HACKERNOON

Testing in DevOps? [Take the survey]

How do you authenticate, mate?

October 21st 2017 TWEET THIS

A definitive guide for software developers to authentication

ıed
mim
san
ɛ_sha

We all know the common procedure for authenticating a user in our application. It’s the old school method of registering a user with his/her basic info, like- email, password etc. and then at the time of login, match the given email, password with the previously stored data. If matched then give him/her access, otherwise not.

But time has changed and a lot of other authentication methods have been introduced over time. To keep yourself valuable as a programmer/developer at this fast paced, ever changing software development world, you need to know about these new methods.

It's an undeniable fact that 'Authentication' is of the utmost importance in any type of application or system to keep safe the user data and proper access to information. To determine which authentication method is better for you would require the knowledge about the authentication methods, their trade-offs, and mostly how they works.

Here I'll try to introduce you with the most common and popular methods of authentication using these days. It won't be a detail technical guide for these methods but just an attempt to make you familiar with them. Though the following topics are discussed having the 'web' in mind, but the concepts are not limited to only that. These concepts and methods can be proved useful in other domains too.

I'm gonna continue my article assuming that you already know the fact that most of the web/internet—as we know it—is built upon the HTTP protocol. Also before proceeding, you should already know how a web application works, what does it mean by authenticating a user to the application and what is client-server architecture.

Ready for the journey? Let's dive.

Session based authentication:

Since the HTTP protocol is *stateless*, this means that if we authenticate a user with a username and password, then on the next request, our application won't know this is the same person from the previous request. We would have to authenticate again. At each request, HTTP doesn't know anything about what happened before, it just carries the request. So for any request for private data, you would have to log in again to make sure the application knows this is really you. This would be very annoying.

To avoid this problem, session/cookie based authentication was introduced. It makes the authentication process **'stateful'**, This means that an authentication record or session must be kept both server and client-side. The server needs to keep track of active sessions in a database or memory, while on the front-end a cookie is created that holds a session identifier, thus the name cookie based authentication. This one is the most common and widely known method of authentication which is being used for a long time.

Basic flow of session based authentication:

1. In the browser User enters his username and password and the request goes from the client application to the server.

2. Server checks for the user, authenticates it and sends a unique token to the user's client application. (also saves this unique token in memory or database)

3. The client application stores the token in cookies, and sends it back with each subsequent request.

4. The server receives every request that requires authentication and uses the token to authenticate the user and return the requested data back to the client application.

5. When someone logs out, the client application removes that token, and so that subsequent request to rails from the client becomes unauthorized.

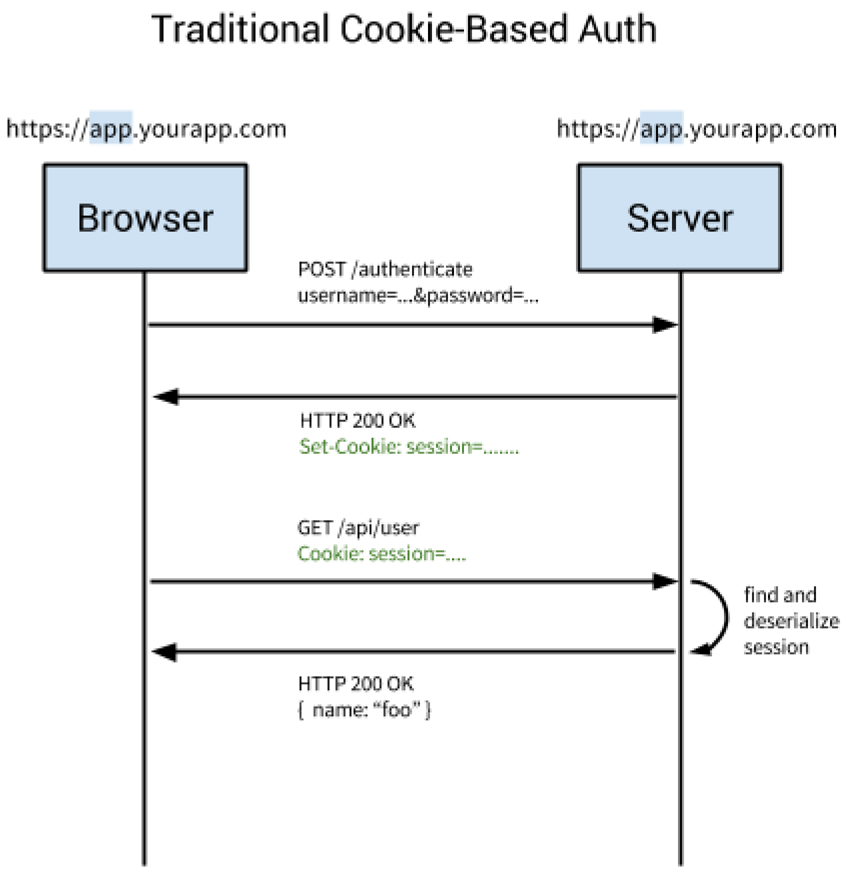


image source: auth0 blog

A few major problems arose with this method of authentication.

- Every time a user is authenticated, the server will need to create a record somewhere on the server. This is usually done in memory and when there are many users authenticating, the overhead on the server increases.

- Since sessions are stored in memory, this provides problems with scalability. If you replicate your server to multiple instances then you have to replicate all of the user sessions to all your servers., which complicates the scalability process. (Though it can be avoided by having a single dedicated server for session management but is not always feasible and easy to implement)

Token based authentication:

Token-based authentication has gained prevalence over the last few years due to rise of single page applications, web APIs, and the Internet of Things (IoT). Token used for *'Token based authentication'* is mostly Json Web Tokens(JWT). Though there are different implementations of tokens but the JWT have become the de-facto standard.

Token based authentication is **stateless**. We will not store any information about our user on the server or in a session, not even which JWTs have been issued to the clients.

Basic flow of token based authentication:

1. User enters their login credentials

2. Server verifies the credentials are correct and returns a signed token (the JWT) which can contain some additional information as a metadata like user_id, permissions etc.

3. This token is stored client-side, most commonly in local storage—but can be stored in session storage or a cookie as well

4. Subsequent requests to the server include this token generally as an additional Authorization header in form of *Bearer {JWT}*, but can additionally be sent in the body of a POST request or even as a query parameter.

5. The server decodes the JWT and if the token is valid, processes the request

6. Once a user logs out, the token is destroyed client-side, no interaction with the server is necessary

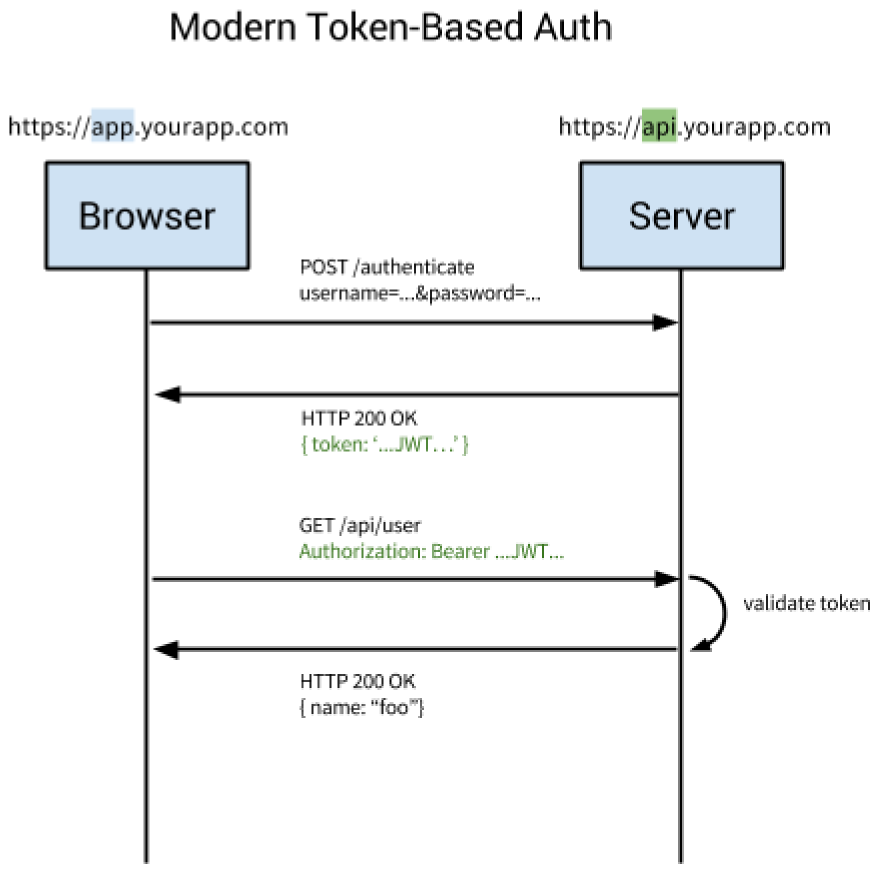


image source: auth0 blog

If you want a more detailed explanation, try here.

There are several benefits of this method:

- The biggest advantage of this method is that it is completely stateless as in the server doesn't need to store any record of the user tokens/sessions. Each token is self-contained, containing all the data required to check its validity as well as convey user information through claims. That's why it doesn't add any complexity in scalability.

- With a cookie based approach, you simply store the session id in a cookie. JWT's on the other hand allow you to store any type of metadata, as long as it's valid JSON.

- When using the cookie based authentication, the back-end has to do a lookup, whether that be a traditional SQL database or a NoSQL alternative, and the roundtrip is likely to take longer compared to decoding a token. Additionally, since you can store additional data inside the JWT, such as the user's permission level, you can save yourself additional lookup calls to get and process the requested data.
  For example, say you had an API resource /api/orders that retrieves the latest orders placed via your app, but only users with the role of admin have access to view this data. In a cookie based approach, once the request is made, you'd have one call to the database to verify that the session is valid, another to get the user data and verify that the user has the role of admin, and finally a third call to get the data. On the other hand, with a JWT approach, you can store the user

role in the JWT, so once the request is made and the JWT verified, you can make a single call to the database to retrieve the orders.

- While possible, there are many limitations and considerations to using cookies with mobile platforms. Tokens on the other hand are much easier to implement on both iOS and Android. Tokens are also easier to implement for Internet of Things applications and services that do not have a concept of a cookie store.

Because of these benefits and simplified approach, Token based authentication is on the rise of popularity these days.

Passwordless:

The first reaction to the term 'Passwordless authentication' could be — *'how could someone be authenticated without a password? Is it even possible?'*
That's because we have had it drilled into our heads that passwords are the ultimate source of protection for our account. But after digging up some information about it, you might realize that not only is Passwordless authentication safe to use, it might even be safer than a traditional username + password login. You also might have heard people who raised their voice trying to establish that Passwords are obsolete.

What is Passwordless?

As you might have already guessed that, *'Passwordless authentication'* is a way to configure login and authenticate users without a password. The general idea to implement a Passwordless authentication is following:

Instead of a user giving an email/username and password, they enter only their email address. Your application sends them a one-time-use link to that email, which the user clicks on to be automatically logged in to your website/application. In the case of a password-less login, the app assumes that you will get the login link from your inbox if the email provided is indeed yours.

There is a similar approach, which is instead of sending one-time-link to email, sending a code or one-time-password(OTP) through SMS. But you need to incorporate your application with a SMS service, like twilio, to make it work (also it will cost you money). Also, it’s good to know that code or One-time-passwords can be sent to email too.

Another less (yet) popular (and only available to apple devices) Passwordless process is to use Touch ID, which uses fingerprint for authentication. Here is a good read about it.

If you are using Slack, you might already get a taste of Passwordless login.

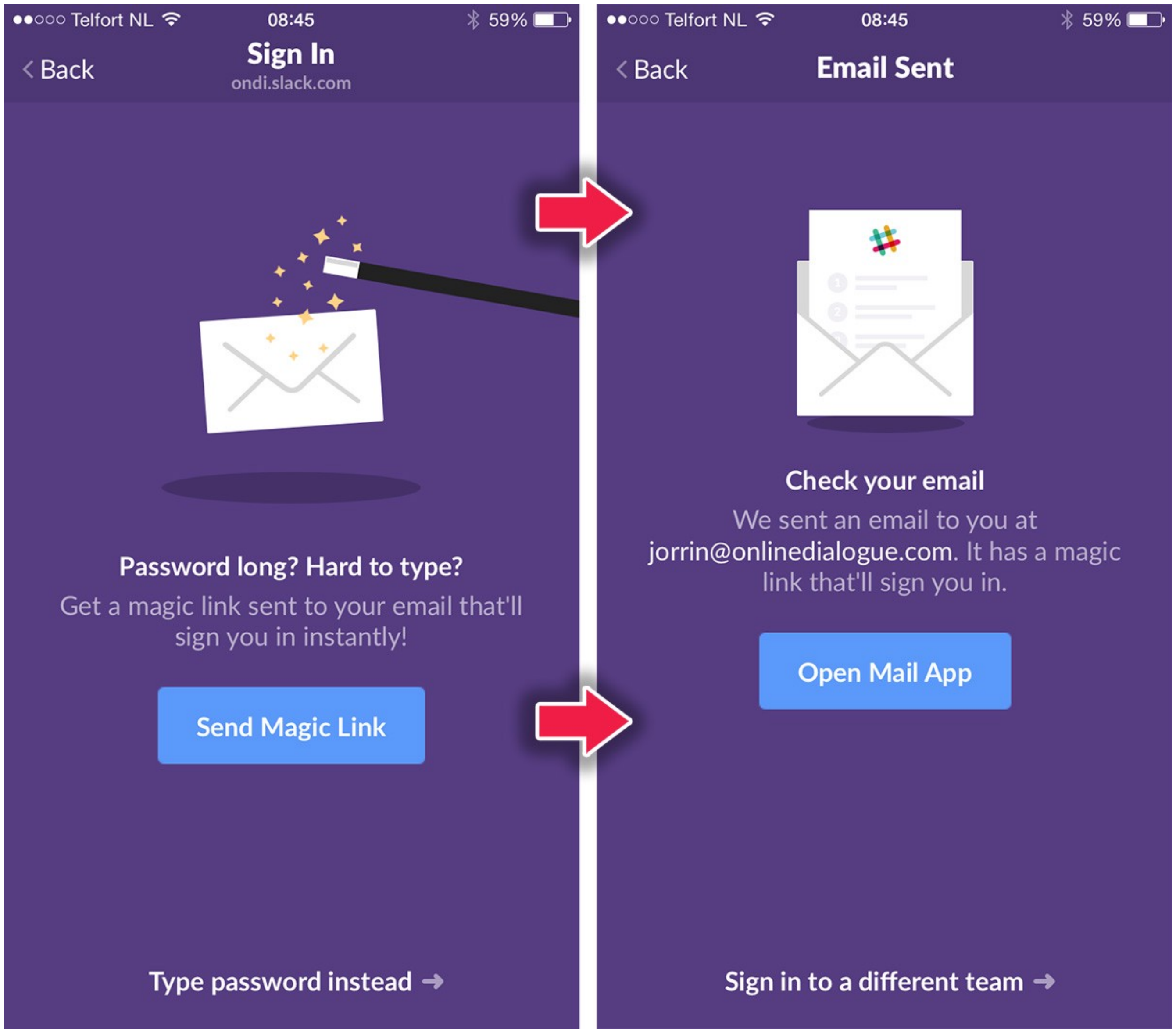

Telfort NL 08:45 59%
Back
Sign In
ondi.slack.com
Password long? Hard to type?
Get a magic link sent to your email that'll sign you in instantly!
Send Magic Link
Type password instead
Telfort NL 08:45 59%
Back
Email Sent
Check your email
We sent an email to you at jorrin@onlinedialogue.com. It has a magic link that'll sign you in.
Open Mail App
Sign in to a different team

image source: auth0 blog

Also Medium is giving user's access to their site using only Email. I recently found that, Auth0 or Facebook's AccountKit could be a very good option if you want to implement a Passwordless system in your application.

What could go wrong with Passwordless?

If someone has access to the user's email account, they would have access to their account in your website/application too. Well, don't worry about that. Securing the user's email account is not our(as a developer) responsibility at all. Also, if someone gains access to one's email account they can exploit their 'password-based authenticated' applications too using 'reset-password' feature. So, let's move on as we have nothing to do about it.

What's the benefit?

Before answering that I ask you to just think about how frequent you use 'forgot password' to reset your passwords, not mentioning the several failed attempt before that, to login a site/application just because you can't remember the goddamn password? Well, that's not only the case for you my friend. We all are in the same boat. It's no wonder that remembering password is hard, specially if you are concerned with your account safety and set different password in every site (following the *at least 'a number', 'a capital letter', 'a symbol', 'minimum 8 character length'* rule). Using a Passwordless authentication will save you from this headache. (I know, now you might thinking, *'I use password manager you idiot'*. Respect to you. But remember, mass users of your application are not tech-savvy like you. You need to consider that.)

Not only for users, as a developer it is good for you too. Now you don't need to implement a 'forgot password', 'reset password' flow. This is a win-win for all. Cheers!

If you really think that some of the users might still want to use that age old email-password login, then you should give both of the options like Slack did, so the users could opt-in.

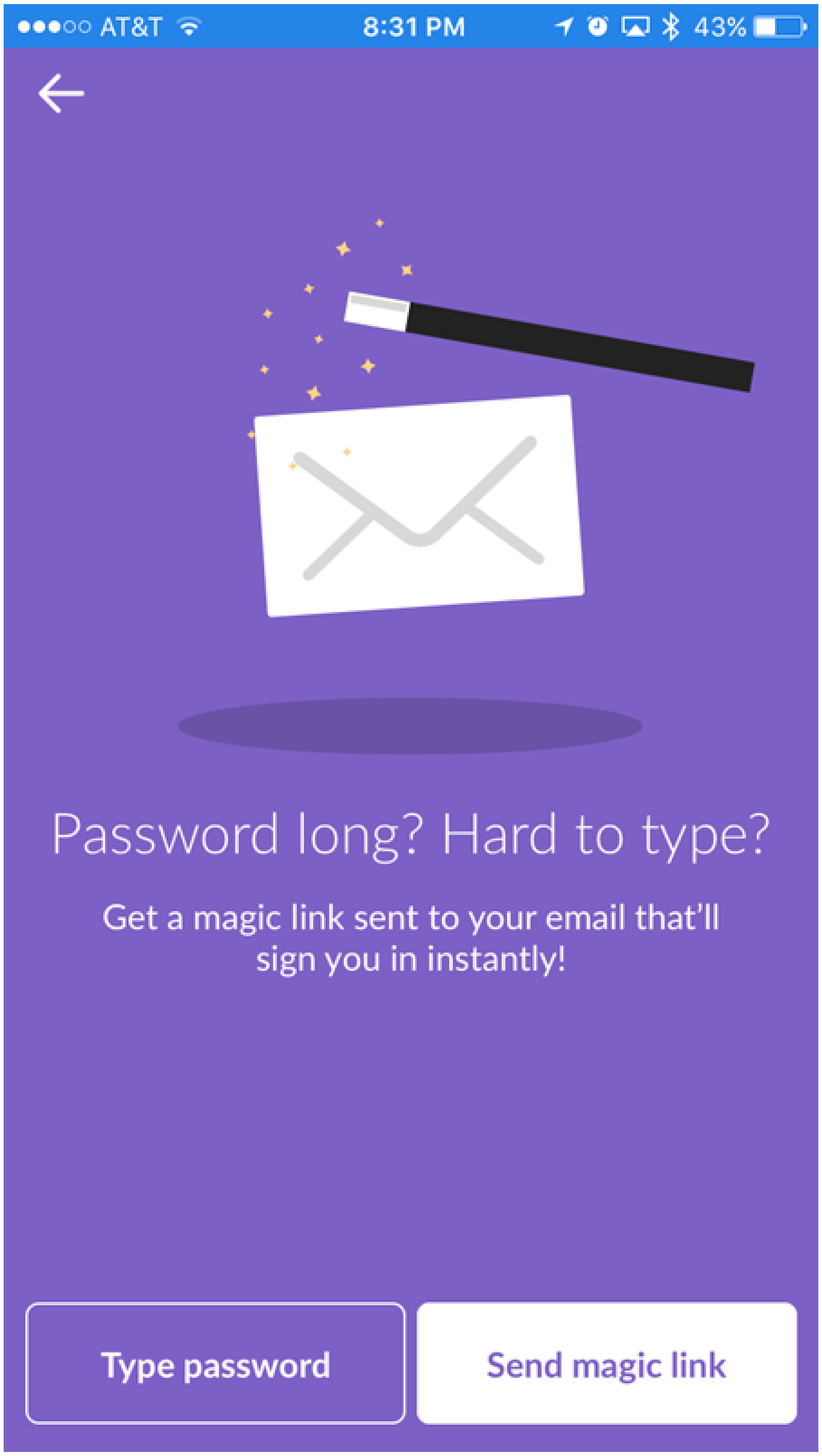

●●●○○ AT&T
8:31 PM
43%
Password long? Hard to type?
Get a magic link sent to your email that'll sign you in instantly!
Type password
Send magic link

image source: smashingmagazine

Needless to say that Passwordless is becoming an increasingly relevant option for login and gaining popularity these days.

Single Sign On (SSO):

Did you notice that if you login to any of the google services like your gmail account from your browser, and then you go to youtube or any other google based service, you don’t need to separately login for that service? You automagically gain access to all of the google services. Fascinating, isn’t it? Though gmail and youtube are both products of google, but they are separate products, right? So, how do they authenticate a user after single login to all of their products?

This method is known as Single Sign On (SSO).

Single sign on can be achieved in many ways. One of them is to makes use of a Central Service which orchestrates the single sign on between multiple clients. In the example of Google, this central service is Google Accounts. When a user first logs in, Google Accounts creates a cookie, which persists with the user as they navigate to other Google-owned services. The process flow is as follows:

1. The user accesses the first Google product.

2. The user receives a Google Accounts-generated cookie.

3. The user navigates to another Google product.

4. The user is redirected again to Google Accounts.

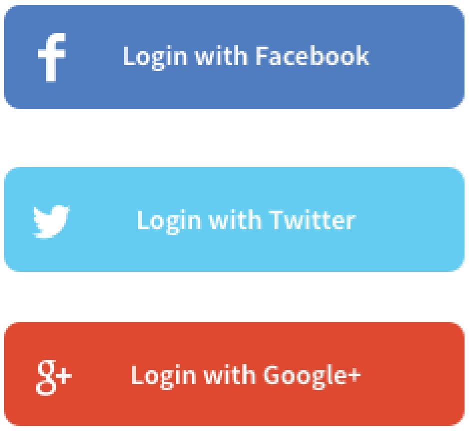
5. Google Accounts sees that the user already has an authentication-related cookie, so it redirects the user to the requested product.

Single Sign On (SSO) can be described very simply as “user logs in once and gains access to all systems without being prompted to log in again at each of them”. This boils down to three different entities who trust each other directly and indirectly. A **user** enters a password (or some other authentication method) to their **identity provider (IDP)** in order to gain access to a **service provider (SP)**. User trusts IdP, SP trusts IDP so SP can in-turn trust user.

This seems so simple, but custom implementations of this would be very complicated. A details explanation on how SSO works can be found here.

Social Sign-in:

I bet the following image is pretty familiar as you frequently see it in most of the sites, right?



This is the thing which is famously known as **‘Social sign-in’** or **‘Social Login’**. Using this you can authenticate a user based on their social networking accounts. Users don’t need to register separately in your application.

Social sign-in or Social login is not technically a different authentication method. Rather it’s a form of single-sign-on which simplifies the registration/login process of a user to your application and that’s why you should know about it (as a developer).

Social login is best of the both worlds. Why?
First of all, for the users, the login to your application is just one click away as they can use their existing social network account and don’t need to remember username, password for services like your application. This results in a rich user experience. Also as a developer you don’t need to worry about securing user’s authentication credentials and also you are ensured that user’s email address is already verified (by the social service providers). Another bonus point, social provider will also handle the password recovery process. Yay!

How do I do it then, huh?
As a developer you need to know a little bit more about the underlying process. Most of the social providers use OAuth2 (some uses OAuth1, e.g. twitter) for authorization as well as authentication mechanism behind the scene. The major key points to understand in OAuth is that, the social provider is the **‘resource server’**, your application is the **‘client’** and the user trying to login to your application is the **‘resource owner’**, because the key ‘resource’ here is the user’s profile/authentication information. So, when the user wants to login to your application using the social provider your application will redirect them to the social provider for authentication (generally a popup window opens with the social providers URL). Along with the successful authentication, the user needs to approve your

application's permission to access the user's profile information from social provider. Then the social provider will redirect back the user to your application with some access token. Next time using that access token your application can ask the social provider about the user's profile information. This is how OAuth works in a nutshell (skipping some technical details for easy clarification)

To implement social login in your application you might need to register your application in the social providers site which will give you some app_id and other related keys for configuration to communicate them. You will get these information in the respective social provider's site. Also there are several popular library/packages (like Passport, Laravel Socialite etc.) which might simplifies the process for you and release you from the burden to know the nitty gritty details.

Two-factor authentication (2FA):

Two-factor authentication (2FA) strengthens access security by requiring two methods (also referred to as **factors**) to verify a user's identity. It is a type of multi-factor authentication which provides an extra layer of security. You might not realized before but when you go to an ATM booth or Cash machine to withdraw money, you are being authenticated by a Two-factor authentication system. You must have the correct combination of the bank card(*something you possesses*) and the PIN(*something you know*) to be authenticated. If someone steals your ATM card, it is of not much use until they also know your PIN. That means in a Two-factor authentication system a user is granted access only after successfully presenting several separate pieces of evidence to an authentication mechanism.
Another example you might be familiar with is the 2-step verification of

Google, facebook etc. Where after enabling 2 factor authentication in your account, every time you need to login to your account, first you provide the login credentials, such as email, password(verify that you know the credentials) and then a one-time password(OTP) is sent to you through SMS(verify that you possesses the device), and you have to enter it correctly to complete your login process. If your password has been compromised, your account is still safe, as the attacker cannot complete the second step without having the verification code.

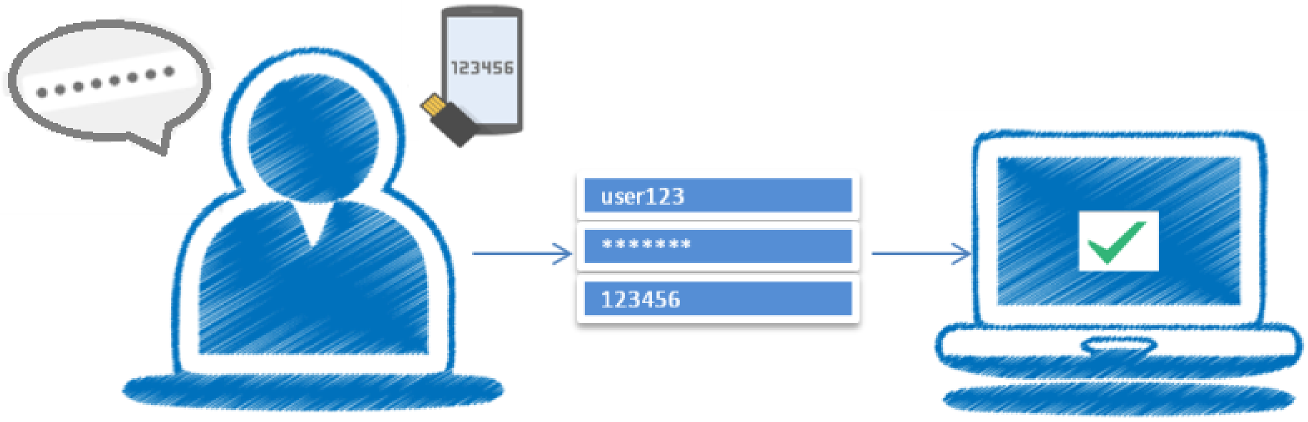


image source: https://dzone.com/articles/implementing-two-factor-authentication-using-authe

Instead of OTP, another common method is to use the user's biometric data such as fingerprints or retina as a second factor.

Two factor authentication is based on the user providing two of the following three *"somethings"*:

- **Something you Know**—the password or pin for an account
- **Something you Have**—a physical device such as a mobile phone or a software application that can generate one-time passwords

- **Something you Are**—a biologically unique feature to you such as your fingerprints, voice or retinas

Learning the password or pin for an account is what most hackers go after. Accessing a physical token generator or getting biological features is harder and the reason why 2FA is effective in providing greater security for user accounts.

So, Is 2FA the one-size-fits-all solution? Maybe Not.

But, still it will help you to strengthen the authentication security in your application. How would you implement a 2FA solution in your system? Well, it might be better to use some existing solution like Auth0 or Duo rather than rolling your own.

Bonus Topic: Authentication vs Authorization:

Some of us might mistakenly use the terms 'authentication' and 'authorization' interchangeably. But these two terms doesn't mean the same thing.

- **Authentication** is the process of verifying who you are. When you log on to an application with a username and password you are authenticating.

- **Authorization** is the process of verifying that you have access to something. That means the set of permissions which you are allowed to do. As an example, if you created a resource in an application you might be the only person allowed to remove it (as an owner), other users are not ‘authorized’ to remove this resource.

Are you still here?

Congratulations, you’ve successfully completed reading a long, tedious, boring article. □

Hope you get a brief overview about the topics. If you find any mistakes or think any improvements needed in this article please leave a comment.

Authentication # Software Development # Programming # How Do You Authenticate # Software

Continue the discussion

More by Ahmed shamim hassan

Observer vs Pub-Sub pattern

Ahmed shamim hassan

# Software Development

Open office environment should be like ‘Library’, not ‘Cafeteria’

Ahmed shamim hassan

# Work

Probabilistic Data structures: Bloom filter

Ahmed shamim hassan

# Data Structures

Webhook for you and me

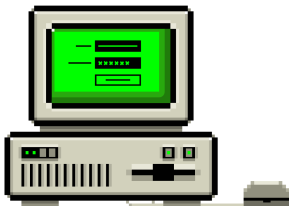
Ahmed shamim hassan

# Webhooks

What every programmer should know about ‘String’

Ahmed shamim hassan

# Programming



Hackernoon Newsletter curates great stories by real tech professionals

Get solid gold sent to your inbox. Every week!

Email Address *

First Name

Last Name

TOPICS OF INTEREST

✓ Software Development

✓ Blockchain Crypto

✓ General Tech

✓ Best of Hacker Noon

Get great stories by email

More Related Stories

A Brief History in Authentication

Karan Shah

# Security

Desktop apps for Front-end development. My workspace.

Alexander Buzin
Feb 15

# Web Development

0–100 in Django: Starting an app the right way

Jeremy Spencer
Sep 04

# Python

9 Product Design Themes That Will Improve Your Product’s User Adoption,

Your Team's Design Process...

Poornima Vijayashanker
May 23

# Design

Help
About
Start Writing
Sponsor:
Brand-as-Author
Sitewide Billboard

Contact Us
Privacy
Terms