

2023 EDITION

The Definitive Guide to Passwordless Authentication

Table of Contents



Passwords are being phased out of the SaaS space. While most B2C companies are already embracing passwordless, B2B companies are still playing catch up when it comes to embracing the security, usability, and productivity benefits of this authentication methodology. This guide will cover all aspects of Passwordless and help you get started in the best way possible.

- 02 What is Passwordless Authentication?
- 03 The Pros and Cons of Passwordless SSH Authentication
- 05 Main Types of Passwordless Authentication
- 08 Passwordless Authentication Best Practices
- 12 Passwordless: The New Authentication Standard



What is Passwordless Authentication?

Passwordless authentication is all about stopping the use of passwords to bolster security, improve brand performance, and conserve valuable IT resources. This kind of authentication works well for all [kinds of SaaS applications](#) - legacy, on-prem, cloud-based, and even ones with hybrid setups. It's also better for users on-the-go who are becoming more dependent on smartphones and tablets.

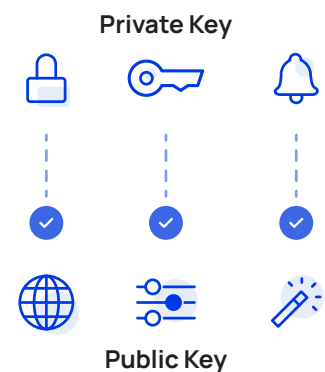
DID YOU KNOW?

As per [Statista](#), global revenue from passwordless authentication sales, currently at around 15 billion USD, is expected to cross the 25 billion USD by 2025 and touch 30 billion USD in 2030.



For example, let's focus on the fingerprint authentication methodology and how it works behind the scenes when an end-user opts for this technique.

The private key and public key are two separate entities. The private key can be tied to the [fingerprint authentication](#) that the end-user creates with private tools like smartphones or laptops. This private key, stored on the device itself, can only be accessed by the end-user. The public key is provided to the SaaS application or website, where the user account is actually being created.



Opting for Passwordless can help you enforce enhanced security standards while also implementing an improved user experience (UX) to increase customer satisfaction. You can also significantly reduce the total cost of ownership (TCO), since passwords are extremely expensive to maintain. For example, you can have IT staff resource wastage and cumbersome damage control processes after data breaches.




Related: [Security Measures to Prevent Authentication Attacks](#)



The Pros and Cons of Passwordless SSH (Secure Socket Shell) Authentication

So, does this mean that you should quickly run tomorrow morning to delete ALL passwords from your database and solely provide SSH passwordless based authentication? Should you unconditionally opt for this move that employs a pair of public (resides on the server) and private keys (brought by clients for authentication) for asymmetric encryption? Let's consider the pros and cons.

The pros when it comes to passwordless authentication are rather obvious:

-  **Brute Force Attack Immunity**
More often than not, passwords tend to be weak. Human nature drives people to maintain the same password across all SaaS apps, which leads to an increased risk of password breaches.
-  **Improved User Experience (UX)**
Users do not need to remember passwords, nor do they have to change them constantly and follow strict password policy rules. Passwordless option offers an easy flow.
-  **Resource Friendly**
Getting rid of passwords simply allows organizations to use up less resources, not to mention the cost saving that comes with it. There are also no password resets ([\\$70 average cost per reset](#)).



The cons of the passwordless approach, amongst other things, are:

✗ **Hard to Implement**

In most cases, email + passwords are very easy to implement but a flow where we need to maintain expirations on tokens and ship out emails, makes the implementation complex and costly.

✗ **Still Not an Established Standard**

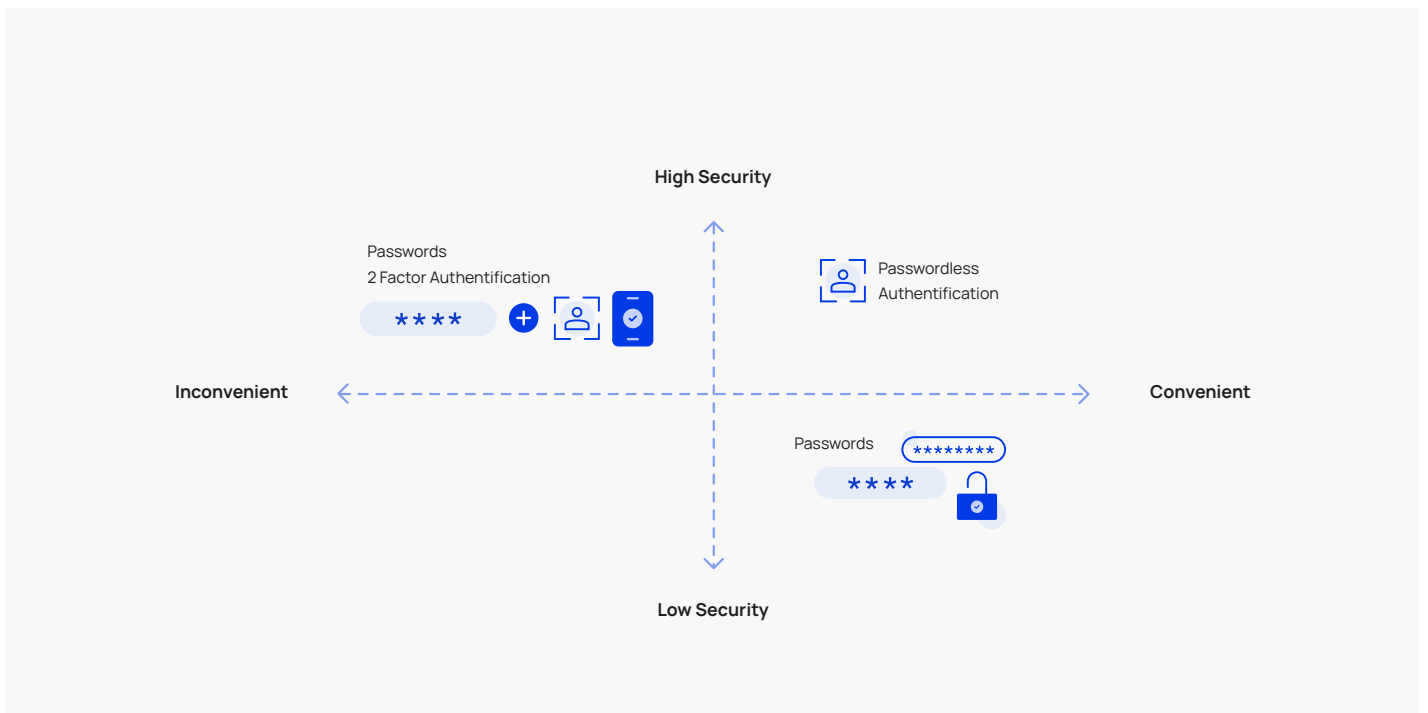
While users are used to email and password-based authentication, the “entry point” for passwordless authentication is somehow limited.

✗ **Dependency on 3rd Parties**

Using password+email authentication means we can take care of activation immediately. When one of the users is not getting his activation email, the dependence makes it harder to integrate.

✗ **Less Relevant in the Case of IDP / SSO Authentication**

With [SAML/SSO](#), there is no need for passwordless SSO authentication (at least on the app side). Users have one password, the same one used for their email login.





Main Types of Passwordless Authentication

Before getting started with passwordless authentication, you will need to pick your flavor. There are basically three main variations you should be considering.

1. One-time passwords (otps)

OTPs are essentially numeric codes that have been linked to a unique reference. Since these unique codes are sent directly to the user, only the server knows about it. All the user needs to do is enter the code into the platform, following which authentication is completed and access is granted. How are these codes sent? Mobile is the primary medium, but emails can also be used.

✓ PROS

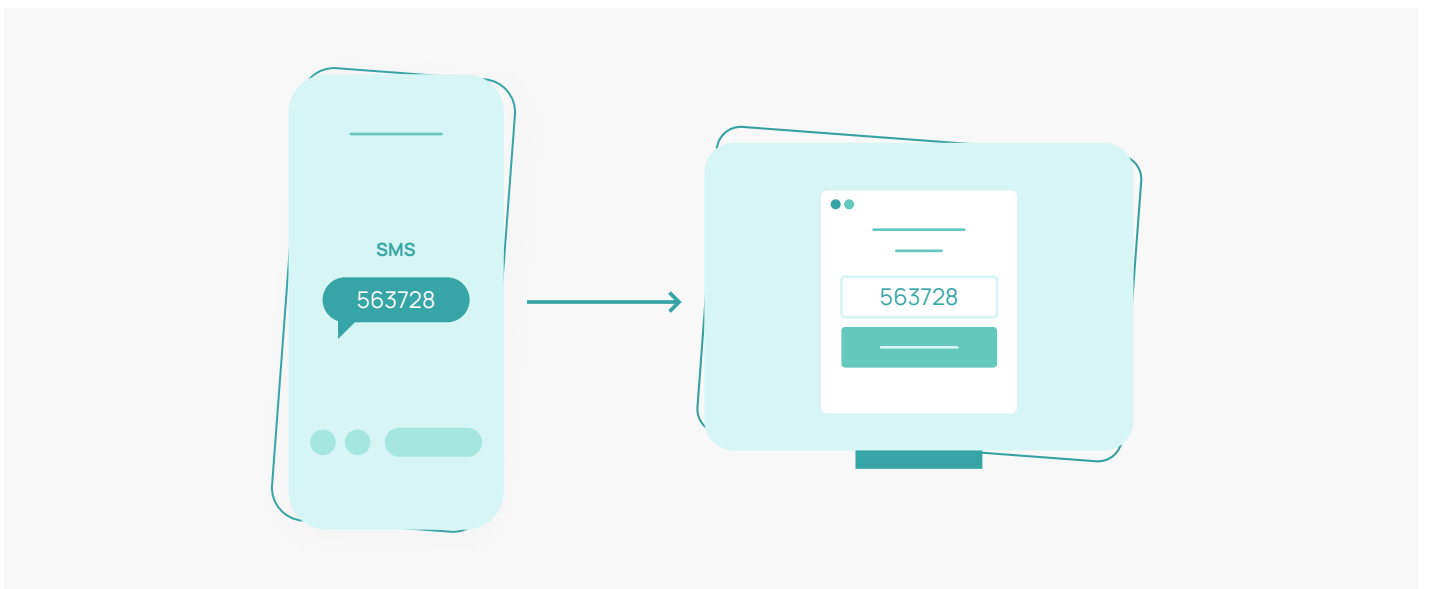
Other users can't access the unique reference - very secure, The OTPs can be sent in voice format - more versatile. Also serves as device verification

✗ CONS

Hacking or breaching the user's phone or laptop can be disastrous. Using two devices can create friction in some use cases

+ USE CASES

Finance, banking platforms, insurance companies, healthcare, government and educational services





2. Magic Links

Magic links, also known as one-click solutions, have become extremely popular due to their ease-of-use and email-centric characteristics, which are extremely suitable for B2B settings. These one-step solutions are basically URLs with authentication tokens that are sent via email. All the user needs to do is to click on them to trigger the authentication process and get redirected to the app.

✓ PROS

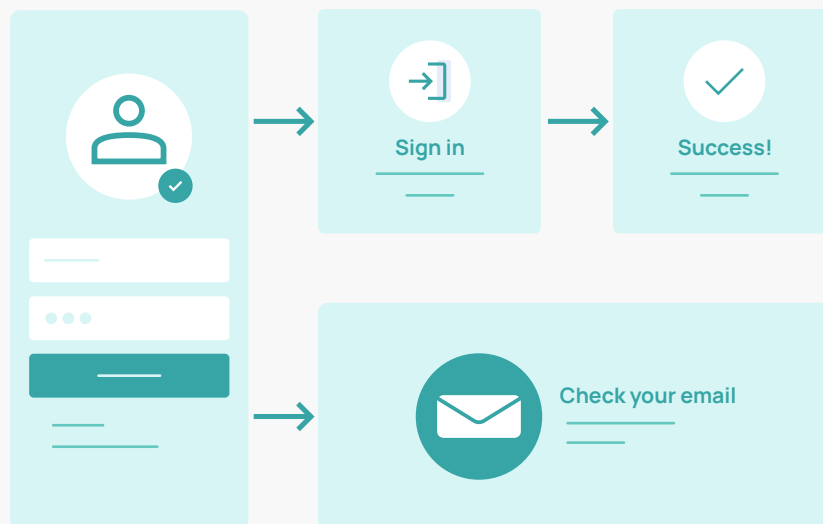
Minimal friction because only one action is required. The URLs can also be sent via mobile if needed. Users can be redirected with parameterized URLs

✗ CONS

Hackers with unauthorized access to the user's email can wreak havoc

+ USE CASES

Software-as-a-Service applications and platforms across all industries productivity, entertainment, utility, and more





3. Hardware/Physical Authentication

Magic links, also known as one-click solutions, have become extremely popular due to their ease-of-use and email-centric characteristics, which are extremely suitable for B2B settings. These one step solutions are basically URLs with authentication tokens that are sent via email. All the user needs to do is to click on them to trigger the authentication process and get redirected to the app.

✓ PROS

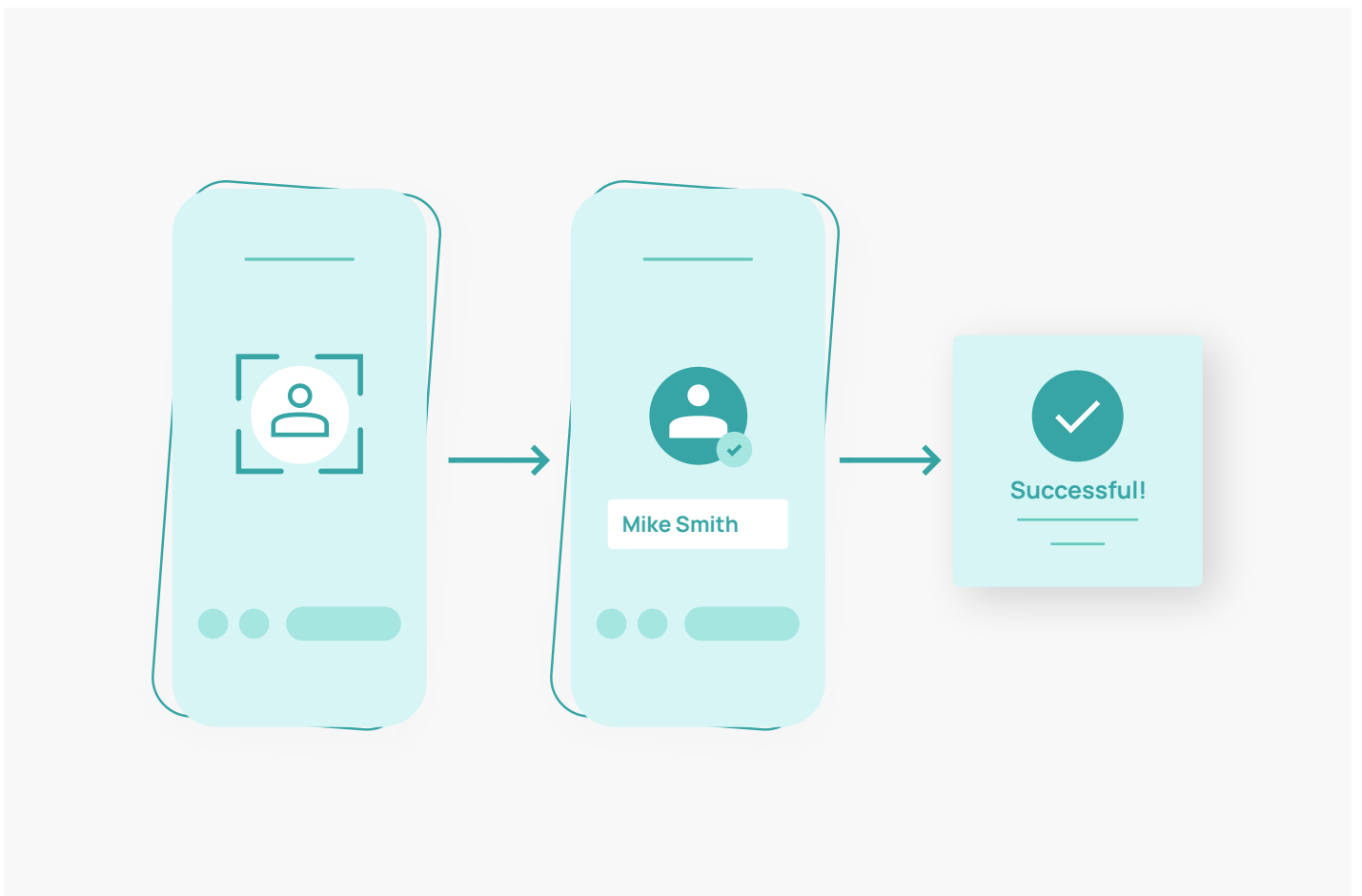
Considered to be extremely secure. Can also be used to verify devices. Enhanced user satisfaction metrics

✗ CONS

Requires investment in infrastructure. Privacy implications that can potentially conflict with regulatory requirements

+ USE CASES

Travel (biometric passports and fingerprint readers), online gaming and betting, hotel check-ins





Passwordless Authentication Best Practices

Before even diving into the specifics of passwordless authentication, you need to brainstorm with your team and try to answer some key questions. What is your target audience? What's their average age? Can you pinpoint specific geolocations? What kind of data are you handling? What regulatory rules do you need to comply with? These questions will help you get a better feel for what is really needed.



Enjoying passwordless authentication is possible only when you have taken the right steps while setting it up. Once you have decided on your strategy, you should embrace the following best practices while getting started.



1. Going Passwordless is Not a One-Off Move

Passwordless authentication is here to stay, but most SaaS users are still using passwords in one way or another. The shift to passwordless is inevitable, but it has to be done with user acceptance and onboarding. Make sure your users know about the upcoming shift to make adoption as smooth as possible. Social media logins can be helpful add-ons while making the move.

+ PRO TIP: Introduce passwordless as an optional method before making the switch

2. Connect Identities to Devices Tightly

As obvious as this requirement may sound, many companies fail to create air-tight connections between end-devices and identities. The reason for this is simple. When it comes to passwordless authentication, the user's device is the authentication origin. Seamless and ongoing identity binding is key when it comes to blocking out spoofing attempts and other kinds of malicious activity.

+ PRO TIP: Register all devices properly while ensuring explicit transitive trust

3. Ensure Secure and Smooth Account Recovery

Another often overlooked aspect of passwordless authentication is account recovery. The recovery has to be smooth to minimize friction, but also secure to stay clear of privacy and compliance issues. Therefore, the recovery should be client-side and never on the server side, something that significantly reduces the risk of injection-based threats or credential exploits.

+ PRO TIP: Always opt for secured and authenticated channels to transfer data



4. Strive to be Verifier Compromise Resistant

It's also best to be prepared for the worst. You should ideally make sure that any IdP exploits will not result in data theft. With privacy laws like GDPR, HIPAA, and CCPA in full force, any such leak can result in hefty fines and brand damage that can be hard to recover from. For example, if user authentication secrets are being stored, your ecosystem is not really verifier compromise resistant.

+ PRO TIP: Pen test your app from time to time to gauge your security posture

5. Invest in an Authentication Ecosystem

Passwordless authentication is much more than a one-time action. Companies should strive to get a holistic view of their entire authentication ecosystem and take control of the user experience, while securing it. How? Try combining user trust scoring and Mobile Threat Defense (MTD). The concept of continuous authentication has to be instilled relentlessly to create a robust habitat.

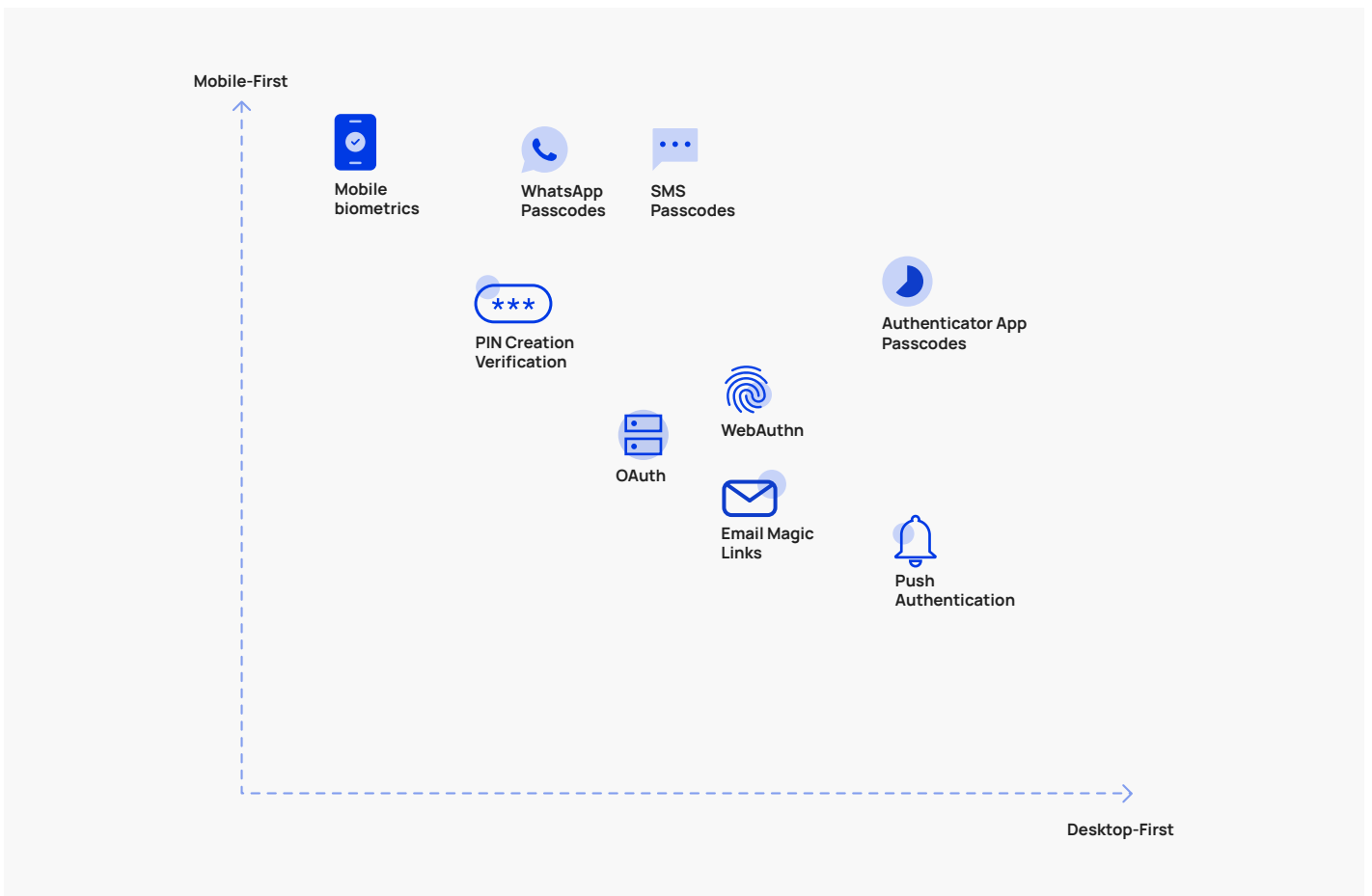
+ PRO TIP: Use API integrations to eliminate siloed data and gain a 360 view

Now that you have the fundamental understanding of passwordless authentication and what crucial steps need to be taken before getting started, you need to find the right vendor for your specific needs. We've got you covered on that front as well. **Our [Top 10 Passwordless Vendors](#) list has the best options you can find today, all analyzed and compared for your convenience.**



Picking the right passwordless method can also be a challenging task.

This should be done by understanding what your customers are doing and how they are using your product. For example, mobile vs desktop is a major consideration today. B2B SaaS businesses are desktop heavy. So it may be a great idea to go with email magic links or OAuth based authentication. eCommerce use cases are mobile-heavy, requiring more smartphone verifications and social logins.





Passwordless: The New Authentication Standard

Passwordless authentication is soon becoming the industry standard. The idea of not requiring a user to remember new passwords for multiple accounts enhances the level of trust in the authentication flow, eventually boosting engagement and satisfaction metrics. Getting started with it and implementing it correctly however, can be challenging. Let's take a closer look at things.

For defining the front-end for your email magic link implementation, you'll need:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Passwordless Authentication</title>
5     <script src="./frontend.js"></script>
6   </head>
7   <body>
8     <h1>This is where you'll put your email to get a magic link.</h1>
9     <form>
10      <div>
11        <label for="email_address">Enter your email address</label>
12        <input type="email" id="email_address" />
13      </div>
14      <button type="submit" id="submit_email">Get magic link</button>
15    </form>
16  </body>
```

As a result, you'll be looking at frontend.js file along these lines:

```
1 window.onload = () => {
2   const submitButton = document.getElementById("submit_email");
3   const emailInput = document.getElementById("email_address");
4   submitButton.addEventListener("click", handleAuth);
5   /** This function submits the request to the server for sending the user a magic
6   link.
7   * Params: email address
8   * Returns: message
9   */
10  async function handleAuth() {
11    const message = await axios.post("http://localhost:4300/login", {
12      email: emailInput.value
13    });
14    return message;
15  }
16  };
```



Now, we can shift our focus to the back-end (for example, Node.js). Let's start by creating an express app and installing a few packages on the way:

```
1 import cors from "cors";
2 import express from "express";
3
4 const PORT = process.env.PORT || 4000;
5 const app = express();
6 // Set up middleware
7 app.use(cors());
8 app.use(express.json());
9 app.use(express.urlencoded({ extended: false }));
10 // Login endpoint
11 app.post("/login", (req, res) => {
12   const email = req.body.email;
13   if (!email) {
14     res.status(403);
15     res.send({
16       message: "There is no email address that matches this.",
17     });
18   }
19   if (email) {
20     res.status(200);
21     res.send(email);
22   }
23 });
24 // Start up the server on the port defined in the environment
25 const server = app.listen(PORT, () => {
26   console.info("Server running on port " + PORT)
27 })
28 export default server
```

With this basic server in place, we can start adding more functionality. Let's go ahead and add the email service we're going to use. First, add nodemailer to your package.json and then import it.

```
import nodemailer from "nodemailer";
```

With this basic server in place, we can start adding more functionality. Let's go ahead and add the email service we're going to use. First, add nodemailer to your package.json and then import it.

```
1 // Set up email
2 const transport = nodemailer.createTransport({
3   host: process.env.EMAIL_HOST,
4   port: 587,
5   auth: {
6     user: process.env.EMAIL_USER,
7     pass: process.env.EMAIL_PASSWORD
8   }
9 });
10 // Make email template for magic link
11 const emailTemplate = ({ username, link }) => `
12 <h2>Hey ${username}</h2>
13 <p>Here's the login link you just requested:</p>
14 <p>${link}</p>
```



Next, we need to make our token that holds the user's info. This is just an example of some of the basic things you might include in a token. You could also include things like, user permissions, special access keys, and other information that might be used in your app.

```
1 // Generate token
2 const makeToken = (email) => {
3   const expirationDate = new Date();
4   expirationDate.setHours(new Date().getHours() + 1);
5   return jwt.sign({ email, expirationDate }, process.env.JWT_SECRET_KEY);
6   };
```

Now we can update the login endpoint to send a magic link to registered users and they'll be logged in to the app as soon as they click it.

```
1 // Login endpoint
2 app.post("/login", (req, res) => {
3   const { email } = req.body;
4   if (!email) {
5     res.status(404);
6     res.send({
7       message: "You didn't enter a valid email address.",
8     });
9   }
10  const token = makeToken(email);
11  const mailOptions = {
12    from: "You Know",
13    html: emailTemplate({
14      email,
15      link: `http://localhost:8080/account?token=${token}`,
16    }),
17    subject: "Your Magic Link",
18    to: email,
19  };
20  return transport.sendMail(mailOptions, (error) => {
21    if (error) {
22      res.status(404);
23      res.send("Can't send email.");
24    } else {
25      res.status(200);
26      res.send(`Magic link sent. : http://localhost:8080/account?token=${token}`);
27    }
28  });
29 });
```

There are only two more things we need to add to the code to get the server finished. Let's add an account endpoint. Then we'll add a simple authentication method.

```
// Get account information
app.get("/account", (req, res) => {
  isAuthenticated(req, res)
});
```



This gets the user's token from the front-end and calls the authentication function.

```
1  const isAuthenticated = (req, res) => {
2    const { token } = req.query
3    if (!token) {
4      res.status(403)
5      res.send("Can't verify user.")
6      return
7    }
8    let decoded
9    try {
10     decoded = jwt.verify(token, process.env.JWT_SECRET_KEY)
11   } catch {
12     res.status(403)
13     res.send("Invalid auth credentials.")
14     return;
15   }
16   if (!decoded.hasOwnProperty("email") || !decoded.hasOwnProperty("expirationDate")) {
17     res.status(403)
18     res.send("Invalid auth credentials.")
19     return;
20   }
21   const { expirationDate } = decoded
22   if (expirationDate < new Date()) {
23     res.status(403)
24     res.send("Token has expired.")
25     return;
26   }
27   res.status(200)
28   res.send("User has been validated.")
29 }
```

This authentication check gets the user's token from the URL query and tries to decode it with the secret that was used to create it. If that fails, it returns an error message to the front-end. If the token is successfully decoded, a few more checks occur and then the user is authenticated and has access to the app!

What if you could skip all of this work and get started with an end-to-end user management solution that already has the coding covered?

At Frontegg, we have already taken all of these requirements into consideration when building our self-served user management platform, with passwordless authentication enabled by default. If you have any questions and are not sure what implementing passwordless authentication should look like in your use case, feel free to reach out and get in touch with our experts. We are here to help.

[CONTACT US](#)