Identity

- index
- Help users change passwords easily by adding a well-known URL for changing passwords
- Sign in with a passkey through form autofill
- Designing the user experience of passkeys on Google accounts
- Create a passkey for passwordless logins
- Sign-in form best practices
- Sign-up form best practices
- SMS OTP form best practices

index

layout: collection override:tags: [] date: 2021-09-30 pathName: identity

Help users change passwords easily by adding a well-known URL for changing passwords

Set a redirect from `/.well-known/change-password` to the change password page of your website. This will enable password managers to navigate your users directly to that page.

Introduction

As you may know, passwords are not the best way to manage accounts. Luckily, there are emerging technologies such as WebAuthn and techniques such as one-time passwords that are helping us get closer to a world without passwords. However, these technologies are still being developed and things won't change rapidly. Many developers will still need to deal with passwords for at least the next few years. While we wait for the emerging technologies and techniques to become commonplace, we can at least make passwords easier to use.

A good way to do this is to provide better support for password managers.

How password managers help

Password managers can be built into browsers or provided as third-party apps. They can help users in various ways:

Autofill the password for the correct input field: Some browsers can find the correct input heuristically even if the website is not optimized for this purpose. Web developers can help password managers by correctly annotating HTML input tags.

Prevent phishing: Because password managers remember where the password was recorded, the password can be autofilled only at appropriate URLs, and not at phishing websites.

Generate strong and unique passwords: Because strong and unique passwords are directly generated and stored by the password manager, users don't have to remember a single character of the password.

Generating and autofilling passwords using a password manager have already served the web well, but considering their lifecycle, updating the passwords whenever it's required is as important as generating and autofilling. To properly leverage that, password managers are adding a new feature:

Detect vulnerable passwords and suggest updating them: Password managers can detect passwords that are reused, analyze the entropy and weakness of them, and even detect potentially leaked passwords or ones that are known to be unsafe from sources such as Have I Been Pwned.

A password manager can warn users about problematic passwords, but there's a lot of friction in asking users to navigate from the homepage to a change password page, on top of going through the actual process of changing the password (which varies from site to site). It would be much easier if password managers could navigate the user directly to the change-password URL. This is where a well-known URL for changing passwords becomes useful.

By reserving a well-known URL path that redirects the user to the change password page, the website can easily redirect users to the right place to change their passwords.

Set up "a well-known URL for changing passwords"

`.well-known/change-password` is proposed as a well-known URL for changing passwords. All you have to do is to configure your server to redirect requests for `.well-known/change-password` to the change password URL of your website.

For example, let's say your website is `https://example.com` and the change password URL is `https://example.com/settings/password`. You'll just need to set your server to redirect a request for `https://example.com/.well-known/change-password` to `https://example.com/settings/password`. That's it. For the redirection, use the HTTP status code `302 Found`, `303 See Other` or `307 Temporary Redirect`.

Alternatively you can serve HTML at your `.well-known/change-password` URL with a `<meta>` tag using an `http-equiv="refresh"`.

```
<meta http-equiv="refresh" content="0;url=https://example.com/settings/password">
```

Revisit your change password page HTML

The goal of this feature is to help the user's password lifecycle be more fluid. You can do two things to empower the user to update their password without friction:

- If your change-password form needs the current password, add `autocomplete="current-password"` to the `<input>` tag to help the password manager autofill it.
- For the new password field (in many cases it's two fields to ensure that the user has entered the new password correctly), add `autocomplete="new-password"` to the `<input>` tag to help the password manager suggest a generated password.

Learn more at Sign-in form best practices.

How it is used in real world

Examples

Thanks to Apple Safari's implementation, `/.well-known/change-password`, has already been available on some major websites for a while:

- Google
- GitHub
- Facebook
- Twitter
- WordPress

Try them yourself and do the same for yours!

Browser compatibility

A well-known URL for changing passwords has been supported in Safari since 2019. Chrome's password manager is starting to support it from version 86 onwards (which is scheduled for Stable release in late October 2020) and other Chromium-based browsers may follow. Firefox considers it worth implementing, but has not signalled that it plans to do so as of August 2020.

Chrome's password manager behavior

Let's have a look at how Chrome's password manager treats vulnerable passwords.

Chrome's password manager is able to check for leaked passwords. By navigating to `about://settings/passwords` users can run **Check passwords** against stored passwords, and see a list of passwords that are recommended for update.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/2qTEnaWJhxDcvX6hfgrY.png", alt="", width="1478", height="845" %}
>
> Chrome's **Check passwords** functionality

By clicking the **Change password** button next to a password that is recommended to be updated, the browser will:

- Open the website's change password page if `/.well-known/change-password` is set up correctly.
- Open the website's homepage if `/.well-known/change-password` is not set up and Google doesn't know the fallback.

{% Details %}

{% DetailsSummary %} What if the server returns `200 OK` even `/.well-known/change-password` doesn't exist? {% endDetailsSummary %}

Password managers try to determine if a website supports a well-known URL for changing passwords by sending a request to `/.well-known/change-password` before actually forwarding a user to this URL. If the request returns `404 Not Found` it is obvious that the URL is not available, but a `200 OK` response doesn't necessarily mean that the URL is available, because there are a few edge cases:

- A server-side-rendering website displays "Not found" when there is no content but with `200 OK`.
- A server-side-rendering website responds with `200 OK` when there is no content after redirecting to the "Not found" page.
- A single page app responds with the shell with `200 OK` and renders the "Not found" page on the client side when there is no content.

For these edge cases users will be forwarded to a "Not Found" page and that will be a source of confusion.

That's why there's a proposed standard mechanism to determine whether the server is configured to respond with `404 Not Found` when there is genuinely no content, by requesting a random page. Actually, the URL is also reserved: `/.well-known/resource-that-should-not-exist-whose-status-code-should-not-be-200`. Chrome for example uses this URL path to determine whether it can expect a proper change password URL from `/.well-known/change-password` in advance.

When you are deploying `/.well-known/change-password`, make sure that your server returns `404 Not Found` for any non-existing contents.

{% endDetails %}

Feedback

If you have any feedback on the specification, please file an issue to the spec repository.

Resources

- A Well-Known URL for Changing Passwords
- Detecting the reliability of HTTP status codes
- Sign-in form best practices

Photo by Matthew Brodeur on Unsplash

Sign in with a passkey through form autofill

Passkeys replace passwords and make user accounts on the web safer, simpler, easier to use. However, the transition from password-based to passkey-based authentication can complicate the user experience. Using form autofill to suggest passkeys can help create a unified experience.

Why use form autofill to sign-in with a passkey?

With a passkey, a user can sign in to a website just by using fingerprint, face, or device PIN.

Ideally, there would be no password users and the authentication flow could be as simple as a single sign-in button. When the user taps the button, an account selector dialog pops up, the user can pick an account, unlock the screen to verify and sign in.

However, the transition from password to passkey-based authentication can be challenging. As users switch to passkeys, there will still be those who use passwords and websites will need to accommodate both types of users. Users themselves should not be expected to remember on which sites they've switched to passkeys, so asking users to select which method to use up front would be poor UX.

Passkeys are also a new technology. Explaining them and making sure users are comfortable using them can be a challenge for websites. We can rely on familiar user experiences for autofilling passwords to solve both problems.

Conditional UI

To build an efficient user experience for both passkey and password users, you can include passkeys in autofill suggestions. This is called conditional UI and it's a part of the WebAuthn standard.

{% Video src="video/YLflGBAPWecgtKJLqCJHSzHqe2J2/ZaxgMtkOh5CX2B1ZT9gd.mp4", autoplay="true", loop="true" %}

As soon as the user taps on the username input field, an autofill suggestion dialog pops up which highlights the stored passkeys along with password autofill suggestions. The user can then choose an account and use the device screen lock to sign in.

This way, users can sign in to your website with the existing form as if nothing has changed, but with the added security benefit of passkeys if they have one.

{% Aside 'success' %}

When a user signs in with a password, consider using that opportunity to offer them to create a passkey.

{% endAside %}

How it works

To authenticate with a passkey, you use the WebAuthn API.

The four components in a passkey authentication flow are: the user:

- **Backend**: Your backend server that holds the accounts database storing the public key and other metadata about the passkey.
- **Frontend**: Your frontend which communicates with the browser and sends fetch requests to the backend.
- **Browser**: The user's browser which is running your Javascript.
- **Authenticator**: The user's authenticator which creates and stores the passkey. This may be on the same device as the browser (e.g. when using Windows Hello) or on another device, like a phone.

{% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/vAwnMiLi7U1spDopYg2Q.png", alt="Passkey authentication diagram", width="800", height="254" %}

1. As soon as a user lands on the frontend, it requests a challenge from the backend to authenticate with a passkey and calls `navigator.credentials.get()` to initiate authenticating with a passkey. This returns a `Promise`.
2. When the user puts the cursor in the sign-in field, the browser displays a password autofill dialog including passkeys. An authentication dialog appears if the user selects a passkey.
3. After the user verifies their identity using the device's screen lock, the promise is resolved and a public key credential is returned to the frontend.

4. The frontend sends the public key credential to the backend. The backend verifies the signature against the matched account's public key in the database. If it succeeds, the user is signed in.

Prerequisites

Conditional WebAuthn UI is publicly supported in Safari on iOS 16, iPadOS 16 and macOS Ventura. It's also available on Chrome on Android, macOS and Windows 11 22H2.

Authenticate with a passkey through form autofill

When a user wants to sign in, you can make a conditional WebAuthn `get` call to indicate that passkeys may be included in autofill suggestions. A conditional call to WebAuthn's `navigator.credentials.get()` API does not show UI and remains pending until the user picks an account to sign-in with from the autofill suggestions. If the user picks a passkey the browser will resolve the promise with a credential rather than filling in the sign-in form. It's then the page's responsibility to sign the user in.

Annotate form input field

Add an `autocomplete` attribute to the username `input` field, if needed. Append `username` and `webauthn` as its tokens to let it suggest passkeys.

```
<input type="text" name="username" autocomplete="username webauthn" ...>
```

Feature detection

Before invoking a conditional WebAuthn API call, check if:

- The browser supports WebAuthn.
- The browser supports WebAuthn conditional UI.

```
// Availability of `window.PublicKeyCredential` means WebAuthn is usable.
if (window.PublicKeyCredential &&
```

```
    PublicKeyCredential.□□isConditionalMediationAvailable) {
  // Check if conditional mediation is available.
  const isCMA = await PublicKeyCredential.□□isConditionalMediationAvailable();
  if (isCMA) {
    // Call WebAuthn authentication
  }
}
```

Fetch a challenge from the RP server

Fetch a challenge from the RP server that is required to call `navigator.credentials.get()`:

- `challenge`: A server-generated challenge in an ArrayBuffer. This is required to prevent replay attacks. Make sure to generate a new challenge on every sign-in attempt and disregard it after a certain duration or after a sign-in attempt fails to validate. Consider it like a CSRF token.
- `allowCredentials`: An array of acceptable credentials for this authentication. Pass an empty array to let the user select an available passkey from a list shown by the browser.
- `userVerification`: Indicates whether user verification using the device screen lock is `"required"`, `"preferred"` or `"discouraged"`. The default is `"preferred"`, which means the authenticator may skip user verification. Set this to `"preferred"` or omit the property.

{% Aside 'caution' %}

For requests with `userVerification` set to `"preferred"`, authenticators may skip the user verification check, for example if the device doesn't have any biometric sensors, the user hasn't set it up (no enrolled fingerprints), or if the sensor is temporarily unavailable (a laptop running with a closed display lid). The UV bit in the authenticator data of the response always indicates whether user verification was performed.

If you'd like to always require a user verification, set `userVerification` to `"required"`. Don't forget to check that the UV flag is `true` on the server as well.

{% endAside %}

{% Aside %}

ArrayBuffer values transferred from the server such as `challenge`, `user.id` and credential `id` for `excludeCredentials` need to be encoded on transmission. Make sure to decode them on the RP frontend before passing the values to the WebAuthn API call. We recommend using Base64URL encode.

{% endAside %}

Call WebAuthn API with the `conditional` flag to authenticate the user

Call `navigator.credentials.get()` to start waiting for the user authentication.

```
// To abort a WebAuthn call, instantiate an `AbortController`.
const abortController = new AbortController();

const publicKeyCredentialRequestOptions = {
  // Server generated challenge
  challenge: ****,
  // The same RP ID as used during registration
  rpId: 'example.com',
};

const credential = await navigator.credentials.get({
  publicKey: publicKeyCredentialRequestOptions,
  signal: abortController.signal,
  // Specify 'conditional' to activate conditional UI
  mediation: 'conditional'
});
```

- `rpId`: An RP ID is a domain and a website can specify either its domain or a registrable suffix. This value must match the rp.id used when the passkey was created.

Remember to specify `mediation: 'conditional'` to make the request conditional.

{% Aside %}

The WebAuthn call here will be pending and the promise won't be resolved until the user interacts with the autofill input element and completes the authentication flow. The RP frontend can use `AbortController` to abort the WebAuthn call.

{% endAside %}

Send the returned public key credential to the RP server

After the user selects an account and consents using the device's screen lock, the promise is resolved returning a `PublicKeyCredential` object to the RP frontend.

A promise can be rejected due to several different reasons. You need to handle the errors accordingly, depending on the `Error` object's `name` property:

- `NotAllowedError`: The user has canceled the operation.
- **Other exceptions**: Something unexpected happened. The browser shows an error dialog to the user.

The public key credential object contains the following properties:

- `id`: The base64url encoded ID of the authenticated passkey credential.
- `rawId`: An ArrayBuffer version of the credential ID.
- `response.clientDataJSON`: An ArrayBuffer of client data. This field contains information such as the challenge and the origin that the RP server will need to verify.
- `response.authenticatorData`: An ArrayBuffer of authenticator data. This field contains information such as the RP ID.
- `response.signature`: An ArrayBuffer of the signature. This value is the core of the credential and needs to be verified on the server.
- `response.userHandle`: An ArrayBuffer that contained the user ID that was set at creation time. This value can be used, instead of the credential ID, if the server needs to pick the ID values that it uses, or if the backend wishes to avoid creating an index on credential IDs.
- `authenticatorAttachment`: Returns `platform` when this credential came from the local device. Otherwise `cross-platform`, notably when the user used a phone to sign in. If the user needed to use a phone to sign-in, consider prompting them to create a passkey on the local device.
- `type`: This field is always set to `"public-key"`.

If you use a library to handle the public-key credential object on the RP server, we recommend that you send the entire object to the server after encoding it partially with base64url.

Verify the signature

When you receive the public key credential on the server, pass it to the FIDO library to process the object.

Look up the matching credential ID with the `id` property (If you need to determine the user account, use the `userHandle` property which is the `user.id` you specified when creating the credential). See if the credential's `signature` can be verified with the stored public key. To do so, we recommend using a server side library or a solution instead of writing your own code. You can find open source libraries in the awesome-webauth GitHub repo.

Once the credential is verified with a matching public key, sign the user in.

Resources

- Create a passkey for passwordless logins
- Passkeys
- Apple document: Authenticating a User Through a Web Service
- Google document: Passwordless login with passkeys

Designing the user experience of passkeys on Google accounts

Passkeys are a simple and secure cross-device authentication technology that enables creating online accounts and signing in to them without entering a password. To log in to an account, users are simply shown a prompt to to use the screen lock on their device, such as touching the fingerprint sensor.

Google has been working with the FIDO Alliance for years, alongside Apple and Microsoft, to bring passkeys to the world. In 2022 we rolled out platform support for passkeys so that Android and Chrome users can seamlessly sign in to apps and websites across all their devices. In May 2023, we enabled signing in to Google Accounts with passkeys, bringing the security and convenience of passkeys to our users.

Google is in a unique position, as we are both working on the infrastructure for passkeys and are one of the largest services using them. We are rolling out passkeys for Google Accounts carefully and deliberately, so we can measure the results and use that feedback to continue to improve the passkey infrastructure and the Google account experience.

Transitioning users to passkeys

Passwords have been the standard sign-in method since the advent of personalized online experiences. How do we introduce the passwordless experience of passkeys?

Research indicates that when it comes to authentication, users value the convenience the most. They want a smooth and fast transition to the real experience, which only comes after signing in.

Still, the transition to passkeys requires changing muscle memory and users need to be convinced it's worth making a switch.

The user experience of passkeys for Google.com has been strategically designed to emphasize two principles at every step of the authentication process: ease of use and security.

Leading with convenience

{% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/UI5EioUkpDiMtUAdARnw.png", alt="For most users, this will be the first time they see passkeys", width="800", height="600" %}
For most users, this will be the first time they see passkeys.

The first passkey screen users see is light and easy-to-digest. The header is focusing on the user benefit, saying "Simplify your sign in."

The body copy explains "With passkeys you can now use your fingerprint, face or screen lock to verify it's really you".

The illustration is intended to ground the message in the value proposition made by the page. The large blue primary action invites the user to proceed. "Not now" is included as a secondary action to allow users to choose whether or not to opt in at this time, leaving the user in control. And "Learn more" is offered for the most curious users who would like to understand passkeys better before proceeding.

We explored many iterations of the pages used to introduce users to passkeys during sign in. This included trying content that emphasized the security, technology, and other aspects of passkeys—yet convenience was really what resonated most. Google's content strategy, illustration, and interaction design demonstrates this core principle for our implementation of passkeys.

Associating the term "passkeys" with familiar security experiences

Passkeys are a new term for most users so we are intentionally gently exposing the users to the term to build familiarity. Guided by internal research, we are strategically associating passkeys with security.

The word "passkey" is included throughout the sign-in flow in the less-prominent body copy position. It's consistently nestled amongst the familiar security experiences that enable passkey use: fingerprint, face scan, or other device screen lock.

Our research has shown that many users associate biometrics with security. While passkeys don't require biometrics (a passkey can be used with a device PIN, for example), we are leaning into the association of passkeys with biometrics to boost user perception of passkeys' security benefits.

The additional content behind the "Learn more" has lots of valuable information for users, including reassurance for users that their sensitive, biometric data stays on their personal device and is never stored or shared when creating or using passkeys. We took this approach because most users found the convenience aspect of passkeys appealing, but only a few took into account the

biometric element during testing.

Introducing passkeys when it's relevant to the user

Google's heuristics carefully determine who will see the introductory screen. Some of the factors are whether a user has two-step verification enabled and whether they access that account regularly from the same device.

Users who are most likely to succeed with passkeys are selected first, and over time more users will be introduced (though, anyone can get started at g.co/passkeys today).

Select users are prompted to create a passkey after signing in with a username and password. There are a few reasons we chose this point in the user journey:

- The user has just signed in, they're aware of their credentials and second step.
- We are confident that the user is on their device–they just signed in, so it's unlikely they walked away or put their device down.
- Statistically, signing in isn't always successful the first time–so a message around making it easier next time has tangible value.

Positioning passkeys as an alternative to passwords and not yet a replacement

Initial user research shows that many users still want passwords as a backup sign-in method. And not all users will have the technology necessary to adopt passkeys.

So while the industry, Google included, is moving towards a "passwordless future", Google is intentionally positioning passkeys as a simple and secure **alternative** to passwords. Google's UI focuses on the benefits of passkeys and avoids language that implies getting rid of passwords.

The creation moment

When users choose to enroll, they'll see a browser-specific UI modal that enables them to create a passkey.

The passkey itself is shown with the industry-aligned icon and the information used to create it. This includes the display name (a friendly name for your passkey, like your user's real name) and

the username (a unique name on your service–an email address can work great here). When it comes to working with the passkeys icon, the FIDO alliance recommends using the proven passkeys icon–and encourages making it your own with customizations.

Passkeys icon is shown consistently across the user journey to create a familiarity with what the user will see when using or managing the passkey. The passkey icon is never presented without context or supporting material.

> {% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/kPbCZqaBDx5AFTZyi54a.png", alt="When users create their passkey, they'll see this page", width="800", height="600" %}
> When users create their passkey, they'll see this page.

Above, we outlined how the user and the platform work together to create a passkey. When the user clicks "Continue" they'll be presented with a unique UI depending on the platform.

With that in mind, we learned through internal research that a confirmation screen once the passkey is created can be very helpful in terms of comprehension and closure at this step of the process.

> {% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/sNqODRG8qqaGQc8ygbjO.png", alt="Once the passkey has been created, users will see this page", width="800", height="600" %}
> Once the passkey has been created, users will see this page.

The confirmation screen is a deliberate ‘pause' to bookend the journey of introducing a user to passkeys and going through the process of creating one of their own. As it is (likely) the first time a user has engaged with passkeys, this page aims to provide clear closure to the journey. We chose a standalone page after trying some other tools like smaller notifications, and even a post-creation email–simply to provide a structured, stable end to end experience.

Once the user clicks "Continue" here, they're brought to their destination.

> {% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/u7ugXsWYbGcylhCWZ21M.png", alt="When users sign in again, they'll likely see this page", width="800", height="600" %}
> When users sign in again, they'll likely see this page.

Signing in

Next time a user tries to sign in, they'll be greeted with this page. This uses the same layout, illustration, and primary call to action to evoke the first ‘creation' experience outlined above. Once the user has made a choice to enroll in passkeys, this page should feel familiar and they will recognize what steps they need to take to sign in.

{% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/kPbCZqaBDx5AFTZyi54a.png", alt="The user will use this WebAuthn UI to sign in", width="800", height="600" %}
The user will use this WebAuthn UI to sign in.

The same principle of familiarity applies here. Intentionally, this uses the same iconography, illustration, layout and text. The text within the WebAuthn UI is kept brief, broad, and re-usable–so everyone can use this both for authentication and reauthentication.

Passkeys management

Introducing a whole new page within the Google Account settings pages required careful consideration to ensure a cohesive, intuitive, and consistent user experience.

To achieve this, we analyzed the patterns regarding navigation, content, hierarchy, structure, and established expectations that existed across the Google Account.

{% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/C0xuS6j8vd98Y17jckE3.png", alt="Passkeys management page in the Google Account", width="800", height="655" %}
Passkeys management page in the Google Account.

Describe passkeys by ecosystem

To create a high level category system that would be logical to understand we settled on describing passkeys by ecosystem. This way, a user could recognize where a passkey was created and where it is used. Each identity provider (Google, Apple, and Microsoft) has a name for their ecosystem, so we chose to use those (Google Password Manager, iCloud keychain, and Windows Hello respectively).

To support this, we added additional metadata, such as when it was created, when it was last used, and the specific OS that it was used on. In terms of user management actions, the API only supports renaming, revoking, and creating.

Renaming allows users to assign personally meaningful names to passkeys, which could help particular cohorts of users keep track and understand them more easily.

Revoking a passkey doesn't delete it from the user's personal credential manager (like Google Password Manager), but renders it unusable until it is set up again. That's why we chose a cross, instead of a trash or delete icon, to represent the action of revoking a passkey.

When describing the action of adding a passkey to their account, the phrase "Create passkey" resonated better with users compared to "Add a passkey." This is a subtle language choice to distinguish passkeys from tangible, hardware security keys (though it should be noted that

passkeys can be stored on some hardware security keys).

Providing additional content

Internal research showed that using passkeys is a relatively seamless and familiar experience. However as with any new technology, there are lingering questions and concerns that will come up for some users.

How the technology works behind the screen lock, what makes it more secure, and the most common "what if" scenarios Google came across in testing are addressed in Google's passkey Help Center content. Having support content ready with launch of passkeys is critical for an easy transition for users on any site.

Falling back from passkeys

Reverting to the old system is as simple as clicking "try another way" when a user is asked to authenticate with a passkey. Additionally, exiting the WebAuthn UI will start users on a path to try their passkey again, or sign into their Google Account in traditional ways.

Conclusion

We are still in the early days of passkeys, so when designing the user experience keep a few principles in mind:

- Introduce passkeys when it's relevant to the user.
- Highlight the benefits of passkeys.
- Use opportunities to build familiarity the concept of passkeys.
- Position passkeys as an alternative to passwords and not a replacement.

The choices we made for passkeys for Google Accounts were informed by best practices and internal research and we'll continue to evolve the user experience as we gain new insights from users in the real world.

Create a passkey for passwordless logins

Using passkeys instead of passwords is a great way for websites to make their user accounts safer, simpler, easier to use and passwordless. With a passkey, a user can sign in to a website or an app just by using their fingerprint, face or device PIN.

{% Aside %}

To learn basic concepts of passkeys, check out Passwordless login with passkeys first.

{% endAside %}

A passkey has to be created, associated with a user account and have its public key be stored on your server before a user can sign in with it.

How it works

A user can be asked to create a passkey in one of the following situations:

- When a user signs in using a password, or a passkey from another device (that is, the `authenticatorAttachment` is `cross-platform`).
- On a dedicated page where users can manage their passkeys.

  {% Video src="video/YLflGBAPWecgtKJLqCJHSzHqe2J2/2gY9wppN8tivRRWDH8gV.mp4", autoplay="true", loop="true" %}

To create a passkey, you use the WebAuthn API.

The four components of the passkey registration flow are:

- **Backend**: Your backend server that holds the accounts database storing the public key and other metadata about the passkey.
- **Frontend**: Your frontend which communicates with the browser and sends fetch requests to the backend.
- **Browser**: The user's browser which is running your Javascript.
- **Authenticator**: The user's authenticator which creates and stores the passkey. This may be on the same device as the browser (for example, when using Windows Hello) or on

another device, like a phone.

{% Img src="image/YLflGBAPWecgtKJLqCJHSzHqe2J2/vmxdOrHbR2kW17Ipv6hk.png", alt="Passkey registration diagram", width="800", height="258" %}

The journey to add a new passkey to an existing user account is as follows:

1. A user signs in to the website.
2. Once the user is signed in, they request to create a passkey on the frontend, for example, by pressing a "Create a passkey" button.
3. The frontend requests information from the backend to create a passkey, such as user information, a challenge, and the credential IDs to exclude.
4. The frontend calls `navigator.credentials.create()` to create a passkey. This call returns a promise.
5. A passkey is created after the user consents using the device's screen lock. The promise is resolved and a public key credential is returned to the frontend.
6. The frontend sends the public key credential to the backend and stores the credential ID and the public key associated with the user account for future authentications.

Compatibilities

WebAuthn is supported by most browsers, but there are small gaps. Refer to Device Support - passkeys.dev to learn what combination of browsers and an operating systems support creating a passkey.

Create a new passkey

Here's how a frontend should operate upon a request to create a new passkey.

Feature detection

Before displaying a "Create a new passkey" button, check if:

- The browser supports WebAuthn.
- The device supports a platform authenticator (can create a passkey and authenticate with the passkey).
- The browser supports WebAuthn conditional UI.

```
// Availability of `window.PublicKeyCredential` means WebAuthn is usable.
// `isUserVerifyingPlatformAuthenticatorAvailable` means the feature detection is usable.
// `□□isConditionalMediationAvailable` means the feature detection is usable.
if (window.PublicKeyCredential &&
    PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable &&
    PublicKeyCredential.□□isConditionalMediationAvailable) {
  // Check if user verifying platform authenticator is available.
  Promise.all([
    PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable(),
    PublicKeyCredential.□□isConditionalMediationAvailable(),
  ]).then(results => {
    if (results.every(r => r === true)) {
      // Display "Create a new passkey" button
    }
  });
}
```

Until all the conditions have been met, passkeys will not be supported on this browser. The "Create a new passkey" button shouldn't be displayed until then.

Fetch important information from the backend

When the user clicks the button, fetch important information to call `navigator.credentials.create()` from the backend:

- **`challenge`**: A server-generated challenge in ArrayBuffer for this registration. This is required but unused during registration unless doing attestation—an advanced topic not covered here.
- **`user.id`**: A user's unique ID. This value must be an ArrayBuffer which does not include personally identifying information, for example, e-mail addresses or usernames. A random, 16-byte value generated per account will work well.
- **`user.name`**: This field should hold a unique identifier for the account that the user will recognise, like their email address or username. This will be displayed in the account selector. (If using a username, use the same value as in password authentication.)
- **`user.displayName`**: This field is an optional, more user-friendly name for the account. It need not be unique and could be the user's chosen name. If your site does not have a suitable value to include here, pass an empty string. This may be displayed on the account selector depending on the browser.

- `excludeCredentials`: Prevents registering the same device by providing a list of already registered credential IDs. The `transports` member, if provided, should contain the result of calling `getTransports()` during the registration of each credential.

{% Aside %}

ArrayBuffer values transferred from the server such as `challenge`, `user.id` and credential `id` for `excludeCredentials` need to be encoded on transmission. Don't forget to decode them on the frontend before passing to the WebAuthn API call. We recommend using Base64URL encode.

{% endAside %}

Call WebAuthn API to create a passkey

Call `navigator.credentials.create()` to create a new passkey. The API returns a promise, waiting for the user's interaction displaying a modal dialog.

```
const publicKeyCredentialCreationOptions = {
  challenge: *****,
  rp: {
    name: "Example",
    id: "example.com",
  },
  user: {
    id: *****,
    name: "john78",
    displayName: "John",
  },
  pubKeyCredParams: [{alg: -7, type: "public-key"},{alg: -257, type: "public-key"}],
  excludeCredentials: [{
    id: *****,
    type: 'public-key',
    transports: ['internal'],
  }],
  authenticatorSelection: {
    authenticatorAttachment: "platform",
    requireResidentKey: true,
  }
};

const credential = await navigator.credentials.create({
```

```
  publicKey: publicKeyCredentialCreationOptions
});


// Encode and send the credential to the server for verification.
```

The parameters not explained above are:

- `rp.id`: An RP ID is a domain and a website can specify either its domain or a registrable suffix. For example, if an RP's origin is `https://login.example.com:1337`, the RP ID can be either `login.example.com` or `example.com`. If the RP ID is specified as `example.com`, the user can authenticate on `login.example.com` or on any subdomains on `example.com`.
- `rp.name`: The RP's name.
- `pubKeyCredParams`: This field specifies the RP's supported public-key algorithms. We recommend setting it to `[{alg: -7, type: "public-key"},{alg: -257, type: "public-key"}]`. This specifies support for ECDSA with P-256 and RSA PKCS#1 and supporting these gives complete coverage.
- `authenticatorSelection.authenticatorAttachment`: Set it to `"platform"`. This indicates that we want an authenticator that is embedded into the platform device, and the user will not be prompted to insert e.g. a USB security key.
- `authenticatorSelection.requireResidentKey`: Set it to a boolean "true". A discoverable credential (resident key) stores user information to the passkey and lets users select the account upon authentication.
- `authenticatorSelection.userVerification`: Indicates whether a user verification using the device screen lock is `"required"`, `"preferred"` or `"discouraged"`. The default is `"preferred"`, which means the authenticator may skip user verification. Set this to `"preferred"` or omit the property.

{% Aside 'caution' %}

For requests with `userVerification` set to `"preferred"`, authenticators may skip the user verification check, for example if the device doesn't have any biometric sensors, the user hasn't set it up (no enrolled fingerprints), or if the sensor is temporarily unavailable (laptop running with a closed display lid). The UV bit in the authenticator data of the response always indicates whether user verification was performed.

If you'd like to always require a user verification, set `userVerification` to `"required"`. Don't forget to check that the UV flag is `true` on the server as well.

{% endAside %}

{% Aside %}

Note: Some browsers don't require any parameters on `authenticatorSelection` to create a passkey, but others may. We recommend specifying those parameters explicitly.

{% endAside %}

Send the returned public key credential to the backend

After the user consents using the device's screen lock, a passkey is created and the promise is resolved returning a PublicKeyCredential object to the frontend.

The promise can be rejected for different reasons. You can handle these errors by checking the `Error` object's `name` property:

- **`InvalidStateError`**: A passkey already exists on the device. No error dialog will be shown to the user and the site should not treat this as an error—the user wanted the local device registered and it is.
- **`NotAllowedError`**: The user has canceled the operation.
- **Other exceptions**: Something unexpected happened. The browser shows an error dialog to the user.

The public key credential object contains the following properties:

- **`id`**: A Base64URL encoded ID of the created passkey. This ID helps the browser to determine whether a matching passkey is in the device upon authentication. This value needs to be stored in the database on the backend.
- **`rawId`**: An ArrayBuffer version of the credential ID.
- **`response.clientDataJSON`**: An ArrayBuffer encoded client data.
- **`response.attestationObject`**: An ArrayBuffer encoded attestation object. This contains important information such as an RP ID, flags and a public key.
- **`authenticatorAttachment`**: Returns `"platform"` when this credential is created on a passkey capable device.
- **`type`**: This field is always set to `"public-key"`.

If you use a library to handle the public key credential object on the backend, we recommend sending the entire object to the backend after encoding it partially with base64url.

{% Aside %}

Optionally, the `response` member of the credential - an `AuthenticatorAttestationResponse` — provides `getTransports()`, `getPublicKey()`, `getAuthenticatorData()`, and `getPublicKeyAlgorithm()` functions to retrieve elements of the credential.

{% endAside %}

Save the credential

Upon receiving the public key credential on the backend, pass it to the FIDO library to process the object.

{% Aside %}

We recommend using a server-side library or a solution instead of writing your own code to process a public-key credential. Check out open-source libraries to process public-key credentials.

{% endAside %}

You can then store the information retrieved from the credential to the database for future use. The following list includes some typical properties to save:

- **Credential ID (Primary key)**
- **User ID**
- **Public key**

The public-key credential also includes the following information that you may want to save in the database:

- **Backup Eligibility flag**: `true` if the device is eligible for passkey synchronization.
- **Backup State flag**: `true` if the created passkey is actually set to be synchronized.
- **Transports**: A list of transports the device supports: `"internal"` means the device supports a passkey, `"hybrid"` means it also supports authentication on another device.

To authenticate the user, read Sign in with a passkey through form autofill.

Resources

- Sign in with a passkey through form autofill
- Passkeys
- Apple document: Authenticating a User Through a Web Service
- Google document: Passwordless login with passkeys

Sign-in form best practices

{% YouTube 'alGcULGtiv8' %}

If users ever need to log in to your site, then good sign-in form design is critical. This is especially true for people on poor connections, on mobile, in a hurry, or under stress. Poorly designed sign-in forms get high bounce rates. Each bounce could mean a lost and disgruntled user—not just a missed sign-in opportunity.

{% Aside 'codelab' %} If you would prefer to learn these best practices with a hands-on tutorial, check out the Sign-in form best practices codelab. {% endAside %}

Here is an example of a simple sign-in form that demonstrates all of the best practices:

{% Glitch { id: 'sign-in-form', path: 'index.html', height: 480 } %}

Checklist

- Use meaningful HTML elements: `<form>`, `<input>`, `<label>`, and `<button>`.
- Label each input with a `<label>`.
- Use element attributes to access built-in browser features: `type`, `name`, `autocomplete`, `required`.
- Give input `name` and `id` attributes stable values that don't change between page loads or website deployments.
- Put sign-in in its own <form> element.
- Ensure successful form submission.
- Use `autocomplete="new-password"` and `id="new-password"` for the password input in a sign-up form, and for the new password in a reset-password form.
- Use `autocomplete="current-password"` and `id="current-password"` for a sign-in password input.
- Provide Show password functionality.
- Use `aria-label` and `aria-describedby` for password inputs.
- Don't double-up inputs.
- Design forms so the mobile keyboard doesn't obscure inputs or buttons.
- Ensure forms are usable on mobile: use legible text, and make sure inputs and buttons are large enough to work as touch targets.

- Maintain branding and style on your sign-up and sign-in pages.
- Test in the field as well as the lab: build page analytics, interaction analytics, and user-centric performance measurement into your sign-up and sign-in flow.
- Test across browsers and devices: form behaviour varies significantly across platforms.

{% Aside %} This article is about frontend best practices. It does not explain how to build backend services to authenticate users, store their credentials, or manage their accounts. 12 best practices for user account, authorization and password management outlines core principles for running your own backend. If you have users in different parts of the world, you need to consider localizing your site's use of third-party identity services as well as its content.

There are also two relatively new APIs not covered in this article which can help you build a better sign-in experience:

- **WebOTP**: to deliver one-time passcodes or PIN numbers via SMS to mobile phones. This can allow users to select a phone number as an identifier (no need to enter an email address!) and also enables two-step verification for sign-in and one-time codes for payment confirmation.
- **Credential Management**: to enable developers to store and retrieve password credentials and federated credentials programmatically. {% endAside %}

Use meaningful HTML {: #meaningful-html }

Use elements built for the job: `<form>`, `<label>` and `<button>`. These enable built-in browser functionality, improve accessibility, and add meaning to your markup.

Use `<form>` {: #form }

You might be tempted to wrap inputs in a `<div>` and handle input data submission purely with JavaScript. It's generally better to use a plain old `<form>` element. This makes your site accessible to screenreaders and other assistive devices, enables a range of built-in browser features, makes it simpler to build basic functional sign-in for older browsers, and can still work even if JavaScript fails.

{% Aside 'gotchas' %} A common mistake is to wrap a whole web page in a single form, but this is liable to cause problems for browser password managers and autofill. Use a different <form> for each UI component that needs a form. For example, if you have sign-in and search on the same

page, you should use two form elements. {% endAside %}

Use `<label>` {: #label }

To label an input, use a `<label>`!

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

Two reasons:

- A tap or click on a label moves focus to its input. Associate a label with an input by using the label's `for` attribute with the input's `name` or `id`.
- Screenreaders announce label text when the label or the label's input gets focus.

Don't use placeholders as input labels. People are liable to forget what the input was for once they've started entering text, especially if they get distracted ("Was I entering an email address, a phone number, or an account ID?"). There are lots of other potential problems with placeholders: see Don't Use The Placeholder Attribute and Placeholders in Form Fields Are Harmful if you're unconvinced.

It's probably best to put your labels above your inputs. This enables consistent design across mobile and desktop and, according to Google AI research, enables quicker scanning by users. You get full width labels and inputs, and you don't need to adjust label and input width to fit the label text.

> {% Img src="image/admin/k0ioJa9CqnMI8vyAvQPS.png", alt="Screenshot showing form input label position on mobile: next to input and above input.", width="500", height="253" %}
> Label and input width is limited when both are on the same line.

Open the label-position Glitch on a mobile device to see for yourself.

Use `<button>` {: #button }

Use `<button>` for buttons! Button elements provide accessible behaviour and built-in form submission functionality, and they can easily be styled. There's no point in using a `<div>` or some other element pretending to be a button.

Ensure that the submit button says what it does. Examples include **Create account** or **Sign in**, not **Submit** or **Start**.

Ensure successful form submission {: #submission }

Help password managers understand that a form has been submitted. There are two ways to do that:

- Navigate to a different page.
- Emulate navigation with `History.pushState()` or `History.replaceState()`, and remove the password form.

With an `XMLHttpRequest` or `fetch` request, make sure that sign-in success is reported in the response and handled by taking the form out of the DOM as well as indicating success to the user.

Consider disabling the **Sign in** button once the user has tapped or clicked it. Many users click buttons multiple times even on sites that are fast and responsive. That slows down interactions and adds to server load.

Conversely, don't disable form submission awaiting user input. For example, don't disable the **Sign in** button if users haven't entered their customer PIN. Users may miss out something in the form, then try repeatedly tapping the (disabled) **Sign in** button and think it's not working. At the very least, if you must disable form submission, explain to the user what's missing when they click on the disabled button.

{% Aside 'caution' %} The default type for a button in a form is `submit`. If you want to add another button in a form (for **Show password**, for example) add `type="button"`. Otherwise clicking or tapping on the button will submit the form.

Pressing the `Enter` key while any form field has focus simulates a click on the first `submit` button in the form. If you include a button in your form before the **Submit** button, and don't specify the type, that button will have the default type for buttons in a form (`submit`) and receive the click event before the form is submitted. For an example of this, see our demo: fill in the form, then press `Enter`. {% endAside %}

Don't double up inputs {: #no-double-inputs }

Some sites force users to enter emails or passwords twice. That might reduce errors for a few users, but causes extra work for *all* users, and increases abandonment rates. Asking twice also makes no sense where browsers autofill email addresses or suggest strong passwords. It's better to

enable users to confirm their email address (you'll need to do that anyway) and make it easy for them to reset their password if necessary.

Make the most of element attributes {: #element-attributes }

This is where the magic really happens! Browsers have multiple helpful built-in features that use input element attributes.

Keep passwords private—but enable users to see them if they want {: #show-password }

Passwords inputs should have `type="password"` to hide password text and help the browser understand that the input is for passwords. (Note that browsers use a variety of techniques to understand input roles and decide whether or not to offer to save passwords.)

You should add a **Show password** icon or button to enable users to check the text they've entered—and don't forget to add a **Forgot password** link. See Enable password display.

> {% Img src="image/admin/58suVe0HnSLaJvNjKY53.png", alt="Google sign-in form showing Show password icon.", width="300", height="107" %}
> Password input from the Google sign-in form: with **Show password** icon and **Forgot password** link.

Give mobile users the right keyboard {: #mobile-keyboards }

Use `<input type="email">` to give mobile users an appropriate keyboard and enable basic built-in email address validation by the browser… no JavaScript required!

If you need to use a telephone number instead of an email address, `<input type="tel">` enables a telephone keypad on mobile. You can also use the `inputmode` attribute where necessary: `inputmode="numeric"` is ideal for PIN numbers. Everything You Ever Wanted to Know About inputmode has more detail.

{% Aside 'caution' %} `type="number"` adds an up/down arrow to increment numbers, so don't use it for numbers that aren't meant to be incremented, such as IDs and account numbers. {% endAside %}

Prevent mobile keyboard from obstructing the **Sign in** button {: #keyboard-obstruction }

Unfortunately, if you're not careful, mobile keyboards may cover your form or, worse, partially obstruct the **Sign in** button. Users may give up before realizing what has happened.

> {% Img src="image/admin/rLo5sW9LBpTcJU7KNnb7.png", alt="Two screenshots of a sign-in form on an Android phone: one showing how the Submit button is obscured by the phone keyboard.", width="400", height="360" %}
> The **Sign in** button: now you see it, now you don't.

Where possible, avoid this by displaying only the email/phone and password inputs and **Sign in** button at the top of your sign-in page. Put other content below.

> {% Img src="image/admin/0OebKiAP4sTgaXbcbvYx.png", alt="Screenshot of a sign-in form on an Android phone: the Sign in button is not obscured by the phone keyboard.", width="200", height="342" %}
> The keyboard doesn't obstruct the **Sign in** button.

Test on a range of devices {: #devices }

You'll need to test on a range of devices for your target audience, and adjust accordingly.

BrowserStack enables free testing for open source projects on a range of real devices and browsers.

> {% Img src="image/admin/jToMlWgjS3J2WKmjs1hx.png", alt="Screenshots of a sign-in form on iPhone 7, 8 and 11. On iPhone 7 and 8 the Sign in button is obscured by the phone keyboard, but not on iPhone 11", width="800", height="522" %}
> The **Sign in** button: obscured on iPhone 7 and 8, but not on iPhone 11.

Consider using two pages {: #two-pages }

Some sites (including Amazon and eBay) avoid the problem by asking for email/phone and password on two pages. This approach also simplifies the experience: the user is only tasked with one thing at a time.

> {% Img src="image/admin/CxpObjYZMs0MMFo66f4P.png", alt="Screenshot of a sign-in form on the Amazon website: email/phone and password on two separate 'pages'.", width="400", height="385" %}
> Two-stage sign-in: email or phone, then password.

Ideally, this should be implemented with a single <form>. Use JavaScript to initially display only the email input, then hide it and show the password input. If you must force the user to navigate to a new page between entering their email and password, the form on the second page should have a hidden input element with the email value, to help enable password managers to store the correct value. Password Form Styles that Chromium Understands provides a code example.

Help users to avoid re-entering data {: #autofill }

You can help browsers store data correctly and autofill inputs, so users don't have to remember to enter email and password values. This is particularly important on mobile, and crucial for email inputs, which get high abandonment rates.

There are two parts to this:

1. The `autocomplete`, `name`, `id`, and `type` attributes help browsers understand the role of inputs in order to store data that can later be used for autofill. To allow data to be stored for autofill, modern browsers also require inputs to have a stable `name` or `id` value (not randomly generated on each page load or site deployment), and to be in a <form> with a `submit` button.
2. The `autocomplete` attribute helps browsers correctly autofill inputs using stored data.

For email inputs use `autocomplete="username"`, since `username` is recognized by password managers in modern browsers—even though you should use `type="email"` and you may want to use `id="email"` and `name="email"`.

For password inputs, use the appropriate `autocomplete` and `id` values to help browsers differentiate between new and current passwords.

Use `autocomplete="new-password"` and `id="new-password"` for a new password {:

#new-password }

- Use `autocomplete="new-password"` and `id="new-password"` for the password input in a sign-up form, or the new password in a change-password form.

Use `autocomplete="current-password"` and `id="current-password"` for an existing password {: #current-password }

- Use `autocomplete="current-password"` and `id="current-password"` for the password input in a sign-in form, or the input for the user's old password in a change-password form. This tells the browser that you want it to use the current password that it has stored for the site.

For a sign-up form:

```
<input type="password" autocomplete="new-password" id="new-password" …>
```

For sign-in:

```
<input type="password" autocomplete="current-password" id="current-password" …>
```

{% Aside %} Browsers such as Chrome can use the browser's password manager to autofill fields in the sign-in process for returning users. For these features to work, the browser needs to be able to distinguish when a user changes their password.

Specifically the form for changing the user's password should be cleared or hidden from the page after the new password is set up. If the form for changing the user's password stays filled out on the page after the password change has occurred, the browser may not be able to record the update. {% endAside %}

Support password managers {: #password-managers }

Different browsers handle email autofill and password suggestion somewhat differently, but the effects are much the same. On Safari 11 and above on desktop, for example, the password manager is displayed, and then biometric authentication (fingerprint or facial recognition) is used if available.

{% Img src="image/admin/UjBRRYaLbX9bh3LDFcAM.png", alt="Screenshots of three stages of sign-in process in Safari on desktop: password manager, biometric authentication, autofill.", width="800", height="234" %}
Sign-in with autocomplete—no text entry required!

Chrome on desktop displays email suggestions, shows the password manager, and autofills the password.

{% Img src="image/admin/mDm1cstWZB9jJDzMmzgE.png", alt="Screenshots of four stages of sign-in process in Chrome on desktop: email completion, email suggestion, password manager, autofill on selection.", width="800", height="232" %}
Autocomplete sign-in flow in Chrome 84.

Browser password and autofill systems are not simple. The algorithms for guessing, storing and displaying values are not standardized, and vary from platform to platform. For example, as pointed out by Hidde de Vries: "Firefox's password manager complements its heuristics with a recipe system."

Autofill: What web devs should know, but don't has a lot more information about using `name` and `autocomplete`. The HTML spec lists all 59 possible values.

{% Aside %} You can help password managers by using different `name` and `id` values in sign-up and sign-in forms, for the `form` element itself, as well as any `input`, `select` and `textarea` elements. {% endAside %}

Enable the browser to suggest a strong password {: #password-suggestions }

Modern browsers use heuristics to decide when to show the password manager UI and suggest a strong password.

Here's how Safari does it on desktop.

{% Img src="image/admin/B1DlZK0CllVjrOUbb5xB.png", alt="Screenshot of Firefox password manager on desktop.", width="800", height="229" %}
Password suggestion flow in Safari.

(Strong unique password suggestion has been available in Safari since version 12.0.)

Built-in browser password generators mean users and developers don't need to work out what a "strong password" is. Since browsers can securely store passwords and autofill them as necessary, there's no need for users to remember or enter passwords. Encouraging users to take advantage of built-in browser password generators also means they're more likely to use a unique, strong password on your site, and less likely to reuse a password that could be compromised elsewhere.

{% Aside %} The downside with this approach is that there's no way to share passwords across platforms. For example, a user may use Safari on their iPhone and Chrome on their Windows laptop. {% endAside %}

Help save users from accidentally missing inputs {: #required-fields }

Add the `required` attribute to both email and password fields. Modern browsers automatically prompt and set focus for missing data. No JavaScript required!

> {% Img src="image/admin/n5Nr290upVmQGvlc263U.png", alt="Screenshot of desktop Firefox and Chrome for Android showing 'Please fill out this field' prompt for missing data.", width="600", height="392" %}
> Prompt and focus for missing data on Firefox for desktop (version 76) and Chrome for Android (version 83).

Design for fingers and thumbs {: #mobile-design }

The default browser size for just about everything relating to input elements and buttons is too small, especially on mobile. This may seem obvious, but it's a common problem with sign-in forms on many sites.

Make sure inputs and buttons are large enough {: #tap-targets }

The default size and padding for inputs and buttons is too small on desktop and even worse on mobile.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/lJNO6w2dOyp4cYKl5b3y.png", alt="Screenshot of unstyled form in Chrome for desktop and Chrome for Android.", width="800", height="434" %}

According to Android accessibility guidance the recommended target size for touchscreen objects is 7–10 mm. Apple interface guidelines suggest 48x48 px, and the W3C suggest at least 44x44 CSS pixels. On that basis, add (at least) about 15 px of padding to input elements and buttons for

mobile, and around 10 px on desktop. Try this out with a real mobile device and a real finger or thumb. You should comfortably be able to tap each of your inputs and buttons.

The Tap targets are not sized appropriately Lighthouse audit can help you automate the process of detecting input elements that are too small.

Design for thumbs {: #design-for-thumbs }

Search for touch target and you'll see lots of pictures of forefingers. However, in the real world, many people use their thumbs to interact with phones. Thumbs are bigger than forefingers, and control is less precise. All the more reason for adequately sized touch targets.

Make text big enough {: #size-text-correctly }

As with size and padding, the default browser font size for input elements and buttons is too small, particularly on mobile.

> {% Img src="image/admin/EeIsqWhLbot15p4SYpo2.png", alt="Screenshot of unstyled form in Chrome on desktop and on Android.", width="800", height="494" %}
> Default styling on desktop and mobile: input text is too small to be legible for many users.

Browsers on different platforms size fonts differently, so it's difficult to specify a particular font size that works well everywhere. A quick survey of popular websites shows sizes of 13–16 pixels on desktop: matching that physical size is a good minimum for text on mobile.

This means you need to use a larger pixel size on mobile: `16px` on Chrome for desktop is quite legible, but even with good vision it's difficult to read `16px` text on Chrome for Android. You can set different font pixel sizes for different viewport sizes using media queries. `20px` is about right on mobile—but you should test this out with friends or colleagues who have low vision.

The Document doesn't use legible font sizes Lighthouse audit can help you automate the process of detecting text that's too small.

Provide enough space between inputs {: #size-margins-correctly }

Add enough margin to make inputs work well as touch targets. In other words, aim for about a finger width of margin.

Make sure your inputs are clearly visible {: #visible-inputs }

The default border styling for inputs makes them hard to see. They're almost invisible on some platforms such as Chrome for Android.

As well as padding, add a border: on a white background, a good general rule is to use `#ccc` or darker.

> {% Img src="image/admin/OgDJ5V2N7imHXSBkN4pr.png", alt="Screenshot of styled form in Chrome on Android.", width="250", height="525" %}
> Legible text, visible input borders, adequate padding and margins.

Use built-in browser features to warn of invalid input values {: #built-in-validation }

Browsers have built-in features to do basic form validation for inputs with a `type` attribute. Browsers warn when you submit a form with an invalid value, and set focus on the problematic input.

> {% Img src="image/admin/Phf9m5J66lIX9x5brzOL.png", alt="Screenshot of a sign-in form in Chrome on desktop showing browser prompt and focus for an invalid email value.", width="300", height="290" %}
> Basic built-in validation by the browser.

You can use the `:invalid` CSS selector to highlight invalid data. Use `:not(:placeholder-shown)` to avoid selecting inputs with no content.

```
input[type=email]:not(:placeholder-shown):invalid {
  color: red;
  outline-color: red;
}
```

Try out different ways of highlighting inputs with invalid values.

Use JavaScript where necessary {: #javascript }

Toggle password display {: #password-display }

You should add a **Show password** icon or button to enable users to check the text they've entered. Usability suffers when users can't see the text they've entered. Currently there's no built-in way to do this, though there are plans for implementation. You'll need to use JavaScript instead.

> Google sign-in form showing Show password icon.
> Google sign-in form: with **Show password** icon and **Forgot password** link.

The following code uses a text button to add **Show password** functionality.

HTML:

```
<section>
  <label for="password">Password</label>
  <button id="toggle-password" type="button" aria-label="Show password as plain text. Warning:
this will display your password on the screen.">Show password</button>
  <input id="password" name="password" type="password" autocomplete="current-password"
required>
</section>
```

Here's the CSS to make the button look like plain text:

```
button#toggle-password {
  background: none;
  border: none;
  cursor: pointer;
  /* Media query isn't shown here. */
  font-size: var(--mobile-font-size);
  font-weight: 300;
  padding: 0;
  /* Display at the top right of the container */
  position: absolute;
```

```
  top: 0;
  right: 0;
}
```

And the JavaScript for showing the password:

```
const passwordInput = document.getElementById('password');
const togglePasswordButton = document.getElementById('toggle-password');

togglePasswordButton.addEventListener('click', togglePassword);

function togglePassword() {
  if (passwordInput.type === 'password') {
    passwordInput.type = 'text';
    togglePasswordButton.textContent = 'Hide password';
    togglePasswordButton.setAttribute('aria-label',
      'Hide password.');
  } else {
    passwordInput.type = 'password';
    togglePasswordButton.textContent = 'Show password';
    togglePasswordButton.setAttribute('aria-label',
      'Show password as plain text. ' +
      'Warning: this will display your password on the screen.');
  }
}
```

Here's the end result:

> {% Img src="image/admin/x4NP9JMf1KI8PapQ9JFh.png", alt="Screenshots of sign-in form with Show password text 'button', in Safari on Mac and on iPhone 7.", width="800", height="468" %}
> Sign-in form with **Show password** text 'button', in Safari on Mac and iPhone 7.

Make password inputs accessible {: #accessible-password-inputs }

Use `aria-describedby` to outline password rules by giving it the ID of the element that describes the constraints. Screenreaders provide the label text, the input type (password), and then the description.

```
<input type="password" aria-describedby="password-constraints" …>
<div id="password-constraints">Eight or more characters with a mix of letters, numbers and
symbols.</div>
```

When you add **Show password** functionality, make sure to include an `aria-label` to warn that the password will be displayed. Otherwise users may inadvertently reveal passwords.

```
<button id="toggle-password"
        aria-label="Show password as plain text.
                    Warning: this will display your password on the screen.">
  Show password
</button>
```

You can see both ARIA features in action in the following Glitch:

{% Glitch { id: 'sign-in-form', path: 'index.html', height: 480 } %}

Creating Accessible Forms has more tips to help make forms accessible.

Validate in realtime and before submission {: #validation }

HTML form elements and attributes have built-in features for basic validation, but you should also use JavaScript to do more robust validation while users are entering data and when they attempt to submit the form.

{% Aside 'warning' %} Client-side validation helps users enter data and can avoid unnecessary server load, but you must always validate and sanitize data on your backend. {% endAside %}

Step 5 of the sign-in form codelab uses the Constraint Validation API (which is widely supported) to add custom validation using built-in browser UI to set focus and display prompts.

Find out more: Use JavaScript for more complex real-time validation.

Analytics and RUM {: #analytics }

"What you cannot measure, you cannot improve" is particularly true for sign-up and sign-in forms. You need to set goals, measure success, improve your site—and repeat.

Discount usability testing can be helpful for trying out changes, but you'll need real-world data to really understand how your users experience your sign-up and sign-in forms:

- **Page analytics**: sign-up and sign-in page views, bounce rates, and exits.
- **Interaction analytics**: goal funnels (where do users abandon your sign-in or sign-in flow?) and events (what actions do users take when interacting with your forms?)
- **Website performance**: user-centric metrics (are your sign-up and sign-in forms slow for some reason and, if so, what is the cause?).

You may also want to consider implementing A/B testing in order to try out different approaches to sign-up and sign-in, and staged rollouts to validate the changes on a subset of users before releasing changes to all users.

General guidelines {: #general-guidelines }

Well designed UI and UX can reduce sign-in form abandonment:

- Don't make users hunt for sign-in! Put a link to the sign-in form at the top of the page, using well-understood wording such as **Sign In**, **Create Account** or **Register**.
- Keep it focused! Sign-up forms are not the place to distract people with offers and other site features.
- Minimize sign-up complexity. Collect other user data (such as addresses or credit card details) only when users see a clear benefit from providing that data.
- Before users start on your sign-up form, make it clear what the value proposition is. How do they benefit from signing in? Give users concrete incentives to complete sign-up.
- If possible allow users to identify themselves with a mobile phone number instead of an email address, since some users may not use email.
- Make it easy for users to reset their password, and make the **Forgot your password?** link obvious.
- Link to your terms of service and privacy policy documents: make it clear to users from the start how you safeguard their data.
- Include the logo and name of your company or organization on your signup and sign-in pages, and make sure that language, fonts and styles match the rest of your site. Some forms don't feel like they belong to the same site as other content, especially if they have a significantly different URL.

Keep learning {: #resources }

- Create Amazing Forms
- Best Practices For Mobile Form Design
- More capable form controls
- Creating Accessible Forms
- Streamlining the Sign-in Flow Using Credential Management API
- Verify phone numbers on the web with the WebOTP API

Photo by Meghan Schiereck on Unsplash.

Sign-up form best practices

{% YouTube 'Ev2mCzJZLtY' %}

If users ever need to log in to your site, then good sign-up form design is critical. This is especially true for people on poor connections, on mobile, in a hurry, or under stress. Poorly designed sign-up forms get high bounce rates. Each bounce could mean a lost and disgruntled user—not just a missed sign-up opportunity.

{% Aside 'codelab' %} If you would prefer to learn these best practices with a hands-on tutorial, check out the Sign-up form best practices codelab. {% endAside %}

Here is an example of a very simple sign-up form that demonstrates all of the best practices:

{% Glitch { id: 'signup-form', path: 'index.html', height: 700 } %}

{% Aside 'caution' %} This post is about form best practices.

It does not explain how to implement sign-up via a third-party identity provider (federated login) or show how to build backend services to authenticate users, store credentials, and manage accounts.

Integrating Google Sign-In into your web app explains how to add federated login to your sign-up options.

12 best practices for user account, authorization and password management outlines core back-end principles for managing user accounts. {% endAside %}

Checklist

- Avoid sign-in if you can.
- Make it obvious how to create an account.
- Make it obvious how to access account details.
- Cut form clutter.
- Consider session length.
- Help password managers securely suggest and store passwords.
- Don't allow compromised passwords.

- Do allow password pasting.
- Never store or transmit passwords in plain text.
- Don't force password updates.
- Make it simple to change or reset passwords.
- Enable federated login.
- Make account switching simple.
- Consider offering multi-factor authentication.
- Take care with usernames.
- Test in the field as well as the lab.
- Test on a range of browsers, devices and platforms.

Avoid sign-in if you can {: #no-forced-sign-in }

Before you implement a sign-up form and ask users to create an account on your site, consider whether you really need to. Wherever possible you should avoid gating features behind login.

The best sign-up form is no sign-up form!

By asking a user to create an account, you come between them and what they're trying to achieve. You're asking a favor, and asking the user to trust you with personal data. Every password and item of data you store carries privacy and security "data debt", becoming a cost and liability for your site.

If the main reason you ask users to create an account is to save information between navigations or browsing sessions, consider using client-side storage instead. For shopping sites, forcing users to create an account to make a purchase is cited as a major reason for shopping cart abandonment. You should make guest checkout the default.

Make sign-in obvious {: #obvious-account-creation}

Make it obvious how to create an account on your site, for example with a **Login** or **Sign in** button at the top right of the page. Avoid using an ambiguous icon or vague wording ("Get on board!",

"Join us") and don't hide login in a navigational menu. The usability expert Steve Krug summed up this approach to website usability: Don't make me think! If you need to convince others on your web team, use analytics to show the impact of different options.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/KeztoU8KgAqrQ5CKBSWw.jpg", alt="Two screenshots of a mockup ecommerce website viewed on an Android phone. The one on the left uses an icon for the sign-in link that's somewhat ambiguous; the one on the right simply says 'Sign in'", width="800", height="737" %}
> Make sign-in obvious. An icon may be ambiguous, but a **Sign in** button or link is obvious.

{% Aside %} You may be wondering whether to add a button (or link) to create an account and another one for existing users to sign in. Many popular sites now simply display a single **Sign in** button. When the user taps or clicks on that, they also get a link to create an account if necessary. That's a common pattern now, and your users are likely to understand it, but you can use interaction analytics to monitor whether or not a single button works best. {% endAside %}

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/WUgCNqhEgvoWEVwGjfrA.jpg", alt="Screenshots of sign-in for Gmail: one page, showing Sign in button, when clicked leads to form that also has a Create account link.", width="800", height="545" %}
> The Gmail sign-in page has a link to create an account.
> At window sizes larger than shown here, Gmail displays a **Sign in** link and a **Create an account** button.

Make sure to link accounts for users who sign up via an identity provider such as Google and who also sign up using email and password. That's easy to do if you can access a user's email address from the profile data from the identity provider, and match the two accounts. The code below shows how to access email data for a Google Sign-in user.

```
// auth2 is initialized with gapi.auth2.init()
if (auth2.isSignedIn.get()) {
  var profile = auth2.currentUser.get().getBasicProfile();
  console.log(`Email: ${profile.getEmail()}`);
}
```

{: #obvious-account-details}

Once a user has signed in, make it clear how to access account details. In particular, make it obvious how to change or reset passwords.

Cut form clutter {: #cut-clutter}

In the sign-up flow, your job is to minimize complexity and keep the user focused. Cut the clutter. This is not the time for distractions and temptations!

> Don't distract users from completing sign-up.

On sign-up, ask for as little as possible. Collect additional user data (such as name and address) only when you need to, and when the user sees a clear benefit from providing that data. Bear in mind that every item of data you communicate and store incurs cost and liability.

Don't double up your inputs just to make sure users get their contact details right. That slows down form completion and doesn't make sense if form fields are autofilled. Instead, send a confirmation code to the user once they've entered their contact details, then continue with account creation once they respond. This is a common sign-up pattern: users are used to it.

You may want to consider password-free sign-in by sending users a code every time they sign in on a new device or browser. Sites such as Slack and Medium use a version of this.

> Password-free sign-in on medium.com.

As with federated login, this has the added benefit that you don't have to manage user passwords.

Consider session length {: #session-length}

Whatever approach you take to user identity, you'll need to make a careful decision about session length: how long the user remains logged in, and what might cause you to log them out.

Consider whether your users are on mobile or desktop, and whether they are sharing on desktop, or sharing devices.

{% Aside %} You can get around some of the issues of shared devices by enforcing re-authentication for sensitive features, for example when a purchase is made or an account updated. You can find out more about ways to implement re-authentication from the codelab Your First WebAuthn App. {% endAside %}

Help password managers securely suggest and store passwords {:

#help-password-managers}

You can help third party and built-in browser password managers suggest and store passwords, so that users don't need to choose, remember or type passwords themselves. Password managers work well in modern browsers, synchronizing accounts across devices, across platform-specific and web apps—and for new devices.

This makes it extremely important to code sign-up forms correctly, in particular to use the correct autocomplete values. For sign-up forms use `autocomplete="new-password"` for new passwords, and add correct autocomplete values to other form fields wherever possible, such as `autocomplete="email"` and `autocomplete="tel"`. You can also help password managers by using different `name` and `id` values in sign-up and sign-in forms, for the `form` element itself, as well as any `input`, `select` and `textarea` elements.

You should also use the appropriate `type` attribute to provide the right keyboard on mobile and enable basic built-in validation by the browser. You can find out more from Payment and address form best practices.

{% Aside %} Sign-in form best practices has lots more tips on how to improve form design, layout and accessibility, and how to code forms correctly in order to take advantage of built-in browser features. {% endAside %}

Ensure users enter secure passwords {: #secure-passwords}

Enabling password managers to suggest passwords is the best option, and you should encourage users to accept the strong passwords suggested by browsers and third-party browser managers.

However, many users want to enter their own passwords, so you need to implement rules for password strength. The US National Institute of Standards and Technology explains how to avoid insecure passwords.

{% Aside 'warning' %} Sign-up forms on some sites have password validation rules that don't allow the strong passwords generated by browser and third-party password managers. Make sure your site doesn't do this, since it interrupts form completion, annoys users, and requires users to make up their own passwords, which may be less secure than those generated by password managers. {% endAside %}

Don't allow compromised passwords {: #no-compromised-passwords}

Whatever rules you choose for passwords, you should never allow passwords that have been exposed in security breaches.

Once a user has entered a password, you need to check that it's not a password that's already been compromised. The site Have I Been Pwned provides an API for password checking, or you can run this as a service yourself.

Google's Password Manager also allows you to check if any of your existing passwords have been compromised.

If you do reject the password that a user proposes, tell them specifically why it was rejected. Show problems inline and explain how to fix them, as soon as the user has entered a value—not after they've submitted the sign-up form and had to wait for a response from your server.

> Be clear why a password is rejected.

Don't prohibit password pasting {: #allow-password-pasting}

Some sites don't allow text to be pasted into password inputs.

Disallowing password pasting annoys users, encourages passwords that are memorable (and therefore may be easier to compromise) and, according to organizations such as the UK National Cyber Security Centre, may actually reduce security. Users only become aware that pasting is disallowed *after* they try to paste their password, so disallowing password pasting doesn't avoid clipboard vulnerabilities.

Never store or transmit passwords in plain text {: #salt-and-hash}

Make sure to salt and hash passwords—and don't try to invent your own hashing algorithm!

Don't force password updates {: #no-forced-password-updates}

Don't force users to update their passwords arbitrarily.

Forcing password updates can be costly for IT departments, annoying to users, and doesn't have much impact on security. It's also likely to encourage people to use insecure memorable passwords, or to keep a physical record of passwords.

Rather than force password updates, you should monitor for unusual account activity and warn users. If possible you should also monitor for passwords that become compromised because of data breaches.

You should also provide your users with access to their account login history, showing them where and when a login happened.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/zZXmhWc9bZ1GtvrE5Ooq.jpg", alt="Gmail account activity page", width="800", height="469" %}
>
> Gmail account activity page.

Make it simple to change or reset passwords {: #password-change}

Make it obvious to users where and how to **update** their account password. On some sites, it's surprisingly difficult.

You should, of course, also make it simple for users to **reset** their password if they forget it. The Open Web Application Security Project provides detailed guidance on how to handle lost passwords

.

To keep your business and your users safe, it's especially important to help users change their password if they discover that it's been compromised. To make this easier, you should add a `/.well-known/change-password` URL to your site that redirects to your password management page. This enables password managers to navigate your users directly to the page where they can change their password for your site. This feature is now implemented in Safari, Chrome, and is coming to other browsers. Help users change passwords easily by adding a well-known URL for changing passwords explains how to implement this.

You should also make it simple for users to delete their account if that's what they want.

Offer login via third-party identity providers {: #federated-login}

Many users prefer to log in to websites using an email address and password sign-up form. However, you should also enable users to log in via a third party identity provider, also known as federated login.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/jubgwX1shLB7qAIiioTU.jpg", alt="WordPress login page", width="800", height="513" %}
> WordPress login page, with Google and Apple login options.

This approach has several advantages. For users who create an account using federated login, you don't need to ask for, communicate, or store passwords.

You may also be able to access additional verified profile information from federated login, such as an email address—which means the user doesn't have to enter that data and you don't need to do the verification yourself. Federated login can also make it much easier for users when they get a new device.

Integrating Google Sign-In into your web app explains how to add federated login to your sign-up options. Many other identity platforms are available.

{% Aside %} "First day experience" when you get a new device is increasingly important. Users expect to log in from multiple devices including their phone, laptop, desktop, tablet, TV, or from a car. If your sign-up and sign-in forms aren't seamless, this is a moment where you risk losing users, or at least losing contact with them until they get set up again. You need to make it as quick and easy as possible for users on new devices to get up and running on your site. This is another area where federated login can help. {% endAside %}

Make account switching simple {: #account-switching}

Many users share devices and swap between accounts using the same browser. Whether users access federated login or not, you should make account switching simple.

> {% Img src="image/tcFciHGuF3MxnTr1y5ue01OGLBn2/sPDZJIY5Vo2ijqyuofCy.jpg", alt="Gmail, showing account switching", width="800", height="494" %}
> Account switching on Gmail.

Consider offering multi-factor authentication {: #multi-factor-authentication}

Multi-factor authentication means ensuring that users provide authentication in more than one way. For example, as well as requiring the user to set a password, you might also enforce verification using a one-time-passcode sent by email or an SMS text message, or by using an app-based one-time code, security key or fingerprint sensor. SMS OTP best practices and Enabling Strong Authentication with WebAuthn explain how to implement multi-factor authentication.

You should certainly offer (or enforce) multi-factor authentication if your site handles personal or sensitive information.

Take care with usernames {: #username}

Don't insist on a username unless (or until) you need one. Enable users to sign up and sign in with only an email address (or telephone number) and password—or federated login if they prefer. Don't force them to choose and remember a username.

If your site does require usernames, don't impose unreasonable rules on them, and don't stop users from updating their username. On your backend you should generate a unique ID for every

user account, not an identifier based on personal data such as username.

Also make sure to use `autocomplete="username"` for usernames.

{% Aside 'caution' %} As with personal names, ensure that usernames aren't restricted to characters from the Latin alphabet. Payment and address form best practices explains how and why to validate using Unicode letter matching. {% endAside %}

Test on a range of devices, platforms, browsers and versions {: #test-platforms}

Test sign-up forms on the platforms most common for your users. Form element functionality may vary, and differences in viewport size can cause layout problems. BrowserStack enables free testing for open source projects on a range of devices and browsers.

Implement analytics and Real User Monitoring {: #analytics-rum}

You need field data as well as lab data to understand how users experience your sign-up forms. Analytics and Real User Monitoring (RUM) provide data for the actual experience of your users, such as how long sign-up pages take to load, which UI components users interact with (or not) and how long it takes users to complete sign-up.

- **Page analytics**: page views, bounce rates and exits for every page in your sign-up flow.
- **Interaction analytics**: goal funnels and events indicate where users abandon the sign-up flow and what proportion of users click buttons, links, and other components of your sign-up pages.
- **Website performance**: user-centric metrics can tell you if your sign-up flow is slow to load or visually unstable.

Small changes can make a big difference to completion rates for sign-up forms. Analytics and RUM enable you to optimize and prioritize changes, and monitor your site for problems that aren't

exposed by local testing.

Keep learning {: #resources }

- Sign-in form best practices
- Payment and address form best practices
- Create Amazing Forms
- Best Practices For Mobile Form Design
- More capable form controls
- Creating Accessible Forms
- Streamlining the Sign-up Flow Using Credential Management API
- Verify phone numbers on the web with the WebOTP API

Photo by @ecowarriorprincess on Unsplash.

SMS OTP form best practices

{% YouTube 'sU4MpWYrGSI' %}

Asking a user to provide the OTP (one time password) delivered via SMS is a common way to confirm a user's phone number. There are a few use cases for SMS OTP:

- **Two-factor authentication.** In addition to username and password, SMS OTP can be used as a strong signal that the account is owned by the person who received the SMS OTP.
- **Phone number verification.** Some services use a phone number as the user's primary identifier. In such services, users can enter their phone number and the OTP received via SMS to prove their identity. Sometimes it's combined with a PIN to constitute a two-factor authentication.
- **Account recovery.** When a user loses access to their account, there needs to be a way to recover it. Sending an email to their registered email address or an SMS OTP to their phone number are common account recovery methods.
- **Payment confirmation** In payment systems, some banks or credit card issuers request additional authentication from the payer for security reasons. SMS OTP is commonly used for that purpose.

This post explains best practices to build an SMS OTP form for the above use cases.

{% Aside 'caution' %} While this post discusses SMS OTP form best practices, be aware that SMS OTP is not the most secure method of authentication by itself because phone numbers can be recycled and sometimes hijacked. And the concept of OTP itself is not phishing resistant.

If you are looking for better security, consider using WebAuthn. Learn more about it from the talk "What's new in sign-up & sign-in" at the Chrome Dev Summit 2019 and build a reauthentication experience using a biometric sensor with "Build your first WebAuthn app" codelab. {% endAside %}

Checklist

To provide the best user experience with the SMS OTP, follow these steps:

- Use the `<input>` element with:
  - `type="text"`

    - `inputmode="numeric"`
    - `autocomplete="one-time-code"`
- Use `@BOUND_DOMAIN #OTP_CODE` as the last line of the OTP SMS message.
- Use the WebOTP API.

Use the `<input>` element

Using a form with an `<input>` element is the most important best practice you can follow because it works in all browsers. Even if other suggestions from this post don't work in some browser, the user will still be able to enter and submit the OTP manually.

```
<form action="/verify-otp" method="POST">
  <input type="text"
         inputmode="numeric"
         autocomplete="one-time-code"
         pattern="\d{6}"
         required>
</form>
```

The following are a few ideas to ensure an input field gets the best out of browser functionality.

`type="text"`

Since OTPs are usually five or six digit numbers, using `type="number"` for an input field might seem intuitive because it changes the mobile keyboard to numbers only. This is not recommended because the browser expects an input field to be a countable number rather than a sequence of multiple numbers, which can cause unexpected behavior. Using `type="number"` causes up and down buttons to be displayed beside the input field; pressing these buttons increments or decrements the number and may remove preceding zeros.

Use `type="text"` instead. This won't turn the mobile keyboard into numbers only, but that is fine because the next tip for using `inputmode="numeric"` does that job.

`inputmode="numeric"`

Use `inputmode="numeric"` to change the mobile keyboard to numbers only.

Some websites use `type="tel"` for OTP input fields since it also turns the mobile keyboard to numbers only (including `*` and `#`) when focused. This hack was used in the past when `inputmode="numeric"` wasn't widely supported. Since Firefox started supporting `inputmode="numeric"`,

there's no need to use the semantically incorrect `type="tel"` hack.

`autocomplete="one-time-code"`

`autocomplete` attribute lets developers specify what permission the browser has to provide autocomplete assistance and informs the browser about the type of information expected in the field.

With `autocomplete="one-time-code"` whenever a user receives an SMS message while a form is open, the operating system will parse the OTP in the SMS heuristically and the keyboard will suggest the OTP for the user to enter. It works only on Safari 12 and later on iOS, iPadOS, and macOS, but we strongly recommend using it, because it is an easy way to improve the SMS OTP experience on those platforms.

`autocomplete="one-time-code"` in action.

`autocomplete="one-time-code"` improves the user experience, but there's more you can do by ensuring that the SMS message complies with the origin-bound message format.

{% Aside %} Optional attributes:

- `pattern` specifies the format that the entered OTP must match. Use regular expressions to specify the matching pattern, for example, `\d{6}` constrains the OTP to a six digit string. Learn more about the `pattern` attribute in Use JavaScript for more complex real-time validation
- `required` indicates that a field is required.

For more general form best practices, Sam Dutton's Sign-in form best practices is a great starting point. {% endAside %}

Format the SMS text {: #format }

Enhance the user experience of entering an OTP by aligning with the origin-bound one-time codes delivered via SMS specification.

The format rule is simple: Finish the SMS message with the receiver domain preceded with `@` and the OTP preceded with `#`.

For example:

```
Your OTP is 123456


@web-otp.glitch.me #123456
```

Using a standard format for OTP messages makes extraction of codes from them easier and more reliable. Associating OTP codes with websites makes it harder to trick users into providing a code to malicious sites.

{% Aside %} The precise rules are:

- The message begins with (optional) human-readable text that contains a four to ten character alphanumeric string with at least one number, leaving the last line for the URL and the OTP.
- The domain part of the URL of the website that invoked the API must be preceded by `@`.
- The URL must contain a pound sign ("`#`") followed by the OTP.

Make sure the number of characters doesn't exceed 140 in total.

To learn more about Chrome specific rules, read Format the SMS message section of WebOTP API post. {% endAside %}

Using this format provides a couple of benefits:

- The OTP will be bound to the domain. If the user is on domains other than the one specified in the SMS message, the OTP suggestion won't appear. This also mitigates the risk of phishing attacks and potential account hijacks.
- Browser will now be able to reliably extract the OTP without depending on mysterious and flaky heuristics.

When a website uses `autocomplete="one-time-code"`, Safari with iOS 14 or later will suggest the OTP following the above rules.

{% Aside %} If the user is on a desktop with macOS Big Sur with the same iCloud account set up as on iOS, the OTP received on the iOS device will be available on the desktop Safari as well.

To learn more about other benefits and nuances of the availability on Apple platforms, read Enhance SMS-delivered code security with domain-bound codes. {% endAside %}

This SMS message format also benefits browsers other than Safari. Chrome, Opera, and Vivaldi on Android also support the origin-bound one-time codes rule with the WebOTP API, though not through `autocomplete="one-time-code"`.

Use the WebOTP API

The WebOTP API provides access to the OTP received in an SMS message. By calling `navigator.credentials.get()` with `otp` type (`OTPCredential`) where `transport` includes `sms`, the website will wait for an SMS that complies with the origin-bound one-time codes to be delivered and granted access by the user. Once the OTP is passed to JavaScript, the website can use it in a form or POST it directly to the server.

{% Aside 'caution' %} The WebOTP API requires a secure origin (HTTPS). {% endAside %}

```
navigator.credentials.get({
  otp: {transport:['sms']}
})
.then(otp => input.value = otp.code);
```

WebOTP API in action.

Learn how to use the WebOTP API in detail in Verify phone numbers on the web with the WebOTP API or copy and paste the following snippet. (Make sure the `<form>` element has an `action` and `method` attribute properly set.)

```
// Feature detection
if ('OTPCredential' in window) {
  window.addEventListener('DOMContentLoaded', e => {
    const input = document.querySelector('input[autocomplete="one-time-code"]');
    if (!input) return;
    // Cancel the WebOTP API if the form is submitted manually.
    const ac = new AbortController();
    const form = input.closest('form');
    if (form) {
      form.addEventListener('submit', e => {
        // Cancel the WebOTP API.
        ac.abort();
      });
    }
    // Invoke the WebOTP API
    navigator.credentials.get({
      otp: { transport:['sms'] },
      signal: ac.signal
    }).then(otp => {
      input.value = otp.code;
      // Automatically submit the form when an OTP is obtained.
      if (form) form.submit();
```

```
    }).catch(err => {
      console.log(err);
    });
  });
}
```

Photo by Jason Leung on Unsplash.