buttons are placed inside an extra fieldset (and legend) to indicate which class the seat belongs to: first class, or economy. Visually this is fine, but screen readers don't always behave as expected. Sometimes, they announce both legends when the first radio button is focused. Sometimes they don't announce the outer legend at all. You can

read Léonie Watson's article "Using the fieldset and legend elements" for more information about this.[20]

Where possible, you should avoid nested fieldsets, not only for screen reader users, but also because their existence often signifies extra complexity that can be designed better with a little more thought. In our case, we're showing both first class and economy class seats because users were never asked to specify which class they wanted earlier in the journey.

Instead, we can ask users to specify their preference beforehand. At the same time, we can mark "Economy class" as checked by default. Marking the most common choice expedites the process.



Class chooser with two radio buttons: economy, and first class.

20 http://smashed.by/fieldsetelements

CHECKBOXES ARE NEVER ROUND

In "Checkboxes Are Never Round," Daniel De Laney says:[21]

Interactive things have perceived affordances; the way they look tells us what they do and how to use them. That's why checkboxes are square and radio buttons are round. Their appearance isn't just for show—it signals what to expect from them. Making a checkbox round is like labeling the Push side of a door Pull.

A radio button tells you that just one can be selected; checkboxes tell you that more than one can. So if one person is travelling, use radio buttons; otherwise, use checkboxes.

OOPS, WE BROKE THE RULES

Miller's law would have you believe that we shouldn't present users with more than seven radio buttons at a time. If you have more than seven, traditional advice would be to use a select box.[22]

Laws are useful: they work as constraints that drive us to good, creative solutions; they allow us to think less, free up our time to solve other problems, and avoid mistakes others have made in the past. But UX Myth 23 states that:[23]

21 http://smashed.by/checkboxes
22 http://smashed.by/millerslaw
23 http://smashed.by/sevenchoices

Miller's original theory argues that people can keep no more than 7 (plus or minus 2) items in their short-term memory. On a webpage, however, the information is visually present, people don't have to memorize anything and therefore can easily manage broader choices.

In our case, a Boeing 747 has over 400 seats. Call me a rebel, but I'm struggling to see a better way of presenting fewer seats. Choosing a seat is quite a unique interaction and benefits from presenting this many choices in plain site.

Also, using the one thing per page pattern (introduced in chapter 2) gives us maximal screen space to design something better. The screen, while long, works well because it's dedicated to just one thing: choosing a seat.

UNAVAILABLE SEATS

Unavailable seats are marked by disabling the checkbox (or radio button). Browsers will gray them out so that sighted users know they aren't selectable. Similarly, screen readers won't announce them, and keyboard users can't focus them. This is one of the few use cases where disabling elements is appropriate.

```
<input type="checkbox" name="seat" value="1A" disabled>
```

LAYOUT ENHANCEMENTS

Laying out seats in rows can cause seats to wrap in small viewports, which destroys the idea of laying them out as modeled in real life. We could style the seats so they don't wrap, but this would cause horizontal scrolling. Neither of these problems are deal breakers, but if we could reduce the chance of this happening, we should.

One approach involves hiding the checkboxes and styling the label to look clickable (which it is). Hiding checkboxes with CSS alone is dangerous because pressing **Tab** moves focus to the checkbox, not the label. On its own this breaks the interface for sighted keyboard users because as the user focuses each checkbox there's no feedback.

To fix this problem, we can give the (still) visible `<span class="plane-seatNumber">` the appearance of focus using the adjacent sibling selector like this:

```
.enhanced [type="checkbox"]:focus + .plane-seatNumber {
  border: 3px solid #ffbf47;
}
```

Note that `.enhanced` is part of the selector. This is because these styles should only be applied when JavaScript is available. This is done by adding a class of enhanced to the document element in the `<head>` of the document like this:

```
<script>
  document.documentElement.className = 'enhanced';
</script>
```

LIMITING SELECTION

If the user specified two travellers, we need a way to allow users to select only two seats. There's no way natively to limit the amount of checkboxes the user can check. If a user selects more than their quota, they'll get an error message. Without user research, it's hard to know whether this is a problem. But if it is, we can enhance the experience with JavaScript.

One way to do this is to disable the remaining seats as soon as the limit is reached. But this assumes users will pick the right seat the first time. When the user tries to click another seat, the interface won't respond because that seat will be disabled.

Savvy users may realize they have to deselect the currently selected seat first, but why should they have to? And what about less savvy users? As designers, we should do the hard work to make it simple for users.

If a user surpasses their quota, the currently selected seat should be unchecked automatically for them. Here's a little script to do it.

```
function SeatLimiter(max) {
  this.max = max;
  this.checkboxes = $('.plane-seat input');
  this.checkboxes.on('click', $.proxy(this,
'onCheckboxClick'));
}
SeatLimiter.prototype.onCheckboxClick = function(e) {
  var selected = this.checkboxes.filter(':checked');
  if(e.target.checked && selected.length > this.max) {
    selected.not(e.target)[0].checked = false;
  }
};
```

When a checkbox is clicked, the onCheckboxClick method is called. The method first checks to see if the checkbox has been checked or not. If it has, it checks to see if the quota has been surpassed. If both conditions are true, the previously selected checkbox is unchecked.

Summary

In this chapter, we continued to use the one thing per page pattern which allowed us to make use of the total screen estate. We looked at ways of reducing friction, not only through interface design, but also by looking at the journey as a whole.

As much as we tried to use native form controls in their standard format, it became apparent that custom components were necessary to give users the best experience. In the end we designed four custom components:

- An autocomplete control to let users search through a long list of destinations quickly and accurately in a way that matches what they know.
- A date picker component to let users find a date in the future without having to worry about formatting issues.
- A stepper component to let users make small adjustments to an amount of passengers effortlessly.
- A seat chooser to make seat selection simple, even on small viewports.

THINGS TO AVOID

- Using radio buttons that look like checkboxes (or vice versa).
- Using select boxes when better alternatives exist.
- Letting users do the hard work when the interface can be designed to do the hard work for them.
- Nested fieldsets.

Demos

- **Autocomplete**: http://smashed.by/autocompletedemo
- **Memorable date**: http://smashed.by/memorabledatedemo
- **Date picker**: http://smashed.by/datepickerdemo
- **Stepper**: http://smashed.by/stepperdemo
- **Seat chooser (nested)**: http://smashed.by/seatchoosernesteddemo
- **Seat chooser**: http://smashed.by/seatchooserdemo

A Login Form

"As a user, I want to log in to [your service] so that I can [do stuff]"

– Nobody, ever!

Nobody *wants* to log in to your site. They're forced to as a security measure. Without it, everyone has access to everyone else's stuff. Bad.

Given how long login forms have been around for and how basic they are in appearance, you'd be surprised at how often they contain the same usability mistakes that stop users doing something they don't even want to do in the first place. Add social login into the mix and things get even harder.

In this chapter, we'll design a login form, and as we bump into each of the problems, we'll look at ways to remedy them. By process of elimination, users should be left with a straightforward and relatively pleasant login experience.

A Standard Login Form

Log in

Username

Password

Log in

A basic login form with username and password fields.

Username Label and Hint Text

Our login form, like many on the web, has an ambiguous label of “Username,” even though it expects users to enter their email address. Ultimately, the login form should mirror the site’s registration form. In our case, this means the label should be “Email address.”

Username

Email address

Don’t do this

Do this

A field labeled “Username” (left) and “Email address” (right).

Legacy systems sometimes let users enter an email address or a username. In this case, the same rules apply — the label should be “Username or email address” — don’t make users guess.

Username

Username or email address

Don’t do this

Do this

A field labeled “username” (left) and “username or email address” (right).

Some niche sites, such as airlines, ask users to enter their booking reference number. In this case, use the hint pattern to tell users where they can find it.

Booking reference

Booking reference
You'll find this in the confirmation email

Don’t do this

Do this

The booking reference field without hint (left) and with hint (right).

Auto-Capitalization, Autocorrect and Spell-Checking

Some Android and iOS browsers try to help users by auto-capitalizing words in text boxes (`<input type="text">`). For example, if I type "adam," it will be changed to "Adam," which can be helpful depending on the circumstance.

Prior to iOS 5, this behavior also applied to the email input. In the case of the username or email address, we certainly don't want users to exert energy fixing mistakes they didn't even make. So we can disable this behavior like this:

```
<input autocapitalize="none">
```

Similarly, iOS will autocorrect words in a text input that it thinks are mistakes. Continuing with the example above: a username may contain a random string of characters that may look like a mistake but isn't. You can disable this behavior like this:

```
<input autocorrect="off">
```

By the same token, some browsers will mark misspelled words with an underline. Again, you can disable this:

```
<input spellcheck="false">
```

Here's the final HTML for our email address field:

```
<div class="field ">
  <label for="email">
    <span class="field-label">Email address</span>
  </label>
  <input type="email" id="email" name="email" value=""
autocapitalize="none" autocorrect="off" spellcheck="false">
</div>
```

Password Field Design

People often use the same password for different sites and applications. But password rules differ from site to site. Some sites ask for a capital letter, others ask for numbers and symbols; other sites ask for a combination of all three.

Many users will tweak their password to match the rules of the site in question. For example, if their password is "password," and the site requires a capital letter, they'll just capitalize the first letter to "Password." Obviously, this is not recommended, but shows that users usually take the path of least resistance.

Referring back to the registration form in chapter 1 again, we gave users a hint that explained the password rules. But like many sites, our login form fails to provide the same clarity. Why should users have to guess or, worse, reset their password?

Password

"What are the rules?"

Password
8+ characters and an uppercase letter

"Easy, I get it."

Password field without hint (left) and with hint (right).

This sort of ambiguity is often in the name of security because providing a hint would make a hacker's job easier. But first, hackers don't hack this way and even if they did, what's to stop the hacker checking the rules on the registration page? Nothing.

Let's reuse the patterns in "A Registration Form":

1. Give users a hint text. Users will have a greater chance of success without having to wait for a useful error message.
2. Let users reveal their password using the password reveal pattern (see page 39).

Password field with password reveal component added.

Auto-Tabbing

Some login forms, such as those found on bank sites, ask users for certain characters of their password. Or they may ask for certain digits from a security pin. In either case, users are normally given three separate text boxes or select boxes.

Santander bank password field with separate three text boxes for each requested character.

The first problem with this approach is that sites will auto-tab between the fields. That is, focus is moved to the next text box automatically as the user enters a predetermined number of characters. But as the BBC's UX guidance says:[1]

1 http://smashed.by/managingfocus

It can be disorienting and hinder users from verifying information or correcting mistakes if the focus automatically changes when the user is not expecting it.

Léonie Watson, accessibility expert and screen reader user, finds them problematic too:[2]

I strongly dislike having auto-tab functionality imposed on me. It is unexpected, and based on a flawed assumption that it is helpful. […] it takes me more time and effort to correct mistakes caused by auto-tab, than it does to move focus for myself.

This point of view shouldn't be surprising given the technique is founded on assumptions that not only break convention, but also take control away from the user (*inclusive design principle* 4).

In this case, there's just no good reason for it. And splitting up a text box into three is unnecessary. A single, clearly labelled text box lets users type three characters freely.

Password field with three separate text boxes (left) and a single text box (right).

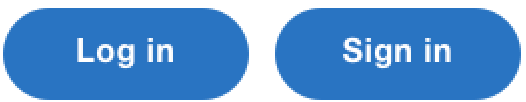
2 http://smashed.by/autotabbing

Submit Button Text: Log In versus Sign In

Having ironed out problems with the username and password fields, our login form is almost identical to the registration form. It contains the same fields in the same order with the same microcopy. The only difference is the button's label. Instead of "Register" it's "Sign in."

"Sign in" is perhaps more human than "Log in." When you visit a spa or office building, signing in grants you entry. And you sign out as you leave. It's usually sensible to use the same language for digital experiences too.

It can, however, depend on the industry. Banks, for example, tend to use "Log in." The notion of logging came along with computers in the 80s — the operations that users do are *logged* for security reasons.



Button labeled "Log in" (left) and "Sign in" (right).

We should design interfaces that speak in a language familiar to the user. Whichever you choose, be consistent (*inclusive design principle 3*). Make URLs, links, headings, and buttons match. And if users click "Log in" to log in, then they should click "Log out" to log out.

The 'Username or Password Doesn't Match' Problem

Sometimes, we deliberately make things difficult to use. A door by its very nature isn't easy to use — it would be far easier if there was no door at all. Login forms need to be somewhat difficult to use, otherwise they wouldn't be secure.

Many login forms make the same mistake of showing users an error message that says "The username and password don't match." But as Jared Spool comically explains in "Is Design Metrically Opposed?":[3]

We know which one doesn't match, we're just not going to tell you, because our security people think that if we told you that it was the password, they would know they had a legal username and they would try every possible password in history.

As noted earlier, hackers don't actually do this. But even if they did, it's easy to check username availability by trying to register an account with that username.

The problem for users, is that they're left to reset their password, which is long-winded and may cause abandonment. Even where a lack of usability or understandability is deliberate, there still needs to be a degree of understandability and usability.

3 http://smashed.by/onelineofcode

In this case, we don't have to tell users what their password is, but we can tell them that it's the password that doesn't correspond to the username (which they have right).

There's a problem

The username and password doesn't match.

"Errr, what?"

Don't: password error message "The username or password don't match."

There's a problem

The password doesn't correspond to your username.

"Oh my password is wrong"

Do: password error message "The password doesn't correspond to your username."

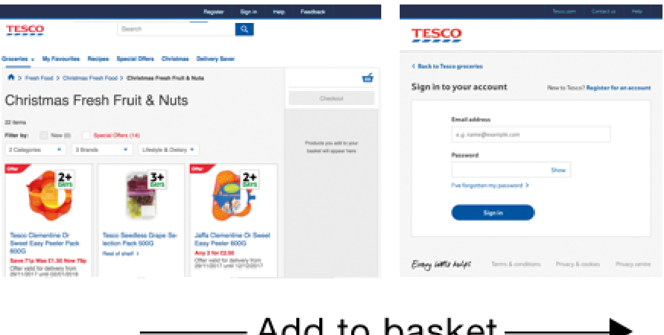
The Form in Context

We've ironed out many of the issues surrounding standard login forms, but we've done so while zoomed in on the form itself. We also need to consider the form in the context of the page and the overall experience. This includes looking at various journeys through the login form. Let's start with layout.

LAYOUT

When trying to perform an action anonymously that requires being logged in, users will be sent to the login page.

Many sites design the login page with a unique, minimalist layout. For example, when users try to add a product to their basket on Tesco's website, they're taken to a login page with a very different layout.



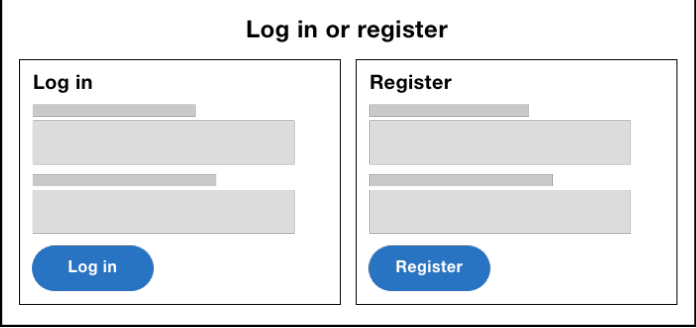
Left: Tesco product list page. Right: Tesco login page with a different layout.

Giving users a different layout is disorienting, especially for screen reader users and cognitively impaired users, as they have to familiarize themselves with a new structure.

Where possible, the login form should inherit the layout of the rest of the site.

ONE FORM PER PAGE

Some sites put both registration and login forms on one page, either next to each other on desktop, or below each other on mobile. This is problematic for a number of reasons.



Page containing login and register forms.

- Putting similar forms next to each other makes it hard to decide between them, especially for cognitively impaired users.
- Arriving on a page containing two forms, with a heading of "Log in or register," is confusing when you consider that many users would have clicked a link labelled "Log in."
- On mobile, one of the forms will be off-screen and effectively deprioritized.

- Screen reader and keyboard users are going to have to wade through more of the interface to get to the relevant form.

Instead, put each form on a separate page, and give users a link to each form at the top.

Left: the login page with a link to the register page. Right: the register page with a link to the login page.

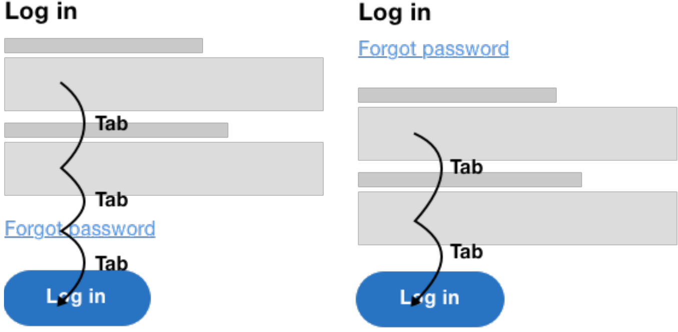
FORGOTTEN PASSWORD LINK PLACEMENT

Human beings are forgetful. Password managers mitigate this problem by storing all your passwords in one place — you just have to remember a single, master password. That's great, but password managers aren't infallible. If you don't remember to save your credentials into it, you're in the same position as everyone else. Moreover, not everyone uses one, nor should they have to.

Most sites give users a way to reset their password if they forget it. The feature itself isn't especially problematic. It's

the placement of the link within the login form that can cause usability issues. If the link is just above the password field, when users tab from the email field, it's the link that will receive focus, not the password field. Some users will tab and start typing, not realizing what's happened.

Worse still is when the link is placed before the submit button. When keyboard and screen reader users tab from the password field and press **Enter**, they'll expect the form to submit. But instead, they'll be taken to reset their password. When they realize what's happened, they'll need to go back, reenter their credentials, and be careful not to make the same mistake again.



Left: forgot password link between last field and submit button. Right: forgot password link before the form.

When the feature is considered in isolation, having the reset password link in close proximity to the password field makes sense. But the primary need is to log in, and the link shouldn't disturb the experience of logging in.

The submit button should be the last interactive element in the form because that's what users expect. Solving this problem is simple: place the forgotten password link before the form, which makes it easy to discover, especially for screen reader and keyboard users.

Social Login

Sites have recently started to offer users the ability to log in with social networks, such as Facebook, Twitter, and Google. This saves users having to type credentials they may not remember.

Medium lets users login in with Facebook.[4] This is a boon for Medium users because they'll then have the option to post articles to Facebook automatically.

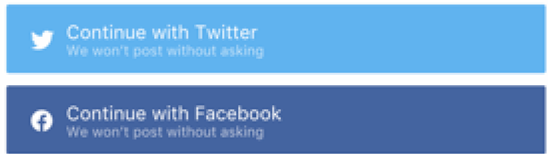
Social login is not without its problems though.

4 https://medium.com

PRIVACY

Users are worried about their privacy. They don't know what you'll do automatically, and they want to feel as though their information is safe and any actions they perform are intended.

Medium does this well: on the login page it says, "We won't post without asking," which puts users' minds at ease.



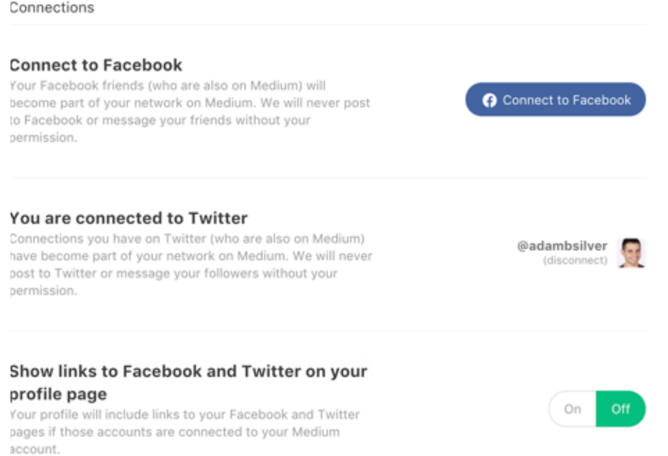
Medium's social login buttons explaining "We won't post without asking."

SEAMLESS INTERCHANGE

Some users won't remember how they originally created an account; therefore, they won't know which login method to choose.

At Kidly, we handled this by showing users an error message. For example, if users had signed up with standard login, then tried to sign in with social login, we'd show an error message saying so. But this puts the burden on the user.

Again, Medium lets users log in interchangeably without ever knowing what happened. For example, if I log in to Medium with my email but have registered previously with Facebook, Medium logs me in automatically and merges my accounts. Users can visit the settings page to see what accounts are hooked up, which keeps users informed and in control.



Medium's settings page lets users connect and disconnect different social media accounts easily.

CHOICE VERSUS CHOICE PARALYSIS

Standard thinking is that choice is good. It offers freedom, autonomy, and personal responsibility. Heck, even one of *the design principles* is "Offer choice."

But *more* choice is not necessarily better when it comes to features or products. Barry Schwartz, author of *The Paradox of Choice*, presents a case study in which researchers set up two displays of jams at a gourmet food store. Customers could try samples, and were then given a coupon for a dollar off if they bought a jar. One display had 24 jams, the other had just 6. Around 30% of people exposed to the smaller selection bought a jam, but only 3% of those exposed to the larger selection did.

There's also some interesting data on companies that offer pension plans to their employees. One of Barry's colleagues was granted access to the records of Vanguard, a mutual fund company, and found that for every 10 mutual funds the employer offered, the rate of participation went down 2%. Consider that employees knew that by not participating, they were passing up as much as $5,000 a year.

This phenomenon is actually called Hick's law (named after psychologist William Edmund Hick), which states that the time taken to make a decision increases as the number of choices expand.

The point, of course, is that we need to be wary of giving users multiple ways to log in. It might seem useful, but it may also be a burden on them. We have to balance the value in doing so.

Summary

In this chapter we started by quashing traditional advice that omitting hint text and explicit error messages improve security on login forms. We then looked at some of the subtle usability issues that can be introduced with social login. Finally, we looked at ways of improving the experience for keyboard and mobile users, which meant avoiding auto-tabbing, autocorrecting and auto-capitalizing input.

THINGS TO AVOID

- Using ambiguous microcopy and error messages in the name of security.
- Putting the login form next to the registration form.
- Auto-tabbing between multiple fields.
- Using multiple text boxes for one field.
- Putting the forgot password link inside the form.
- Enabling autocorrect, auto-capitalize and spell check on fields that may not expect real words: username, for example.

Demo

- Log in form: http://smashed.by/loginformdemo

An Inbox

My sister loves to-do lists. In fact, she loves them so much, that one of her favourite things is making new lists out of old ones. The world is full of lists. There's even a list of great people.[1] On the web there are several types of lists, and there are some design patterns that have emerged over the years that help to manage them.

In this chapter, we'll look at an inbox; that is, a list of emails sent from other people. In many respects, an inbox is a list of tasks organized around emails. Besides reading and replying to them, the aim is to achieve a zen-like state of Inbox Zero.[2] To let users get there quickly, we will design the interface so they can delete, archive, and mark emails as spam. But not just one at a time — in bulk. My sister loves pen and paper, but if we get this right, I hope she'll be converted to digital.

List Types

First, we're going to look at how best to mark up a list of emails. Discussing lists may seem out of place in a book

1 http://smashed.by/thegreat
2 http://smashed.by/inboxzero

about forms, but forms rarely form part of an interface on their own. Ignoring their surroundings can result in disagreeable experiences.

The meaning (or semantics) behind elements should influence their appearance. In other words, form should follow function. There are four elements we can use to construct lists, each with different semantics: description lists, tables, ordered lists, and unordered lists. Let's discuss the pros and cons of each now.

DESCRIPTION LISTS

A description list (`<dl>`) — formerly called a definition list — is for grouping a list of terms and corresponding definitions; for example, product details such as size, price, and material. As a list of emails isn't a collection of terms and definitions, this type of list isn't appropriate.

```
<dl>
  <dt>Size:</dt>
  <dd>250cm × 135cm × 90cm</dd>
  <dt>Price:</dt>
  <dd>£429.95</dd>
  <dt>Material:</dt>
  <dd>Reclaimed teak</dd>
</dl>
```

TABLES

A table (`<table>`) is an arrangement of data, laid out in rows and columns. Like a spreadsheet, tables are well-suited for data that needs to be compared, sorted, and totalled.

Unfortunately, tables are difficult to style on small viewports, because there's no room to show more than two or three columns at a time. Even then, it could be a squeeze depending on the data inside the cells, creating layout issues. For example, content could wrap profusely, or it could cause users to scroll horizontally to reveal the hidden content.

Making tables responsive isn't the most straightforward thing to do because they are inherently tied to the way they look. Put another way, to make a table not look like a table is not only very difficult, but it would be deceptive, counterproductive, and inaccessible.

Gmail uses tables and puts recipient, subject, and date sent into columns. Interestingly, though, there are no table headings, which is the first clue that tables have been used for layout purposes rather than their semantic qualities, which causes various access issues. Jeremy Keith talks about this in his book *Resilient Web Design*:[3]

---

3 http://smashed.by/tablelayout

Using TABLEs for layout is materially dishonest. The TABLE element is intended for marking up the structure of tabular data. The end result [...] is a façade. At first glance everything looks fine, but it won't stand up to scrutiny. As soon as such a website is stress-tested by actual usage across a range of browsers, the façade crumbles.

```
<table class="inbox">
  <a href="/email/1">
    <tr>
      <td>John Oates</td>
      <td>Your Amazon.co.uk order #123 is out for
delivery</td>
      <td>10 August</td>
    </tr>
  </a>
</table>
```

See the markup above as an example. The `<tr>` is wrapped in an `<a>` to let users click an email to read it. The problem is that browsers ignore the link. It's simply not allowed and screen reader users will struggle to interpret it.

Gmail makes the row clickable by using JavaScript to listen to click events. But not only is this unclear for screen reader users, not everyone has JavaScript and, quite frankly, it's unnecessary.

ORDERED AND UNORDERED LISTS

The generic list is useful because it itemizes and groups content into related chunks accessibly. But they can be used for more than just bullet points. They come in two flavors: ordered (`<ol>`) and unordered (`<ul>`) lists. And they're far less opinionated than tables.

The difference between the two types lies in their name. If the order of the items matters, use an ordered list. For example, a recipe's instructions require users to follow them in order — not doing so may produce inedible food. On the other hand, an inbox doesn't have to be read or actioned in a predefined order. It sounds simple when put like that, but we tend to overthink these things.

Let's lay out the inbox using a `<ul>`.

```
<ul class="inbox">
  <li>
    <a href="/emails/1/">
      <div class="inbox-recipient">John Oates</div>
      <div class="inbox-subject">Your Amazon.co.uk order
#123 is out for delivery</div>
      <div class="inbox-date">10 August</div>
    </a>
  </li>
</ul>
```

> *Note:* Unfortunately, most screen readers don't differentiate between unordered and ordered lists in any meaningful way. But this shouldn't stop us from using the most appropriate element. As support improves, the benefits will be ready and waiting.

Unlike tables, unordered lists are stylistically malleable and, therefore, responsive. That's because they can be laid out in different ways without having to affect their structure in a way that makes them less accessible non-visually.

With the markup above, on large viewports we can lay the emails out in columns, and on small viewports we can avoid layout issues by stacking them vertically. Moreover, the entire list item can be made clickable without resorting to JavaScript hacks. Wrapping a link around the contents is perfectly valid, which is less work and more robust.

In the case of an inbox, list items are more suited anyway: not only are column headings redundant, but there's no need to compare or total items in the list.

Marking Email for Action

To let users mark emails for action, we need to give each row a checkbox.

```
<ul class="inbox">
  <li>
    <input type="checkbox" name="email">
    <a href="/emails/1/">
      <div class="inbox-recipient">John Oates</div>
      <div class="inbox-subject">Your Amazon.co.uk order
#123 is out for delivery</div>
      <div class="inbox-date">10 Aug</div>
    </a>
  </li>
  ...
</ul>
```

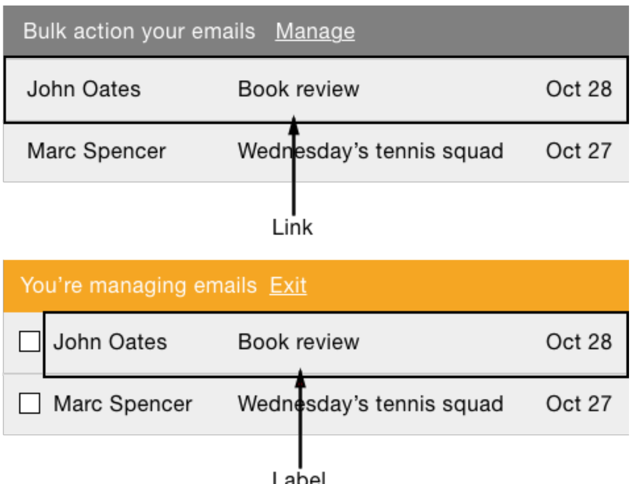
You'll notice each checkbox is missing a label. The problem is that the contents of the link should also be the contents of the label. In other words, two opposing interactions need to occupy the same space. Remember, clicking the label should mark the checkbox, whereas clicking the link should navigate the user to the email.

In this case, you could argue that a visible label is redundant. After all, the label would duplicate the link's content which would make the experience confusing for sighted users.

USING MODES

Trying to meet two user needs (viewing and managing) in a single interface is partially responsible for the problem in the first place. One way to avoid the issue would be to split these needs apart using the concept of modes.

This just means letting users switch between managing email and reading it.



Top: inbox in read mode where each row is a link to read the email. Bottom: inbox in manage mode where each row is a label that toggles the checkbox state.

Clicking "Manage" puts users into manage mode. When in manage mode, the link's label changes to "Exit" (or similar), which, when clicked, takes the user back to read mode.

When in read mode, there are no checkboxes or any other form paraphernalia. The row is a link which takes users to read the email. When in manage mode, the row turns into a `<label>`. When clicked, it marks (or unmarks) the checkbox just like normal.

Modes are best suited when one mode is used more frequently than the other. But when both are used frequently, like an inbox, having to switch back and forth all the time may be undesirable.

> *Note:* We should try to avoid modes, but if they're necessary, the interface must make it obvious which mode is invoked.

VISUALLY HIDE THE LABEL

Instead of using modes, we can add a visually hidden label. There are two ways to do this. The first is to use the `aria-labelledby` attribute (shown below), which uses existing content to label the checkbox. The downside is that it means adding `id` attributes. In any case, ARIA shouldn't be used unless there's no better alternative — something first noted in chapter 1, "A Registration Form."

```
<li>
  <input type="checkbox" name="email" aria-
labelledby="inbox_label1">
  <a href="/emails/1/" id="inbox_label1">
    <div class="inbox-recipient">John Oates</div>
    <div class="inbox-subject">Your Amazon.co.uk order #123
is out for delivery</div>
    <div class="inbox-date">10 Aug</div>
  </a>
</li>
```

Alternatively, a standard `<label>` has better support and adheres to ARIA's first rule (not to use it if a native option is available). But the downside with this approach is that the content has to be duplicated, which would create redundancy for sighted users with missing CSS.

```
<li>
  <input type="checkbox" name="email" id="email1">
  <label for="email1" class="visually-hidden">From John
Oates, subject ‘Your Amazon.co.uk order #123 is out for
delivery’ (10 August 2017)</label>
  <a href="/emails/1/">
    <div class="inbox-recipient">John Oates</div>
    <div class="inbox-subject">Your Amazon.co.uk order #123
is out for delivery</div>
    <div class="inbox-date">10 Aug</div>
  </a>
</li>
```

> *Note:* The CSS for the visually hidden class is set out in “A Checkout Form.”

While duplication isn't a big performance issue, if we're not careful, bloated HTML can eventually diminish the experience by causing some operations to take longer — screen reader software can be unresponsive, for example.

On the other hand, duplication in this case can be advantageous. As the label content is just for screen reader users, we

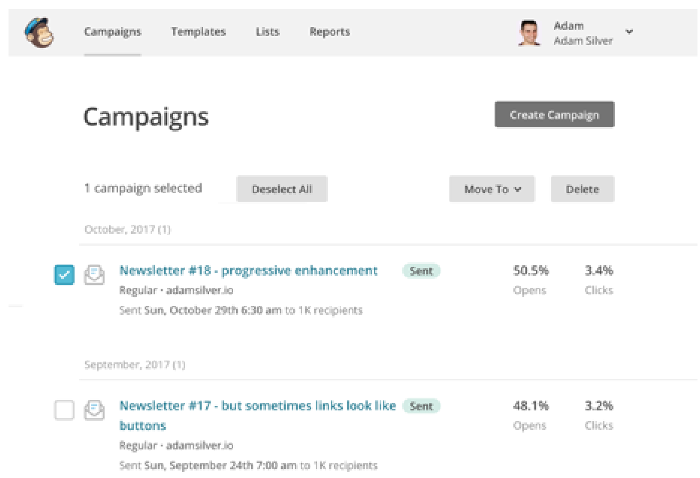
can create a specific message just for them. For example, the label has the word “subject” prefixed, which is useful in this context. This follows *inclusive design principle 1*, “Provide a comparable experience,” which is not about giving users the same experience, but one of comparable value and utility.

HIGHLIGHTING MARKED EMAILS

The deal with human–computer interaction is that when the human does something, the computer should respond. In this case, clicking a checkbox makes a little tick appear (and disappear) accordingly. As with every other checkbox in any other form, this is probably enough feedback.

Left: checkbox checked. Right: checkbox unchecked.

For example, MailChimp, which has a reputation for user-centered design, shows that you don't need to highlight the entire row. It relies solely on the checked state of the checkbox. We can assume their research showed this to be enough. My own research aligns with this too.



MailChimp's campaign list page with one campaign selected.

We could highlight the entire row using CSS and JavaScript, but we should only do that if user research shows this will *add value* (*inclusive design principle* 7).

Actioning Emails

Letting users select multiple emails is all well and good, but we're going to want to facilitate actioning them too. This form has three actions and, therefore, three submit buttons: Archive, Delete, and Mark as spam.

```
<input type="submit" name="archive" value="Archive">
<input type="submit" name="delete" value="Delete">
<input type="submit" name="spam" value="Mark as spam">
<ul class="inbox">...</ul>
```

The nature of this form and the presence of multiple submit buttons create several new problems that previous chapters haven't had to consider. Let's discuss each of those now.

THE MULTIPLE SUBMIT BUTTON PROBLEM

Implicit submission lets users submit the form by pressing **Enter** when focus is within a field. This convention speeds up submission without having to move focus to the submit button. This is especially useful for a single field form such as a search form.

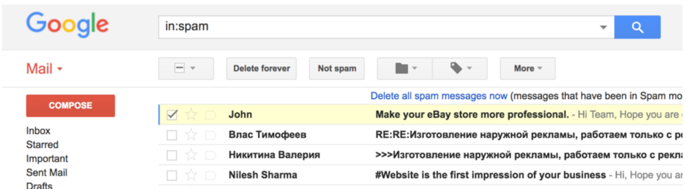
Having multiple submit buttons with differing actions is problematic because if the user presses **Enter**, which action will be taken? The answer is that browsers will choose the first button in the document source.

The best solution to this problem is to avoid it; that is, to have just one action per form. Depending on the design, this may not be easy, which is unfortunately the case with the inbox.

One alternative approach could be to expect users to choose which action they want to perform, before selecting the

emails to apply that action to. But this seems somewhat unconventional and long-winded.

Fortunately, multi-select interfaces usually place the submit buttons at the top of the form in close alignment to the checkboxes. This gives users a way to discover the available actions before making their selection.



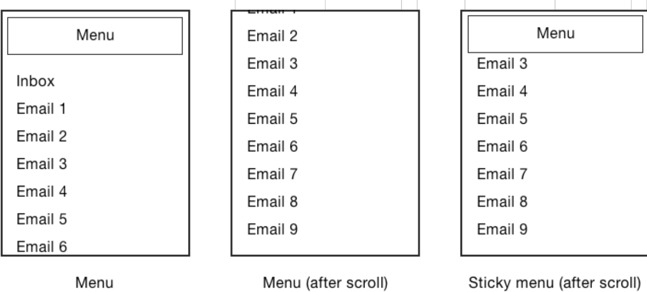
Gmail's inbox screen showing a selected email with an additional menu now available.

It's worth noting that implicit submission is probably less useful on a form consisting solely of checkboxes. In any case, as we have multiple submit buttons, we should put the least critical action first — in this case, Archive. That way, if a user happens to submit the form implicitly, they'll be in less of a predicament.

Also, we can offer users a way to *undo* their last action, which we'll discuss later.

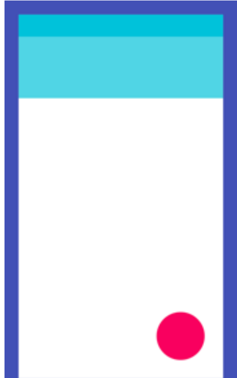
STICKY MENUS

The menu is placed above the list of emails; as users scroll, it might disappear off screen. A sticky menu, however, would stay on-screen as soon as the menu gets to the top edge of the viewport.



Sticky menu in three states. Left: menu positioned above the content as normal, before the page is scrolled. Center: the menu (not sticky) rolls off-screen after the page has been scrolled. Right: a sticky menu still on-screen even after the page has been scrolled.

Similarly, Google's material design has the floating action button. As users scroll, the action button floats on top of the content. Both of these techniques give users quick and easy access to the menu without having to scroll back up to the top.



Floating action button layered on top of the screen at the bottom-right of the viewport.

However, sticky menus are problematic for three reasons.

First, they obscure the content beneath, which is especially distracting on mobile as the menu impedes access to the primary content. In the case of the inbox, the primary need is to read and respond to email, not to bulk action it.

Second, sticky menus are usually employed to solve symptoms that mask the true underlying problems; for example, that the page is often too long in the first place. An inbox typically shows just 20 emails at a time, which means the menu is, at most, a quick flick away on mobile and always in view on desktop.

Third, the items within the sticky menus are difficult to focus with the keyboard. You might be halfway down the page but the menu (which is in close proximity visually) could be a long way away via the keyboard.

For these reasons, it's better to position the menu statically.

> *Note:* Where sticky menus are useful, you can use `position: sticky` as a progressive enhancement. In the past, we had to resort to complicated techniques that created jarring and broken experiences across a range of mobile and tablet devices.[4]

4 http://smashed.by/fixedposition

DISABLING AND HIDING BUTTONS

Some multi-select interfaces will hide or disable the menu buttons until at least one item has been selected. You could argue that showing (or enabling) the buttons in response to selecting an item helps users take the next step. When hiding the buttons, the interface becomes more streamlined as the buttons are only shown as they become relevant. But this is problematic for three reasons.

First, hiding the buttons means the available actions aren't discoverable. This is why designers opt for disabled buttons. But we discussed the problems with disabled buttons in chapter 1, "A Registration Form." As a quick reminder, they don't tell users why they're disabled, and screen reader users can't focus them.

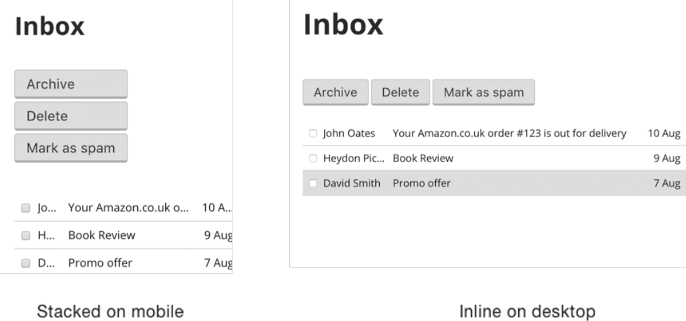
Second, there needs to be space to reveal the buttons in the first place. When there isn't, the page can judder as the page reveals the buttons and moves other parts of the interface around to make space.

Third, having the buttons appear when clicking a checkbox is distracting as users are focused on selecting the right emails. And assuming the change of state is valuable, the buttons would have to be in the viewport for users to see the change anyway.

Just show the buttons at all times.

A Responsive Menu

When there's enough space, the buttons should just be laid out at all times, making them readily available and interactive. But if you have more than three buttons in the menu, or you need to display additional components along the same row, it's going to be hard to fit them on screen, especially on mobile.



Left: on mobile with menu buttons stacked. Right: on desktop with menu buttons laid out in a row.

The problem is that the buttons will start to stack beneath one another, which pushes the main content downward and changes the spatial relationship between the menu and the list of emails. Moreover, having the menu dominate the interface is problematic because dominance is a quality

we should use sparingly. After all, if everything dominates, nothing does. Really, the inbox itself should take center stage, with the menu taking a back-seat role.

We can handle this problem by hiding the buttons behind a menu. There are two ways to create a menu: first, by using a select box; second, by creating a custom menu component. Let's discuss the pros and cons of each next.

A SELECT BOX MENU

Select boxes are a menu of sorts. In fact, sometimes, they're referred to as dropdown menus, among other names.

Like a menu, they group similar items together that users can select. And they hide the items behind a click, keeping the interface compact. They're an attractive option because, as we know, browsers supply them for free. But even though select boxes look like menus and behave like them, and even though they are sometimes referred to as menus, they aren't true menus.

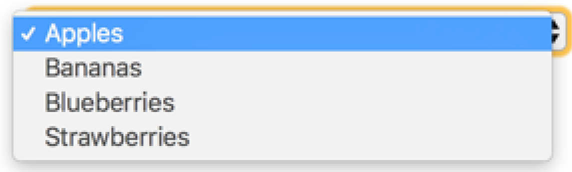
Select boxes are for input. That's why forms that contain select boxes — like any other input — must be accompanied by a submit button to submit the choice. Not only is this convention, but it's also in the Web Content Accessibility Guidelines (WCAG):[5]

5 http://smashed.by/constbehavior

Changing the setting of any user interface component does not automatically cause a change of context [...]

The reason I bring this up is because using a select box as a menu often causes designers to omit the submit button from the interface. And then JavaScript is needed to submit the form when the selected option is changed (`onchange`). But this submits the form, without the user's say-so, which fails *inclusive design principle* 4, "Give control."

There are also problems for screen reader and keyboard users. For example, on Chrome (Windows), the `onchange` event is fired as soon as the user presses **Down** to select the next option. But with this approach in place, the form is immediately submitted, making it impossible to move through all the items in the menu.



Expanded select box with the first option selected. Pressing down immediately submits the second option when the user might have wanted to select the third option.

Other browsers are more forgiving of such techniques — most won't fire the `onchange` event (and thus submit the

form) until the user presses **Space** or **Enter**. But not all browsers are alike nor implement the specification consistently. Ignoring people who use one of the less forgiving browsers doesn't make the problem any less real.

The other problem with using a select box is that it's always collapsed, even when there's enough space to lay out the options. One solution is to use JavaScript to create a completely different component for big screens. This is known as adaptive design.[6]

ADAPTIVE DESIGN VS. RESPONSIVE DESIGN

First a fun history lesson.

When the web came along, we settled on 640 pixel widths (as computer monitors commonly supported this resolution). Then a few years later, when larger monitors came to market, we increased it to 800 and then 960 pixels. We no longer cared about people with smaller monitors. We expected users to maximize their browser window; if they didn't, they'd get a horizontal scroll bar, and that would be their problem.

More years passed. The mobile web was born. Or, more accurately, we could use websites on our phones, which happen to have small screens. A million devices came out.

6 http://smashed.by/rwdadaptive

A million browsers came out. And browsers gave us CSS media queries. Accordingly, we started to design for a width of 320 pixels. Why? Because many of us had iPhones, and this happened to be its width in portrait orientation. The hardcore among us started designing for portrait and landscape sizes according to the most popular devices at that time.

Now we also have tablets, desktops, and really big desktop screens. We can browse on large screen televisions and tiny watches. If your head is spinning, don't worry, so is mine. This is the problem that responsive design solves and adaptive design exacerbates.

The difference between responsive and adaptive design is both subtle and crucial. Both techniques are often based on viewport width. And both use CSS media queries to change the interface. But they are really quite different.

Adaptive Design

Not everyone in the industry agrees on the meaning of adaptive design.[7] Some think it means having different layouts that snap at particular sizes — in other words, not fluid. Others think it's the same as responsive design, which is hardly surprising seeing as the words are synonyms.

7 http://smashed.by/adaptivedesign

The other common understanding of adaptive design is about defining several different (parts of) layouts, made up of different HTML that's rendered. Originally, this was based on user agent string.[8] Sometimes, JavaScript is used to restructure the arrangement of HTML at certain viewport widths. But more commonly these days, it's done using CSS media queries. We'll focus on this flavor of adaptive design from this point.

This involves delivering all the HTML for the different layouts, and hiding and showing these layouts based on CSS media queries that match a particular device's width.

```
<!-- layout 1 -->
<div class="stuff1">...</div>
<!-- layout 2 -->
<div class="stuff2">...</div>
```

```
@media only screen and (min-device-width: 375px) and (max-
device-width : 667px) {
  .stuff1 {
 display: none;
  }
  .stuff2 {
 display: block;
  }
}
```

8 http://smashed.by/ress

This approach is normally unnecessary and counter-productive for a number of reasons.

- There's an endless stream of devices and browsers with different widths: creating specific designs for every device width is impossible.
- The extra code needed to produce such designs would result in slow-loading pages, which are detrimental to the user experience.
- Not all components need a breakpoint. Plenty of components can be designed to work well in exactly the same way on both small and large screens.
- More importantly, users should get a consistent experience (*inclusive design principle 3*) no matter which device they choose to use. Rotating a device from portrait to landscape, for example, shouldn't mean having to relearn an interface because that puts an unnecessary cognitive burden on users.

Responsive Design

Responsive design takes a different approach. It's about designing a single, fluid interface that works well at any size, regardless of device. Specific browsers and device widths become irrelevant. The difference is that you only add a media query when and if something breaks. These media queries are known as content breakpoints.

```
@media only screen and (min-width: 61.37em) {
  /* Fix broken layout for a particular thing at a
particular width here */
}
```

Where adaptive design tries to bend the web to its will, responsive design embraces it. Responsive design understands that you can't possibly design for every device and browser individually. That's just not how the web works. Instead, responsive design encourages us to design interfaces that work on any size screen.

The select box design I mentioned earlier requires an adaptive approach: on small viewports users get a select box; then, when there's enough space, it's swapped out for submit buttons.

Left: select box menu for small screens. Right: menu buttons laid out in a row for large screens.

In this case, the big screen view entirely discards the select box in favour of a different interface using CSS and JavaScript. We either have to change the HTML dynamically with JavaScript, or we have to have both layouts in HTML, ready to be enabled and disabled through a CSS breakpoint.

The server also needs to be aware of how both menus transmit data. In this case, the select box and submit buttons would be sending different data.

Not only is all of this more work, but the page will take longer to load and there are now two vastly different variations of the same feature to maintain indefinitely. Adaptive design should always be a last resort.

HOVER VERSUS CLICK

On the web, menus are sometimes opened on hover. Designers often assume that this aids discovery and saves users the effort of clicking. The thing is, there are many problems with opening a menu (or anything really) on hover and the effort of clicking a part of the interface is extremely low.

First, hovering is not an intention to open the menu. When a user moves over a hover menu it can obscure the content behind it, which disrupts the experience. With the inbox, as the user goes to select the first checkbox, they may accidentally end up clicking one of the items in the menu, which fails *inclusive design principle 4*, “Give Control.”

Second, users have to be careful to keep the cursor within the bounds of the menu, otherwise it will close. This is known as a hover tunnel, and is especially difficult to operate with motor impairments.

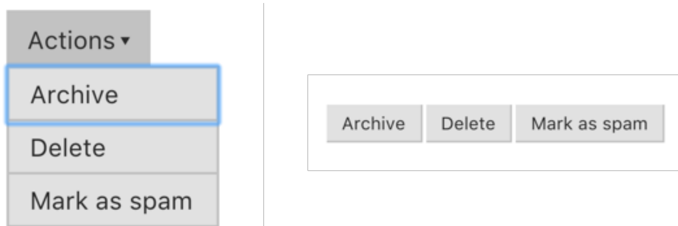
Third, not all users use a mouse (or other types of pointing device) and touchscreen devices are usually operated without one.

You should note that opening a menu on hover on desktop and on click for mobile isn't recommended either. There are many large touchscreen devices and many small screen laptops. Features should never be inferred from screen size.

Needless to say, menus should be triggered on click, which is an explicit intention to activate it, keeping users in control.

A TRUE MENU

Having explored the pitfalls of adaptive design and hover menus, we can now safely proceed to design a true responsive menu that opens on click.



Left: a collapsible menu for small screens. Right: a menu bar for large screens.

The Basic Markup

```
<div class="menu">
  <div role="menu">
    <input type="submit" name="archive" value="Archive"
role="menuitem">
    <input type="submit" name="delete" value="Delete"
role="menuitem">
    <input type="submit" name="spam" value="Mark as spam"
role="menuitem">
  </div>
</div>
```

Notes

- The menu itself has `role="menu"` indicating that it contains menu items. When a menu item is focused, screen readers will announce it as a three-item menu.

- The `menu` role isn't commonly used in web applications. Only ones that mimic desktop applications (like ours, which is essentially a web-based email application) apply.
- The wrapping `<div class="menu">` will be needed for enhancement purposes because the toggle button will be prepended to it.

SMALL MODE VERSUS BIG MODE

When the script initializes, it will need to check to see if the viewport is in small or big mode. We're using the words small and big, as opposed to mobile and desktop, because responsive design doesn't think in terms of devices. Moreover, the media query values for small and big don't necessarily correspond to mobile or desktop — they are determined by the place in which the menu would otherwise break.

The constructor function (shown below), takes two arguments: the `container` element and `mq` (short for media query) string. The media query string is `(min-width: 45em)` because that's where the interface starts to break.

```
function Menu(container, mq) {
  this.container = container;
  this.menu = this.container.find('[role=menu]');
  this.mq = mq;
```

```
  this.keys = { esc: 27, up: 38, down: 40, tab: 9 };
  this.menu.on('keydown', '[role=menuitem]', $.proxy(this,
'onButtonKeydown'));
  // create button and listen to click and down events
  this.createToggleButton();
  // Setup up media query listener and check which applies
on initialisation
  this.setupResponsiveChecks();
}
```

Besides assigning properties to `this` to make them available to other methods (shown later), the constructor is responsible for listening to the keydown event on the menu items and creating the toggle button.

The last line calls the `setupResponsiveChecks()` method, which is responsible for collapsing the menu items behind a traditional menu using a combination of CSS media queries and JavaScript's `matchMedia` API.

```
Menu.prototype.setupResponsiveChecks = function() {
  this.mql = window.matchMedia(this.mq);
  this.mql.addListener($.proxy(this, 'checkMode'));
  this.checkMode(this.mql);
};
Menu.prototype.checkMode = function(mql) {
  if(mql.matches) {
    this.enableBigMode();
  } else {
    this.enableSmallMode();
  }
};
```

The `matchMedia` API is the JavaScript equivalent of a CSS media query. Where `@media() {}` is for CSS, `matchMedia()` is for JavaScript. It's a way of keeping behavior and style in sync, based on the same media query. In this case, when the `(min-width: 45em)` media query is matched, big mode is enabled. When it doesn't match, this means the viewport width is less than `45em`, and so the script calls the `enableSmallMode()` method which constructs a toggle menu.

```
<div class="menu">
  <button type="button" aria-haspopup="true" aria-
expanded="false">
    Actions
    <span aria-hidden="true">&#x25be;</span>
  </button>
  <div role="menu">
    <input role="menuitem" type="submit" name="archive"
value="Archive">
    <input role="menuitem" type="submit" name="delete"
value="Delete">
    <input role="menuitem" type="submit" name="spam"
value="Mark as spam">
  </div>
</div>
```

Notes

- The `aria-haspopup` attribute indicates that the button shows a menu. It acts as warning that, when pressed, the user will be moved to the pop-up menu.
- The `<span>` contains the Unicode character for a down arrow. Conventionally, this indicates visually what `aria-haspopup` does non-visually — that pressing the button reveals something. The `aria-hidden="true"` attribute prevents screen readers from announcing "down pointing triangle" or similar. Thanks to `aria-haspopup`, it's not needed in the non-visual context.
- The `aria-expanded` attribute tells users whether the menu is currently expanded (open) or collapsed (closed) by toggling between `true` and `false` values.

> *Note:* Before `matchMedia`, we had to use flaky techniques to get the width of the viewport, breaking the experience in various browser and device combinations.[9] Even in browsers that returned the correct viewport width, it would only do so in pixels — not ems. Using `em`s is preferred because when the user increases the text size, the layout will adapt in proportion.

9 http://smashed.by/rwdjs

Keyboard and Focus Behaviour

When the menu button is clicked, the script checks to see if the menu is currently open by checking whether `aria-expanded` is set to `false`. If it is, the menu is shown, and focus is moved to the first item; if it isn't, the menu is hidden, and focus moves back to the menu button.

```
Menu.prototype.onMenuButtonClick = function() {
  if(this.menuButton.attr('aria-expanded') == 'false') {
    this.showMenu();
    this.menu.find('input').first().focus();
  } else {
    this.hideMenu();
    this.menuButton.focus();
  }
};
```

We can use the `[aria-expanded]` CSS attribute selector to toggle the menu's display.

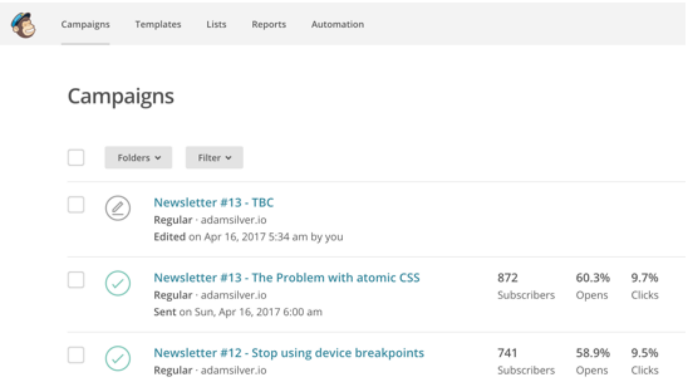
```
[aria-expanded="true"] + [role=menu] {
  display: block;
}
[aria-expanded="false"] + [role=menu] {
  display: none;
}
```

When focus is on a menu item, pressing **Down** or **Up** arrows will move to the next or previous item, on loop.

Pressing **Escape** on a menu item will move focus to the menu button and closes the menu. Sometimes, **Home** and **End** are used to go to the first and last items directly, which is particularly useful if there are lots of menu items.

Select All

Users may want to archive every email in their inbox. Rather than selecting each email one by one, we can provide a more convenient method. One way to service this functionality is through a special checkbox, placed at the top and in vertical alignment with the other checkboxes, creating a visual connection. Clicking it would check every checkbox in one fell swoop.



MailChimp's campaign list page showing a select all checkbox positioned top-left of the list.

Arguably, this standard checkbox has all the ingredients of an accessible control. It's screen reader and keyboard accessible. It communicates through its label and change of state. Its label would be "Select all" and its state would be announced as "checked" or "unchecked." All this behavior without any JavaScript.

By now, the benefits of using standard elements should be well understood. Despite this control being accessible by mouse, touch, keyboard, and screen readers, it just doesn't quite feel right. Accessibility is only a part of inclusive design. This control should look like what it does.

The trouble with using a checkbox is that they don't signal what they do. Like select boxes, they are associated with collecting data for submission. We should match people's expectations by using the same interface component for the same job. In doing so, the interface becomes familiar and consistent which speaks to *inclusive design principle 3*, "Be consistent."

Instead, we can employ a simple button, labeled "Select all," that when clicked will check all the checkboxes. At the same time, the button's label will change to "Deselect all." Clicking the button will then uncheck all the checkboxes putting them back into their original state.

```
<!-- When unselected -->
<button type="button">Select all</button>
<!-- when selected -->
<button type="button">Deselect all</button>
```

Note: We looked at how to implement an alternative toggle button using the `aria-pressed` attribute in chapter 1 for the password reveal pattern (see page 39).

Success Messages

When the user submits the form, the selected emails will disappear from the inbox. When an action has been completed, telling users is the respectful thing to do. Not doing so leaves users wondering what happened, if anything.

In chapter 1, "A Registration Form," we designed and constructed an error summary panel that resides at the top of the page. A success message needs a similar treatment with a couple of tweaks.

Instead of being red, it should be green, which is conventionally associated with success. Second, the content should be "You've successfully archived 15 emails" (or similar).

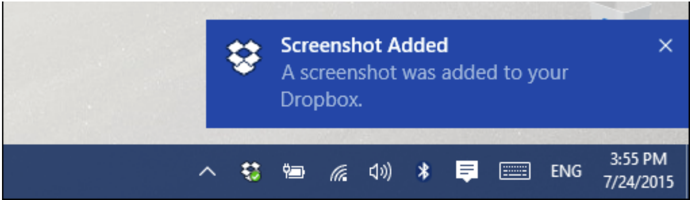
You've successfully archived 15 emails

```
<div class="successMessage" role="alert">
  <h2>You've successfully archived 15 emails</h2>
</div>
```

Both the error and success message panels are placed within the natural flow of the document and toward the top of the page to indicate their importance. The `role="alert"` attribute ensures screen readers will announce it when the page loads or if it is updated on the client.

TOAST MESSAGES

Some applications employ what is known as a "toast" message or notification. When the application needs to notify users, a little (non-modal) dialog will pop up on the page — a bit like a piece of toast from a toaster. Then, after a certain amount of time, the notification disappears automatically, usually with a fading animation.



A toast notification on Windows, positioned bottom-right, just above the taskbar.

This is all very interesting from a design perspective, but it's hardly a useful way to communicate. First, the message obscures the content beneath. Second, users have to read the message before it disappears. This makes comprehension a stressful task and takes control *away* from the user.

A success message should be laid out bare and placed within the natural flow of the page. There's no need to obscure parts of the interface. After all, the message is temporary and will naturally disappear when the user leaves the page.

However, if users are likely to stay on the page for a long time after, research might show that dismissing a message is valuable after all. Offer that functionality with a button. When clicked, it hides the message.

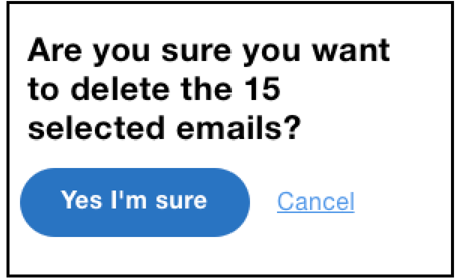
You've successfully archived 15 emails Dismiss

A success message panel with "Dismiss" button.

Be careful to inject the `<button>` with JavaScript. If we put it in directly in the HTML, then the interface will appear broken when JavaScript is unavailable; that is, nothing will happen when it's clicked.

CONFIRMING VERSUS UNDOING

As a safety measure, some roads have speed bumps. They compel drivers to slow down on roads that are more likely to cause accidents. We can create a digital speed bump by asking users to confirm their action by asking them if they're sure.



An "Are you sure" confirmation screen with an option to confirm or cancel the action.

This is fine for infrequent tasks, but it quickly becomes tedious when that action needs to be performed more often. Continuing with the driving analogy, then: it's a bit like putting speed bumps on the motorway. They'd probably cause more accidents than they'd stop.

An alternative approach would be to let users perform the action immediately, without any warning. Then, along with the success message, give users the chance to undo the

action. Clicking "Undo" would reverse the action by restoring the emails back to the inbox. If only we could undo accidents on the road.

A success message panel with "Undo" button.

Summary

In this chapter we began by choosing the right way to present a collection of emails and the impact of combining two disparate modes — reading email and actioning it — into one interface.

We then looked at how a multi-select form is different from most other types of form, and how this caused us to consider several other visual and interactive design treatments.

Finally, we looked at ways in which to add value and put users firmly in control by designing consistent interfaces that give users feedback and a way to undo their actions.

THINGS TO AVOID

- Using the wrong element for the job and fixing it with JavaScript.
- Using ARIA when standard HTML can be used instead.
- Using checkboxes and select boxes for buttons and menu components.
- Doing work that doesn't add value, such as highlighting selected rows.
- Disabling submit buttons until the form becomes valid.
- Hiding notifications automatically.
- Putting speed bumps in front of repetitive tasks.

DEMOS

- Inbox: http://smashed.by/inboxdemo

A Search Form

I'm an organized person. Even as a boy, I remember always having a place for things. To be fair, I've always been minimalist too. Organizing when you only own a few things is easy. So it's no surprise I rarely lost things. On the odd occasion I did, I just shouted in the general direction of the resident search engine: "Where's my…," and I'd have my answer.

By search engine, I mean Mum! Mum knew where everything was, not just my stuff — everyone's. This was one of her many excellent qualities. She didn't just know where stuff was, she knew the answer to everything (at least, that's how I remember it). If I had grown up with search engines, I might have nicknamed her Google.

As I've gotten older and become a husband and father, my life is richer but also less minimalist. Even if I meticulously organize all our belongings, it's still hard to remember where it all is. Worst case scenario: I have to peruse each cupboard hoping that the thing hasn't been lost — which is time-consuming.

In this chapter, we're going to design a responsive search form. Like Mum, we'll want it to be readily available and on hand to answer *any* question users have. To make this happen, there are some crucial things to consider.

Search Everything

Not only was I able to ask Mum where my stuff was, really I was able to ask her anything. When designing a global search form, users should be able to do the same thing. Too often, users can only find stuff that lives in the database. On Amazon, search will only return products. On YouTube, search will only return videos.

In "Content and Design Are Inseparable Work Partners" Jared Spool explains that "content is the thing the user needs right now."[1] He recounts a story from user research where someone was trying to buy a purse.

The woman was happy enough to buy the purse on the proviso she could return it. But the returns policy wasn't on the product page, or in the FAQ. In the end, she tried typing "Refund policy" into the search box — but it didn't return any results. That was the end of the research session.

1 http://smashed.by/contentanddesign

The returns policy isn't a product. Nor does it reside in the database. But this is what she wanted. We often hear how content is king, but the design of the search function let the content down.

Wherever possible, search should search everything. And if it doesn't, the label should be explicit. If the search function only returns products, make that clear within the interface.

Search | **Product search**

Bad | Good

Left: generic search label "Search." Right: specific search label "Search products."

> *A tip*: use analytics to track what users are searching for. If the most popular searches retrieve empty results, make provisions to improve the experience based on data.

The Basic Form

The search form is simple enough and contains just three elements: the label, search input, and submit button.

Search

Search

A search form.

```
<div role="search">
  <form>
    <div class="field">
      <label for="search">
        <span class="field-label">Search</span>
      </label>
      <input type="search" id="search" name="search">
    </div>
    <input type="submit" value="Search">
  </form>
</div>
```

Notes

- As noted in chapter 3, “A Flight Booking Form,” the search input (`<input type="search">`) lets users clear the field more conveniently than a standard text box, by clicking the delete cross or by pressing **Escape**.

- The search form has a search landmark role: `role="search"`. Like other landmarks, this means it will be listed as a shortcut in most screen readers. The extra `<div>` is necessary because putting the landmark attribute directly on the `<form>` would override its semantics.[2]

There's No Room

Typically, search is placed within the header. Like navigation, this makes it easily discoverable and quick to access. Putting such an integral feature elsewhere on the page would be counterintuitive and unconventional.

The challenge, of course, is that it's hard to fit the search form inside the header along with everything else. The header is premium screen real estate. That is, there isn't much room available and it's highly sought after. The more we put into the header, the more the main content is pushed down the page.

As noted in earlier chapters, we're often seduced by novel, space-saving techniques, such as the hamburger menu,[3] but hiding content should always be a last resort. On desktop,

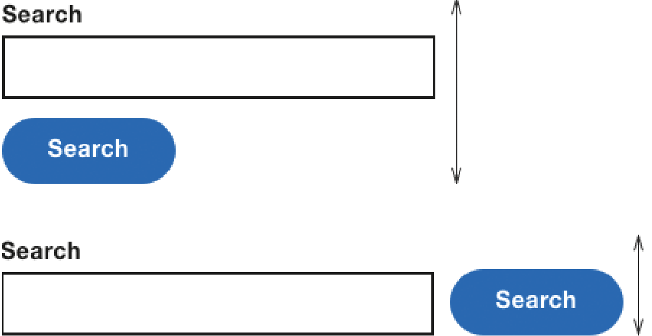
2 http://smashed.by/searchrole
3 http://smashed.by/hamburgermenu

the issue of space isn't much of, well, an issue — there's usually plenty of room. On mobile, though, we're going to have to think a bit more. There's only so much space available to play with.

Let's look at some ways to reduce the size of the search form, so that it might fit inside the header more easily. We'll discuss various problems and considerations we need to think about with each technique.

PLACE THE SUBMIT BUTTON INLINE

As explained in chapter 1, "A Registration Form," the best place for the submit button is directly below the final field. But one way to reduce the amount of vertical space the search form takes up is to place the submit button next to the field.



Left: submit button below the search field. Right: submit button next to the text box to save vertical space.

It's a bit of a special case, and it's acceptable if the search form consists of just one form field. But some search forms offer more than one field. In this case, the submit button should go directly below the last field as usual.

HIDING THE LABEL

Another space-saving technique is to hide the label. You might consider using the `placeholder` attribute to supplant the label, but this is problematic for a number of reasons. See chapter 1, "A Registration Form," for an in-depth rundown.

We could forgo a visible label altogether because arguably the submit button acts as a quasi-label for sighted users. But you should still include a label for screen reader users: they shouldn't have to skip ahead to the button in the hope that its label provides a clue.

```
<div class="field">
  <label for="search" class="visually-hidden">
    <span class="field-label">Search</span>
  </label>
  <input type="search" id="search" name="search">
</div>
```

> *Note:* The CSS for the visually hidden class is set out in chapter 2, "A Checkout Form" (page 116).

If search only retrieves products, for example, then the button would be better labeled “Search products.” This way, users know what they can and can’t search for. As noted above, users should be able to search everything — although technological and resource limitations may come into play.

Also, if you recall the hint and error patterns from “A Registration Form,” (on page 56) the text is injected into the `<label>`. If you need to show this information, you’ll have to come up with a new solution, which seems unnecessary. Besides, hiding the label doesn’t usually save enough space to fit the form inside the header anyway, especially in smaller viewports.

HIDING THE BUTTON

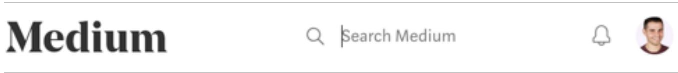
We might also consider hiding the button, but this has several pitfalls. *First*, without a button you can’t use it as a quasi label; the label would need to be reinstated, taking up room again.

Second, the button is a fundamental part of a form — without it, it’s not clear how users are meant to perform the search. While *we* may be aware that forms can be submitted implicitly (by pressing **Enter**), not all users are. See chapter 5, “An Inbox,” for more information about implicit submission.

Third, if your search form contains more than one field, omitting the submit button stops implicit submission from

working. Fortunately, if your search form consists of a single field, implicit submission will still work without a button.

Interestingly, many of the sites that omit the submit button normally find room to include a magnifying glass icon to signify its otherwise hidden affordance. In this case, they may as well place the icon inside a submit button solving both problems at the same time.



Medium's search form lacks a submit button but has a magnifying glass icon

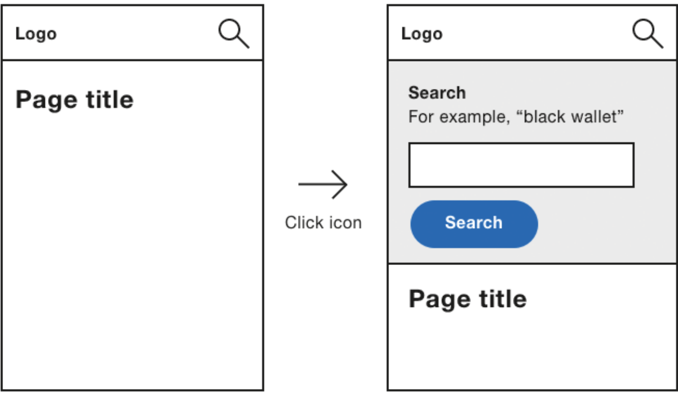
> *Note:* If you decide to hide the button using the `visually-hidden` class, remember the button will still be focusable. This means sighted screen reader users, for example, will find this problematic.[4] You can fix this by adding the `tabindex="-1"` attribute.

Toggling the Form's Visibility

Even if removing the label and submit button didn't degrade the usability of the form, it still wouldn't really solve the space problem enough to fit the form comfortably within the header.

4 http://smashed.by/notallblind

Instead of messing around with pixels to this extent, we can toggle the entire form's visibility by letting users click a button. Finding room in the header for a button is relatively straightforward. And including a visible label, a hint (if needed), and a submit button is an easy task.



Left: hidden search form. Right: search form revealed.

THE BASIC MARKUP

The basic page consists of a header containing the logo and navigation. The search form is positioned underneath the header.

```
<header>
  ...
</header>
<div role="search">...</div>
```

THE ENHANCED MARKUP

When JavaScript is available, the markup is enhanced like this:

```
<header>
  ...
  <button type="button" aria-haspopup="true" aria-
expanded="false">Search form</button>
</header>
<div role="search" class="hidden">...</div>
```

Notes

- A toggle button is injected into the header.
- The search form is hidden using the `hidden` class as introduced in chapter 1.
- The `aria-haspopup` attribute indicates that the button reveals a part of the interface (the search form). It acts as warning that, when pressed, the focus will be moved to the search form.
- The `aria-expanded` attribute tells screen reader users whether the search form is currently expanded or collapsed by toggling between `true` and `false` values.

A SMALL SCRIPT

```
function SearchForm() {
  this.header = $('header');
  this.form = $('.searchForm');
  this.form.addClass('hidden');
  this.button = $('<button type="button" aria-
haspopup="true" aria-expanded="false">
<img src="/public/img/magnifying-glass.png" width="20"
height="20" alt="Search products"></button>');
  this.button.on('click', $.proxy(this, 'onButtonClick'));
  this.header.append(this.button);
}

SearchForm.prototype.onButtonClick = function() {
  if(this.button.attr('aria-expanded') == 'false') {
    this.button.attr('aria-expanded', 'true');
    this.form.removeClass('hidden');
    this.form.find('input').first().focus();
  } else {
    this.form.addClass('hidden');
    this.button.attr('aria-expanded', 'false');
  }
};
```

Notes

- The constructor is responsible for enhancing the HTML and listening to the button's click event.
- When the button is clicked, the script checks the `aria-expanded` attribute to see if the form is currently expanded or collapsed.

- If the form is collapsed, then the `aria-expanded` attribute is set to `true`, and the `hidden` class is removed to reveal the form. Finally, the first input is focused, which saves users an unnecessary extra click.
- If the form is expanded, the `aria-expanded` attribute is set to `false` and the form is hidden by adding the class of `hidden` to the form.

Displaying Search Results

Displaying search results is somewhat out of scope for a book about form design, but let's run through some important details quickly now:

- **Maintain search text.** When the user arrives at the search page, what they typed should persist. This way users can make tweaks without having to retype the entire query.
- **Display result count.** Tell users how many results have been returned. A simple approach would be to update the page's `<title>` text to read "Search results for [search term]" or similar. Users can then decide what their next action is. For example, if there are many results, they may decide to filter them (more on this in the next chapter).

- **Let users sort.** Depending on the dataset being searched, it's often useful to let users sort by relevance, popularity, or recency, for example.
- **Don't employ infinite scrolling by default.** It's an anti-pattern with several usability issues.[5] This leaves "Show more" or standard pagination. "Show more" is more appropriate for sites with a lot of user-generated content, where the location of the result is not important. Pagination is more appropriate for ecommerce sites, where users are looking for a specific item, not just browsing for entertainment.

Summary

In this chapter we started by looking at how important it is to give users what they searched for — not just products, or articles, but anything the site contains.

We then went on to look at the interface; specifically, how we can accommodate a search form as part of the header so that it's readily accessible from every page in the site.

We also enhanced the experience for screen reader users by using the `role="search"` landmark.

5 http://smashed.by/infinitescrolling

CHECKLIST

- Search everything, not just what's stored in a database.
- Use the `role="search"` landmark to help screen reader users access search quickly.
- If the form can't fit easily in the header, let users toggle its display with JavaScript.
- When displaying search results, avoid infinite scrolling.

Demos

- Search form: http://smashed.by/searchformdemo

A Filter Form

In the introduction to "A Search Form" you'll recall the type of conversation I used to have with Mum. Sometimes I would ask, "Where's my black top?" But this was so vague that Mum would respond with questions like, "Is it a football or tennis top?" This question is a filter on a large set of results. Without knowing the answer to this question, Mum couldn't respond with an accurate answer.

Filters (also referred to as facet navigation or guided navigation) let users refine a large set of search results. This helps users home in on what they're looking for.

First of all, though, it must be said that if you don't need a filter, don't include one. They're only useful if searching returns a vast amount of results. On Google, where searching can yield thousands, if not millions of results, most people aren't willing to click beyond the first or second page.

Letting users filter out irrelevant results is important. The ability to filter not only offers an additional dimension of control, but it does so in a way that matches each user's own mental model. In "Designing for Faceted Search," Stephanie Lemieux says:

Think of a cookbook: authors have to organize the recipes in one way only — by course or by main ingredient — and users have to work with whatever choice of organizing principle that has been made, regardless of how that fits their particular style of searching. An online recipe site using faceted search can allow users to decide how they'd like to navigate to a specific recipe, e.g. by course type, cuisine or cooking method.[1]

At first glance, filters might look similar across different sites, but their behavior varies quite widely. When and how filters should be applied, how to denote selected filters, what elements should be used, how to give users feedback, how they'll work on mobile and desktop: all of these things need to be taken into account.

Interactive Filters versus Batch Filters

There are two ways to let users filter: one at a time (interactive filtering), or selecting multiple filters at once (batch filtering). In "User Intent Affects Filter Design," Katie Sherwin describes an excellent way to think about this:[2]

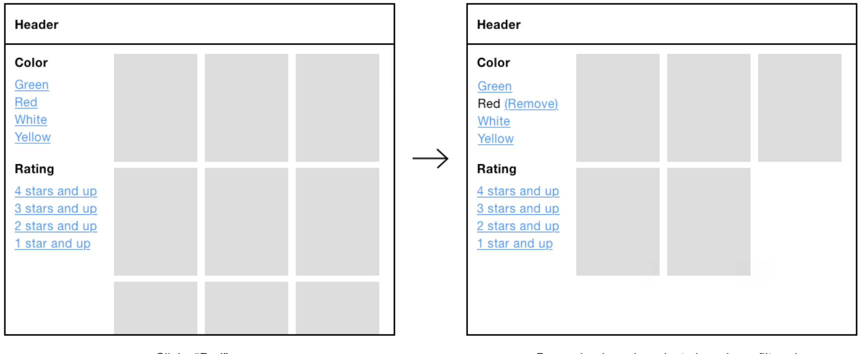
[…] think about how you might order appetizers at a restaurant. Say you want to order three appetizers for the table, but as soon as you name the first one, the waiter snatches the menu out of your

1 http://smashed.by/facetedsearch
2 http://smashed.by/applyingfilters

hands and walks back to the kitchen to get the chefs started on cooking that dish. Instead, a good waiter understands that you're still in the process of ordering and knows to give you more time before taking away the menu. A good waiter ***allows you time to make a batch decision****, even if that might slightly delay the delivery of the first item ordered. (However, sometimes the waiter may take the appetizer order, and then give you more time to decide on the main course. A good waiter is flexible and adapts to the needs of the customers.)*

INTERACTIVE FILTERS

Interactive filters update as soon as the user clicks a filter. The advantage is that users will see the results update as they go.

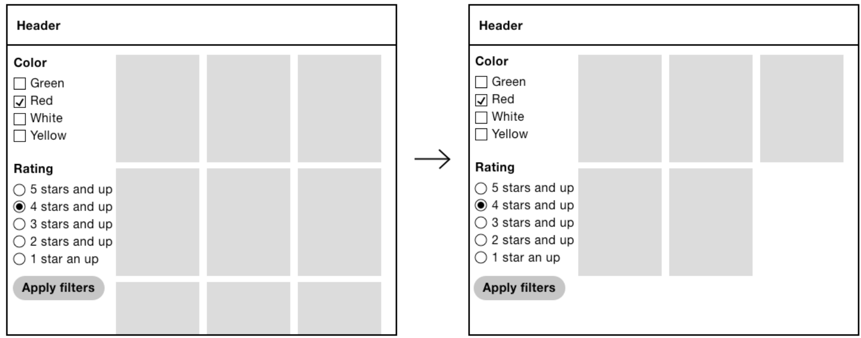


Left: an interactive filter with no filters selected. Right: the same, but with the red filter selected.

One disadvantage is that each click causes a page refresh, which can cause frustration due to lag and getting scrolled back to the top of the page — something that will happen every time the user selects a filter. This is especially problematic for keyboard users as they'll have to tab back to where they were.

BATCH FILTERS

Batch filters work by letting users set a number of options before submitting and reloading the page (see above). One advantage of this approach is that it's faster, as users just make one request for several filters.



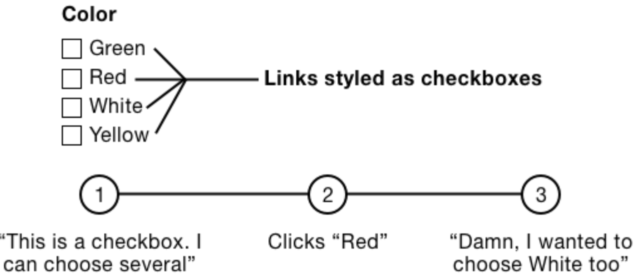
Left: a batch filter with filters about to be submitted. Right: the page with the filters applied.

One disadvantage of this approach is that a combination of filters could lead to zero results.

MATERIALLY DISHONEST INTERFACES

We've already discussed the concept of material honesty several times in the book. This is because, rather unfortunately, dishonest interfaces are prevalent on the web. As a reminder, one material shouldn't be used as a substitute for another because the end result is deceptive. In short, we should use the right material for the job. But what does this mean for filters?

As shown above, interactive filters tend to use links because they provide the expected behavior — that clicking a filter would immediately request the new page of results. But some sites style links to look like checkboxes by using CSS background images, for example.[3] But checkboxes are for input, not for requesting new pages.



Links styled as checkboxes.

3 http://smashed.by/cssbackground

The problem is that a link should look and behave like a link, not a checkbox. Batch filters, made from real checkboxes, let users select several filters. Making links look like checkboxes means users wouldn't expect clicking a filter would immediately request the new results. That's materially dishonest and therefore deceptive.

Why would designers do this? One possibility is that without making the links look like checkboxes, users wouldn't know that they could choose multiple filters within the same category. Or, in the case of radio buttons, they wouldn't know they could choose only one filter.

Breaking widely understood conventions without a very good reason can seriously harm the user experience. That's all well and good as a theory, but how do we take this information and design a filter that works?

We'll find out by taking things step by step.

WHICH TYPE OF FILTER IS BEST?

Only conducting your own research for your problem can tell you the answer to that, but we'll go with batch filters. This seems prudent because:

- users can choose multiple filters at once creating a faster experience.

- radio buttons and checkboxes have natural signifiers that indicate that one or multiple filters can be selected within the same category.
- users can still, if they wish, select just one filter at a time with a batch filter, which speaks to *inclusive design principle 4*, “Give control.”

Layout

Before tackling the complexity of the filter form itself, it’s important to look at it in the context in which users are likely to use it. In modular design, we can fall prey to focusing so deeply on the individual components that we forget to check how everything works when they combine to form the page (or journey).

The wallets category page as seen on desktop, with filters on the left and results on the right.

In this case, the page has been carefully arranged so the relationship between the filter and the products is clear. This has been achieved by using the correct heading levels:

1. The primary heading (level 1) is at the top: “Wallets.”
2. Then there is a subheading (level 2) for each component: “Filter” and “Products.”

We’ve married the correct heading hierarchy *semantically* (as we’ll see in the code to follow) with how they’re sized and positioned *visually*. That is, the top-level heading comes first and is the biggest. The second-level headings come after and are smaller. This is important because not only does the filter let users find products more quickly, but it also provides context for the products in view.

The Markup

As the form is made from standard form components, the code for our filter form should be familiar.

```
<main>
  <h1>Wallets</h1>
  <aside class="filter" aria-labelledby="filter-heading">
    <h2 id="filter-heading">Filters</h2>
    <form role="form" method="get" aria-labelledby="filter-
heading?">
```

```
        <fieldset class="field">
          <legend>
            <span class="field-legend">Color</span>
          </legend>
          <div class="field-options">
              <div class="field-checkbox">
                  <label for="color">
                    <input type="checkbox" name="color"
value="green" id="color">
                    Green
                  </label>
             </div>
          <div class="field-checkbox">
             <label for="color1">
                <input type="checkbox" name="color"
value="red" id="color1">
                  Red
                </label>
             </div>
             <!-- more checkboxes -->
          </div>
        </fieldset>
        <fieldset class="field">
          <legend>
             <span class="field-legend">Rating</span>
          </legend>
          <div class="field-options">
             <div class="field-radioButton">
                <label for="rating">
                   <input type="radio" name="rating"
value="4" id="rating">
                   4 stars and up
                  </label>
            </div>
```

```
          <div class="field-radioButton">
            <label for="rating1">
                <input type="radio" name="rating"
value="3" id="rating1">
                3 stars and up
              </label>
          </div>
          <!-- more radio buttons -->
        </div>
      </fieldset>
      <!-- other filter categories -->
      <input type="submit" value="Apply filters">
    </form>
  </aside>
  <div class="results">
    <h2>Products</h2>
    <!-- products -->
  </div>
</main>
```

Notes

- There are three headings on the page: the top level (`<h1>Wallets</h1>`) and a level-two heading for the filter and products components. Many sites provide an incomplete and broken heading structure — for example, by replacing `<h2>Products</h2>` with `<h1>Wallets</h1>`. However, this orphans the `<h2>Filter</h2>`, which deceives both sighted and non-sighted users because users expect that a second-level heading comes after the first.

- The specification has recently changed to allow headings inside legend elements. We could, then, consider marking up the text inside the legends as `<h3>`s. This would give screen reader users an alternative way to navigate the filter, which is sometimes called multimodality and speaks to *inclusive design principle* 5, "Offer choice."
- Notice the form has a `role="form"` attribute. This may seem counterintuitive, but it turns the form into a landmark, which makes it navigable in screen readers using shortcuts. Since the basic functionality works without JavaScript and triggers a page refresh, this helps users navigate back to the form from the top of the document. It also means users can browse the products and still get back to the filter component quickly.
- Similarly, the filter is marked up as an `<aside>`, which is another type of landmark, typically used to denote a sidebar. An aside should be tangentially related to the main content, which suits the filter component well.
- We're using the GET method on the form to rebuild the page from the server without relying on client-side JavaScript at this stage. For example, submitting the form with the "Red" and "3 stars and above" options selected will build a page with

?color=red&rating=3 as the query parameter. We'll look at enhancing the page with Ajax later.

- The form contains a number of fields which have been included for the purpose of example. The type of form control you use should be based on the type of behavior your users need. As noted in previous chapters, checkboxes should be used for multiple selections; radio buttons if only one can be selected.
- Keyboard users can press the arrow keys (left and right, up and down) to select a radio button. Pressing the down arrow key, for instance, will focus and select the next radio button with the `name="rating"` attribute: "3 stars and up." In screen readers this will announce "rating, three stars and up, selected, radio button, two of four" (or similar).

Automatic Submission

As noted above, the filter form (like any other form, I might add) lets users select as many filters as they like before submitting them. This standard and conventional behavior should be familiar to users — except this is not always the case.

I interviewed Dave House, a former designer for Gumtree, a site which uses filters extensively. Dave and his team conducted many rounds of usability tests and here's what he said:

> *On desktop, Gumtree users would select filters without submitting them. They didn't expect to have to submit their choices. We heard a lot of feedback saying "Your filters are broken."*

Owing to the materially dishonest design of filters as explained earlier, it seems some people have come to expect that clicking a checkbox (or radio button) will reload the results without having to submit. This phenomenon is known as Jakob's law:[4]

> *Users spend most of their time on other sites. This means that users prefer your site to work the same way as all the other sites they already know.*

Automatically submitting the form when a filter is selected would effectively convert our batch filter into an interactive one. That's a shame, as not only would we be exacerbating the problem of dishonest design, but we'd be forgoing the inherent advantages of batch filters.

4 http://smashed.by/endofwebdesign

And this may work for radio buttons and checkboxes, but what if there were text boxes that could be used to enter a price range? When would users expect the form to submit? Submitting while typing is out of the question. This leaves submitting the form on blur (tabbing or clicking out of the field), which is odd and unintuitive. We'd need a submit button just for that box.

If your users have the same expectations as Gumtree's users, then you may have no choice. But before going to such lengths, let's explore some other techniques to help users realize that submission is necessary.

First, the button should look like a button and be styled prominently to stand out on the page. And second, the button should be within easy reach (within reason). Some options include:

- Making each filter category collapsible (we'll look at how to do this later)
- Duplicate the submit button at the top of the form
- Consider making the buttons stay on screen by using `position: sticky`.[5]

5 http://smashed.by/positionsticky

Should We Just Change to Links?

Perhaps we should just throw away the form and use links. The advantage would be we wouldn't need JavaScript — users would get standard (link) behavior for free.

But there are several disadvantages:

- We can't use links for the dynamic price range inputs, for example, which is unnecessarily constraining.
- We may need to signify the "select one" or "select multiple" behavior across different link filters, which is challenging without making links look like checkboxes (here we go again).
- If users select many filters then they may exceed the max limit of the query string. Posting a form averts this problem.

Really, we're exchanging one set of problems for another, arguably larger set. Let's see how we can submit the form automatically using JavaScript and the issues that may arise from doing so.

SUBMITTING THE FORM AUTOMATICALLY

If, despite our efforts to thwart breaking convention, users still expect the form to submit automatically, we can use JavaScript to submit the form in response to the form's change event.

As our filter only contains checkboxes and radio buttons, we can first remove the now redundant submit button. However, we can't completely remove the button from the document (with `display: none`, for example) because some platforms (iOS for one) will not submit forms when a submit button isn't present. And, as mentioned in chapter 6, omitting the submit button stops users being able to submit the form implicitly, with the **Enter** key. In which case, we can use our special `visually-hidden` class, plus `tabindex="-1"` to make sure the button isn't user-focusable.

```
function FilterRequester() {
  this.form = $('.filter form');
  this.form.find('[type=submit]')
    .addClass('visually-hidden')
    .attr('tabindex', '-1');
}
```

Now the submit button has been hidden accessibly, we can submit the form when a filter is changed by listening to the change event:

```
function FilterRequester() {
  //...
  this.form.find('[type=radio], [type=checkbox]').
on('change', $.proxy(this, 'onInputChange'));
}
FilterRequester.prototype.onInputChange = function() {
  this.form.submit();
};
```

You should note that this fails Web Content Accessibility Guidelines Success Criterion 3.2.2:[6]

Changing the setting of any user interface component does not automatically cause a change of context

Additionally, keyboard users operating the filters must use their arrow keys to move through the radio buttons. Each arrow keypress not only focuses adjacent radio buttons, but also selects them. As a result, keyboard users won't be able to move from one radio button to another without the form being submitted. What if they want the third or fourth radio button?

Even if you ignore the difficulties associated with certain interaction modalities, having the page refresh in the middle of choosing filters is a poor user experience. Let's see if Ajax can fix these issues.

6 http://smashed.by/constbehavior

Ajax

Ajax is a technology that lets users dynamically update parts of an interface without a page refresh. The advantage for our filter form is that users can select as many filters as they like without being interrupted by a page refresh and the focus moving to the top of the document.

To do this, we can listen to the change event and fire off an Ajax request:

```
FilterRequester.prototype.onInputChange = function() {
  var data = this.form.serialize();
  this.requestResults(data);
};
FilterRequester.prototype.requestResults = function(data)
{
  $.ajax({ data: data, ...});
}
```

While the main issue has been solved, the introduction of Ajax has created additional problems. Let's discuss each of these now and see how we might deal with them.

COMMUNICATING LOADING STATES

When a web page is loading, the web browser shows a loading indicator. This loading indicator is accurate, accessible, and as it's part of the browser shell it appears in the same

place no matter the website, making it trustworthy and familiar.

When Ajax is used, we have to provide our own mechanism to inform users that the request is loading. This is normally the purview of a loading spinner.

A loading spinner.

But you should note that unlike the browser, it doesn't tell users how long is left, or if the connection is slow. In the next chapter, we'll look at ways to provide an accurate progress bar with Ajax.

Also, the loading spinner, in its current form, is only determinable by sighted users. To provide a comparable experience (*inclusive design principle 1*) for screen reader users, we'll employ a live region (as first set out in "A Checkout Form").

```
<div aria-live="assertive" role="alert"
class="visually-hidden">Loading products.</div>
```

When the products are loaded:

```
<div aria-live="assertive" role="alert" class="visually-
hidden">Loading complete. 13 products listed.</div>
```

Note: As the loading spinner is enough communication for sighted users, the live region is given the special `visually-hidden` class as set out in "A Checkout Form."

BREAKING THE BACK BUTTON

When a user loads a web page, the browser refreshes and puts the previous page into its history. This allows users to press the back button to return to it quickly. But when Ajax is used to make updates, it's not put into the browser's history: technically, the user hasn't navigated to another page.

Despite this, if the page looks like it has significantly changed, then users will expect the back button to work as normal. Baymard's usability study found that breaking the back button's behavior caused confusion, disappointment, anger, and even abandonment.[7]

This is the sort of thing that can happen when a little enhancement breaks convention. This is why the best experiences are usually the simplest.

Fortunately, we can use HTML5's History API, which was designed to help solve this problem.[8] First, we have to create a history entry when the Ajax request succeeds, like this:

7 http://smashed.by/macysfilter
8 http://smashed.by/historyapi

```
FilterRequester.prototype.onInputChange = function() {
  var data = this.form.serialize(); // color=red&rating=3
  this.requestResults(data);
  history.pushState(data, null, '/path/to/?'+data);
};
FilterRequester.prototype.requestResults = function(query)
{
  $.ajax({ ..., success: $.proxy(this, 'onRequestSuccess',
query)});
};
FilterRequester.prototype.onRequestSuccess =
function(query, response) {
  history.pushState(response, null, '/path/to/?'+query);
  //...
};
```

Notes

- The first parameter is the state which we want to be stored with the history entry. In this case, it's the JSON response that's used to render the updated page.
- The second parameter is the title. As it's not well supported and it's not necessary in our case, we're ignoring it by passing null.
- The third parameter is the history's URL, which is the URL including the query string.

With this in place, we need to listen to when the history changes:

```
function FilterRequester() {
  //...
  $(window).on('popstate', $.proxy(this, 'onPopState'));
}
FilterRequester.prototype.onPopState = function(e) {
  this.requestResults(e.originalEvent.state);
};
```

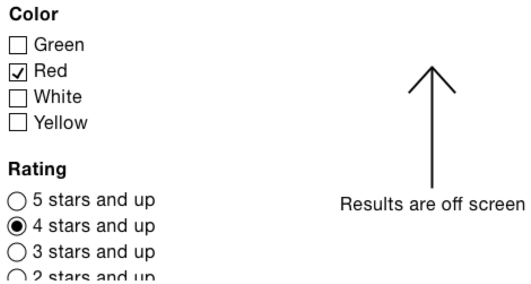
Notes

- The `state` property contains the JSON response we associated with the history entry on creation. It's then passed to the already written `requestResults` method so it can be used to render the page again without an AJAX call.
- As we're using jQuery to listen to the `onpopstate` event, the state (normally `e.state`) property is found in `e.originalEvent.state`.

Users May Not Notice the Results Update

Another problem with making updates using Ajax is ensuring that users notice the results update. Take a situation where the filter component is very long and scrolls beyond the fold. Imagine that while selecting a filter toward the

bottom, it returns too few results such that the user sees a blank screen.



Page showing a long list of filters with the few results now offscreen.

We can't move focus as that defeats the entire point of introducing Ajax in the first place. There are two ways we might solve this problem.

First, we can set a maximum height on the filter and give it an additional inner scroll bar. However, inline scroll areas are really hard to use; so much so, that Baymard Institute says they should be avoided.[9]

Second, we can use progressive disclosure to collapse the filter categories. We'll be looking at this in detail shortly.

9 http://smashed.by/inlinescroll

BATTERY DRAIN AND DATA ALLOWANCE

As we're now making heavy use of Ajax to rerender the page every time the user selects a filter, this will cause users' data and battery to be eaten up at a more rapid rate.

AJAX IS NOT NECESSARILY FASTER

Despite popular belief, Ajax is not necessarily faster than a page refresh.

First, it must be said, that Ajax is subjected to much of the same latency as a standard page request. If the site (or connection) is slow, then so too will be the Ajax request. In fact, an Ajax request can be even slower.

This is mostly because it engineers away progressive rendering (also called chunking) which the browser provides for free. In "Fun hacks for faster content," Jake Archibald explains the concept of progressive rendering:[10]

When you load a page, the browser takes a network stream and pipes it to the HTML parser, and the HTML parser is piped to the document. This means the page can render progressively as it's downloading. The page may be 100k, but it can render useful content after only 20k is received.

10 http://smashed.by/fastercontent

As Ajax has to wait for the entire 100Kb before showing anything, users have to wait a lot longer to see something.

This is not to say Ajax is bad. It's just that we should use it judiciously and when we know that users will benefit from it. We introduced Ajax on the assumption that users needed it, but where possible we should, generally speaking, reserve the use of Ajax for making smaller page updates, for which it is better suited.

In the end, we can only be sure of what's best by conducting user research with a diverse group of people, using a broad range of browsers and devices, on varying connection speeds in the context of our own problem.

> *Note:* Screen size shouldn't be used to infer fast or slow connection speeds. Like many other people, most of my time on mobile involves me being connected to Wi-Fi. An inclusive experience is one that is made fast for all, by not adding superfluous features and bloat to a page in the first place.

Collapsible Filters

If your filter has many categories and many options within those categories, we need to be sure users aren't overloaded with too much choice (as explained in chapter 4, "A Login

Form”). The best way to do this is to have fewer filters, so don’t include ones that users don’t need.

Additionally, we can collapse the categories. This is advantageous for a number of reasons:

1. The submit button is more likely to draw users’ attention as it will be in view.
2. Ajax-injected results will likely be in view.
3. Users shouldn’t have to scroll nearly as much, while still being able to scan the filter categories.
4. Keyboard users won’t have to tab through all the filters to get to the one they want. This is because hidden content isn’t focusable.

ENHANCING THE MARKUP

The basic markup consists of standard form fields. As an example, this is what the color field markup looks like:

```
<fieldset class="field">
  <legend>
    <span class="field-legend">Color</span>
  </legend>
  <div class="field-options">
    <!-- checkboxes here -->
  </div>
</fieldset>
```

The JavaScript-enhanced markup will look like this:

```
<fieldset class="field">
  <legend>
    <button type="button" aria-expanded="false">Color</button>
  </legend>
  <div class="field-options hidden">
    <!-- checkboxes here -->
  </div>
</fieldset>
```

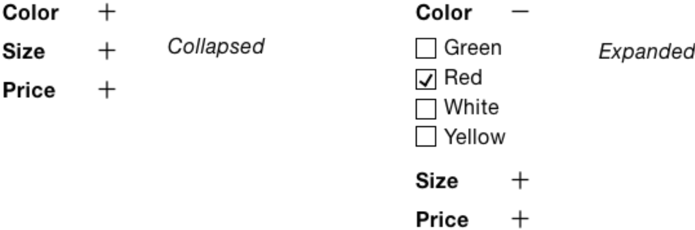
The legend now contains a button element with a `type="button"` attribute, which stops it from submitting the form. We don't want it do that: we just want it to expand and collapse the filters.

Had we instead converted the legend into a button using ARIA's `role="button"`, we would be overriding the legend's semantics. This means screen readers wouldn't announce the legend as the group's accessible label. We would have also had to recreate all the free browser-provided behavior associated with the `<button>` element, such as being focusable and activated by pressing **Space** and **Enter** keys.

STATE

The checkboxes are hidden by the `hidden` class, as first explained in chapter 1, "A Registration Form." Removing it will reveal the checkboxes.

The button has an `aria-expanded` attribute, initially set to false, which denotes that the section is collapsed. When the button is clicked this will be switched to true, which means it's expanded. For screen readers, the button will be announced as "Color, collapsed, button" (or similar, depending on the screen reader).



Left: collapsed filters. Right: expanded filters.

We also need to communicate the state of the component visually. Replacing the entire legend with a button element is not ideal because we still want the legend to look like what it is: a legend. By the same token, the interface needs to make it clear that clicking the legend will toggle the filter.

We can signify this functionality with the conventional plus (can be expanded) and minus (can be collapsed) symbols, though up and down triangles may work just as well. Let's make use of a lightweight SVG icon placed inside the button:

```
<button type="Button" aria-expanded="false">
  Color
  <svg viewBox="0 0 10 10" aria-hidden="true"
focusable="false">
    <rect class="vert" height="8" width="2" y="1" x="4" />
    <rect height="2" width="8" y="4" x="1" />
  </svg>
</button>
```

The `aria-hidden="true"` attribute hides the icon from screen readers — the button's text is enough. The `focusable="false"` attribute fixes the issue that in Internet Explorer SVG elements are focusable. And the `class="vert"` attribute on the vertical line allows us to show and hide it based on the state using CSS, like this:

```
[aria-expanded="true"] .vert {
  display: none;
}
```

Script

All the script does is create and inject a button and toggle visibility when clicked. Here's the entire script:

```
function FilterCollapser(fieldset) {
  this.fieldset = fieldset;
  this.options = this.fieldset.find('.field-options');
  this.legend = this.fieldset.find('legend');
```

```
  this.createButton();
  this.hide();
}
FilterCollapser.prototype.createButton = function() {
  this.button = $('<button type="button" aria-
expanded="true">'+this.legend.text()+'<svg viewBox="0 0 10
10" aria-hidden="true" focusable="false"><rect class="vert"
height="8" width="2" y="1" x="4" /> <rect height="2"
width="8" y="4" x="1" /></svg></button>');
  this.button.on('click', $.proxy(this, 'onButtonClick'));
  this.legend.html(this.button);
};
FilterCollapser.prototype.onButtonClick = function(e) {
  this[this.button.attr('aria-expanded') == 'true' ? 'hide'
: 'show']();
};
FilterCollapser.prototype.hide = function() {
  this.button.attr('aria-expanded', 'false');
  this.options.addClass('hidden');
};
FilterCollapser.prototype.show = function() {
  this.button.attr('aria-expanded', 'true');
  this.options.removeClass('hidden');
};
```

Small-Screen Experience

Up to now, we've only considered the interface in the context of desktop-sized screens, where there's enough space to fit the filter next to the results. But what about the small-screen experience?

With a mobile-first mindset, if you cut out the superfluous content and lay out what remains, the experience usually works well. And this approach scales up easily for large viewports: increasing the font size and white space is usually enough.

However, the two components (the filter and the results) are closely weighted in terms of importance. Really, the filter needs to be as prominent as the results, something we've been able to achieve in desktop-sized screens.

We can't just put the filters first, as this will push the results down the page. And we can't just put them after the results, as users would have to move beyond them — most users wouldn't know they exist.

We're left with having to collapse the filters behind a toggle button. We've covered this behavior extensively in chapters 5 and 6. Now we're going to focus on another related problem.

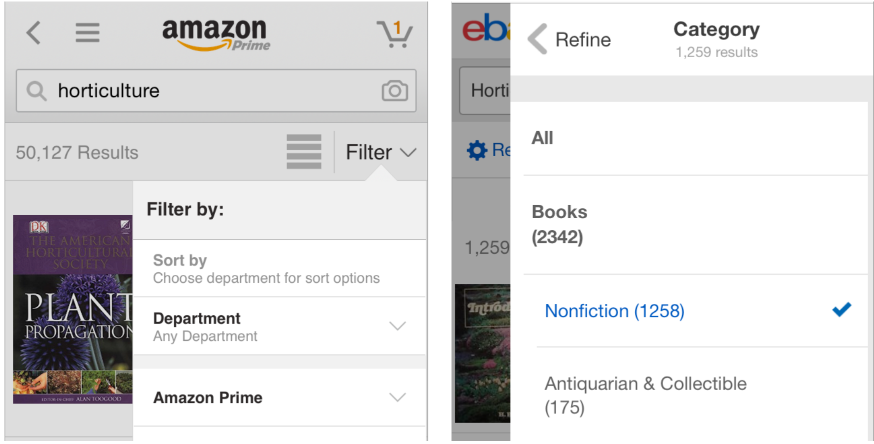
Earlier, I mentioned part of the interview I had with Dave House, a former Gumtree designer, who conducted a lot of research on how to deal with filters. In particular, that desktop users expected the filter to update the results as the user clicked filters with Ajax. However, Gumtree's research also showed that on mobile this wasn't desirable because users couldn't see the results refresh. Here's what Dave said:

On mobile, Ajax wasn't desirable because users couldn't see any visible refresh of the results. The amount of filters available in some categories meant this would have happened off-screen. To the user this would have looked like nothing had changed. We didn't want to move focus to the results on every interaction because users often wanted to pick more than one filter. We reluctantly had to use an adaptive approach. On mobile, clicking the "Refine" menu button would reveal a fullscreen filter panel where users could build up their refinements and submit when they were done.

Gumtree reluctantly went for an adaptive approach, giving mobile users and desktop users different experiences. But there might be a way to give users a more responsive experience — one that conforms to *inclusive design principle 3*, "Be consistent."

By applying this principle we ensure that users get the same, familiar conventions applied consistently. In this case, users learn how to use the filters once, to then use them on any device or screen size.

We need to make sure that users can see the results update as they filter. We can achieve this by having the filters appear on top of the results without completely covering them. It works because users can see the results on the left, while filters are selected on the right.



Both Amazon (left) and eBay (right) have the filter appear on top of the results, allowing users to see the results update as filters are selected.

I found this technique described in “Mobile Faceted Search with a Tray: New and Improved Design Pattern” by Kathryn Whitenton, which is worth reading in full.[11]

Denoting Selected Filters

When the user selects a filter, it becomes checked as standard. This is an important signifier. Keeping selected filters in their original position alongside other selected filters is useful, because some users will remember where the filter was when they originally selected it.

11 http://smashed.by/mobilefacetedsearch

We can also help users by grouping the selected filter in a separate list at the top. This confirms to users that what they selected is being viewed. It also gives users easy access should they want to remove any of the active filters, without having to scroll through them scanning for the marked filters.

We can achieve this by adding a little component to the top of the filter, like this:

```
<div class="current-filters" aria-labelledby="current-
filters">
  <h3 id="current-filters">Current filters</h3>
  <ul>
    <li><a href="...">Red (Remove)</a></li>
  </ul>
</div>
```

Notes

- The section is appropriately labeled with a third-level (`<h3>`) heading as it sits under the second-level “Filter” heading.
- The surrounding `div` is labeled by the heading. This means the heading will be announced for screen reader users who have tabbed to (focusable) links.

Summary

In this chapter we've nimbly covered several design details that often crop up with filters. While we've persistently tried to keep to convention, non-conventional approaches have been explored that may be needed to satisfy users' new expectations — expectations that have, unfortunately, been born out of the many materially dishonest interfaces present on the web today.

With that said, we've carefully made the effort to include a number of provisions that give users a good and inclusive user experience, should they need the Ajax-driven, automatically submitted and responsive filter component. And we've done that, by applying five of the seven inclusive design principles set out in the introduction; namely: provide a comparable experience, consider situation, be consistent, give control, and offer choice.

THINGS TO AVOID

- Making links look like radio buttons and checkboxes.
- Automatically submitting forms without exhausting other simpler techniques.
- Assuming Ajax always delivers a faster and better user experience.
- Prioritizing best practice above user needs.

Demos

- Filter Form: http://smashed.by/filterformdemo

An Upload Form

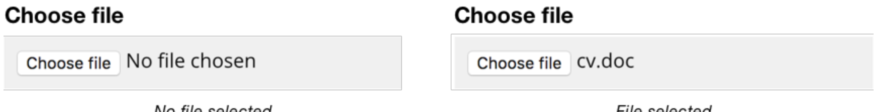
The web is more than just text. Whether it's sending a CV to a recruiter by email, or adding photos to an eBay advert, we need to let users upload files. Forms have this capability baked in, of course.

On one hand, uploading a file is only marginally more complex than, say, inputting text or clicking a checkbox. On the other hand, there are number of unique design challenges and opportunities that arise, especially when there's a need to upload multiple files at the same time.

As usual, we'll start by looking at what browsers give us for free. After that, we'll look at adding various enhancements and the various issues that surface as a result of those enhancements. We'll end up with a number of different ways to upload a file, appropriate for several different occasions.

A File Picker

A file picker (`<input type="file">`) is another type of form control. When clicked, it will spawn a dialog that lets users browse files on their computer or device. Once a file is selected, the dialog closes and the picker updates to reflect the file has been chosen.



Left: a file picker without a file selected. Right: a file picker with a file selected.

If all users need to do is upload a single file, then you can add a file picker to your form, and you're pretty much done:

```
<form enctype="multipart/form-data">
  <div class="field">
    <label for="documents">
      <span class="label">Choose file</span>
    </label>
    <input class="field-file" type="file" id="file" name="file">
  </div>
  <input type="submit" value="Upload" name="upload">
</form>
```

Notes

- The form has an `enctype="multipart/form-data"` attribute, which ensures the file is transmitted to the server for uploading.
- The file picker uses the same pattern as first described in "A Registration Form" and throughout the book, which can take a hint and error message.

RESTYLING THE FILE PICKER IS DANGEROUS TERRITORY

Some designers like to restyle the file picker to:

- achieve consistency between different browsers and operating systems
- match the brand's look and feel
- be able to configure the control's text

Whether you agree with all of these reasons or not, it must be said that *pretty and useless* is considerably worse than *ugly and useful*. But this doesn't mean aesthetics aren't important: where possible we should marry form and function together.

However, it's just as important to make sure that any techniques we employ to achieve these goals don't cause any adverse usability issues. That's a bit like taking one pill to fix one symptom, only to need additional pills to relieve the side effects that come from the first.

Styling file pickers has always been tricky because browsers ignore any attempt at doing so with CSS. We have to resort to hacking, which is not usually a good idea — that's why it's called hacking. But let's walk through how it could work, what can be achieved, and the pitfalls that are involved.

Hiding the Input

The most robust way of styling the file picker is to visually hide it, like this:

```
<div class="field">
  <label for="file">
    <span class="label">Choose file</span>
  </label>
  <input class="visually-hidden" type="file" id="file"
name="file">
</div>
```

> *Note:* The CSS for the `visually-hidden` class is set out in "A Checkout Form."

Now that it's hidden, we can style the control's label, which *is* easy to style. As described in "A Registration Form," this works because a control's label acts as a proxy to the control itself: clicking the label is like clicking the input.

Styling the Label

Now the input is hidden, we need to style the label so it looks interactive. We need to style it as a button and change its text to "Upload file."

Left: a label styled as normal. Right: the modified label styled as a button.

Focus States

Now the label looks and is clickable, we need to think about focus states.

As the input is visually hidden, the user won't get any feedback that it's in focus when they tab to it. To do this, we can use JavaScript to add a class of `focused` to the label when the input is focused, which will allow us to style it:

```
.focused {
  /* focus styles */
}
```

Reflecting the Chosen File

When the user selects a file from the dialog, it's the input that will change state (as shown earlier). To reflect the chosen file, we'll need to update the label text when the input's `onchange` event fires.

```
$('[type=file]').on('change', function(e) {
  // change label
});
```

Upload file | cv.doc

No file selected | File selected

Left: the button-styled label before file selection. Right: after selection.

Pitfalls

On the face of it, this implementation is visually pleasing and still accessible. Keyboard, mouse, and touch users can operate it normally, and screen readers will announce the value of the input.

But that's not all it takes to design a fully inclusive and robust custom file picker interface. There are a number of additional considerations that this solution doesn't solve very well at all.

1. Updating the label to reflect the input's value is confusing because the label should describe the input and remain unchanged. In this case, screen reader users will hear "cv.doc" as opposed to "Attach document."

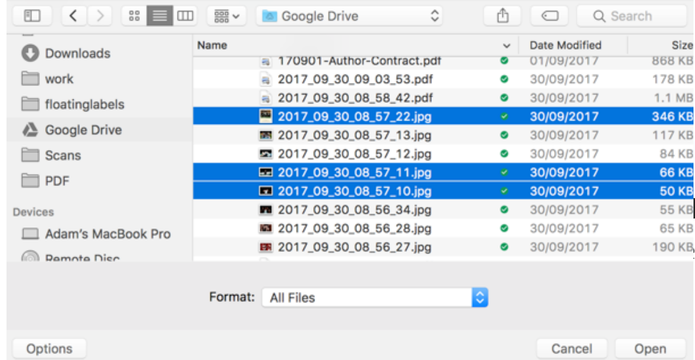
2. The interface doesn't fit with the established convention of providing hint and error text, as set out in "A Registration Form." Not only would we need to think of another way to provide this information, but it creates an inconsistent and unfamiliar user experience.
3. File inputs are actually drop zones, which means they let users drag and drop files (instead of going through the dialog). Hiding the input means forgoing this behavior, which some users may prefer.
4. There's not much room inside the button for a large file name. Remember, good design adapts well to varying lengths of content.

Considering the pitfalls, the improvement to aesthetics doesn't seem to justify the downgrade in usability and utility.

A Multiple File Picker

Very few tasks on the web, require a user to upload just a single file at a time. Take the two examples from the introduction to this chapter. Both attaching files to an email and uploading photos to an eBay advert involve uploading several files in one go.

The easiest but most problematic way to solve this would be to add the `multiple` boolean attribute to the file input:



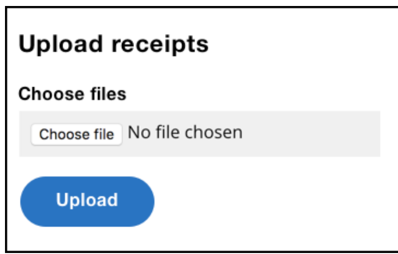
Multiple file dialog on macOS.

That would be it, if you ignored two significant problems.

First, users can only select files within a single folder. If they need to upload files from different folders, they can't. Of course, users could move all the files into a single folder beforehand but this puts the onus on the user.

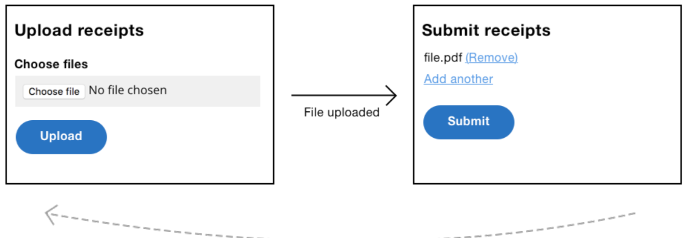
Second, not all browsers support the `multiple` attribute. And when support is lacking, a single file input may be found wanting.

For example, take a form which asks users to submit receipts. When the `multiple` attribute is supported, users can upload all the relevant receipts and submit them. Without support, users can only upload a single receipt.



A multiple file form.

One way to solve this problem involves giving users a way to add more files as part of the flow:



Multiple file form with extra screen to let users continue adding files.

Not only does this design let users upload multiple files in unsupported browsers, but it also lets the user review their submission, which is a useful addition regardless.

A Persistent Upload Form

Most forms are ephemeral — users submit a form and they're taken to another page without a form. For example, when registering, users are taken to a confirmation screen.

But the task of uploading files means it's useful to give users a form that persists until they're finished on their own terms. This is the *persistent form pattern* in action.

HOW IT MIGHT LOOK

The user can choose and upload a file repeatedly until they've uploaded all the desired files. At which point, they can click the Continue button.

Left: a persistent upload form before a file has been uploaded. Right: the same form with a file uploaded and the upload form beneath.

THE MARKUP

```
<form enctype="multipart/form-data">
  <div class="field">
    <label for="documents">
      <span class="label">Attach file</span>
    </label>
    <input class="field-file" type="file" id="documents"
name="documents" multiple>
  </div>
  <input type="submit" value="Upload" name="upload">
</form>
```

Note the file input has the `multiple` attribute. When used in conjunction with a persistent form, the `multiple` attribute becomes a robust enhancement. Where supported, users can select multiple files at a time, meaning fewer requests and a streamlined experience.

However, when not supported, users can keep uploading a single file at a time, as many times as they need to until the task is finished. This solves the problem I mentioned earlier regarding uploading files from different folders.

A Drag-and-Drop Enhancement

As noted earlier, the native file input acts as a drop zone to let users drag and drop files. However, there are two problems with it.

First, it's not immediately obvious that dragging and dropping is even possible — there are no signifiers that make this behavior perceivable. Second, the drop zone has a small hit area, which makes it hard to use, especially for motor-impaired users.

To solve these issues, we're going to take our persistent upload form and progressively enhance it with better drag-and-drop functionality.

WARNING: IS DRAG-AND-DROP NECESSARY?

Depending on the situation, the humble file picker may be all that users need to upload files. In this case, you may not need to worry about adding a drag-and-drop enhancement at all. This way, there's less code to send to the user. As a result, page load times are faster, and the interface is simplified at the same time.

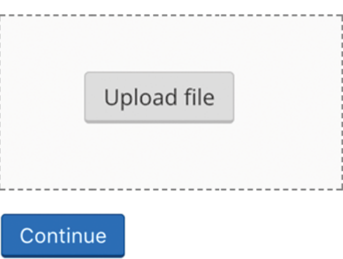
It's also worth noting that a drag-and-drop enhancement is just that — an enhancement. It should be used in conjunction with a standard file picker. First, because users can't actually drag and drop files on mobile, for example. Second, users with dexterity problems, such as tremors, may have difficulty dragging a file.

By giving users both choices, we're safely following *inclusive design principle 5*, "Offer choice."

HOW IT MIGHT LOOK

The large drop zone is more ergonomic, especially for people with motor impairments. It's conventionally styled with a dashed border. However, if your users aren't familiar with this convention, you can add instructional text.

Inside the drop zone sits a button. When clicked, it triggers the dialog as normal. The button is actually a label *styled* as a button using the ill-advised technique from earlier. But I haven't gone mad, there's good reason for this.



An upload form enhanced with large drop zone.

WHY STYLE THE LABEL AS A BUTTON

The drop zone has two methods of interaction: dropping files onto the drop zone, and clicking the button. Browsers don't let you programmatically update a file input's value owing to security reasons.[1] Because of this, we can't update the file input's value, for example, when the user drops files onto the drop zone. Therefore, files will be uploaded immediately with Ajax (which we'll cover shortly).

1 http://smashed.by/dragdropupload

Remember, the form has two methods of interaction: dropping files onto the drop zone, and clicking the button. For consistency we want both approaches to upload files immediately (drop zone `ondrop` and input `onchange`). This way, users don't have to think about when (or when not) to submit the form — that interaction is no longer an option.

THE ENHANCED MARKUP

Here's the JavaScript-enhanced markup:

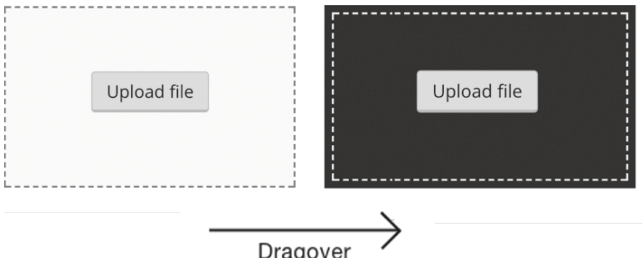
```
<form class="dropzone">
  <div class="field">
    <label for="files">Upload file</label>
    <input type="file" name="files" id="files" multiple>
  </div>
</form>
```

Notes

- The button has been removed because the files will be uploaded with Ajax `onchange`.
- The "dropzone" class exists as a way to target this particular form for enhancement.

DRAGOVER AND DRAGLEAVE EVENTS

When the user is dragging files over the drop zone, they should be given feedback so they know that the files can be dropped.



Left: drop zone. Right: drop zone while file is being dragged over it.

We can achieve this by adding a class to the drop zone when the `ondragover` event is fired. Similarly, we need to remove the class when the user leaves the drop zone (`ondragleave`):

```
Dropzone.prototype.onDragOver = function(e) {
  e.preventDefault();
  this.dropzone.addClass('dropzone-dragover');
};
Dropzone.prototype.onDragLeave = function() {
  this.dropzone.removeClass('dropzone-dragover');
};
.dropzone-dragover {
  /* styles here */
}
```

Notes

- `e.preventDefault()` is called to allow the file to be dropped onto the drop zone. Without doing this, the browser will try and load the dropped file instead.
- We can't just use `:hover` because the feedback should only be given when a user is dragging a file over the drop zone — not just when the cursor happens to be over the drop zone.

DROPPING FILES

Next we need to handle the file drop, which we can do by listening to the `ondrop` event.

```
Dropzone.prototype.onDrop = function(e) {
  e.preventDefault();
  this.dropzone.removeClass('dropzone-dragover');
  $('.fileList').removeClass('hidden');
  this.uploadFiles(e.originalEvent.dataTransfer.files);
};
```

Notes

- `e.preventDefault()` is called to allow the file to be dropped onto the drop zone. Without this, the browser will attempt to load the file instead.

- The `dragover` highlight is removed as the file has now been dropped.
- The file list component is revealed, ready to give users feedback as the files are uploaded. More on this shortly.
- The event object (`e`) contains information about the files, which is handed over to the `uploadFiles` method (shown below)

```
Dropzone.prototype.uploadFiles = function(files) {
  for(var i = 0; i < files.length; i++) {
    this.uploadFile(files[i]);
  }
};
```

This method loops through each file and calls the `uploadFile` method, which is explained next.

UPLOADING THE FILE

Uploading the file involves two steps: creating the data to be sent, and actually sending it.

```
Dropzone.prototype.uploadFile = function(file) {
  var formData = new FormData();
  formData.append('documents', file);
  $.ajax({
    data: formData
    url: '/ajax-upload',
```

```
    type: 'post',
    processData: false,
    contentType: false
  });
};
```

The FormData API is designed to construct key/value pairs that represent form fields and their values, which can then be sent with Ajax, including forms that contain files (like ours does). First, we create a new instance, then we append the file data to it.

For convenience, we're using jQuerys `$.ajax` method. Here's a rundown of the properties used:

Property	Description
`data`	The data constructed with `FormData`.
`type`	Set to "post" because data is being sent.
`url`	The URL/endpoint for which the server will process the uploaded files.
`processData`	Set to `false`, which tells jQuery not to convert the data into a query string. This is important as we're sending files, not just text.
`contentType`	Set to `false`, which tells jQuery not to override the automatically created header appropriate for sending files. [2]

2 http://smashed.by/multipartform

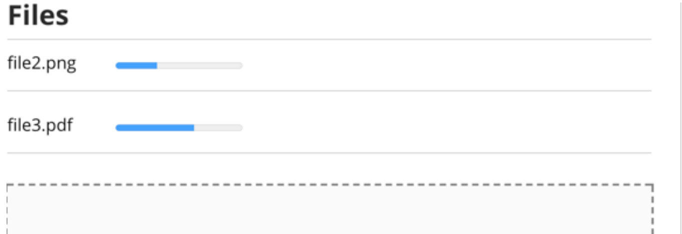
FEEDBACK

It's all well and good having uploaded the files to the server, but at this moment the user hasn't been given any feedback as to what's happened. Perhaps the file couldn't be uploaded, for example. There are a number of times we need to give users feedback: during upload, on success, and on error.

Progress

Files can take a long time to upload, especially if the connection is slow. It's important to give users feedback during upload — not just on completion.

We can show feedback with a progress bar. Each file is represented separately as there's a separate request for each one. This way, some small files will upload quickly, while others load more slowly in parallel.



File list with progress bar for each one.

```
<ul>
  <li>
    <span class="file-name">file.pdf</span>
    <progress max="100" value="80">80% complete</progress>
  </li>
  <li>
    <span class="file-name">file.pdf</span>
    <progress max="100" value="50">50% complete</progress>
  </li>
</ul>
```

Supporting browsers display the `<progress>` element as a progress bar. The element has two attributes: `max` and `value`. The `max` attribute describes how much work there is to be done. In our case, it's set to 100 as we're working in percentages. The `value` specifies how much is complete, which we'll be updating with JavaScript.

```
$.ajax({
  xhr: function() {
    var xhr = new XMLHttpRequest();
    xhr.upload.addEventListener('progress', function(e) {
      if (e.lengthComputable) {
        var percentComplete = e.loaded / e.total;
         percentComplete = parseInt(percentComplete * 100);
         li.find('progress')
           .prop('value', percentComplete)
           .text(percentComplete + '%');
      }
    }, false);
    return xhr;
  }
});
```

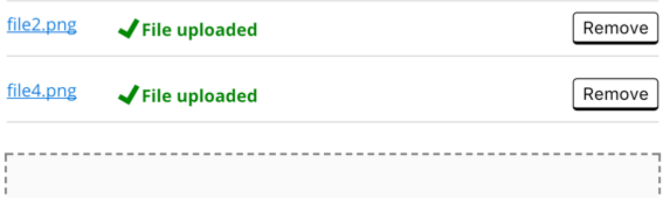
As jQuery (at the time of writing) doesn't expose the `onprogress` event, we've created an `XMLHttpRequest` object ourselves.

The handler first checks to see if the server has correctly sent a `Content-Length` header by seeing if `e.lengthComputable` is `true`. If it is, then we can determine how much of the file has been uploaded, which is calculated by dividing `e.loaded` by `e.total`. That value is then converted to a percentage before updating the progress bar.

The progress bar's inner text is also set. This is so users with a browser that lacks support for the `<progress>` element can still see it. That's inclusive.

Success

Next, we want to show users when a file has been successfully uploaded. First, the file name is converted into a link so users can download and verify the file if they wish. Second, we inject a success message of "File uploaded" and a Remove button, which is useful if the file was uploaded by mistake.



File list with successfully uploaded files marked as such.

```
<li>
  <a class="file-name" href="/path/to/file.pdf">file.pdf</
a>
  <span class="success">
    <svg width="1.5em" height="1.5em">
      <use xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="#tick"></use>
    </svg>
    File uploaded
  </span>
  <input type="submit" name="remove1" value="Remove">
</li>
$.ajax({
  success: $.proxy(function(response){
    if(response.file) {
      li.html(this.getSuccessHtml(response.file));
    }
  }, this)
});
```

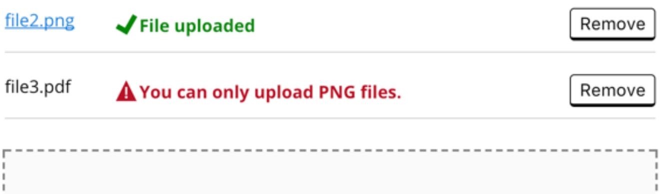
We're using the `success` callback, which receives the response from the server as an object. The response contains a `file` property, which contains the path and name of

the file. This is used to create the HTML that is injected into the list item.

> *Note:* The demo uses Multer[3] and Express[4] to process the request and generate the response object. You can use whatever you like.

Error

If the uploaded file is too big, or in the wrong format, we'll need to show users an error message. This is similar to the success message, but instead of showing a green success message with a tick, we'll show a red message with a warning symbol. Note that the error markup below is the same as the error markup used to show validation errors in a standard form.



File list with unsuccessfully uploaded files marked with error messages.

3 http://smashed.by/multer
4 http://smashed.by/expressjs

```
<li>
  <span class="file-name">file.pdf</span>
  <span class="error">
    <svg width="1.5em" height="1.5em">
      <use xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="#warning-icon"></use>
    </svg>
    File.pdf is too big.
  </span>
  <button type="button">Remove</button>
</li>
$.ajax({
  success: $.proxy(function(response){
    if(response.error) {
      li.html(this.getErrorHtml(response.error));
    } else if(response.file) {
      li.html(this.getSuccessHtml(response.file));
    }
  }, this)
});
```

The updated success function now checks to see if the response has an `error`. If it does, the error state will be constructed and injected into the list item instead.

Screen Reader Feedback

While the feedback is useful for sighted users, screen reader users won't hear any feedback. To provide a comparable experience (*inclusive design principle 1*), we'll need to use a live region — something we've extensively covered in "A Checkout Form" and "A Flight Booking Form."

```
<div class="visually-hidden" role="status" aria-
live="polite">
  Uploading files. Please wait.
</div>
```

> *Note:* The live region is visually hidden because sighted users have already been catered for with the live progress bar and its various states. The CSS for the `visually-hidden` class is set out in “A Checkout Form” (on page 116).

The live region will change at various points:

When	Description
Upload starts	Uploading files. Please wait.
Upload completes	[Name of file] has been uploaded.
Upload fails	For example: [Name of file] is too big. The size must be less than 2Mb.

FEATURE DETECTION AND INITIALIZATION

Feature detection was introduced in the very first chapter and also demonstrated in “A Flight Booking Form” (on page 164). We’ll use it once again here because it’s important to detect features before using them, otherwise we’ll create a broken experience for users of browsers that lack support.

The drag-and-drop enhancement uses a number of APIs that not all browsers recognize. Here are the feature detection functions with a usage example at the end.

```
function dragAndDropSupported() {
  var div = document.createElement('div');
  return typeof div.ondrop != 'undefined';
}
function formDataSupported() {
  return typeof FormData == 'function';
}
function fileApiSupported() {
  var input = document.createElement('input');
  input.type = 'file';
  return typeof input.files != "undefined";
};
if(dragAndDropSupported() && formDataSupported() &&
fileApiSupported()) {
  var Dropzone = function(container) {
 //...
  };
}
```

There are three feature detection functions; one for each of the features that browsers might not recognize. We then make sure that there is support for all of them before defin-ing our Dropzone component.

As the Dropzone is conditionally defined based on feature detection, we need to detect the Dropzone function during initialization too. If it's defined, then the browser supports it,

meaning it's safe to initialize; otherwise users will get the basic (but not broken!) experience.

```
if(typeof Dropzone != 'undefined') {
  new Dropzone($('.dropzone'));
}
```

A NOTE ABOUT OLDER BROWSER SUPPORT

Uploading files immediately `onchange` and `ondrop` might be confusing to users because, at least conventionally speaking, forms are submitted with a separate action. However, this isn't only a conventional problem. It doesn't work cross-browser either.

For example, some older browsers won't trigger the dialog when the label is used as proxy,[5] and while the `onchange` event is supported there are two problems:

1. Choosing the same file (or a file with the same name) for a second time, won't fire the `onchange` event, which creates a broken interface.[6] The solution is to replace the entire file input after the event with a clone of itself. As the cloned input would need to be refocused, screen readers will announce it for a second time, which is mildly annoying.

5 http://smashed.by/ieinput
6 http://smashed.by/selectevent

2. The `onchange` event won't fire until the field is blurred.[7] Newer browsers offer the `oninput` event, which solves this problem because it fires the event as soon as the value changes.

Whether you need to support such browsers depends on your situation but it's worth being aware of the problems. Fortunately, the feature detection above happens to rule out the offending browsers.

Other Considerations

There's a number of additional design considerations for uploading files, some of which are deep topics in their own right. Let's run through them quickly now for completeness.

CONVERT FORMATS AUTOMATICALLY

If users need to upload a spreadsheet, we should let them upload proprietary formats, like Microsoft Excel, but also non-proprietary ones like CSV.

Whichever they choose, we can convert it into the right format when it's processed. As you can see, the "Give choice" principle is more than just offering different interaction modalities.

7 http://smashed.by/ieinput

This is another example of doing the hard work so users don't have to.

MANY USERS STRUGGLE TO FIND FILES

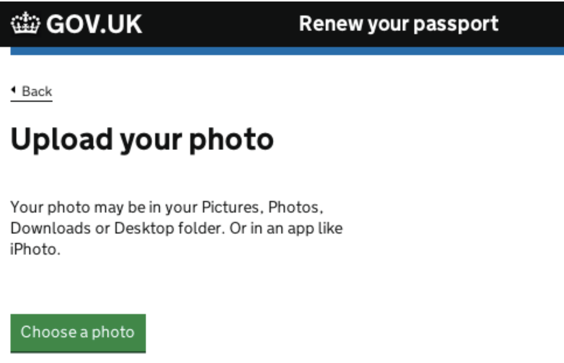
Many users, particularly less digitally savvy users, may struggle to use a file picker because they don't know where the file lives on their computer. And sometimes the file needs to be transferred to the computer from an external camera or device, creating additional effort and confusion.

Ed Horsford, a designer at GDS, conducted research around this for the UK's passport renewal service, which involves users uploading a passport photo. He said that desktop users who used a file importer (such as Windows photo importer) struggled to find where the file was imported when it came to selecting it from the file picker.

I'd class myself as digitally savvy, but I've also struggled to locate files I've downloaded from different applications. My personal workaround involves remembering to save my files to desktop. This makes the files easy to find — I just have to remember to delete the clutter every now and then.

Ed also said that while users generally found picking a photo easy on mobile, some Android devices house the photos inside "Documents" and not a folder named "Photos," which research revealed threw most users.

One simple way to help users with this is to provide additional guidance and instructions about how and where to save files, or alternatively tell users where they're likely to find the file.



Upload form with a field hint explaining that "Your photo may be in your Pictures, Photos, Downloads or Desktop folder. Or in an app like iPhoto."

IT'S EASIER TO TAKE A PHOTO ON MOBILE

Your photos are taken and stored on your mobile device, so it's easier to upload them using that device: there's no need to transfer files from elsewhere. That's all well and good if people are using your service on mobile, but what if they're on desktop?

For services that require users to already have an account, this is quite easy. If I create an advert on eBay but want to

upload photos from my mobile, I just have to login on my mobile to continue where I left off — my in-progress advert will be there ready and waiting for me.

For services that don't require being logged in, such as the passport renewal service, it's trickier. In such cases, consider directing users to their phone with one-time security codes or unique URLs that they can type easily into their phone.

Switch to your other device

Using your other device, go to:
http://bit.ly/check-photo

▸ Once you've uploaded your photo

Screen guiding users to continue the process on their mobile with a special link.

THIRD-PARTY INTEGRATION

Some digitally savvy users may already use third-party services, such as Dropbox, to store files. Giving users a way to upload or provide files from these services may well be easier for them, especially if your service is already connected with theirs.

Be warned, however, that it may be unhelpful or even confusing to users who don't know what Dropbox is. Be sure

to make the different choices clear, and test widely in user research sessions.

MICROCOPY: "UPLOAD" OR "ATTACH"

Generally speaking, there are two ways to communicate to users about uploading files. The first is to use "Attach," but this seems best suited for email. In almost every other situation "Upload" seems more common, which is what we used for our generic drag-and-drop upload form earlier.

THE ACCEPT AND CAPTURE ATTRIBUTES

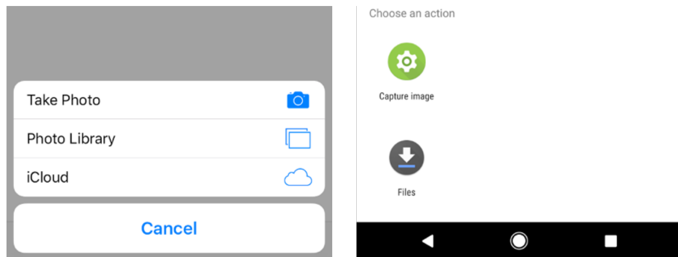
The file input has two interesting attributes that affect the file uploading experience: `accept` and `capture`.

The accept Attribute

The `accept` attribute takes a string that indicates which types of file the picker will accept.

```
<input type="file" accept="image/*">
```

When supported, the browser/device may offer users a more stringent experience when choosing files. In Chrome and Safari on iOS and Android, it will give users a choice of which app to use to capture the image, including the option of taking a photo with the camera or choosing an existing image file.



iOS (left) and Android (right) dialogs for selecting images on a device when the accept attribute is used.

But on desktop browsers it will prompt the user to upload an image file from the file system, disabling files that aren't accepted (images in the above case). The problem is, users won't be told why the files are disabled as there's no feedback.

The capture Attribute

The `capture` attribute, when supported, indicates the preference of getting an image from the camera:

```
<input type="file" accept="image/*" capture>
<input type="file" accept="image/*" capture="user">
<input type="file" accept="image/*" capture="environment">
```

Adding the `capture` attribute without a value lets the browser decide which camera to use (if there's one available), while the `user` and `environment` values tell the browser to prefer the front and rear cameras respectively. The `capture` attribute works on Android and iOS, but is ignored on desktop. Beware that on Android this means the user will no longer have the option of choosing an existing picture as the camera app will be started directly instead, which is probably undesirable.

Summary

In this chapter we began by looking at the native file picker as the browser gives us quite a bit of power for free. However, we also looked at the various problems that crop up with multiple file uploads.

From there, we looked at various solutions that started with the persistent upload form, before enhancing the interface with a more ergonomic and inclusive drag-and-drop interface.

THINGS TO AVOID

- Prioritizing form over function.
- Using the `multiple` file input without considering browsers that lack support for it.
- *Replacing* a standard file picker with a drag-and-drop interface.
- Forcing users to use a particular file format, when we can convert it for them automatically.

Demos

- Dropzone: http://smashed.by/dropzonedemo

An Expense Form

As a self-employed freelancer I have to submit expenses for my tax return. It's a pain but if I do it correctly I get tax breaks. The problem is that I have so many expenses to enter and a limited amount of time to enter them.

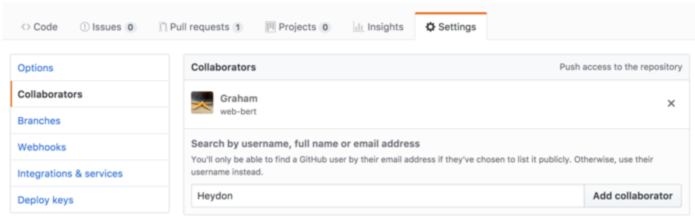
The anatomy of an expense depends on the system you use and, perhaps, the country you live in. It might include a description, company name, date, amount, and proof of purchase. How can we design a form that makes inputting multiple entries easy, fast, performant, and inclusive? Of course, if you know how many entries are needed in advance, then give users a form with that number of fields, make them required, and that's about it. But if we don't know how many entries are needed in advance (like expenses), keep reading.

The Persistent Form Pattern (Again)

In the previous chapter, I introduced the persistent form pattern. In short, we gave users an upload form that users can keep using until they've finished uploading as many files as they need, at which point they can proceed or exit the page, whichever is best (see page 324).

Left: an upload form that stays on screen to be used as many times as necessary. Right: file list above the upload form with uploaded files. Each file can be deleted.

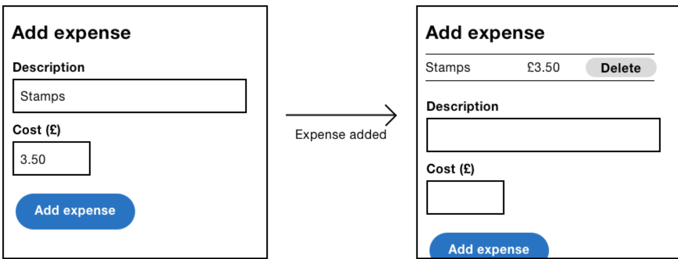
There are a number of other forms on the web that use the persistent form pattern. For example, GitHub's "Add collaborators" form and the infamous to-do list form that many JavaScript frameworks use to demonstrate their approach.[1]



GitHub's "Add collaborators" form.

This pattern works for adding expenses too. Each time the user submits an expense, it will be added to the list that sits above the form.

1 http://smashed.by/todomvc



Left: expense form before adding an expense. Right: same form with a list of added expenses above.

This pattern is well-suited to short, simple forms that can be submitted in one go. The pattern does, however, have a number of downsides:

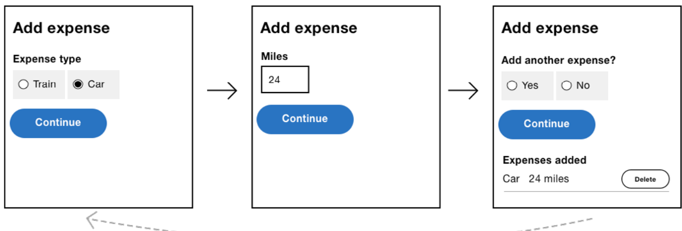
1. Users might need to use a form that has dynamic questions (branching) that are conditionally shown based on previous answers. In this case, the pattern doesn't work so well. We'll look at branching in more detail shortly.
2. As the list of added expenses grow, the form moves further down the page. This could be a problem, especially on mobile, as users would have to scroll down to see and use the form. Alternatively, you could put the list below the form, but after adding an expense users may not see the newly added item.

3. Having multiple calls to action (to submit, and to proceed or exit) might be confusing, especially for cognitively impaired users. Where possible, one call to action is preferable as it requires less thinking.
4. Each submission requires a separate request to the server. This could be frustrating when lots of entries need to be added.

Branching with One Thing Per Page

One of the major problems with the persistent form pattern is that it can't handle branching. Branching involves users being asked different questions depending on previous answers. For example, if users are expensing a car, they'll need to enter mileage; if they're expensing a train ticket, then they'll need to enter its price.

In this case, the *one thing per page* pattern (first discussed on page 70) is more suitable. This is because it presents one question at time, meaning we can show users different pages depending on their answers. This solves the branching problem elegantly and simply, but what if users need to enter multiple expenses and submit them in one go? We can add an additional screen to the end of the journey asking users if they'd like to add another expense. Selecting *yes* takes the user down the same flow again; selecting *no* completes the task.



The add expense flow using "one thing per page," with a question at the end of the flow asking if the user wants to add another one.

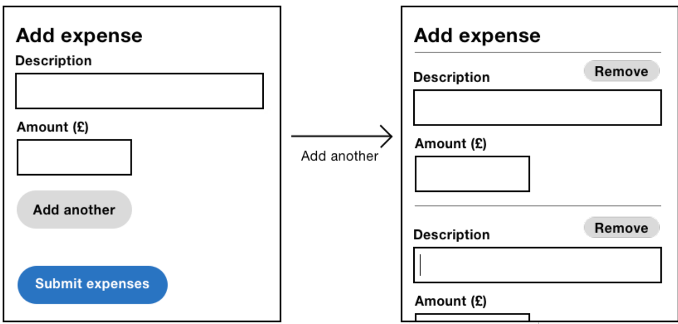
The Add Another Pattern

Both the persistent form pattern and the *one thing per page* pattern suffer from the same problem — that each expense created requires at least one trip to the server, which is slow. How might we solve this problem?

The *add another pattern* works by giving users a single form, on a single page, submitted in a single step. However, the user can keep adding fields for as many expenses as they need. For demonstration purposes, let's simplify the anatomy of an expense down to just a description and cost.

HOW IT WORKS

The form starts with enough fields to enter one expense. However, there's an Add Another button, that when pressed, will instantly clone the fields so that users can enter the details of an additional expense.



Left: add expense form with "Add another" button. Right: after having clicked the button to clone the fields allowing the user to add multiple expenses.

Users can keep on doing this until they're done, at which point the user is able to submit all their expenses at once, with just a single trip to the server — speeding up the process drastically.

> *Note:* The basic experience (before adding the JavaScript enhancement), works the same way except that pressing the "Add another" button will generate the new fields on the server.

THE BASIC MARKUP

```
<form>
  <div class="addAnother">
    <div class="addAnother-item">
      <div class="field">
        <label for="items[0][description]">
          <span class="field-label">Description</span>
        </label>
        <input
          type="text"
          id="items[0][description]"
          name="items[0][description]">
      </div>
      <div class="field">
        <label for="items[0][amount]">
          <span class="field-label">Amount</span>
        </label>
        <input
          type="text"
          id="items[0][amount]"
          name="items[0][amount]">
      </div>
    </div>
    <input type="submit" name="addAnother" value="Add
another expense">
  </div>
  <input type="submit" name="submitexpenses" value="Submit
expenses">
</form>
```

There are two important notes about this form:

1. The expense fields are wrapped in a `<div class="addAnother-item">`. This lets us target the expense with CSS and JavaScript (explained shortly).
2. The input's `name` and `id` attributes have a special array-like naming convention, which we'll discuss now.

PROCESSING MULTIPLE (DYNAMIC) EXPENSES

When the form is submitted, the payload will consist of multiple expenses. The server will need to process this payload, but it won't know how many expenses will be sent ahead of time; that is, the amount of expenses is dynamic.

To help the server recognize and process the expenses, a contract must be made between the client and the server. When it comes to forms, the contract is forged by the `name` attribute.

```
<input type="text" name="items[0][description]">
<input type="text" name="items[0][amount]">
```

Note the special `name` attribute value. By formatting it this way, the request payload can be parsed and converted into an array of expenses that the server can process easily.

Some server-side frameworks, such as Express,[2] are designed to recognize this convention. Let's run through each part and its meaning.

- `items` is the name of the group. You can use whatever name you like. On the server, this will be used to identify the list of expenses.
- `[0]` represents the particular expense in the list and starts from zero, like a JavaScript array. That is, the second expense will be represented by `[1]`, and so on.
- `[description]` and `[amount]` represent the specific attributes about the expense; in this case, the description and the amount. For each unique attribute you want to process, you'll need a name to identify it with.

ADDING ANOTHER EXPENSE

Pressing the "Add another" button needs to create a new set of expense fields. There are several ways we might go about doing this. For example, we might use templating — which could be written in JavaScript[3] or in HTML.[4]

2 http://smashed.by/expressjs
3 http://smashed.by/templateliterals
4 http://smashed.by/template

Both approaches, however, lack browser support. A simple, alternative approach involves cloning the already existing expense fields.

```
function AddAnotherForm(container) {
  container.on('click', '.addAnother-addButton',
$.proxy(this, 'onAddButtonClick'));
}
AddAnotherForm.prototype.onAddButtonClick = function(e) {
  var item = this.getNewItem();
  this.getItems().last().after(item);
};
AddAnotherForm.prototype.getNewItem = function() {
  return this.getItems().first().clone();
};
AddAnotherForm.prototype.getItems = function() {
  return this.container.find('.addAnother-item');
};
```

There are three small functions that have been split out for readability and maintainability. When the button is clicked, we get a clone of the first `<div class="addAnother-item>` in the form. Then we add the clone to the end of the form.

Injecting Remove Buttons

The form initially starts out with just a single expense. There's no "Remove" button because the user has to submit at least one expense. However, when the user adds another expense, we need to add a remove button to the

first expense. To do this, when the “Add another” button is pressed, we’ll need to check whether a remove button should be added.

```
AddAnotherForm.prototype.onAddButtonClick = function(e) {
  // previous code
  var firstItem = this.getItems().first();
  if(!this.hasRemoveButton(firstItem)) {
    this.createRemoveButton(firstItem);
  }
};
AddAnotherForm.prototype.hasRemoveButton = function(item) {
  return item.find('.addAnother-removeButton').length;
};
AddAnotherForm.prototype.createRemoveButton =
function(item) {
  item.append('<button type="button" class=" addAnother-
removeButton">Remove</button>');
};
```

Now, when the button is clicked, the function checks to see if the first expense has a remove button. If it doesn’t, one is created. The reason we have to check for its existence is because the first expense may or may not have a remove button, depending on how many expenses the user has added.

Having given the first expense a remove button, we’ll need to apply the same provision for the newly cloned expense, like this:

```
AddAnotherForm.prototype.getNewItem = function() {
  var item = this.getItems().first().clone();
  if(!this.hasRemoveButton(item)) {
    this.createRemoveButton(item);
  }
  return item;
};
```

This function uses the same helper methods in the exact same way. Now, whenever an item is cloned it will always be cloned with a remove button.

Updating the Attributes

Having cloned the fields and ensured each field has a remove button, we need to update the `name` and `id` attributes. This ensures that the newly cloned fields adhere to the naming convention so that the server can process the submission (as explained earlier).

But how can our script know what `name` and `id` values to use? To make this easy, we'll store the naming convention inside data attributes.

```
<input data-name="items[%index%][description]" data-
id="items[%index%][description]">
<input type="text" name="items[0][amount]">
```

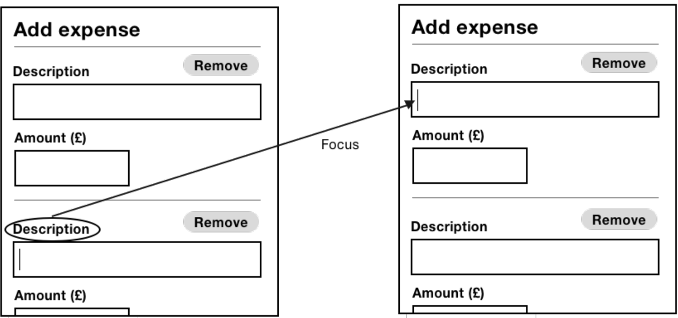
The reason for both the `name` *and* the `id` data attributes is because some fields consist of multiple inputs with the same name. For example, as laid out in chapter 2, "A Checkout Form," the `name` of each radio button is the same because it identifies the set to which they belong. The `id` identifies the individual radio button.

Now all we need to do is replace `%index%` with the new index of the cloned item, like this:

```
AddAnotherForm.prototype.updateAttributes = function(index,
item) {
  item.find('[data-name]').each(function(i, el) {
    el.name = $(el).attr('data-name').replace(/%index%/,
index);
    el.id = $(el).attr('data-id').replace(/%index%/,
index);
    ($(el).prev('label')[0] || $(el).parents('label')[0]).
htmlFor = el.id;
  });
};
```

The function works by searching for all form controls that have the `data-name` data attribute. For each control it finds, it will update the control's `name` and `id` attributes by replacing `%index%` with the new index, which has to increase by 1 each time.

Finally, the label's `for` attribute is set to the control's `id` attribute. If we didn't do this, then when the user clicks the cloned label, focus will be moved to the first field instead.



Left: clicking the second description field sets focus to the first description field erroneously.

> *Note:* The code for retrieving the label uses a logic OR operator. This is because the label appears in different places depending on the type of field. In the case of a text field, for example, it will be the previous sibling. However, for radio buttons, should an expense need radio buttons, it will be the parent.

Managing Focus

When the "Add another" button is clicked, it should focus the first newly created form field. Screen readers will announce the field, prompting the user to fill out the expense.

```
AddAnotherForm.prototype.onAddButtonClick = function(e) {
  // code
  item.find('input, textarea, select').first().focus();
};
```

REMOVING AN EXPENSE

When the user clicks an item's remove button, there's a number of tasks we need to implement. First, of course, is that it should be removed from the form.

```
function AddAnotherForm(container) {
  // code
  this.container.on('click', '.addAnother-removeButton',
$.proxy(this, 'onRemoveButtonClick'));
}
AddAnotherForm.prototype.onRemoveButtonClick = function(e)
{
  $(e.currentTarget).parents('.addAnother-item').remove();
};
```

We're using jQuery's event delegation. This is useful because we're adding and removing the remove buttons dynamically. Without using delegation we'd have to keep adding and removing event listeners, which is a pain.

When the event listener is called, the remove button is referenced by `e.currentTarget`. Then it's just a matter of searching for the parent container for the expense and removing it.

Removing the Remove Buttons

When the penultimate expense is removed, we need to remove the first item's remove button. The user shouldn't be able to remove every expense, because they must enter at least one.

```
AddAnotherForm.prototype.onRemoveButtonClick = function(e)
{
  $(e.currentTarget).parents('.addAnother-item').remove();
  var items = this.getItems();
  if(items.length === 1) {
    items.find('.addAnother-removeButton').remove();
  }
};
```

The function checks to see if there's just one expense item left in the form; if there is, the remove button is removed.

Updating the Attributes

When the user has added a number of expenses, they are free to remove any they might have provided by mistake. The interface gives users control to remove any item they want.

Remember, the names of the fields use array-like indexes so the form can be processed on the server. If there are three expenses, for example, but the user deletes the second expense, then the data will be out of sync.

We can fix this by running through all the expenses in the form and updating the fields' indexes:

```
AddAnotherForm.prototype.onRemoveButtonClick = function(e)
{
  // code
  var items = this.getItems();
  // code
  items.each($.proxy(function(index, el) {
    this.updateAttributes(index, $(el));
  }, this));
};
```

This loops through all the items in the form and invokes the `updateAttributes()` method from earlier.

Managing Focus

When the user clicks the remove button, everything inside the `<div class="addAnother-item">` will be removed. But what happens to the focus when you delete the currently focused element? Heydon Pickering answers this question in "A Todo List":[5]

[...] browsers don't know where to place focus when it has been destroyed in this way. Some maintain a sort of "ghost" focus where the item used to exist, while others jump to focus the next focus-

5 http://smashed.by/a11ytodolist

able element. Some flip out completely and default to focusing the outer document — meaning keyboard users have to crawl [...] back to where the removed element was.

We could set focus to the previous or next expense item, but this seems arbitrary and confusing. Alternatively, we could set focus to the “Add another” button, but that’s presumptuous and odd — why delete an item if you’re just going to add another one — users are better off just typing over the fields that are already there.

Instead, we can set focus to the heading, which puts users back to the beginning of the expense form while announcing itself (“Expenses, heading, level 1,” or similar) to screen reader users.

```
AddAnotherForm.prototype.onRemoveButtonClick = function(e)
{
  // code
  this.container.find('.addAnother-heading').focus();
};
```

The problem is that, by default, headings aren’t focusable. But we can fix that by giving the heading a `tabindex="-1"` attribute. The -1 value allows us to focus the element programmatically using JavaScript, without making it user-focusable, using the **Tab** key, for example.

```
<h1 tabindex="-1">Expenses</h1>
```

When the heading is focused, browsers will give the heading an outline. This should be removed because the heading is not an interactive element and so shouldn't appear interactive.

```
.addAnother-heading { outline: none; }
```

Once the heading is focused, pressing **Tab** will focus the first form field, which makes orientation simple.

FEEDBACK

As it stands, the act of adding and removing expenses provides sufficient feedback for sighted users: items in the form can be seen to appear or disappear from the list. Giving users an additional notification bar would draw users' attention in two directions. And as more expenses are added, the notification bar will be out of the viewport so users wouldn't see it anyway.

Screen reader users are also catered for because the act of adding and removing items moves the users focus and announces the focused element accordingly. This may not tell the user explicitly that the item has been added, but it should be enough. If research shows otherwise, you can add

a hidden live region (as set out in chapter 3, "A Flight Booking Form") to give screen reader users explicit feedback.

Animation Isn't Necessarily Valuable

It's possible that judicious animation effects can help users understand an interface.

But all too often, designers want to add animation for the sake of it. Users just want to get things done. Needless animation is jarring and actually *detracts* from the user experience. Like anything else, animation should only be used if it adds value.

Even when animation is valuable to some users, it can be harmful to users with cognitive impairments, such as attention deficit hyperactivity disorder (ADHD) and autism.[6]

In the case of the expense form, had we not moved focus from the button to the field, an animation might mean the item's arrival is more likely to be noticed. Alas, we *are* moving focus to the form field and so animation is unnecessary here.

6 http://smashed.by/needlessanimations

Summary

In this chapter we looked at three different patterns that let users submit multiple expenses into a system. Really though, expenses were used just for demonstration purposes — these patterns are applicable to all kinds of data, not just expenses.

The persistent form and *one thing per page* patterns are more suitable for infrequent use and users with a lower digital literacy. The *add another pattern* is more suitable for frequent usage that doesn't require branching.

The downside to the *add another pattern* is that despite the usability provisions we've put in place, the interface is a little more complicated to operate.

There's no right or wrong here. It's about choosing the most appropriate pattern for your users.

THINGS TO AVOID

- Letting browsers manage the focus when the focus element is removed from the document.
- Giving non-interactive (but programmatically fo-cusable elements) an outline when focused.
- Giving users multiple sources of feedback for a single action.
- Animating parts of the interface to no practical purpose.

Demos

- Add another: http://smashed.by/todolistdemo

A Really Long and Complicated Form

Different types of tasks take different amounts of time to complete. The one thing per page pattern (introduced in chapter 2, "A Checkout Form") helps users complete tasks in one sitting, but what about those that take hours, or even days, to complete?

In MailChimp, for example, I'll usually start drafting an email campaign weeks before I send it. There are a number of steps to complete, and in a particular order too. First, I check the content reads well. Then I need to make sure it looks good in various email clients. Then days or weeks later, I'll run through some final checks, decide the subject line and schedule it for release.

Other tasks might be performed by several people using the same application. For example, processing a return may involve someone receiving the goods at the warehouse; then a decision maker may look at the goods to make sure it satisfies the returns policy.

Some services, such as applying for a mortgage or registering for a bank account, involve supplying personal information, and this may involve offline processes such as sending identification documents in the post.

How can we design forms that play nicely with this complex and long-form process that can take weeks to complete? And by different people across digital and non-digital journeys?

First, though, it must be said that if you can simplify the process so it's short — or even remove it altogether — do that. If not, this chapter explores some useful patterns specifically designed to solve these things.

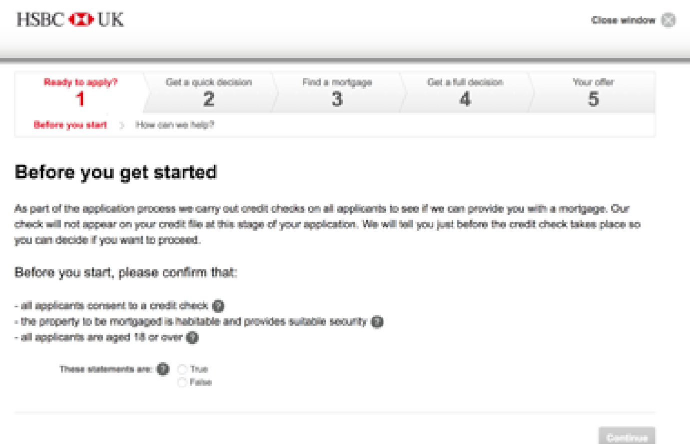
The Check Before You Start Pattern

One of the best ways we can help users save time is by not wasting it in the first place. One way to do that is to tell users what they need to know before they start the process.

For example, to apply for an HSBC mortgage in the UK, users must: consent to a credit check; confirm the property is in a habitable condition; and be aged 18 or over. HSBC tells users this on the first page of the application, *before* they start.

This is helpful, but users may also want to know:

- what the overall process involves
- if they need to visit a branch
- how long it takes to get a decision
- what documents they'll need (and what alternative options they might have, if any)

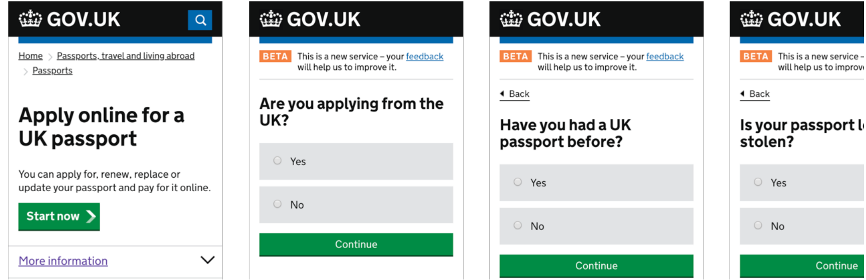


The first page of the HSBC mortgage application process explaining what users need to know before applying.

Knowing this up front not only saves users a lot of time, but it can also lower the operating costs of the service: by reducing the time and effort support teams spend working out and explaining all of this to people over the phone.

Other processes and services are more complicated and can depend on the individual's circumstances. For example, the Renew Your Passport service (shown below) needs to ask a series of questions to determine someone's eligibility to apply for a passport.[1]

1 http://smashed.by/renewpassport



Renew Your Passport service asking questions to determine the user's best course of action. The final screen tells users what to do having told the service that their password has been stolen.

If the applicant has lost their passport, for example, then they aren't able to use the main service. The Government Digital Service calls this "Check Before You Start," which you can read about in "We've published the check before you start pattern" by Harry Trimble and Rob Le Quesne.[2]

The Task List Pattern

In "The Psychology of Checklists," Lauren Marchese explains the importance of breaking down big tasks into smaller ones, which is proven to motivate people.[3] When we experience even small amounts of success, our brains release a chemical called dopamine, which gives us feelings of pleasure, learning, and motivation.

2 http://smashed.by/govukchecklist
3 http://smashed.by/smallgoals

Most of us work in teams employing agile methodologies. This involves breaking down a large project into epics, stories, and tasks. Complete enough tasks and the story is done. Complete enough stories and the epic is done. Complete enough epics and the project is done. Of course, our work is never done, but you know what I mean.

What's really happening is that tasks seem far easier to achieve when they're broken down. Crucially, if tasks are small enough, then we'll get that hit of dopamine more frequently, which creates momentum. Momentum improves morale, and morale improves productivity.

That's not all checklists are good for. In *The Design of Everyday Things*, Don Norman says:

Checklists are powerful tools, proven to increase the accuracy of behavior and to reduce error [...]. They are especially important in situations with multiple, complex requirements, and even more so where there are interruptions.

As people perform digital tasks on the go using different devices, the chances of interruption are high. Designing for interruption and being able to jump back into the middle of a long and complex task is crucial. As mentioned earlier, some parts of the process happen offline too.

The task list pattern, as coined by the Government Digital Service (GDS), shows a page with several top-level tasks.[4] Each top-level task is broken down into several subtasks. Each one of those takes users through a flow — whether each flow consists of one or several screens doesn't really matter, as long as it's achievable in a reasonable time. Once a subtask is completed, users come back to the task list view with that particular task marked as complete.

1. **Check before you start**

Check eligibility COMPLETED

Read declaration COMPLETED

2. **Prepare application**

Company information COMPLETED

Your contact details COMPLETED

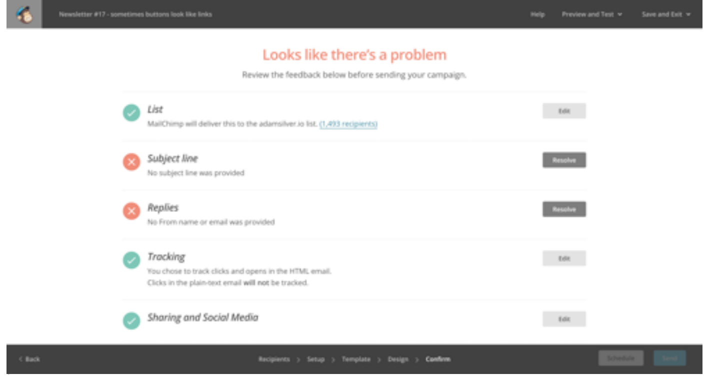
The GDS task list pattern with all tasks marked as completed.

This is not a pattern reserved for government services: MailChimp users have a similar need. The only difference is the visual design and the flatter hierarchy of tasks.

Instead of text, MailChimp uses iconography to mark tasks as complete. I discussed the pros and cons of iconography in chapter 3, "A Flight Booking Form" (see page 184).

4 http://smashed.by/govuktasklist

Additionally, instead of standard-looking links, they use call-to-action buttons labeled as “Resolve” to prompt the user and reduce the effort on their part.



MailChimp’s campaign task list page showing that two tasks still need to be completed

The exact design details you choose will come down to your product’s design language and user research, but it’s key to ensure that:

- each task status is clearly marked so users can see what’s left at a glance
- users can get a feel for how long is left until completion
- previous information is saved, so that users can return easily to it later

WHEN ALL TASKS ARE COMPLETE

When the last task in the list is completed, the user can still go back to the task list page with all the tasks marked as completed. This let's users bask in the glory of completing everything (which is another hit of dopamine). After all, the best part of completing a long to-do list is seeing all those ticks.

It also gives users a chance to review all their answers and make any amendments if necessary, which speaks to inclusive design principle 4, "Give control." The only other thing to remember is to give users an obvious way to proceed once everything is complete. Offer users a clear, single call to action.

ADDITIONAL CONSIDERATIONS

The points discussed above are probably applicable to any very long form you're designing, but you might also want to consider:

- Explaining what users need, such as documents, in the context of each individual task.
- Indicating how long each task will take. An estimate or a range can work well. If you can't offer this information, then you may need to break down the tasks further.

- Using verbs for task names. For example, “Agree to the terms,” “Create subject line,” “Choose template.”
- Listing tasks in order. If so, use an ordered list, the advantages of which are discussed in chapter 5, “An Inbox.”
- Marking who needs to complete the task. This is only useful if the tasks are performed by different people.
- Sizing all the tasks the same. Don’t take this too literally, but if one task is twenty questions and another is two, then take another look.

Summary

In this chapter we looked at two important patterns. The check before you start pattern makes a process transparent and saves users a lot of time. It can even be designed for a range of personal circumstances by asking users a series of questions about themselves.

We then looked at how the task list pattern breaks down a very long process into a series of smaller, more manageable tasks that users can return to in their own time.

These patterns can make a very long and very complex journey relatively easy to complete.

CHECKLIST

- Avoid really long and complex forms if you can.
- Tell users as much as they need to know up front so they can decide whether they are eligible for a service.
- In more complex situations, ask users a series of questions to determine the best course of action, and save their time and the operating costs of the service.
- Break up large tasks into smaller pieces, and allow users to save their progress so they can return later easily.

Thank you for reading my book!

One of the best things about this topic is that there are always going to be new challenges to solve and new patterns to define. If you can think of a new form problem you'd like me to look at, please send me a message on Twitter (*@adambsilver*). You never know — with enough new problems, a sequel could be on the cards.

Index

3rd-party integration 345
Accept attribute 346
Accessibility 20, 68
Accordions (checkout) 68
Adaptive design 241
Address field 79, 82, 105
Ajax 296, 302
Analytics (search) 265
Auto-Capitalization 203
Auto-Tabbing 206
Autocomplete (countries) 127
Autocorrect 203
Autofill (credit card) 97
Back button / links 119, 298
Before You Start Pattern 374
Billing/shipping address 105
Button Text (Call to Action) 48
Character countdown 90
Checkboxes 193, 226
Checklist affirmation pattern 62
Checkout Form 68
Choosing a Flight 186
Choosing A Seat 190
Collapsible Filters 303
Checkout Confirmation 109
Confirming versus undoing 259
Country input/autocomplete 127, 152
CVC input (credit cards) 99, 104
Date picker 155, 158
Delivery notes 88
Delivery Options 84
Denoting Selected Filters 311
Design Principles XV
Disabling/hiding buttons 63, 237
Drag-and-Drop 340
Dragover and dragleave 329
Dropdown menus 124
Email Address Field 33
Error messages 53, 57, 63
Expense Form 351
Feature Detection 164, 339
File upload 315
Filters 150, 280, 303, 311
Flight Booking Form 123
Float Labels 25
Focus styles 33, 253, 364
Forgotten password link 213
Form Feedback (errors) 51
Form Field Width 81
Form Labels 20, 31
Form Placeholders 22
Form Submission 50, 58, 240, 290
Form Validation 49, 59
Guest Checkout 74
Hiding labels 269
Highlighting marked emails 231
Hover versus click 246
Html5 validation 50
IE problems 46, 86
Inclusive Design Principles XV
Inline erros 56
Inline validation 61
Input mistakes 60, 104
Interactive filters 280
JavaScript 240
Keyboard interaction 135, 146, 174, 206, 253
Label position 31

Legend element 85
Limiting text 89
List Types 221
Live regions 92, 134
Loading states 296
Login Form 200
matchMedia API 250
Mobile Phone field input 77
Multiple File Picker 321
Multiple submit buttons 233
No password sign-in 28
Number input 77, 97
Offer choice 83
Onchange event 240
One Thing Per Page 212, 354
Optional fields 78
Optional or required fields 79
Order Summary (checkout) 117
Ordered and unordered lists 225
Password Field Design 204
Password reveal 39
Payment forms (credit cards) 94
Persistent Form Pattern 324, 351
Phone number input 77
Photo upload 344
Placeholder 22
Postcode field 81
Privacy 216
progress bar / Indicator 113
Progressive enhancement (code examples) 35, 52, 82, 130, 164, 183, 273, 305, 328, 340
Radio buttons 84, 87, 126, 193
Registration Form 18
Remove Buttons 360, 366
Responsive Design 241, 308
Responsive Menu 238
Returns policy 265
Review Page (checkout) 107
Screen reader enhancements 52, 92, 116, 134, 269
Search Form 263
Search input 127
Search Results 275
Second-Time Experience 112
Select All 254
Select box 124, 132, 239
Show password 40
Social Login 215
Stepper component 181
Sticky menus 235
Submit Button 208, 233, 268
Success Messages 256
Tab order 137
Tables 223
Task List Pattern 376
Toast messages 257
Toggles 40, 105, 271
Undo pattern 260
Upload Form 315
Username Label 201
Username or Password Doesn't Match 209
Users May Not Notice the Results Update 300
Validation 187
visually-hidden 116, 135, 229, 269
When to Fly 154
Where to Fly 123
ZIP code field 81

More From Smashing Magazine

- *Apps For All: Coding Accessible Web Applications*
  by Heydon Pickering
- *Art Direction For The Web (Oct. 2018)*
  by Andy Clarke
- *Design Systems*
  by Alla Kholmatova
- *Digital Adaptation*
  by Paul Boag
- *Inclusive Design Patterns*
  by Heydon Pickering
- *Smashing Book #6: New Frontiers in Web Design*
  Written by Laura Elizabeth, Marcy Sutton, Rachel Andrew, Mike Riethmueller, Harry Roberts, and others.
- *The Sketch Handbook*
  by Christian Krammer
- *User Experience Revolution*
  by Paul Boag

Visit smashingmagazine.com/printed-books/ for our full list of titles.