

The Ultimate Guide to Outsourcing Your Auth



The Ultimate Guide to Outsourcing Your Auth

Edited by Dan Moore

This book is for sale at

<http://leanpub.com/theultimateguidetooutsourcingyourauth>

This version was published on 2021-07-30



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 Edited by Dan Moore

Contents

| | |
|---|-----------|
| What Is an Auth System | 1 |
| Authentication | 1 |
| Authorization | 2 |
| User Management | 3 |
| Categories of Auth Functionality | 3 |
| Conclusion | 7 |
| Evaluation | 8 |
| Why Outsource Your Auth System | 9 |
| Speed To Market | 10 |
| Consequences Of an Auth Breach | 10 |
| Consequences Of an Auth Outage | 11 |
| Maintainability | 12 |
| Cost of In-House vs Outsourced Auth | 13 |
| Auth May Be Unrelated to Your Core Competency | 14 |
| Getting Buy-In For Outsourcing Auth | 15 |
| Your Boss | 16 |
| Your Developers | 16 |
| Project Management | 17 |
| Product Management | 18 |
| Legal | 18 |
| Quality Assurance | 19 |
| UX and Design | 20 |
| Security | 20 |

CONTENTS

| | |
|--|-----------|
| Infrastructure | 20 |
| Opportunity Cost | 21 |
| Performing Due Diligence on Authentication Vendors . . | 22 |
| Examining the Authentication Provider's Security Stan- dards | 23 |
| Measuring Performance With Benchmarks | 25 |
| Engineering Effort to Implement Authentication | 27 |
| Pricing | 28 |
| Conclusion | 29 |
| The Value of Standards-Compliant Authentication | 31 |
| Why Use a Standardized Authentication Protocol? | 31 |
| Survey Of Authentication Standards | 34 |
| Conclusion | 40 |
| Open Source Vs Commercial Auth Providers | 41 |
| Open-Source Authentication Providers | 41 |
| Commercial Authentication Providers | 46 |
| The Final Showdown | 49 |
| The Value of Trying Your Auth Provider Before You Commit | 51 |
| Advantages of Trying an Auth Provider Before You Buy | 52 |
| Running an Effective Trial | 55 |
| Free Trials Are Important | 56 |
| Risks | 58 |
| Common Authentication Implementation Risks and How to Mitigate Them | 59 |
| Security and Privacy | 60 |
| Performance | 61 |
| Regulatory Compliance | 62 |
| Cost and Time | 63 |
| Integration and Features | 64 |

CONTENTS

| | |
|--|------------|
| Vendor Assessment | 64 |
| Conclusion | 66 |
| Avoiding Authentication System Lock-in | 67 |
| Look For Open Standards | 68 |
| Consider Portability | 68 |
| Limit Your Usage Where You Can | 69 |
| Insulate Your Application | 70 |
| Have a Backup Plan | 72 |
| Wrapping Up | 73 |
| What To Do When Your Auth System Vendor Gets Acquired | 74 |
| Short-Term Concerns | 75 |
| Short-Term Benefits | 76 |
| Long-Term Concerns | 76 |
| Long-Term Benefits | 78 |
| Mitigating Concerns | 78 |
| Conclusion | 81 |
| Implementation | 82 |
| Multi-Tenant Vs Single-Tenant IDaaS Solutions | 83 |
| Multi-Tenant | 84 |
| Single-Tenant | 86 |
| Multi-Tenant Within a Single-Tenant Solution | 88 |
| Private Labeled Identity | 88 |
| Dev, Stage and Prod | 91 |
| Conclusion | 93 |
| Making Sure Your Auth System Can Scale | 94 |
| Why Scaling Auth Is Hard | 95 |
| How Can You Effectively Scale Your Authentication? | 99 |
| Conclusion | 107 |
| When to Self-Host Critical Application Architecture | 108 |
| Evolution Of Self-Hosting | 109 |

CONTENTS

| | |
|--|------------|
| The Benefits Of Self-Hosting | 110 |
| The Downsides Of Self-Hosting | 113 |
| Legal Considerations | 116 |
| A Non-Permanent Choice | 116 |
| Conclusion | 117 |
| Migration of Auth Data | 118 |
| Why Migrate User Data | 118 |
| Types Of Migration | 119 |
| Planning and Mapping | 125 |
| Big Bang Implementation | 127 |
| Segment By Segment Implementation | 134 |
| Slow Migration Implementation | 135 |
| Conclusion | 143 |
| Best Practices for Registration Forms | 145 |
| Is a Registration Form Needed At All? | 146 |
| Ease the Pain Of Registration | 148 |
| Multi-Step Registration | 151 |
| Final Thoughts | 155 |

What Is an Auth System

Almost every software application has users. In the online world, applications may have one or few users, if internal, or billions of users, if Facebook (hi Zuck!). Most applications customize functionality and the user interface for each user because of business rules or customer expectations.

Users are a fundamental entity. If you are building an application, you'll almost certainly have users. At [FusionAuth¹](https://fusionauth.io), we've built a flexible auth system that makes managing users easy and secure.

Auth systems are referred to as identity and access management (IAM) or customer identity and access management (CIAM) as well. In general, IAM systems are used to manage employees and CIAM systems are used to manage customers, possibly in addition to employees.

We talk to people about their auth systems every day and have helped plenty of folks migrate to FusionAuth from other systems, whether commercial, open source, or homegrown. While modern auth systems can be quite complex, three pieces of functionality appear in almost every one: authentication, authorization and user management. Let's look at each of these pieces.

Authentication

Authentication is how an application knows who a user is, and is sometimes referred to as AuthN. In online applications, this is typically done by presenting a set of credentials. Username and

¹<https://fusionauth.io>

password are common credentials. A system may require additional factors of authentication, such as a code the user has been texted, a time based one time password (TOTP), or possession of a physical object like a Yubikey.

At the end of an authentication process, the application should know who the user is.

Authorization

Authorization controls user access to resources, whatever they may be, and is sometimes referred to as AuthZ. These could be API endpoints, functionality in an application or anything else. This is a logical next step after knowing who the user is, as most applications have different levels of access.

Consider the example of a blog. Some users may be administrators, able to change configuration settings. Others may be editors, able to approve and publish articles. Yet other users may be writers, unable to publish, but who can log in, add content, and submit it for approval. Each action (log in, change config settings, publish articles, etc), has a permission associated with it.

The auth system must have some way of mapping users to these permissions. This can happen directly, with permissions being attached to the user, or indirectly, with users assigned to a group or role, and that group or role having permissions. Members of a group or role then inherit permissions.

These permissions must be shared with the application, which can then allow or prohibit any action. If I am a writer for a blog, I should not see the 'publish' button. When an action is requested, you should always check permissions in the back end of your application, since the front end may have been tampered with.

User Management

This is an administrative function. Once an application has users, a nice interface to manage them helps people supporting those users. This interface should be both user facing as well as an API. A smart looking user interface will allow customer support and other non-technical folks to find users' information and fulfill user requests, such as profile changes.

An API will allow efficient bulk operations. Do you want to know how many users logged in during the last month? This is the kind of question that an API will let you answer.

Now that we've examined key pieces of functionality in an auth system, let's examine four ways a developer might handle functionality supporting users.

Categories of Auth Functionality

Like anything else, there's a spectrum of complexity for user management and auth, ranging from the non-existent to scalable and complicated solutions.

Who Are You?

Maybe you aren't building a complex application. Maybe you are building an app where all functionality will either be static or available to everyone. An example of this might be a directory or a calculator application.

In this case, you get to avoid the problem of users.

You can stop reading now!

Not Invented Here Syndrome

We see this solution a lot. I have done this myself. The application might start out with a users table. Engineers implement a hashing algorithm to protect passwords, a login form, and ship the code. They have other features to write.

A home grown system seems simple at first, but then the engineering team needs to build out more and more of the auth flows users expect. This can include “forgot password”, “self service registration”, and more complicated integrations such as social sign-ons.

At FusionAuth we’ve cataloged the following user auth flows:

- Log in
- Log out
- Registration
- Forgot password
- Email verification
- Changing a password
- Two factor authentication
- Passwordless authentication
- Single sign on
- Password expired
- Account lockout by administrative decision or failed attempts
- Password validation failed
- Breach password detection
- Federated log in with IdPs such as Google and Microsoft Active Directory
- Advanced self service registration forms
- Multi application logout (OAuth front channel logout)

While users may not need all of these initially, as time goes on, more and more will be required.

Eventually, events typically happen that make teams wish they hadn't rolled their own auth system. Sometimes it is engineering time spent building auth features; other times it is something more traumatic like a data breach due to an oversight or bug.

Avoid building homegrown auth systems if at all possible. An auth system is similar to a database. Yes, there may be a valid use case for building your own database, but 99.99% of systems will be better served integrating an existing one.

Many Eyes Make For Shallow Bugs

Many modern frameworks have built in auth systems. For example, Wordpress and Django have user auth systems included. There are also open source libraries which can be pulled into an application. For example Spring Java applications can use Spring Security, Ruby on Rails applications can use Devise and Omniauth, and Express applications can use Passport.js.

These solutions are great. They are battle tested by many users. They are often open source so teams can contribute code or patch issues they run into. Whatever framework or language is being used, research this type of option.

Where these built in solutions often fall down is in a heterogeneous environment. Oftentimes these solutions are tightly coupled to the language or ideas of the environment they exist in. But when you get to a certain number of applications, you may want to extract authentication and authorization to a central, focused auth system.

This may happen sooner than you think. For example, if building a SaaS todo application, there may also be a forum for community support, GSuite for employees and a support ticket system for paying customers. All these applications have users that need to be authenticated, authorized and managed. That's in addition to the user model for the todo application.

While one could bolt each application into the todo application's user database, the system will have more flexibility if a standalone auth system is used.

No App Is an Island

Just as a database is good at storing structured data, standalone auth systems are good at authentication, authorization and user management. These systems implement various standards, can be secured and scaled separately, and are a quick way to provide much of the functionality your users need.

There are many of these systems available, and they all solve the fundamental problems in slightly different ways. Dimensions to consider when you are evaluating a standalone auth system include:

- Self hosted or SaaS
- Authentication, authorization, user management or all three?
- Specialized functionality
- Standards support
- Single sign-on support
- Open source or commercial
- Integrations with other auth tech (such as LDAP)
- Which OAuth grants are supported
- Cost
- Operational complexity/support for your deployment environment
- Specific features if needed (for example, customization of the look and feel with themes, or customization of the login flow)
- Documentation and developer experience

Conclusion

Almost every application will have some concept of users. Your needs can range from “I don’t want any users at all” to “I need a standalone auth system”. Being aware of the options, as well as knowing how to avoid a homegrown system, will help you build more secure applications faster.

Now that you know what an auth system encompasses, let’s look at some other aspects. This book has three sections:

- Evaluation: what you need to know when you are looking at auth systems, including whether to outsource this critical functionality to a third party, the value of trying providers before you choose one, and other critical issues.
- Risks: risks with an auth system and how can you mitigate them.
- Implementation: more specific implementation guidance, including migration, scaling and more.

This book features a variety of perspectives on these topics. Enjoy!

Evaluation

It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.

– Isaac Asimov

You should spend a fair amount of time evaluating an auth system. As mentioned previously, this is an architectural choice akin to choosing a database or datastore. Auth systems can help accelerate development, provide security for crucial PII, and help your application succeed, but they get embedded in your applications. It's worth taking some time to ensure whatever you choose is a good fit for your needs.

Like any architectural choice, picking an auth system is about understanding tradeoffs and planning for the future. In this section, you'll learn:

- Why you might choose outsource your auth
- How to sell that choice to your organization
- What due diligence you should perform on vendors
- The value of standards and standard compliant implementations
- How to compare open source versus commercial solutions
- Why you should do a proof of concept (POC) before committing

Let's dive in.

Why Outsource Your Auth System

By Joe Stech

You're a software engineering leader, and you're great at your job. You know that the optimal path for software development lies in figuring out which components of your design to implement from scratch and which have already been implemented by specialists and can be reused.

You also know that these aren't decisions that you can only make once – you have to keep reevaluating based on environment changes and the needs of new products.

Authentication is one of those components that you deal with all the time. Auth is a necessary part of any software product, but how you implement auth is not necessarily always the same. Careful consideration is needed, because your decision to outsource will not only impact speed of development, but also long-term product maintenance – you don't want to slow down time to market because you re-implemented an entire auth system unnecessarily, but you also don't want to use an auth system that is going to cause problems down the road.

What are the primary considerations when making an outsourcing decision, especially around a component as critical as your identity management system? This document is a blueprint for both what you should consider as well as who to get on board if you decide using a third party auth provider makes sense.

Speed To Market

This is the most obvious consideration. Depending on the features you need, it could literally take months to implement auth in-house, whereas it could take less than a day to incorporate an outsourced solution.

You could say “but what if we only need a bare-bones implementation? Some salted hashes in a database and we’re good to go!” That’s a totally valid point, and if you don’t anticipate needing sophisticated auth features then your best bet might be to do a quick in-house implementation and move on.

However, time and time again I’ve seen product developers underestimate the sophistication of features that will be required when their userbases grow. Most of the time development organizations then fall prey to the sunk costs fallacy and double down on augmenting their in-house solution, even when it may be more efficient to abandon the home-grown effort and replace it with an outsourced solution. This will cause huge issues for maintainability, which I’ll talk about further below.

Consequences Of an Auth Breach

Planning for the worst possible case can prevent total financial ruin for your company or division. If a breach of security happens and PII (Personally Identifiable Information) is leaked from your in-house auth system, it can not only cause your company reputational damage but also significant financial penalties (not to mention potential jail time if you try to cover up the breach).

If you outsource your auth system you can limit your liability, and also protect your reputation – if there is a breach on your auth provider’s side, it’s likely that the breach will extend beyond your company. A breach in an outsourced auth provider that is used by

many different companies will be big news, and customers will be more likely to forgive you for making a mistake in your choice of auth provider than for implementing a poor auth system yourself.

A not-insignificant addendum is that I believe your in-house system is much more likely to suffer a breach than an outsourced provider who is an expert in security. I have no studies to support this claim, but I have never seen a major auth provider compromised, and I've definitely seen companies suffer breaches due to their own in-house auth implementations – [this article discusses a compilation of 21 million plaintext passwords collected from various breaches²](#) wherein passwords were not hashed and salted by auth systems.

Properly storing passwords is an incredibly low bar, and yet companies that manage their own auth still do it incorrectly all the time.

Consequences Of an Auth Outage

While less damaging than breaches, outages can still cause reputational damage and liability issues if your SLAs make uptime guarantees. Similarly to breaches, if you outsource your auth system it's likely that any auth outage will extend beyond your company.

As an example, when Microsoft's Azure Active Directory (AAD) [went down for a good portion of the afternoon late last year³](#), logins for applications across the internet stopped working.

When your competitors' authorization systems are down at the same time yours are, nobody blames you for it, but when you're the only company having issues, you suffer reputationally. No matter what your outsourced auth system is (FusionAuth, Cognito, AAD,

²<https://arstechnica.com/information-technology/2019/01/hacked-and-dumped-online-773-million-records-with-plaintext-passwords/>

³<https://www.zdnet.com/article/microsofts-azure-ad-authentication-outage-what-went-wrong/>

etc), you can be almost certain that you won't be alone in the event of an outage.

Maintainability

There are trade-offs here. The benefits to an in-house system include:

- Your engineers can design the exact system to fit your needs, and you'll have unrestricted ability to add very specific features if requirements change over time.
- If the same engineers who built the system are maintaining it, then they'll have the context required to anticipate issues as they add features.

However, the drawbacks can be large:

- Complex new features can take significant time to build. Outsourced auth systems likely have these features already built (things like multi-factor authentication, user management interfaces, analytics and audit logs, and brute force hack detection, among others).
- With an in-house solution, you'll have to budget time to monitor new security threats and patch your system in a rapidly evolving threat landscape.
- You won't get the benefits of a dedicated team that are constantly improving your auth system. This is actually a bigger deal than it seems, because if you outsource auth then other companies will also be filing issue reports and feature requests on your behalf, so you reap the benefits of those extra eyes as well.

- In the case of mergers or acquisitions, an in-house solution is likely to be terrible at combining different databases of users and managing things like duplicates or incomplete data. Such enterprise identity unification efforts can founder on internal auth systems. Third party solutions often support modeling different user bases with tenants.

Cost of In-House vs Outsourced Auth

When building an in-house auth system your costs are all ostensibly sunk (engineer salaries). However, if building your system in-house delays time to market or prevents creation of other features, the build could cost you a significant amount of real income. There will also be on-going maintenance costs with an in-house solution.

When making cost calculations, you should compare:

Revenue lost by slower time-to-market PLUS engineering cost to implement in-house solution PLUS on-going maintenance costs of in-house solution PLUS increased risk of data breach PLUS increased risk of outage PLUS increased risk during a merger or acquisition

vs

The monthly cost incurred by an outsourced provider PLUS the lack of complete control

When evaluating different auth providers, you'll also want to consider whether an outsourced provider charges on a sliding scale based on number of users or if the cost is fixed. AWS Cognito, for instance, will continue to charge more as your application gains more users. FusionAuth, in contrast, has options that charge a single rate for unlimited users. If your app is small and you don't expect it to grow much, a sliding scale may be better for you. If you don't want a large unexpected bill as you gain more users, a provider that allows for fixed costs may be more appealing.

Auth May Be Unrelated to Your Core Competency

As a final consideration, you may want to evaluate if your engineers even have the ability to implement a secure in-house auth system without a significant investment in education. This is something that many leaders dismiss, since they have great faith in the intelligence and skill of their people.

However, knowledge and intelligence aren't the same, and just because your engineers are capable of becoming auth experts doesn't mean you want them to spend the time to do so.

As an engineering leader, you have a responsibility to ensure that your engineers are spending their time on efforts that will optimally contribute to the long-term success of your organization. Auth is a necessary component, but is it really a differentiator for your application?

Joe Stech is a former director of software engineering who has designed and implemented software for some of the largest (and smallest) software companies in the world. He is currently a consultant specialized in cloud software and data engineering. He writes about cloud topics at cloudconsultant.dev.

Getting Buy-In For Outsourcing Auth

By Joe Stech

If you've decided that outsourcing your auth system makes sense, how do you go about talking to all the relevant stakeholders about this choice?

Not all orgs are the same, but many companies have similarities. I'm going to assume you are an engineering leader, possibly the only engineering leader. Who to talk to and what to emphasize will be contingent upon the size and structure of your company, so you'll have to adapt this advice to your own circumstances.

However, I'm going to go through the main categories of stakeholders you might have, and suggest ways to talk to them about outsourcing your auth system. Even though you've convinced yourself that you've found the best course of action for your company, everyone else has to believe it too (or at the very least not actively oppose it).

You'll have to talk to each type of stakeholder on their own terms in order to achieve the best outcome for your company. For each person in each type of role you'll want to emphasize what the benefits are to them and how it's going to help the company as a whole.

Each of the following sections addresses a different stakeholder. If you've been in your current role for any length of time, you'll know which roles exist in your org. If not, your boss can help. For each stakeholder type, I'll make suggestions on how to help them see what you see.

Your Boss

Unless you're the CEO, you're going to have a boss. Your boss will likely be a software engineering director or VP. They will be responsible for delivering against the product roadmap in the most efficient, maintainable way possible. In many ways your boss should be perfectly aligned with you – ideally your concerns are a more detailed subset of theirs.

With that in mind, emphasize a few points:

- **Speed to market:** Depending on the features you need, it could literally take months to implement auth in-house, whereas it could take less than a day to incorporate an outsourced solution.
- **Cost:** There are a few things to consider here, which are covered in the “Cost of in-house vs outsourced auth” section of the “Why Outsource Your Auth System?” chapter.
- **Mitigating risk:** Two of the scariest things to an engineering leader are the risk of implementation time ballooning out of control and the risk of serious security breaches. Outsourced auth can help mitigate both of those concerns. Here, talking through the benefits in detail will be very helpful.

Your Developers

If you lead a team of developers, it's crucial that they don't oppose the decision to use third-party auth. **I can't emphasize this enough.**

If you convince every other stakeholder but fail to convince your devs, many of the benefits of outsourcing auth will be severely reduced. More importantly, if you try to ram through your agenda

without adequately conveying your vision, you can lose hard-earned trust.

This is the ‘lead’ part of ‘leadership’ – in this instance your job is not to dictate, it is to motivate and guide. Your team is smart, and if plugging in an external auth component is the most appropriate way forward for your specific use case you’ll likely be able to forge consensus.

If not, take a long hard look at your situation and ask yourself if you need to reconsider based on new points of view! What have you missed that they have seen? All of your reasoning should be laid before the team, but in particular emphasize:

- **The minimal maintenance required by an outsourced system:** The vendor will be continually improving the system so you don’t have to.
- **Support from the vendor:** Documentation, engineering support, and best practices should all easily be obtainable from your vendor of choice, which will greatly speed implementation and mitigate long-term risk.
- **The ability to work on more interesting features:** With the implementation time savings developers will be able to work on more interesting core features, which will also likely give them more visibility and resume-building achievements than re-implementing auth for the Nth time.
- **Stress reduction:** Having a vendor be responsible for some of the high-security aspects of the product will help reduce developer stress.

Project Management

I don’t want to over-simplify the role of project management, but in general they’re going to care the most about a stable, well-thought-out implementation schedule that can be acted on in a

consistent, predictable manner. Does that sound like most software development projects in your organization or that you've seen in your career? Maybe not! All the more reason to make choices that help your team get a little closer to that ideal.

Thus, when talking to project management about outsourcing your auth system it's important to emphasize the benefits of **speed of implementation** and **reduction of scheduling risk**. With pre-built components and vendor support it's unlikely that you'll be blowing up your schedule with this choice.

Product Management

They probably don't care about implementation details of your auth system, frankly. However, it can be helpful to convey your reasoning to them. Emphasize to them that outsourcing auth will allow the team to **work on new features more quickly**. This can help you even if product doesn't have a direct say in implementation decisions, since by addressing their main concerns, they'll likely be your ally in discussions with other decision makers.

Product management will probably also appreciate the fact that outsourced auth has a large feature set out of the box – you can point out all the functionality that your proposed auth system already has for which they won't need to draft requirements. This conversation might even remind product management of things they need to ensure have been specified in the main product, like internationalization and localization.

Legal

I feel like this one is pretty clear: **emphasize risk mitigation**. Both outages and auth breaches can come with significant legal

ramifications, especially if you have stringent SLAs in place for your product.

Legal will likely appreciate the fact that you want to use a tried and tested solution rather than building something brand new. Legal will also appreciate that your proposed auth solution will help you comply with data protection laws like the GDPR, COPPA and the CCPA, as opposed to your team debating adding such support in a last minute sprint.

Quality Assurance

QA would love to live in a world where they could set up a bunch of regression tests on a system built out of stable third-party components and then move on to the next thing. They don't want to have to come back to dev with a huge list of issues they found in your home-grown auth system and then iterate with you on it over and over until all the kinks are worked out.

They *especially* don't want to miss a crucial test and then have stakeholders across a panicked company asking them why a production auth system wasn't tested thoroughly enough.

Talk to them about how your proposed auth solution has been pre-tested and battle-hardened (if it has been). Talk to them about the large community using the components and how the vendor responds to bugs; do they release a fix quickly or does the bug linger? Talk to them about how they can treat it like any other third-party component, which will let them focus on other concerns, like testing the features customers pay you for!

UX and Design

Good designers want to spend their time designing new exciting aesthetics and workflows, not auth flows that have been done a million times.

Talk to them about how they don't have to reinvent the authentication flows, they just have to tweak them. Have them review vendor documentation about how to customize existing auth flows and screens. That will also inform you on the flexibility you have (or don't) when doing such customization.

Security

If your org is big enough that security needs to sign off on new externally facing architecture decisions, it's likely they'll already be on board with using a pre-existing auth solution.

In talking to people in this area, it still doesn't hurt to emphasize:

- The monitoring benefits provided by a dedicated auth system.
- Standards certifications that your proposed auth system complies with (ISO 27001, SOC2, HIPAA BAAs, etc).
- **Actual laws** that your proposed auth system can help you comply with, as mentioned in the legal section above (GDPR/CCPA are big ones!).

Infrastructure

If your organization is large enough for a dedicated infrastructure team it's likely that you already have a robust auth solution, but there are cases where you have several internal apps with different

auth systems and you want to standardize on a new outsourced auth system.

It might also be that your infrastructure team is young and they're just considering a company-wide auth solution. They might appreciate you doing the legwork here. The infrastructure team might even be where you work!

Talking points of concern to this role mainly focus on **the benefits of single sign-on**. The benefits are numerous, but some of the big ones are reducing help desk tickets, increasing productivity and security (fewer password post-its floating around), and just generally making life better.

Opportunity Cost

Above all else, emphasize spending time doing new things!

Now that we've discussed the majority of potential stakeholders, it should be clear that there's an overarching message to disseminate: **build new things!**

Don't spend your time reimplementing boilerplate functionality that already has robust solutions. Auth is like a datastore; yes, you could build your own, but that'll only make sense in specific circumstances.

Get your team excited about pushing your product forward with new features that will solve problems for your customers. Out-source what already has tried-and-true implementations.

Joe Stech is a former director of software engineering who has designed and implemented software for some of the largest (and smallest) software companies in the world. He is currently a consultant specialized in cloud software and data engineering. He writes about cloud topics at cloudconsultant.dev.

Performing Due Diligence on Authentication Vendors

By Mihir Patel

Within today's software development ecosystem, third-party vendors are a common part of system architecture. Specifically, [Authentication-as-a-Service \(AaaS\) is growing fast](#)⁴. Their out-of-the-box capabilities enable engineering teams to focus on building features valuable to business rather than spending time and resources on reinventing the wheel of securing application access.

But outsourcing isn't as simple as it sounds. Vendor management is time-consuming and can introduce significant risks to the business if due diligence isn't observed.

For example, [the 2017 Equifax data breach](#)⁵ consumed many organizations, including mine where I was on the information security (Infosec) team. The breach exposed the personal data of hundreds of millions of people: social security numbers, names, addresses, and more. Thankfully, our organization had the right policies in place to safeguard the personal data and no data was compromised.

We had communicated to our clients right away about the breach and ensured all servers were being patched up in case there was a leak. Our AppSec team rolled out a patch over the weekend to all of our internal servers including the ones where we hosted our third-party vendors. The team deliberately worked with our clients

⁴<https://www.marketsandmarkets.com/Market-Reports/identity-access-management-iam-market-1168.html>

⁵https://en.wikipedia.org/wiki/2017_Equifax_data_breach

ensuring them the steps taken to protect them and their data. We had mitigated a huge security event. We consistently maintained information on internal servers which also listed all on-premise servers that hosted third-party vendors.

Obviously, it pays to invest effort in vendor management, especially in light of a security breach. It's important that a vendor is able to provide risk assurance, meet compliance standards, and provide analytics and reliable service when partnering with you for functional responsibilities. But if managing vendors is so critical, then you may wonder why you'd want to outsource identity management to a third-party provider in the first place. If your core competencies are in building UX and UI, you want to focus on that, and not the effort of building (and learning how to build) a complex and costly authentication system. Using a reliable authentication system can mitigate operational and security risks for your business.

Of course, if you do outsource authentication capability, then putting in your due diligence is a must.

Due diligence is a series of steps that require research and testing the capabilities of a third-party vendor. Going through this very intentional exercise is absolutely crucial before you onboard an AaaS into your system, as it can prevent future issues with security, performance, engineering, and pricing.

I'll discuss the various areas you should be checking into for a potential AaaS, including security, performance, engineering implementation, and pricing.

Examining the Authentication Provider's Security Standards

Security is at the top of this list and should come as no surprise. Letting unauthorized parties get access to systems leads to loss

of consumer confidence and financial penalties from regulators. Putting in effort to make sure an AaaS offers proper security is critical.

Authentication providers should have strong guardrails to protect your users' confidential data and minimize the possibility of security breaches. Work with potential authentication providers and your internal stakeholders on the following items to ensure security standards are met before integrating them:

1. Include all business and technology stakeholders to facilitate the security review. This will allow you to map which business segments will rely on the authentication provider. Make a practice of communicating your findings with these stakeholders as you move through these steps.
2. Ask your vendor to fill out a questionnaire. This is a standard practice to understand security policies established by authentication providers. These questions should cover all security details, for example, how often are passwords reset? How are credentials stored? Where are they stored?
3. Ask for an encryption policy. It should have guidance on hashing, digital signature policy, and cryptography topics, and these policies should align with your internal security standards. For TLS, the standard is to use [128-bit, 192-bit, or 256-bit encryption](#)⁶ to prevent unauthorized access to data in transit. Does your authentication provider offer encryption of data at rest?
4. CVEs happen. How does your vendor respond when a CVE occurs. How quickly is a fix released and how do they communicate the security issues to you?
5. Understand who will own responsibility in case of a cyber attack. Lawsuits arise when responsibilities are not well understood, so take particular care here. You should have a workflow diagram labeling each section with vendor's name

⁶<https://www.clickssl.net/blog/128-bit-ssl-encryption-vs-256-bit-ssl-encryption>

or your company's name to indicate each party's areas of responsibility.

6. During the procurement process, obtain all required security policy documents applicable to your industry or business. Ensure security policies cover [PII](#)⁷, [HIPAA](#)⁸, and [GDPR, Article 33](#)⁹ standards if your business falls under specific industry criteria. You should also obtain [SOC2](#)¹⁰ reports from any potential AaaS. This auditing procedure ensures that data and privacy are securely managed to protect the interests of your organization and the privacy of your clients.

Of course, we could dive deeper into security due diligence, but the items above should be a minimum place to start.

Measuring Performance With Benchmarks

Naturally, there's a tradeoff between building infrastructure in-house and outsourcing. What you decide will have implications on your services. Auth0, Okta, and FusionAuth are performant services, but you should have preliminary benchmarks to measure initial performance against, with and without AaaS. You can use your performance metric to benchmark against external vendors to make sure they can meet your SLA and performance standards.

First, we need to conduct benchmark testing. If we don't know what we are measuring against, performance metrics don't mean much. Coordinate testing measures on some of the following topics with your QA team:

⁷[https://www.dol.gov/general/ppii#:~:text=Personal%20Identifiable%20Information%20\(PII\)%20is%20defined%20as%3A&text=DOL%20internal%20policy%20specifies%20the,to%20which%20they%20have%20access](https://www.dol.gov/general/ppii#:~:text=Personal%20Identifiable%20Information%20(PII)%20is%20defined%20as%3A&text=DOL%20internal%20policy%20specifies%20the,to%20which%20they%20have%20access)

⁸<https://www.hhs.gov/hipaa/index.html>

⁹https://en.wikipedia.org/wiki/General_Data_Protection_Regulation

¹⁰<https://www.aicpa.org/interestareas/frc/assuranceadvisoryservices/aicpasoc2report.html>

1. Begin with load testing. Some vendors put restrictions on load testing. But if your release process includes [Change Advisory Board \(CAB\)](#)¹¹ approval, you need to conduct end-to-end load testing to get approval for production releases. Reach out to an account manager or sales team from the third-party vendor to request load testing.
2. Many applications error out if response time is not within an implicit timeout range. Every application has different needs. Define explicit timeouts based on your users' location and complexity of their tasks. Then ask yourself if these thresholds sufficiently meet your Service Level Agreements (SLA) with your clients.
3. What is the authentication provider [failover strategy](#)¹²? Are business continuity and disaster recovery policies in place? Outages happen, but your software application won't work without authentication. Understanding your vendor's failover strategy will help you evaluate business risks during outages. For some businesses, and especially in enterprise space, authentication services should be highly available, otherwise SLAs will be breached leading to client loss.
4. Introducing an authentication service in your architecture can lead to latency issues. Sign-ins will have to go through the authentication provider's data center before a user can start interacting with your features. This extra round trip can be costly unless the service is hosted on premise or in the same Availability Zones as your cloud provider.
5. How is maintenance handled and communicated? If your business is global, local maintenance can impact clients globally. If your business is global, you need a vendor who can support your services in different time zones. Vendor maintenance needs to be communicated to all impacted teams across the globe, and you have to make sure you know who

¹¹<https://www.servicenow.com/content/dam/servicenow-assets/public/en-us/doc-type/success/quick-answer/change-advisory-board-setup.pdf>

¹²<https://searchstorage.techtarget.com/definition/failover>

exactly is responsible for that maintenance and releasing communication. Ensure your vendor communicates clearly and directly.

Engineering Effort to Implement Authentication

Authentication is a simple concept but has costly and complex infrastructure, which is exactly why [AaaS is on the rise](#)¹³. Outsourcing critical pieces of software development can help engineering teams deliver high-impact features without losing velocity.

Authentication providers need to keep this in mind while selling you on their service. If implementation is not straightforward, then it's not as beneficial. Here are some ways to ensure the process of AaaS implementation doesn't cost you market velocity:

1. User Experience (UX) is key for developers. While it's arguably not the primary objective of procurement teams, developers are the primary users of AaaS. AaaS UX should be developer-friendly and not too difficult to navigate. Dashboards should have relevant data with appropriate graphs and tooling. Documents should be well-written. Otherwise outsourcing a service that gets underutilized can have financial consequences, such as paying for tools you don't use. Don't wing this. Ask your developers to review the AaaS developer experience or build a proof of concept.
2. Documentation is the lifeblood of implementation teams. No one wants back-and-forth emails or waiting on an engineering sales rep to answer questions about implementation. Well-thought-out documentation should list API calls that are easy

¹³<https://www.globenewswire.com/news-release/2020/05/09/2030657/0/en/Global-Authentication-Services-Industry.html#:~:text=Authentication%20Services%20market%20worldwide%20is,to%20grow%20at%20over%202020>.

to understand. It should also have an education center, FAQ section, change-log, release notes, workflow diagrams, and definitions of technical terms. And just as you should have open communication with the vendor for maintenance, you'll want to ensure your developers have forums or other venues for asking implementation questions.

3. Single sign-on (SSO) via Google, Facebook or GitHub is becoming a common feature for online accounts. It's helpful because users don't have to recreate login credentials every time they create a new retail profile or media streaming account. Your authentication service should provide a widget or page with a single sign-on (SSO) page.
4. Sign-on widgets and other UX features should be hardware agnostic. A smooth implementation should be available regardless of OS, be it Linux, Windows, or macOS. The same goes for mobile devices—both iOS and Android should be a seamless experience.
5. Does the authentication provider integrate with an LDAP directory service? For many organizations, user information and entitlement is stored on LDAP servers. Authentication providers should be able to delegate authorization against the LDAP directory. This is important for midsize to large organizations because LDAP is universal to managing user directory and access management. This allows organizations to add or remove access when an employee is terminated or discharged from an organization.

Pricing

Authentication-as-a-Service (AaaS) can be very economical if you don't want to be in the datacenter business. But before you onboard a provider, you need to understand your own business model. For example, if you have free users and scale overnight, you need to be able to cover those expenses.

Conducting due diligence on pricing is every bit as critical as the previous elements covered. You don't want surprises as your application grows. AWS Cognito is a perfect example when it comes to pricing transparency. Here are some questions to keep in mind when you're considering an AaaS's pricing:

1. For out-of-the-box solutions, what is the pricing per authentication?
2. Have some understanding of your expected user base size and how that will affect the price of the service.
3. Is there an admin tool where pricing is consistently being updated based on current usage?
4. Many organizations are split into several P&Ls (Profit & Loss centers). Is an authentication provider able to segregate pricing based on P&L units?
5. Does the authentication provider charge for trial users who are non-paying customers?
6. What are the pricing level tiers? Does the cost go down as authentication goes up?

Conclusion

To wrap up, vendor management and risk assessment is an important piece of onboarding a third-party service for many reasons. Performing due diligence can address security concerns, performance issues, implementation concerns, and pricing transparency, and understanding the limitations of your vendors is key to preventing security challenges down the road.

As a result, the due diligence process should be extremely detailed, and it should play a critical role before procurement. At minimum, you should:

- Know your security standard and measure it against your vendor's security standard.

- Understand your performance needs and assess whether the vendor can offer the same or better performance.
- Assess whether implementation is going to be easy for your engineering team.
- Know the cost of the services.

At the end of your assessment, understand the tradeoffs. Do the benefits outweigh the risks? Often, partnering up with a third-party provider enables your business to focus on what you do best, delivering value to your end customer.

Mihir Patel is a Product Manager with an extensive background in engineering. He has experience in vendor management and conducting research on third-party vendors.

The Value of Standards-Compliant Authentication

By James Hickey

Software applications regularly need access to data from other services on behalf of their users.

For example, an application may need to grab a list of user's contacts from a third-party service, such as their Google contacts. Or it might need to access a user's calendar so the application can create calendar entries for the user. In addition, larger organizations often require employees to have passwordless access to all the applications and services needed to do their jobs.

How can you make sure your systems are giving proper access to other systems and verifying access requests from other applications? Are there easy and trusted ways to build these integrations?

You'll learn about why it's important to use a standards-compliant authentication protocol when integrating systems.

You'll also learn some specifics about a few of the most commonly used authentication protocols in modern software systems.

Why Use a Standardized Authentication Protocol?

Why should you consider a standards-based protocol in the first place? There are a number of reasons.

Security

If you try to build an authentication protocol or procedure—no matter how simple it may seem—you are putting your system at increased risk. It’s often said in the context of security, “Don’t roll out your own crypto.” The same can be said about an authentication protocol primarily used to integrate systems.

Standardized auth protocols are like open-source software: You can trust open-source code when other experts have examined how they work and have publicly vetted them. Likewise, standardized auth protocols have been publicly vetted by experts and are openly trusted. Because of this, many organizations will trust your solutions only if you are using standardized protocols such as OAuth and SAML.

When you use a standardized protocol, you have the peace of mind that comes with knowing your authentication system is following in the footsteps of industry experts and best practices.

Transferable Learning

What would happen if every time you built a new system’s authentication system, you had to create it from scratch? You would have to learn the nitty-gritty details of authentication over and over again.

This would lead to a scenario where you couldn’t leverage your hardwon knowledge between projects and employers. If you instead use a ubiquitous protocol like OAuth, there may be subtle differences, but you’ll understand the general authentication architecture. If you know how standardized protocols work and what use cases they solve, you can bring that knowledge to other projects and companies.

The same applies when teaching and onboarding new engineers to your team. If you are using a standardized auth protocol, then your new team members are likely to already know about OAuth,

SAML, or other standardized protocols. It will be much easier to get these new team members up and running and contributing to these relevant areas of your system.

Supporting Libraries

Imagine that you have an API with a bespoke authentication implementation. Would you be able to build SDKs or code libraries to interact with your API easily? Since you are using a custom authentication design, you'd have to build the core logic of any code libraries from the ground up every time. And maintain them, forever.

On the other hand, most modern programming languages have code libraries that integrate with standard auth protocols thereby accelerating your development work. By using a standardized auth protocol such as OAuth or SAML, your clients and API consumers can reuse common code libraries for their programming language of choice.

Interoperability With Other Systems

Using an auth protocol such as OAuth makes your system more interoperable with others. If you are working with a larger enterprise organization, then you need to integrate with other systems all the time. Using a standardized way to do this saves time, mental overhead, and overall cost.

The same applies to external systems. If you have created an API that supports SAML, for example, then your customers will understand how to integrate their solutions and systems with yours much more quickly, and with far fewer headaches.

Edge Cases

Standards have been used by many different organizations and systems in many different ways. Often edge cases are either handled or explicitly ruled out.

By leveraging a standard, you will gain the benefits of all that knowledge and experience.

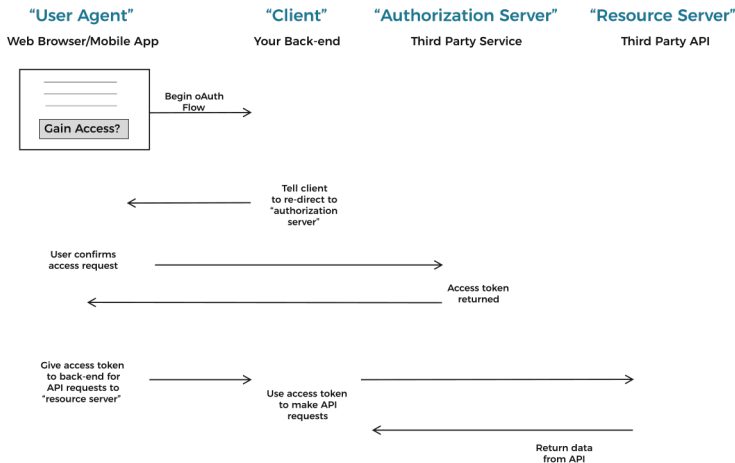
Survey Of Authentication Standards

OAuth

When you need to give a website or application access to your Google account's data, or any other service's data for that matter, OAuth can help to grant access securely. OAuth is perhaps the best-known protocol in the authentication space, so you should know about it. However, it's actually not an authentication protocol, as you'll see later. The current version is OAuth 2.0, though OAuth 2.1 is being drafted.

More specifically, OAuth allows an application to securely gain an access token which can be used to make additional requests to a third-party API or web service. OAuth is a standard that defines this choreography between clients and services to obtain this token.

Here's a diagram of the "implicit" OAuth flow. The official OAuth terms for different actors in the process are in purple:



The OAuth Implicit grant.

It can be useful to look at the implicit flow to understand the concepts. However, please don't use the implicit grant, as it is extremely vulnerable to XSS attacks. You can learn more about that in ["The Modern Guide to OAuth"](#)¹⁴.

Today, it's safer to go with a more secure update to OAuth called [PKCE](#)¹⁵ (often pronounced "pixie") and the Authorization Code grant. Originally intended to enhance OAuth security for mobile applications, this combination provides extra security benefits to all clients using this updated protocol.

Authentication Vs Authorization

It is important to understand OAuth is not an authentication protocol. It's an authorization protocol.

OAuth 2.0 was intentionally designed to provide authorization without providing user identity and authentication, as those problems have very different security

¹⁴<https://leanpub.com/themodernguidetooauth/>

¹⁵<https://oauth.net/2/pkce/>

considerations that don't necessarily overlap with those of an authorization protocol. – [OAuth.com](#)¹⁶

Authentication is about proving you are you. Authorization is about delegating access or permissions to information.

Historically, OAuth has been used as both a way of dealing with both authentication and authorization. However, OAuth doesn't define a standard way to provide user information to the requester—so every auth implementation is a little bit different. This removes some of the benefits of using a standardized protocol.

Many developers may also assume that obtaining an access token from a third-party service means that your user was authenticated. But that's not true. An access token could be granted to your application even if your user was not authenticated (that's 100% up to the service you are requesting access to).

Authentication features like getting identity information, session management, and user registration have to be handled some other way. OAuth servers often support OIDC, which we'll cover below.

When Is OAuth a Good Choice?

If you are creating an app that needs to get information from another source, then OAuth makes sense. It helps to limit your app's access to your customer's resources, and it facilitates the process of requesting and obtaining permission.

While often misunderstood as an authentication protocol, OAuth is not a standardized solution for authentication, as mentioned above. Let's look at some proper authentication protocols that you can use to authenticate your user.

¹⁶<https://www.oauth.com/oauth2-servers/openid-connect/authorization-vs-authentication/#:-:text=OAuth%202.0%20was%20intentionally%20designed,those%20of%20an%20authorization%20protocol>.

OpenID Connect (OIDC)

You’ve seen that OAuth is a great way to gain access to resources. But it doesn’t define a standard way to authenticate your user. [OpenID Connect \(OIDC\)](#)¹⁷ solves authentication by extending the OAuth 2.0 protocol, what is technically called a “profile” of OAuth.



OIDC tokens build on OAuth tokens.

OIDC adds an “ID token” to be returned from the final OAuth request in the flow. This confirms the user was in fact authenticated (unlike the access token, as discussed previously) and also gives you a way to access identity-specific information about the current user.

Beyond these basic features, OIDC can also enable more advanced features such as [session management](#)¹⁸, [log-out ability](#)¹⁹, [user registration standard](#)²⁰, and more.

When Is OpenID Connect a Good Choice?

OIDC is a great choice when you want to allow your users to log into your applications via another service, such as Google, Facebook, Twitter, or other social providers.

Historically, this was done using OAuth along with some customized extras. OIDC provides a solid standard for doing this in

¹⁷<https://openid.net/connect/>

¹⁸https://openid.net/specs/openid-connect-session-1_0.html

¹⁹https://openid.net/specs/openid-connect-rpinitiated-1_0.html

²⁰https://openid.net/specs/openid-connect-prompt-create-1_0.html

a secure and trusted manner.

As mentioned above, OIDC is very often supported by OAuth servers. If you have an OAuth server, check to see if it supports OIDC.

SAML

SAML stands for “Security Assertion Markup Language.” It’s an SSO (single sign-on) standard that is XML-based. It typically enables business users to sign into their organization’s authentication system and automatically log in to the external and third-party applications their employer allows.

SSO offers immense convenience for users. It’s also secure in that businesses can have more control over their user’s security by enforcing policies such as strong passwords and two-factor authentication. These policies then apply, by proxy, to all their external applications.

While transparent to the user, when they access a configured third-party service, it will issue a request to an authentication system called an “identity provider.” If the user is already logged into their organizational identity provider, then it will send a special response to the external application to tell it, “This person is authenticated; let them in!”

When Is SAML a Good Choice?

SAML is a great tool for larger organizations to further enhance their users’ security. Again, the organization can have full control over a user’s authentication policies and force any configured third-party applications and services to “use” the organization’s authentication system to verify users. For organizations that are seeking compliance with security programs such as ISO 27001 or SOC 2, SSO is often necessary.

FIDO

FIDO stands for “Fast Identity Online.” It’s a set of protocols created by the [FIDO Alliance](#)²¹, a nonprofit group seeking to expunge the use of passwords. In general, the FIDO protocols allow users to authenticate themselves via special devices or applications that use fingerprint readers, facial recognition, or external devices such as special USB dongles. Many websites are also now supporting [webauthn](#)²², which is a protocol that enables website logins with FIDO.

FIDO works much like SSH keys. Your device will have a FIDO-enabled “authenticator” application that can generate private and public keys between your device and other services. Your device keeps a private key for that service and sends it a public key. Whenever the device reads your fingerprint, for example, the device’s authenticator application will verify you are you and then use public key cryptography to authenticate you to that other service.

This means that attackers can’t steal your password. It’s also much more convenient than having to enter a password over and over again.

When Is FIDO a Good Choice?

FIDO is a great choice for larger organizations that want to keep their users and devices very secure. If an organization chooses to use some type of specialized hardware, such as a [Yubico device](#)²³, that can be a substantial cost. Not only must the devices be purchased, but they must be delivered when employees are onboarded and retrieved when they leave, incurring additional logistical complexity.

However, with growing support for webauthn and built-in facial

²¹<https://fidoalliance.org/>

²²<https://fidoalliance.org/fido2/fido2-web-authentication-webauthn/>

²³<https://www.yubico.com/solutions/passwordless/>

recognition and fingerprint readers on modern PCs, many cloud-based tools are allowing FIDO-based authentication for their websites, lowering the overall cost of using FIDO. Thanks to FIDO's strong security, you'll often see FIDO used in government, insurance, and healthcare.

Conclusion

You've seen that using a standard authentication protocol is important. It can make sure you are implementing authentication in a trusted and reliable way. Using tools like SSO/SAML can allow your organization to enforce more secure policies that apply across many external applications your organization's users require to do their jobs.

Even better, by using an auth system such as FusionAuth or Auth0, you don't even need to implement all of these protocols. You can use such a system to manage your user authentication and it typically will support OAuth, SAML, and OIDC out of the box.

Whatever tools you choose to use, your understanding of when and why OAuth, OIDC, SAML, and FIDO are appropriate can help you choose the best authentication standard for your organization's goals, security posture, finances, and future growth.

James Hickey is a Microsoft MVP with a background in fintech and insurance industries building web and mobile applications. He's the author of "Refactoring TypeScript" and the creator of open-source tools for .NET called Coravel. He lives in eastern Canada and is a ten-minute drive from the highest tides in the world.

Open Source Vs Commercial Auth Providers

By Keanan Koppenhaver

When it comes to building any web or mobile application, authentication is one of those areas where you generally don't want to take on the risk yourself. Third-party authentication providers have made this their entire business and dedicate lots of time and money to make sure their solutions are performant, easy to use, and most of all, secure. However, some of these providers make the source code for their solutions available and others keep their source code proprietary.

Choosing one of these types of authentication providers over the other is not always a cut-and-dried decision, as there are benefits to each. Support, release frequency, relicensing capability, maintenance, security, and cost are all factors you should consider when making this decision.

Let's take a more in-depth look at the pros and cons of open-source and commercial authentication providers.

Open-Source Authentication Providers

Open-source authentication providers are popular because anyone can review much or all of the code that powers them. This availability can be especially helpful in evaluating whether a particular

authentication provider will work for your use case. In addition, if you want the source code for any number of reasons (eg, the provider could go out of business or get acquired), open source is basically tailor-made for that.

But while open-source providers do have some benefits over proprietary authentication providers, there are some downsides as well.

Support

For solutions that are truly free and open source, often the only support avenue available is submitting a detailed issue via the project's issue tracker and hoping that one of the project maintainers gets back to you. The timeline for this is very undefined, and many open-source maintainers are overwhelmed with the volume of issues that projects get. This can be problematic for something as crucial to an application as an authentication provider, but that's the tradeoff for getting something completely free.

On the plus side, many open-source projects have started introducing paid support tiers, often provided by an affiliated company. If you choose to go with this option, you don't get the product entirely for free, but you usually get an SLA with not only guaranteed support but a guaranteed response time as well. You'll have peace of mind knowing someone knowledgeable about the product is ready and able to answer your questions in a timely manner.

Release Frequency

How frequently product updates are released really depends on the project, but there are two benefits that open-source providers have over closed-source providers in this case.

First, you can look back at the entire release history and see how frequently releases and updates have happened in the past. Have

they shipped one major version a year for the past two or three years? Odds are this will continue. Has the project been dormant for the last couple years? History isn't a perfect predictor, and they may have been working on a huge new release all this time, but that's not a good sign.

And second, open-source projects tend to “work in public,” meaning discussions about release frequency, notes from planning meetings, and more may be publicly accessible. Depending on how the project is structured, you may even be able to be part of these meetings. Some projects have this information in their README file or some similar place. You'd rarely get this kind of transparency with a more proprietary provider.

However, it is worth noting the distinction between open source and open process. A project can be open source but take no feedback from the community; some of Apple's open source projects are like this. On the other hand, commercial software companies can develop “in the open” and even make the source available for inspection using source-available licenses, without allowing for the freedoms that are key to open source licensing. When you evaluate an open-source project, make sure you know what the community feedback process is, if any.

Relicensing

Most open-source projects of any size have the license for the project publicly available, so you can quickly determine whether those license terms will work for you. It's important to review this carefully, as some projects specify that any project that uses them must also be open source, which might be a deal breaker for you and your team. One common license, the GPL, has this restriction.

Some open-source projects, especially if they are backed by a company, support dual licensing. This means that you can choose one license or another as fits your business model. The first license

may be the GPL or a similar restrictive license, while the other may be a commercial license with fewer restrictions that costs money.

Always review the license for any solution.

Maintenance Responsibility

In a true open-source solution, the “community” is responsible for submitting fixes they want to see included in the project. Feature requests from the community tend to need to be contributed to by the community itself, meaning you as the customer bear more of the maintenance burden. The amount of testing done for a new release varies based on the effort the community puts forth.

On the flip side, it could mean you have more say in the product roadmap if you’re able to contribute fixes in a way that is acceptable to the community. Heck, you could even pay a developer to spend most of their day writing code for this system, which would buy you a lot of influence. Consider how much bandwidth your team has to contribute to this process; it might be a dealbreaker.

You can also, if you need to, make a change in your own version of the project. You can then submit it to the maintainers. If it meets their quality standards and goal for the project, it can be integrated; this is also known as upstreaming. However, if it doesn’t get incorporated, you are stuck supporting a critical feature and making sure that future releases don’t break it.

Security Responsibility

Open-source projects [may be more secure, especially over time](https://www.infoworld.com/article/2985242/why-is-open-source-software-more-secure.html)²⁴, because more eyes have inspected the code to find security vulnerabilities. Some projects even undergo specific security audits to ensure their responsibility.

²⁴<https://www.infoworld.com/article/2985242/why-is-open-source-software-more-secure.html>

However, there are also [instances of vulnerabilities remaining open for months²⁵](#). But if the bug is severe and an open-source project is well supported or popular, the timeline may shrink to days or weeks.

In the end, you (and the maintainers) are ultimately responsible for the security of the software. Just like with maintenance responsibility, if you don't have time for your team to frequently review the security issues as they're reported and fixed and then upgrade your system, this may mean that open source isn't the best option for you.

Cost

This is where many open-source solutions really shine. If you're running an open-source authentication provider, the cost for the provider itself in many cases is nothing. Zip. Nada. Who doesn't like free?

The caveat to that price tag is that you generally also have to run this software on your own infrastructure, so there's an implicit cost there. However, if you're already running your own infrastructure, then in many cases, an open-source authentication provider can be added at no direct monetary cost to you.

While this price tag is attractive, it's important not to forget that open-source providers come with added maintenance and security responsibility when compared to proprietary solutions, as we've already discussed.

²⁵<https://github.com/keycloak/keycloak/pull/7612>

Commercial Authentication Providers

An alternative to open-source, commercial authentication providers (such as FusionAuth, Okta, and Auth0) are popular because they take care of all the ambiguity and self-reliance that come with open source.

But just because commercial providers handle many of open source's problems doesn't mean there aren't tradeoffs that need to be considered.

Support

Because you're paying for a commercial authentication provider, some level of support is included in your contract. This can be provided either over email, phone, live chat, or some combination of all of those. Generally when you sign up with one of these providers, some sort of SLA will be provided with a contractually guaranteed response timeframe.

The peace of mind from having dedicated support can be one of the most important reasons to choose a commercial authentication provider over open source. At the end of the day, the buck stops with them. As noted above, you can often pay a company to support an open-source option, too.

Release Frequency

Unlike with open-source providers, it may be difficult to know how often updates for commercial providers are released, especially if all of the provider's infrastructure lives in the cloud and updates don't require any action from the customer. They may release their

software multiple times per day, or it may be months between public releases.

This is one of the questions you can ask the sales team when evaluating different providers, but commercial providers don't always have the transparency associated with open-source providers. You can't always just go and look up previous release dates and research update progress yourself. However, some providers, such as FusionAuth, do offer [comprehensive release notes](#)²⁶ to aid you in your evaluation.

Relicensing

Commercial providers tend to have either more obscure or more restrictive licenses. You won't be able to take the code and do anything you want with it. If you are building a typical web application, this shouldn't be an issue, but if you are embedding or redistributing your code with the auth system, that may violate the license. Ask your vendor.

Many of the issues of relicensing that come with open-source technologies don't apply to commercial authentication providers, because they usually provide a single license that goes with their product. They handle the licensing of any component parts in-house and usually have legal teams or an attorney that's reviewing all of the different aspects of software licensing.

This is another thing you don't have to worry about with a commercial provider, rather than being responsible for reviewing all these licenses if you went with an open-source solution.

Maintenance Responsibility

This is one of the main benefits of a commercial provider. They are responsible for all the maintenance of their product. Because you're

²⁶<https://fusionauth.io/docs/v1/tech/release-notes/>

paying for the product directly, this is one area you no longer have to worry about. You are trading money for time.

However, unlike with open source, you have less freedom to make fixes yourself. So if you have a feature or a fix that you believe should be in the product, you try to convince someone at the vendor to get that prioritized. With an open-source solution, you could fix it yourself, if you are willing to accept the maintenance burden.

That said, many commercial vendors accept community input to their roadmap and let you file bugs. Some vendors even offer professional services agreements which can ensure feature delivery on a schedule.

If you are worried about access to the source code if the commercial vendor goes out of business, ask about a source code escrow. Some providers are willing to add a provision to their contract with you.

Security Responsibility

As with maintenance responsibility, the responsibility for the security of a commercial product is on the vendor. They will be the ones applying patches and performing frequent security audits to ensure their product is secure. These could include running internal security reviews, a bug bounty program, or hiring a pentesting firm to test the product's security.

It may take longer to find bugs than it would with an open-source product where more people from all sorts of different backgrounds are invested in the security of the product and review it frequently. While the security responsibility may not be on you as a customer, you do have the responsibility to perform due diligence on any potential vendor and ensure that they take the security of their product seriously. One way to do that is to examine the vendor's public security policy. Another is to look at the [CVE database](https://cve.mitre.org/)²⁷ and see how vulnerabilities are handled.

²⁷<https://cve.mitre.org/>

Cost

One of the more notable differences between open source and commercial solutions is cost. Because of the benefits detailed above, there is usually a cost to a commercial solution, whereas many open-source solutions are provided free of charge and maintained by the community.

However, some commercial software offers free tiers with limited support. For example, Auth0 allows you a certain number of users for free. FusionAuth has a community edition allowing you an unlimited number of users if you self-host.

Another item to consider is the pricing structure of the commercial solution. Some commercial products may prove cost-effective to large corporations, with high usage, but terribly improbable for a startup wanting a lowering buy-in point. Or more commonly, commercial providers provide different tiers depending on usage. This may make some use cases untenable as your product scales.

Depending on what stage your company is in, this monetary expense can either be something you can absorb or a dealbreaker. You should weigh this cost against the time and effort needed to maintain an open source solution.

The Final Showdown

Using third-party authentication can be a great choice for your application, allowing you to focus on the business logic where your application provides value, but there is more than one type of provider. So in the battle of open source versus commercial authentication, which is the better choice? As with most real-world decisions, the answer is “that depends.”

Open-source authentication providers can be a great choice if you have more time than money. They are usually more cost-effective,

too, but they do require internal resources for maintenance and upkeep.

If you don't have the staff or the time to maintain an open-source solution, or if you do but don't want to make that investment, third party providers can be a great option. Such providers offer high-quality authentication solutions to your business and take care of all the maintenance and security concerns that go with it. This lets you focus on writing the code that matters.

Keanan is the CTO at Alpha Particle where he helps publishers modernize their technology platforms.

The Value of Trying Your Auth Provider Before You Commit

By Keanan Koppenhaver

Sometimes, despite a salesperson overcoming all your concerns or a landing page perfectly crafted to speak your language in every bullet point, there's a lingering fear that the product you're looking at just might not be what you're looking for. Everyone has experienced buyer's remorse at some point in their life, and most people I know don't want to go through it again.

So what's the only way to know if you'll truly love a product? Trying it risk-free at no monetary cost to you.

This holds especially true for larger or more long-term purchases. When you're evaluating authentication providers, one of the main building blocks of any software product, you want to make sure you won't regret your choice a few months or years later.

Choosing incorrectly could mean a major rewrite of your application logic. Getting months down the road only to realize that a particular provider can't support all your use cases can really kill developer momentum.

Advantages of Trying an Auth Provider Before You Buy

Trying your authentication provider before you commit can save you lots of headaches in the future and should be an important part of choosing an authentication partner.

Offering a free trial is an increasingly common practice among authentication providers, but if there's no trial option listed publicly, you should ask a salesperson if it's an option. If there still isn't a way to experiment with an auth system before you pick it, you may want to consider working with a different partner. There are many reasons you want to try before you commit, but let's go over a few of the more important ones.

Your Developers Can Test the API

When making any significant purchase that your development team will use as part of their daily work, it's important to know that your developers can work with it and use it in ways that will benefit the product long term.

If the developers start testing a potential API and they find they can build a relatively simple prototype to prove the API can support all your use cases, then you can be confident that a larger-scale implementation will work as well. Not only will this give you more confidence that the product you're buying is the right one for your business, it will also help your developers work more quickly when they do move to a larger-scale implementation because they will already have experience working with the API.

For example, you may be looking at a few different authentication providers and have one that you've decided is the best candidate for your application. After all, it checks all the boxes! But when your developers start reading the documentation and actually build out

a small prototype, they may find the library for your application's language doesn't support passwordless authentication. If you had committed to this authentication provider before discovering this, you might have already heavily integrated it into your application—which needs passwordless authentication.

This would be a huge problem leading to a major rewrite or possibly having to drop a key feature of your app. However, if you had built a prototype using a free trial and discovered this limitation ahead of time, you could have simply moved on to choosing a different provider.

Better Understanding Of Costs

Many API-based products such as authentication providers are usage-based, which means as your usage increases, the pricing changes. While looking at the pricing table may have you convinced that you'll be on a free or low-cost plan, actually building out a prototype might reveal otherwise.

There are all sorts of potential hidden costs that you may not have considered from just thinking about your application at a high level. Knowing what this product costs on average for other companies of your scale is helpful.

For example, if you're working with an authentication provider that charges you on a per-authentication basis, you might take a look at your users and how active they are, and estimate how many people will be authenticating every month. With this math, you might conclude that you're well under the threshold for the authentication provider's free plan.

But when you build a prototype, you might notice that every incorrect password attempt counts as authentication, demonstrating that your previous estimates might be severely different from what you should expect for actual usage. You might learn that multiple

connections to enterprise identity stores, such as SAML IdPs, cost an arm and a leg.

Catching surprises like this during a free trial can help you understand how to factor your costs correctly, as opposed to finding out when your integration is already in production. It allows your development team a chance to think up different ways to architect your solution to fit within these constraints, or even choose a different provider if a workaround or change in architecture isn't going to be possible.

Developer Buy-In

Any new initiative will move faster and more efficiently if the whole team is on the same page and giving the new integration their full effort.

Developers can be a particularly stubborn group when it comes to efforts like this. However, using a trial period to give even the most resistant developer time to use the proposed solution and discover any necessary workarounds or implementation details will make an eventual integration with your application easier. It can also reveal the strengths of the solution; you may be pleasantly surprised.

And if the development team is split as to which solution they think is best, a free trial like this can help opinions converge around the best solution in practice, rather than wasting time on arguments of opinion. One common attempt to resolve a split between solutions is to just have more demo meetings from vendors and opinion-charged internal meetings. This often leads to frustration and the vendor selection process stalling out all together.

As an alternative, consider dividing the development team based on who backs which solution and having each side develop a small prototype to demo to the team. This will quickly bring to light any challenges with one solution or the other and allow for much more productive conversations after the trial is over.

Running an Effective Trial

It's not enough to simply have a free trial of a potential authentication solution. For your trial to be useful, you need to run it effectively in order to be sure that you've tested for all of your potential use cases and gotten all the information you need to make a more informed decision.

Decide What You Need to Learn

You might not be able to answer every single one of your questions during the free trial period, so it's important to prioritize what you're hoping to learn during your free trial and then work on answering those questions first.

For example, if you have a web application as well as iOS and Android apps and they all need authentication, does the provider you're trying out allow you to use the same platform across all three of these applications? Is one of them going to be more difficult to support than the others?

If this is what's most important to your product team, it's important that you have a solid answer to this question to make the most of your free trial.

Set a Realistic Timeline

Free trials vary in length, usually from around seven to thirty days. However, if you're scoping out a complex integration, a thirty-day trial may not be enough to truly evaluate the product and see whether it's going to work for you.

Some options, like self hosted FusionAuth, are free to run forever. Even if this is the case, setting a deadline can be a forcing function to ensure that a decision is actually made.

It's important to be realistic. Try to work with the vendor to get an extended trial if you think you'll need it. You don't want to get to the end of your trial and find that your time and effort were wasted because you weren't able to answer your important questions about the product. Some vendors will be able to offer an extended trial if you ask.

Otherwise consider reducing your list of important questions to what you can reasonably verify in the allotted time or signing up for a paid plan with the internal understanding that you aren't necessarily committing.

Isolate Your Experiment

If you can, it's best to build out your trial prototype or run your experiments in an environment that's similar to your production environment but not actually your production environment. This way, you're protected from any unintended consequences of your experiment affecting live customer data or causing bugs and unexpected behavior for customers trying to use your application, but you aren't building a toy application which doesn't fully exercise the features you need.

If you don't already have a QA environment, this can be a great chance to create one. And even if you don't want to spin up a new environment, you can have someone on the development team run the experiments in their local environment to make sure you can verify all the necessary functionalities.

Free Trials Are Important

Trying a full version of an authentication provider that you're considering is a very important step in making a final purchasing decision. Because authentication is a crucial component of any

application, it's not a decision you want to be forced to reverse down the road. That would not only be costly in terms of dollars but also in developer time that could be spent working on new features for your product.

Keanan is the CTO at Alpha Particle where he helps publishers modernize their technology platforms.

Risks

Everything in life has some risk, and what you have to actually learn to do is how to navigate it.

– Reid Hoffman

Risk is everywhere, including in auth systems. You can be aware of it and you can plan to mitigate it, but it'll still be there. Beyond any normal software architectural risk, there are specific issues for auth systems.

This section covers some of them. In this section, you'll learn:

- Common implementation risks and how to mitigate them
- What steps to take to avoid lock-in to a particular vendor
- What to do if your auth system vendor is acquired.

Let's take a deeper look at some of these risks.

Common Authentication Implementation Risks and How to Mitigate Them

By James Hickey

Given the increase of [data beaches in the past few years](#)²⁸, it's more important than ever for software engineering leaders to prioritize security, quality development practices, and robust governance controls. Your customers' trust is on the line—and that's the lifeblood of any business that wants to keep growing.

Your authentication system is one of the areas of your software system that you absolutely have to ensure is secure. Not only could a poorly implemented authentication system cause a loss of customer trust, it could also have major implications for your company's finances, overall reputation, and regulatory compliance.

You need to make sure you're aware of the risks and properly mitigate them. While building your authentication system from scratch can give you more flexibility and control over the low-level details of user authentication, you are also potentially introducing more room for error. Depending on your need to have total control over the low-level details of the user login experience, using a third-party service could significantly reduce the risk of failing in these areas. Bonus, it can also speed up development.

Whether you choose to build it yourself or use a third-party service

²⁸<https://havebeenpwned.com/PwnedWebsites>

for auth, let's explore some best practices for keeping yourself and your customers safe.

Security and Privacy

OWASP's list of [top ten web application risks](#)²⁹ is a good place to start as you determine how best to defend your company against potential security risks. You need a plan for making sure these risks are avoided or, if encountered, fixed as soon as possible.

A major part of this plan should be to bake security controls directly into your software development process. This can come in many forms, such as:

- Automated code linting
- Static code analysis focused on security risks
- Periodic third-party penetration testing
- Code review
- Coding standards and conventions
- Security training
- Senior developers coaching less-experienced developers
- Pair programming

If your organization is compliant with one of the major security standards, like [SOC 2](#)³⁰, then you know that even seemingly insignificant elements of your organization—such as how your team shares code, automated code linting, and having clear coding standards—can impact your system's overall security. To avoid leaks, consider implementing some of the policies in the list above.

²⁹<https://owasp.org/www-project-top-ten>

³⁰<https://www.imperva.com/learn/data-security/soc-2-compliance/>

Performance

There are [several considerations](#)³¹ to address in your auth system. These are the first lines of defense against attacks:

- Password strength
- Account lockout policy
- A strong modern hashing algorithm

Let's dive a bit into password hashing. Using a modern hashing algorithm is critical. If an individual password hash is leaked (via an SQL injection attack, for example) or perhaps even an entire database is stolen, then an attacker has the opportunity to try to figure out what the plain text passwords are. They can take a list of common or randomized passwords, hash each potential password, and see if the hashes match any in the database. Salting can help here, but not if they have the salt as well.

Consider the [difference between using a strong and weak algorithm](#)³²:

“If you use SHA1 to hash a password, an attacker can try 10,000,000,000 passwords per second on commodity hardware. If you use PBKDF2 with many iterations, an attacker can try 10 passwords per second. That makes a big difference when brute-forcing a password.”

But wait! Let's look at password hashing from another angle. What if you're already using a very strong hashing algorithm? A strong hashing algorithm is more CPU intensive than a weak one. What would happen if an attacker flooded your login endpoint(s) with a ton of traffic? It's possible your servers wouldn't be able to

³¹https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

³²<https://www.sjoerdlangkemper.nl/2016/05/25/iterative-password-hashing/>

handle the traffic, denying login to your users. Additionally, your cloud usage (and costs) would skyrocket as more login servers were required.

One way to mitigate this is to [implement rate limiting](#)³³. You might choose to limit requests per IP or per specific email address.

When you're building your own authentication system, prepare to handle these kinds of issues. Third-party services, on the other hand, have already put the hard work into tackling many of these security and performance-related considerations.

Regulatory Compliance

I remember when [GDPR](#)³⁴ had everyone panicking. The company I was working for spent a lot of time preparing their system:

- New backup schedules
- New backup retention policies
- Extra data encryption measures
- Changes to password algorithms and work-factors
- Adding functionality to allow users to export their data
- Cookie banners
- Additional staff training

If you've had to deal with other kinds of regulatory compliance, then you know GDPR isn't the only standard that can make such broad strokes affecting an entire company or even an industry. SOC2, PCI, HIPAA, COPPA, and ISO are all standards that can put an additional burden on your software systems. This applies especially to your authentication system, as it holds private and sensitive user data that must be protected.

³³<https://cloud.google.com/solutions/rate-limiting-strategies-techniques>

³⁴https://en.wikipedia.org/wiki/General_Data_Protection_Regulation

If you're building your own auth system, you might need to build additional functionality to meet such standards, like private information export features and advanced encryption measures.

Again, using a reputable third-party authentication service can take some of the load off. They already need to have a high standard of compliance since they're in the business of dealing with private information. For example, FusionAuth is GDPR, HIPAA, and COPPA compliant. You don't need to worry about having to manage the compliance in your auth system with a reputable and compliant third-party vendor.

Cost and Time

Don't forget to consider time when deciding how you want to go about adding an authentication component to a new or existing system. If your team either builds it all or integrates disparate open source libraries, how much time will that take? We all know that whatever your answer is, it'll take longer!

As an engineering leader, you need to consider questions like these before you commit to an authentication approach:

- How many developers will be required to plan, develop, test, and deploy your auth system?
- Do you have any experts or specialized developers who can make sure you don't miss addressing any risks?
- What if the system goes down?
- Who will maintain it as needs change over time?
- Are you ready to bear the burden of improving the scalability of your auth system when traffic to your system increases?

So, is it worth the work, time, and cost of building an auth system yourself? Careful consideration of the questions in the previous list,

and the risks they expose, is a big reason why many companies will choose a third-party service for authentication. Authentication, after all, is usually not a business differentiator, so why spend your developers' time reinventing the wheel when someone else has already solved the problem for you?

Integration and Features

When an organization grows to a certain point, and its software system increases in size commensurately, larger and more established clients will start asking for more advanced security features. They'll also ask for integration with other products.

How are you going to authenticate requests coming from other platforms? How will other platforms validate the requests you make to them? You'll need to implement standardized auth protocols like OAuth, OpenID, and SAML to perform such third-party integrations.

If your company has the time, money, and people to invest, and considers authentication as a core business domain/differentiator, then implementing these kinds of additional security features might be the best approach. But for the majority of us, that's not the case. A third-party solution can typically get you these more advanced features out of the box.

Vendor Assessment

If you're considering using a third-party vendor, then you need to make sure you perform due diligence in assessing how appropriate and risky that vendor is. Many regulatory programs like SOC2 require some kind of controls around vendor risk management.

While we have already touched on areas of risk in implementing an auth system, like application security, financials, and performance, assessing a vendor is where the rubber meets the road. It's where you apply all your knowledge about risk in authentication implementation.

For example, you'll want to consider how coupled you'll become to a vendor's solution (otherwise known as vendor lock-in). You may want the freedom to replace a vendor in the future, perhaps having found a better auth service that's cheaper or supports a compliance program your current vendor does not. Does the vendor you are considering now make it easy or difficult to switch?

Broadly speaking, make sure you answer each of these questions before deciding to move forward with a vendor:

- Does this vendor meet required security standards like SOC2, HIPAA, or COPPA?
- Does this vendor perform regular penetration testing?
- Is there any risk of this vendor and its services affecting my organization's reputation?
- Is the cost of the vendor's service a potential financial risk?
- How does the vendor calculate pricing? Is it usage based, a flat fee, or something else?
- If our organization ever wanted to switch vendors, does this vendor offer some kind of data export functionality?

Of course, this is just a starting point. These questions bring up other fundamental questions like, "What is our threshold for considering a particular vendor's service as risky?" Your organization will have to answer that, and other subjective questions, based on your own risk tolerance, business agility, and company culture.

Conclusion

Authentication is not simple. You've seen just how many areas affect and are affected by authentication: security, your organization's reputation, financials, project timelines, governance, and so much more.

Opting for a third-party service to supply your authentication is a great way to focus your efforts on areas of your system that are true business differentiators.

However, if you do decide to go with a third-party, remember to properly assess potential vendors. This crucial step will go a long way toward protecting you and your organization from an inappropriate vendor and the risks they may expose you to in the future.

James Hickey is a Microsoft MVP with a background in fintech and insurance industries building web and mobile applications. He's the author of "Refactoring TypeScript" and the creator of open-source tools for .NET called Coravel. He lives in eastern Canada and is a ten-minute drive from the highest tides in the world.

Avoiding Authentication System Lock-in

By Cameron Pavey

Years ago your team decided to use a third-party auth system to avoid the time and cost of building one in-house. But now a better option has hit the market and you're wanting to make the switch. Except, hold on, your old system is so deeply ingrained into your organization that you're practically locked-in to your current vendor.

Authentication system lock-in is a big problem. While there are multiple benefits to using a third-party auth system, such as robust security maintained and improved by a team of engineers, out-of-the-box auth UI, and handy features like analytics and logging, there are just as many downsides.

There is always the monetary cost, which for services like this is typically ongoing rather than a one-off. Having less direct control means you don't have to maintain and monitor it; however, there isn't much you can do if there is an outage or performance issue. Finally, the aforementioned vendor lock-in problem can leave you feeling trapped and unable to move away from your provider if you haven't taken steps to mitigate this risk.

Well we can't solve all of those problems, but we can at least show you how to mitigate the risk of vendor lock-in. We will cover a few different tips and strategies you can apply when it comes to auth systems like Auth0, Okta, or FusionAuth.

Look For Open Standards

As with any project, planning plays an important role. You can avoid nasty surprises down the line by conducting a proper analysis of the options available to you and weighing the pros and cons of each potential vendor. Remember, not all auth systems are made equal, and some are bound to be stickier than others, which is not something you want if you are looking to minimize vendor lock-in.

Be sure to focus on standards-based offerings. If your goal is to avoid vendor lock-in, novel or proprietary solutions are not generally going to be suitable for you. In the case of auth providers, look for support for open identity standards like OpenID Connect, OAuth, SAML, etc.

While open standards aren't a silver bullet, you will find it easier to migrate to another vendor in the future if your system relies on open standards also supported by other vendors.

Consider Portability

Another primary concern with any third-party service is portability. If you need to leave in the future, how difficult will it be to get your data out and migrate it to the new provider?

Auth services store varying amounts of data depending on how you have them configured. Still, most of them will maintain at minimum a list of all authenticated users and their supported authentication mechanisms (social logins, enterprise logins, email/-password, etc.).

In some cases, you have more control over how this data is handled and might even be able to provide a self-managed database for the provider to use for data storage. Having this option is good, as the data stays with you, but it isn't always an option or viable to do so.

At a bare minimum, you want to be able to export your data from one provider if you decide to leave, and ideally have a way to import it elsewhere easily. Some third-party services will provide resources on how to do these migrations, as is the case with [FusionAuth, when migrating from Auth0³⁵](#).

If you end up going with a provider who doesn't offer full-data exports, you may be in trouble down the line. Data like Rules and Roles might need to be manually reconfigured in most cases, or scripted with something like Terraform. User data - including password hashes - will be much more problematic if you cannot get it out of the system when you need it, so be sure to check on this before making a decision.

Limit Your Usage Where You Can

Regardless of which provider you go with, many of them offer the ability to store extra data in their system. Being able to store things like user properties and roles on their system seems nice at first, but might be more trouble than it's worth. Generally, it is better to keep details about your user and their relation to your application, in your application. You will doubtlessly have a user model in your system, and this is a better place to store these details (or on related models) because it gives you an extra layer of insulation from the auth provider.

Even if you might want to filter authorization to an application so that only your employees can access it, this would be an excellent time to use provider-side attributes to flag authorized users. In this scenario, you'd likely have a better time adding authorized users to a group in your enterprise identity system (such as a business Google or Microsoft account) and asserting that property in your auth system. This way, even if you need to move to another

³⁵<https://fusionauth.io/resources/auth0-migration/>

provider later, your user attributes will remain intact without needing to be migrated.

Insulate Your Application

When implementing an auth system (and various other services provided by third-party vendors, such as storage), there is usually an integration involved. This integration requires writing code in your application to interact with the vendor's service.

It is usually a good idea to write this code so that the vendor is insulated away from your application code. Often, vendors will provide an SDK for interfacing with their service. By wrapping the usage of this SDK and abstracting it with a more generic interface, we can have greater control over how the vendor ties into our application.

If the vendor uses Open Standards as described above, this makes things easier. We can build an interface that describes how we would ideally like to interact with our auth service and then implement this interface with a wrapper around the vendor's SDK. In the future, if things go south and we need to break away from the vendor, we can just re-implement this interface with an integration for a new provider, and with any luck, things will transition smoothly.

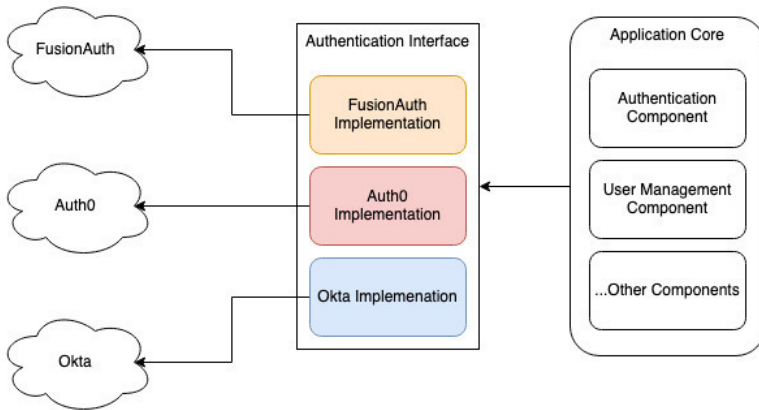


Diagram of Authentication Interface example.

As you can see in this simple diagram, by having your core services rely on a generic abstraction of an auth system rather than a concrete implementation of one, you can retain the freedom to reimplement the interface with code to communicate with whatever provider you like. If you built a concrete dependency on one particular provider instead, you would have lots of refactoring and retesting to do when switching implementations. Naturally, even with a well-abstracted interface, you will still need thorough testing.

It's crucial to build integration tests around this kind of abstraction. Having integration tests in place to give you confidence in your abstraction and interface is always good, but especially so when there is the possibility of re-implementing the underlying code at some point. As long as the new implementation holds to the interface, you will have a good level of confidence that things are working as expected and that your application still works after a migration.

Have a Backup Plan

Generally, you don't implement one solution while actively planning to swap it out in the future. Because of this, it is easy to accidentally put on the proverbial blinders and only focus on what is right in front of you without seeing the bigger picture and the long-term ramifications of certain design decisions.

This can lead to situations where implementations become too specific to the current vendor and are prohibitively expensive to change in the future. "If only we had known we were going to switch providers, we would have done things differently." Factoring in and budgeting for a Plan B or future migration can mitigate such tunnel vision.

This works by identifying an alternative solution-whether it be another provider or building it in-house-and keeping it in mind when designing things and making decisions. You should work it into your project's future budget and just accept it as potentially necessary.

Sure, you hope never to invoke Plan B - and the cost it would entail. Still, if you built everything with this eventuality in mind, your decision-making process is likely to be more deliberative and ultimately less tightly coupled to your chosen vendor. Factoring in the potential resource cost will also help you avoid ending up in a position where you must migrate but don't have the budget.

This is a similar principle to the test-first mindset. If you write your code with testing as a foremost concern, the output will likely be different-and more testable by nature-than if you conducted testing as an afterthought. Making architectural decisions while mindfully being aware of possible future migrations can lead to more robust system design and easier migrations in the future.

Wrapping Up

There will always be the potential for issues when switching out something as central to your application as Authentication, no matter how much you plan and prepare in advance. This risk can be reduced or eliminated by following practices like those described above to protect your application from the uncertainty inherent to third-party integrations. The value that using a service like this brings is not to be understated, as long as you understand the risks and take appropriate measures to minimize them.

Cameron is a full-stack dev living and working in Melbourne. He's committed himself to the never-ending journey of understanding the intricacies of quality code, developer productivity, and job satisfaction.

What To Do When Your Auth System Vendor Gets Acquired

By Eze Sunday Eze

Authentication is an integral part of your application, and as such the acquisition of your auth vendor isn't like other acquisitions. It could mean many things for your business, and you'll have to decide how to respond accordingly.

Will your new provider give you the same support? Pricing? Integration options? All of these might change for better or for worse.

Consider the acquisition of Auth0 by Okta. Auth0 customers praised them for their top-notch customer support. However, Okta customers have lamented their lackluster customer service [in online forums](#)³⁶.

“We moved from Okta a few years ago after we received almost no actual real support for a bunch of issues, even though we were paying a premium cost.

“Nobody cares about issues on [Okta's] GitHub... The kicker was when we received a support response as suddenly something was no longer working after an update, we got help in the form of ‘We have no plans to address this anytime soon’ when asking for an ETA.”

“We ended up switching to Auth0 after we had a few calls with [Okta]. We shaved a decent amount off our

³⁶<https://news.ycombinator.com/item?id=26336270>

costs with Auth0's Enterprise plan, and their webtask based rules worked. While the migration sucked for a bit, in the end, we were much happier."

While clearly an acquisition is cause for concern, it might not be all bad. We'll discuss red flags to be on the lookout for when your auth vendor is acquired, short-term as well as long-term concerns, and how your engineering team can best handle them.

Short-Term Concerns

During the acquisition and transition process, the attention of both the purchasing company and the company being acquired will be given to internal politics. That means a lot of other things get put on the back burner.

New Features Are Given Less Attention

Meetings with key decision-makers will focus on enabling a smooth acquisition rather than rolling out new features. As a customer, this will impact your business, as management will probably be a little disorganized for a while.

Imagine that your own customers are waiting on you to deliver a new feature or fix a bug that's solely your auth vendor's place to fix.

Now what? The new management might not even be interested in rolling out that feature.

Vendor Employees Experience Uncertainty

The employees of companies on either side of the acquisition may feel uncertain about their employment status.

Will the new management decide on a total overhaul of existing staff or will they find a place for everyone? Worried employees can understandably have impacted performance.

Support Deteriorates

All of those reasons—lack of management oversight, shifting plans and roles, delayed rollouts, employee uncertainty—can be felt by your company as less attentive support from your auth vendor during an acquisition.

Short-Term Benefits

There are also some potential positives in the short-term! For example, sales reps might be distracted, resulting in fewer emails.

On a more serious note, your auth vendor might offer you incentives, like more free subscriptions, so you'll stick around through the bumpy transition.

Keep in mind that even though processes might be moving fast during this time, the truth is their business model won't change overnight. The same goes for their pricing. It might eventually fluctuate, but that will most likely happen after everything has settled down.

For most of the transition, your auth vendor will try to make you happy. They know that some of the issues previously mentioned will cause churn and are interested in minimizing that.

Long-Term Concerns

Of course, your concerns aren't magically over after the acquisition completes.

Some Or All Of Your Auth System Features Could Be Phased Out

A company I worked for a few years ago had a very robust and all-encompassing [ERM](#)³⁷ that was used by several hospitals across the country. The company was bought by its competitor, and a year after the acquisition, they announced they were going to retire the software and onboard existing customers to a new software package. The company would not be supporting the acquired ERM anymore.

Customers were upset, and some even threatened lawsuits. That didn't do much to change things. The old software package has been phased out after a couple more years and support has ceased. This happened because the acquiring company figured out the software had too many bugs and would be better off if its functionality was added to other in-house products.

While this was not an auth system, the same logic can be made by an acquirer. A more auth specific example is what happened with Stormpath and Okta in 2017, where customers were given [six months to migrate from the former to the latter](#)³⁸.

What's the use of an auth system that's not maintained or supported? Maybe you could live with an unmaintained software package if it wasn't under regular threat, but that doesn't describe an auth system. Authentication is critical to your customers' privacy and all application functionality. You don't want a security vulnerability in your auth system to shake customer confidence in your application.

³⁷https://en.wikipedia.org/wiki/Electronic_health_record

³⁸<https://techcrunch.com/2017/03/06/okta-stormpath/>

Product Is Milked and Prices Rise

Your auth vendor's new management might decide to modify pricing and plans. They may move some of the best product features to higher payment plans. One thing is for sure, no one acquires a company looking to lose money on the purchase.

Long-Term Benefits

Some acquisitions happen because your previous auth vendor wasn't making enough money to keep running the business but had a very good product that people loved. In that case, it's possible that new management ends up improving your situation.

If the new organization is more financially stable, they could make things a lot better for you with more support staff, better management, more developers, better product maintenance, and an improved feature delivery rate. More resources can mean a better product.

Mitigating Concerns

Once you realize an acquisition is happening, there are some steps you can take to guarantee you'll still be in business after the dust has settled.

Review Your Contract

Not the most exciting first action, but it's important to know what changes your auth provider can make. Find the contract you signed with your vendor and review it.

It's also important to think about how this contract might be expected to change. Get clarity and get yourself ready.

Review Your Usage

Take a close look at how your business solutions integrate with and use your auth vendor's services right now. Figure out the features you use that are absolutely crucial and which of them are proprietary features.

Are they following standard auth protocols? How many of your apps are using this vendor? At a minimum, answering these questions will keep you well informed should you need to migrate to another vendor.

Talk to Your Account Manager

It's time to talk to your account manager about your business relationship. Give them a call or an email and try to negotiate a long-term contract that will protect your business interests and guarantee a level of stability. The research into the contract you did previously? Now is the time to reference it.

Don't forget to ask about migration timelines while you're at it, so you know how soon you need to be prepared and for what changes.

If you don't have an account manager, send an email to the sales or support team. They may send you elsewhere, but are a good starting point.

Evaluate What It Would Take to Switch Vendors

Budget dev team time to look into other options in case it becomes necessary to move. Discuss the possibility with your partners

or stakeholders, touching base with everyone you discussed the issue with during the initial decision process, so that everyone understands what it would take to make the switch.

Even if you stick with your vendor through the acquisition, at least now you know more, and you're prepared for whatever comes afterward.

Consider Impact to Current or Planned Projects

There's never a perfect time for a huge change to your auth system. What current projects will be impacted, for better or worse? Do you have projects in the planning stage that will have to be reimaged due to new standards or a different set of features?

It's best to discuss this with your stakeholders, again so that everyone is on the same page and has consensus about priorities.

Consider Other Options

If the changes are disruptive enough, you may decide you don't want to use third-party solutions anymore. You have a few other options:

- **Use a non SaaS solution:** SaaS solutions are great, but if you use a non-SaaS solution, where you host it yourself, you have far more control over any changes to functionality. You may have to upgrade for security or contractual reasons, but you'll be able to do it on your timeline, not the acquirer's. FusionAuth can be self-hosted and has a community version that is free for unlimited users.
- **An open-source solution:** You'll still have to manage your own source code. You'll just be using a free and community-driven solution like Gluu, Keycloak, or OpenIAM. Your team

will have to explore these projects and choose the one that works best for your system.

- **In-house custom build solution:** In some rare cases, none of the available solutions will be a good fit for your organization, and you'll decide to build a custom auth system for your product. This will require more resources to achieve but of course, when completed, you'll have a solution that works best for you. Further, such a choice can be fine-tuned with more features in the future at your will (and expense)—something you won't get anywhere else.

Conclusion

Your authentication system is one of the most critical pieces in your application, and you have to take it seriously. If your auth vendor gets acquired, understand the implications and discuss them with your team.

Review the ways you expect to be impacted immediately as well as down the road. Weigh your options as to whether you want to migrate to the new system or find an entirely new solution.

If you decide to migrate, take some time to consider how you want to approach it—can you afford to take your time to avoid surprises, or do your users need you to move faster?

Eze Sunday Eze is a software developer and technical writer trying to make sense of the world-building amazing stuff and documenting every step of the journey.

Implementation

There are no secrets to success. It is the result of preparation, hard work, and learning from failure.

– Colin Powell

Implementation of an auth system, whether migrating from an existing one or integrating a new one, is not a trivial task. If you have a large company, the actual rollout of a system may take years. I was at a conference and someone at such a company estimated that a complete consolidation of their identity infrastructure would take 3-5 years, even with unlimited budget.

Rather than poorly covering an implementation process deeply tied to your applications and userbase, this guide will focus on major features to consider when you are in the implementation phase. A detailed implementation plan is beyond the scope of this book.

In this section, you'll learn:

- The difference between multi-tenant and single tenant auth systems
- How to make sure your auth system can scale to meet your needs
- When it makes sense to self host critical infrastructure such as an auth system
- Migration options for moving auth data from an old system to a new one
- Best practices around registration forms

Let's jump in.

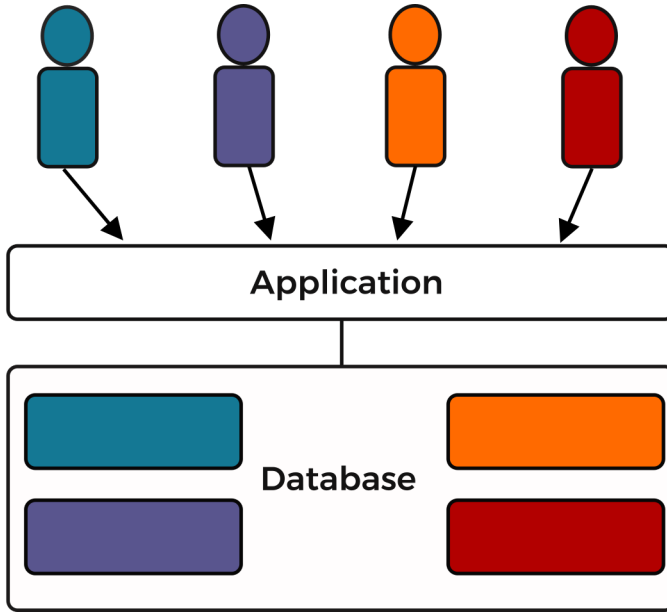
Multi-Tenant Vs Single-Tenant IDaaS Solutions

By Daniel DeGroff

The last few years have brought an explosion of IDaaS (Identity as a Service) solutions giving developers a wide range of choices for how they manage their users' registrations, logins, and identity. It makes sense. Identity and access management are critical components of many applications, but building, testing, and maintaining a secure in-house solution can take weeks or months of senior developer hours. You don't build your own database, so why build a custom identity solution? And just like databases, there are many identity solutions to fit the different requirements of every application.

One important factor to consider when comparing identity platforms is whether you need a multi-tenant or single-tenant identity solution. The choice depends on your business model and requirements. Regulatory compliance, security, data management, and upgrade control are important considerations that will ultimately define the most effective solution for your company. Which trade-offs are you willing and able to make? Read some of the most important considerations below and decide which approach fits your business.

Multi-Tenant



Multi-tenant solutions.

Simply put, multi-tenant is an architecture where multiple companies store their data within the same infrastructure. The entire system can span multiple servers and data centers, but most commonly data is co-mingled in a single database. The tenants are logically isolated, but physically integrated.

Benefits

Cost reduction - One of the big drivers of multi-tenant IDaaS solutions is cost. The sharing of infrastructure and resources across many companies significantly reduces the overhead of the service provider, and as a result, lowers the costs imposed on customers.

Automatic upgrades - Multi-tenant systems ensure that software updates, including security patches, are rolled out to all customers simultaneously. This standardizes software versions utilized by customers and eliminates version control issues.

Instant on-boarding - In most cases, new customers can be set up by creating a new logical tenant. No new servers are provisioned and software installation is not required, which makes this process instantaneous.

Drawbacks

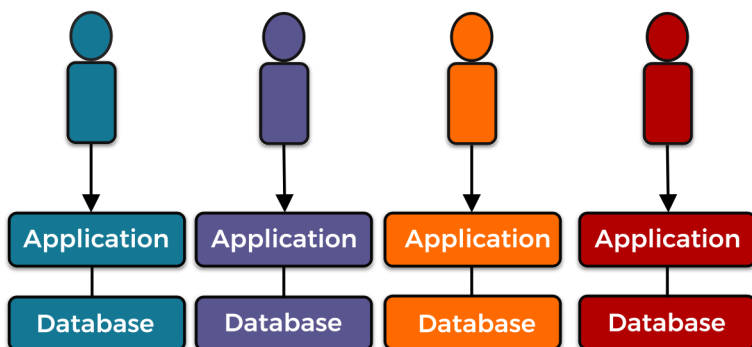
Performance - One tenant's heavy use or load spike may impact the quality of service provided to other tenants. In addition, when software or hardware issues are found on a multi-tenant database, it can cause an outage for all customers.

Security risk - If a hacker gains access to one tenant's data, they can access data from every tenant because all data resides in a single database. This is an important consideration because every tenant is relying on the security practices of the weakest tenant. According to InfoWorld, "System vulnerabilities have become a bigger problem with the advent of multi-tenancy in cloud computing. Organizations share memory, databases, and other resources in close proximity to one another, creating new attack surfaces."

Single point of failure - If the multi-tenant system goes down (and they do regardless of what salespeople will tell you), EVERYONE goes down. Then all tenants sit and wait until the cause is determined and fixed.

Cross-tenant data leaks - In a multi-tenant system, simple programming errors can lead to the data from one tenant leaking into other tenants. This is specifically true for APIs. This not only poses a security risk, but it also be a legal issue since many new data privacy regulations specifically state that cross-tenant data leaks should be prohibited.

Single-Tenant



Single-tenant solutions.

In a single-tenant architecture each company, or tenant, has their own instance, separate from any other customer. With a single-tenant solution the risk of another business accidentally receiving another customer's user data is eliminated.

Benefits

Enhanced security - Single-tenancy delivers true data isolation resulting in maximum privacy and enhanced security. The possibility of data leakage between tenants, whether accidentally or through sabotage, is removed making this architecture a popular choice for large enterprises. To increase security, customers can implement a firewall at any layer to protect data. For example, the identity provider APIs can be located behind a firewall while the OAuth login system resides in the public facing network.

Regulatory compliance - Enforcing regulatory requirements is easier due to complete control of the environment. If your company policy does not allow data to be transmitted outside of your country (i.e. General Data Regulations or GDPR regulations) a multi-tenant

solution needs to be specifically designed for this. A single-tenant solution makes this as simple as installing the software on a server in Germany. Similarly, compliance with regulations such as PCI, HIPAA and SOC2 is simplified because data is secured, encrypted and protected separately for each tenant.

Customization - With a single-tenant architecture, the software environment can be customized to meet customer's business needs; robust plugins can be installed to maximize personalization without limitation.

Upgrade control - Customers have decision authority over the upgrade cycle. Customers can choose what updates they want to install and when. This adds flexibility for scheduling maintenance windows and downtime without impacting others.

Data recovery - Data extraction is an important consideration that is often overlooked. If a service is acquired or shutdown it's wise to consider how you will retrieve your data in advance; it is easier to export data from an isolated, single-tenant cloud.

Drawbacks

Cost - Since this is not a shared infrastructure, customers have to pay the cost of the entire system (hardware and software). However, with the rise of low-cost hosting providers like AWS and Azure, single-tenant solution platforms can have very affordable pricing options. In light of these considerations [technology advisors TechTarget stated their support of single-tenant solutions](https://searchcloudsecurity.techtarget.com/tip/What-a-CPU-cache-exploit-means-for-multi-tenant-cloud-security)³⁹: "Concerns over security in multi-tenant environments have led to many organizations choosing to switch to single-tenant infrastructure as a service to mitigate the risks of co-located data. Despite the extra cost, this is a sensible and advisable solution."

³⁹<https://searchcloudsecurity.techtarget.com/tip/What-a-CPU-cache-exploit-means-for-multi-tenant-cloud-security>

Provisioning - To set up new customers, servers must be provisioned and the software must be installed on each server. This process has been made simpler through the use of APIs provided by hosting providers and tools such as Docker, Kubernetes, and Chef.

Multi-Tenant Within a Single-Tenant Solution

The descriptions above outline multi-tenant and single-tenant solutions at the highest level of an implementation: one company, with one set of users. While this is common, there are other use cases to consider. Here are a few use cases where multi-tenant capability WITHIN a single-tenant instance provides additional flexibility to solve additional challenges.

- Private Labeled Identity
- Dev, Stage and Prod

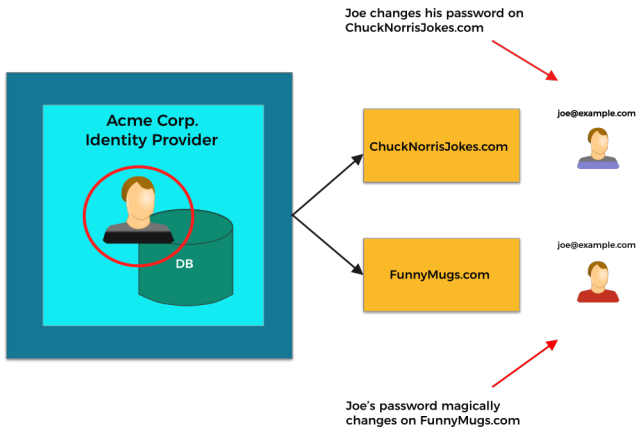
Private Labeled Identity

The proliferation of powerful cloud platforms has made Software as a Service (SaaS) solutions common for all sizes and types of businesses. Functionally, this is simply a multi-tenant architecture: they have many clients using a single instance of their platform.

Let's assume Acme Corp. sells a marketing communication platform that provides commerce, customer relationship management (CRM) and user management to small companies.

Joe uses two different websites, `funnymugs.com` and `chucknorrisjokes.com`. Both of these websites buy their software from Acme Corp., and Acme Corp. provides a single identity backend that stores a single user object for Joe.

Joe will be very (unpleasantly) surprised if he changes his password on chucknorrisjokes.com and magically his password is updated on funnymugs.com. This diagram illustrates why this unexpected password change occurs when Acme Corp. is storing single user objects.



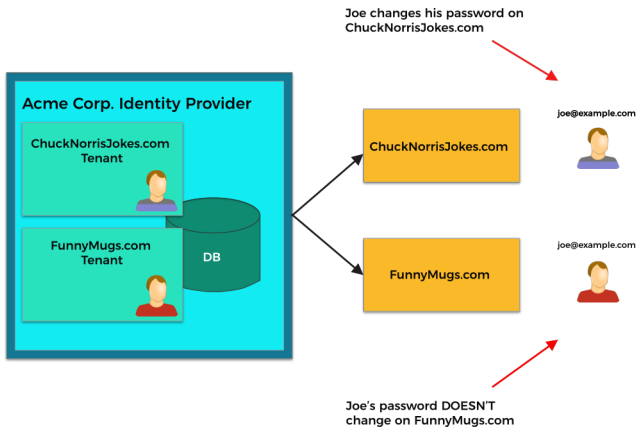
A password change in a single-tenant environment.

This would be a poor user experience and not ideal for Acme Corp. While both users are technically Joe, he is not aware of this nuance in the method that Acme Corp. built their platform.

In most cases we want a user to be considered unique by their email address. You can think of this the same way that a Gmail address works. There is a single Google account that can use a single set of credentials to gain access to Gmail, Blogger, YouTube, Google Play, (ahem..) Google+, Google Analytics and more. Each of these applications are considered an authenticated resource, and Google simply grants the user permission to each of them based on their credentials.

This is a one to many Applications model. A single user can register or be granted permissions to multiple Applications. This is also

where single sign-on comes into play. You login once and then you can access each Application without the need to log into each one separately.



A password change in a multi-tenant environment.

However, as you just saw with Acme Corp., when the platform is opaque to the end user and there is only a single identity for a single email address, surprising side-effects start to occur. In this case, what Acme Corp. needs is a way to partition each of their clients into their own namespace. This is one of the main reasons to use multiple tenants within a single instance.

In this case a tenant is simply a namespace where Applications, Groups and Users exist. Now Acme Corp. can allow Joe to create an account with the same unique email address `joe@example.com` in multiple tenants. They become separate unique identities, one per tenant. Joe can then manage his permissions, passwords, and user details discretely inside each tenant (i.e. each client of Acme Corp.). The second diagram illustrates the new layout of Acme Corp. using multiple tenants.

Dev, Stage and Prod

For this use case, we don't have multiple clients, but instead we have a single production environment.

In addition to production, we need separate environments for development, staging and QA. One option is to stand up a separate deployment of the identity platform for each of these environments. This ensures that the development environment doesn't impact the staging environment, which doesn't impact the QA environment, and so on.

Most SaaS identity products don't solve this problem directly (or easily). Instead, they force you to sign up for multiple accounts, one for each environment. That approach works, but now you have multiple accounts that may or may have a subscription fee associated with each of them. Plus, each account has separate configuration, credentials, and URLs that need to be maintained separately. And if you need to set up an additional environment, it could take quite a bit of work to get it configured properly.

Leveraging tenants in this scenario is a big win because it allows a single instance to service multiple environments, which reduces complexity, infrastructure costs, maintenance and more. You could go as far as letting each developer have their own tenant so they can each develop, create and delete users without affecting the rest of their team.

Here is a specific and common scenario: a customer has completed their integration, written all of their code, written all of their tests and is ready to move into production. If this same customer now wants to use tenants only for staging, test and QA, they can do so without any code change.

This approach is possible when you can authenticate an API request for a particular tenant with a unique API key. This way, none of your API requests change, none of your code changes, you simply

load your API key from an environment variable or inject it based on your runtime mode. Locking an API key to a tenant means that only Users, Groups and Applications in that tenant will be visible to that API key.

To provide you an example of how an API request can be scoped to a tenant, consider the following code that leverages FusionAuth's tenants to illustrate the principle. (Confirm that you can use tenant scoped API keys with your auth system provider.)

This code retrieves a user by email address in FusionAuth. It uses the API key:

```
1 5EU_q5unGCCYv6w_FipDBFevXhAxbRGaRYoxK-nP6t0
```

This key is assigned to the tenant `funnymugs.com`. As you can see, the API call finds Joe successfully.

```
1 FusionAuthClient client = new FusionAuthClient("5EU_q5unG\  
2 CCYv6w_FipDBFevXhAxbRGaRYoxK-nP6t0", "http://localhost:90\  
3 11");  
4 ClientResponse<UserResponse, Errors> response = client.re\  
5 trieveUserByLoginId("joe@example.com");  
6  
7 // API response is 200, success  
8 assertEquals(response.status, 200);
```

Next, we can update the API key to:

```
1 BwLzGhDTYtswDq9hK-ajohectZjFpMvmLeDT1mfiM54
```

This is assigned to the tenant `chucknorrisjokes.com` and that tenant doesn't contain a user with the email address `joe@example.com`. By changing the API key, we have scoped every FusionAuth API call to a different tenant. If you use tenants in this way, you get the best of both worlds.

```
1 FusionAuthClient client = new FusionAuthClient("BwLzGhDTY\  
2 tswDq9hK-ajohectZjFpMvmLeDT1mfiM54", "http://localhost:90\  
3 11");  
4 ClientResponse<UserResponse, Errors> response = client.re\  
5 trieveUserByLoginId("joe@example.com");  
6  
7 // API response is 404, not found.  
8 assertEquals(response.status, 404);
```

These are just some of the use cases that tenants can help solve, but there are many more. For example, when multiple firms merge or are acquired, they often need to combine multiple legacy user databases into one system. Tenants can be leveraged to do this with very low risk over time while preserving the original data from each firm.

Conclusion

There are benefits and drawbacks to both single-tenant and multi-tenant systems. Ultimately, a company must decide what is most important to their business and what can be sacrificed. Is cost a primary driver? Does your industry vertical have unique regulatory constraints? Is security critical for the type of data you are storing? Take the time to explicitly define your specific requirements, and then select the solution that best fits your needs.

Daniel DeGroff is FusionAuth's CTO.

Making Sure Your Auth System Can Scale

By James Hickey

Have you ever worked on a software system that stored passwords in plain text? I once worked on a product that stored encrypted plain-text passwords. Thankfully, this system was eventually fixed to store hashed passwords!

To make password hashes harder to compute, a standard technique is to hash passwords thousands of times before storing them in a database. But there's a trade-off: you get better security at the cost of a performance hit.

This trade-off lies at the heart of authentication. More robust security often means changes to how scalable your solution is. An increase in web traffic might expose these performance issues. If you've split your system into microservices, then you'll be faced with other kinds of challenges, like how to authenticate services-to-service HTTP requests and share authentication logic across services. In fact, the second item on both current OWASP [Web Application Security Top Ten Risks](https://www.cloudflare.com/learning/security/threats/owasp-top-10/)⁴⁰ and [API Security Top Ten Risks](https://owasp.org/www-project-api-security/)⁴¹ is *broken authentication*.

The trade-off between security and performance lies at the heart of authentication.

To be a responsible and effective software engineer, you need to know how to deal with these scalability concerns while keeping

⁴⁰<https://www.cloudflare.com/learning/security/threats/owasp-top-10/>

⁴¹<https://owasp.org/www-project-api-security/>

your application's authentication secure. You'll get some tips on how to scale your authentication functionality and make sure it can meet the demands of your customers.

Why Scaling Auth Is Hard

There's a push-pull relationship between robust security and scalable solutions. Since security is so critical, and frankly nonnegotiable, you'll have to grapple with the challenges of scaling your authentication.

Let's look at a few of these challenges, including hashing performance, chattiness, additional security, and uptime.

Hashing Performance

As mentioned, a standard in securely storing passwords is to iteratively rehash the password X times. The amount of rehashes is often called the work factor.

Having a work factor that's too low ends in hashes that are easier to attack. In the event that an attacker was able to obtain a password hash (via a breached database, SQL injection attack, etc.), it would take significantly longer for an attacker to calculate that hash. This gives more time for your company to inform users to change their passwords. It might even mean that the attackers are only able to "break" (i.e., compute) a smaller number of password hashes altogether.

However, a work factor that's too high can lead to performance degradation. This may alternatively give room for attackers to abuse these CPU-intensive operations and perform denial-of-service attacks by spamming a web application's login.

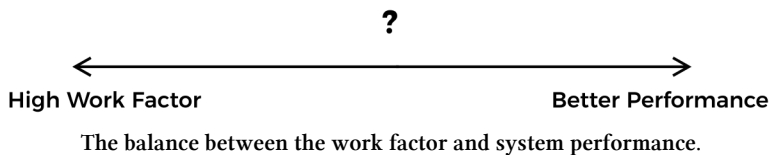
As modern computers quickly increase in power, so must your authentication solution increase its work factor (i.e., the number

of hashing iterations). Again, this makes sure that password hashes are being generated securely enough. This means you may have to update existing systems and perform a migration of sorts to increase the number of hashing iterations if this hasn't been updated for quite a while.

OWASP suggests⁴² increasing the work factor by one (i.e., doubling the number of iterations) every eighteen months:

“Taking Moore’s Law (i.e., that computational power at a given price point doubles every eighteen months) as a rough approximation, this means that the work factor should be increased by 1 every eighteen months.”
–Upgrading the Work Factor, OWASP

Do you know how well your application is balancing its work factor versus performance?



Chattiness

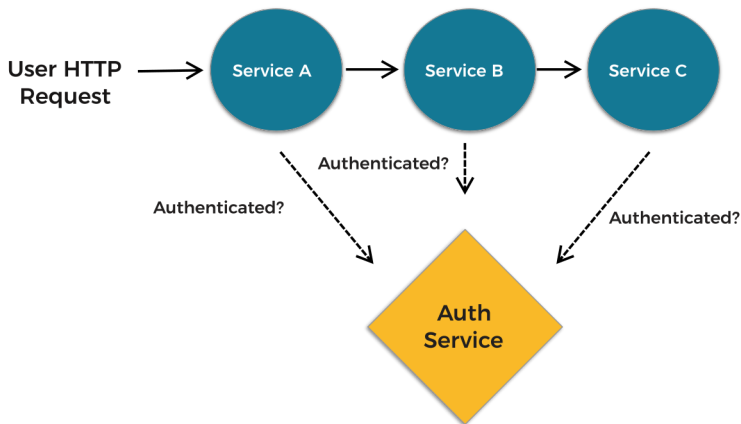
In distributed systems, I've often seen developers implementing dedicated services that manage authentication functionality. These services are often put behind an HTTP API. What could go wrong?

Well, without having a grasp of the pitfalls of common issues with distributed systems, you can quickly end up with “chatty” services⁴³.

⁴²https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#upgrading-the-work-factor

⁴³<https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/chattiness.html>

Imagine an HTTP request or asynchronous process where multiple distributed services need to collaborate. Each service needs to make sure the user is authenticated, right?



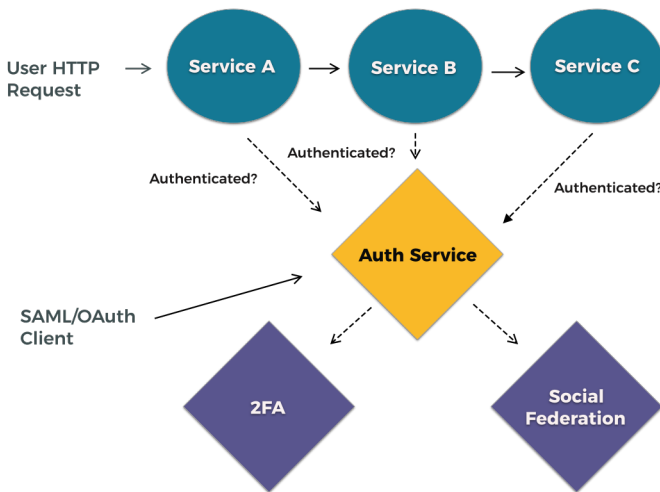
Auth microservice used by other services.

A system where individual services each send an HTTP request to the auth service means a lot of extra network latency and CPU/memory usage. And don't forget the additional costs that come from needing more powerful hardware to handle the extra load.

Additional Security

While basic password-based authentication has its challenges, compliance with standards like SOC 2 or ISO certification often require that software systems implement additional security features like two-factor authentication, federation, etc.

In the scenario described earlier, adding some of these extra security features will degrade scalability even further. More network hops are involved, and more external clients are making requests to the authentication service.



Single sign-on auth with increased traffic.

Implementing performant systems takes more thought than simply throwing everything into a “microservice”.

Uptime

If your authentication functionality is unavailable for any given time, it inevitably means that your application is unavailable, too.

This makes authentication a difficult problem: not only must it be secure and scalable, but highly available, too. Again, this means more hardware, more servers, and more money.

How Can You Effectively Scale Your Authentication?

Now that you've seen some of the challenges that exist for ensuring secure authentication, let's look at how to attack these challenges. We'll cover performance testing, modern hashing algorithms, rate limiting, database scaling, caching session data, auth tokens, API gateways, and third-party auth services.

How Fast Should It Be?

First things first: how scalable and performant should your system be? Do you have any SLA agreements with your clients?

OWASP recommends⁴⁴ that computing a hash should not take more than one second. Does that make sense for your customers?

“As a general rule, calculating a hash should take less than one second, although on higher traffic sites it should be significantly less than this.” –Work Factors, OWASP

Whatever you decide, you need to be able to test the performance of your authentication functionality. One specific tool I love is the open-source HTTP benchmarking tool [bombardier](https://github.com/codesenberg/bombardier)⁴⁵. This is a great tool since it enables you to test and benchmark the entire HTTP pipeline of your authentication endpoints. Perform such benchmarks regularly, preferably on each release of your applications, to ensure there are no performance regressions.

⁴⁴https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#work-factors

⁴⁵<https://github.com/codesenberg/bombardier>

Modern Hashing Algorithms

Most web frameworks have authentication out-of-the-box for you. Modern hashing algorithms are designed to be very secure and performant.

OWASP recommends using an algorithm such as Argon2, PBKDF2, or Bcrypt.

.NET, for example, has historically [used PBKDF2 with an increase in iterations over time](#)⁴⁶.

Rate Limiting

So now you're using a modern hashing algorithm to hash user passwords. What happens when an attacker sends in a flood of HTTP traffic to your authentication endpoints?

Individual HTTP requests seem to work fine, but hundreds and thousands of HTTP requests over a small window of time can bring a system to its knees. Since computing a hash is CPU intensive, this is often an easy target for attackers. How could you defend against this?

Rate limiting is an effective defense against these kinds of attacks. It also ensures the performance of your application's authentication will be more stable on average.

Rate limiting simply means restricting an IP address so it can only try signing in to your application once every X milliseconds or seconds. This technique can help ensure that individual attackers can't flood your system with authentication requests. Depending on your authentication system and architecture, you can rate limit at:

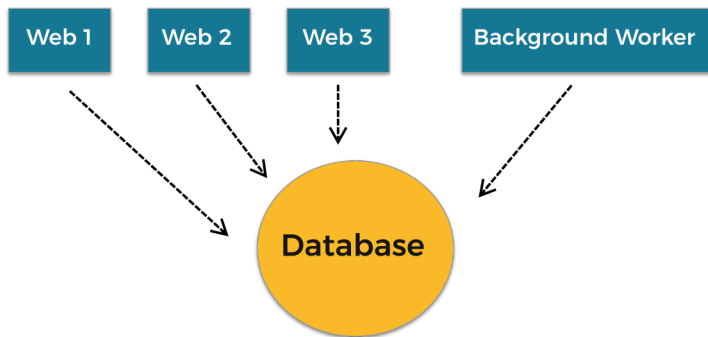
- The network layer using an ACL or a CDN

⁴⁶<https://github.com/dotnet/aspnetcore/blob/60244785aa62dd2a23a82876d26bb0a7aa6f32e8/src/Identity/Extensions.Core/src>PasswordHasher.cs#L29>

- A proxy, such as nginx, in front of your auth system
- Inside the auth system itself

Database Scaling

Perhaps your application has grown to the point where you might need multiple web application processes running behind a load balancer. All of these processes are communicating with the same database.

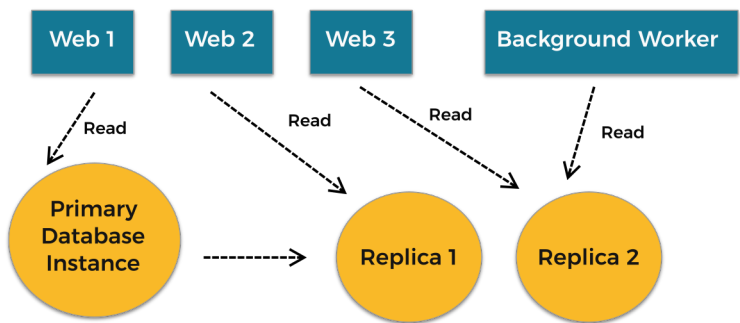


Database load increases.

When your customers (or attackers) try to log in to your application, every login attempt will need to fetch the password hash from the database. Eventually, this may expose that the database has become a bottleneck in your system.

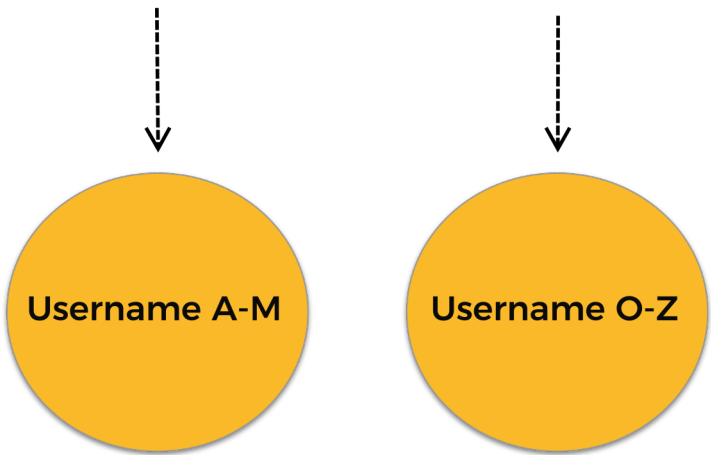
Typical database scaling techniques, like using read replicas or sharding, can help make your application and its authentication functionality more scalable.

Read replicas are like having dedicated secondary instances of your database available for reads. All writes will go to the primary database instance, and changes will be pushed out to the read replicas behind the scenes. The overall traffic to your database will be spread across all instances/replicas.



Read replicas reduce load on primary database instance.

Sharding is another technique where individual records are stored in separate storage locations or database instances based on some algorithm or method of segmenting data.



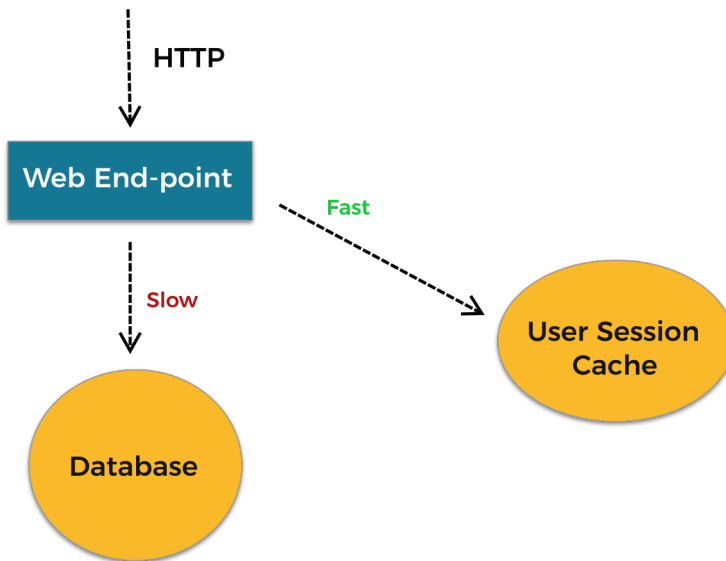
Sharding database instances based on username first letter.

Caching Session Data

One of the most common performance optimizations you might be able to make in web applications is to cache your user session data. Almost every HTTP request to a typical web application will need to verify that a valid session for the user exists.

Out-of-the-box web frameworks usually store user session data in a database. While database scalability optimizations will help (like the aforementioned read replicas and sharding), using a dedicated caching technology can significantly increase the performance of fetching user session data.

[Redis](https://redis.io/)⁴⁷ is a fantastic technology that's often used as a distributed cache for solving issues like this.



In-memory key-value databases can be used to store user sessions.

⁴⁷<https://redis.io/>

By caching user sessions in this way, you can reduce the overall load of the primary database and/or server that houses your authentication data (e.g., password hashes). If your authentication, for example, is a microservice, then this technique could have a large impact on how well your authentication system can scale.

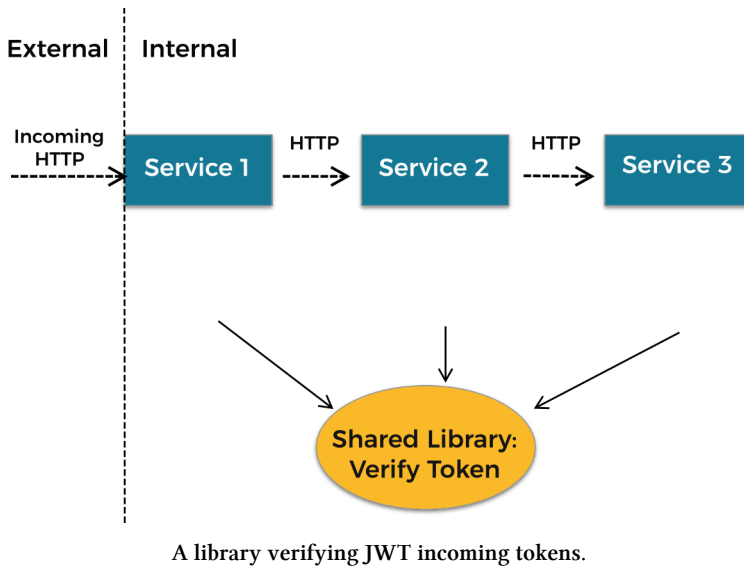
Auth Tokens

One of the challenges mentioned earlier was around microservices having to ensure that any given request is authorized to perform the requested operation. The same would apply to any type of distributed system that's using the same distributed authentication/authorization HTTP service.

The main issue is that your authorization service becomes a bottleneck. Everyone needs to send an HTTP request to this service to verify its own external requests coming in. What if there was a way you could verify that the request was authorized without having to make all those extra HTTP requests?

Digitally signed tokens such as JWTs can be verified by backend server code without communicating with the auth system. By using a private/public key or a secret, you can enable various services to verify a given token within the same process and avoid those extra network hops.

One way to do this is to create a shared code library that can run in-process and verify incoming JWT tokens.

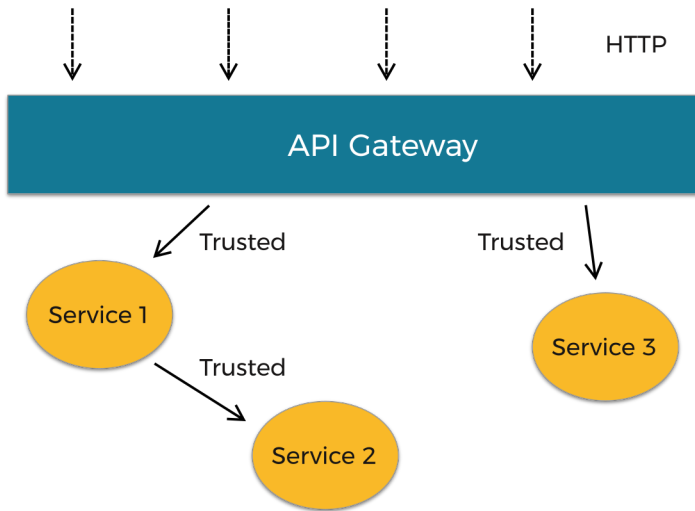


API Gateways

There may be situations where you have dedicated web servers or services that handle incoming HTTP traffic. That service will verify that the requested action is allowed and then delegate work to one or more internal services.

That is called an API gateway. For large solutions, often a dedicated API gateway service from Amazon, Azure, etc. might be used. You can think of it as a lightweight “bouncer” that makes sure incoming HTTP requests are allowed in. It can perform authorization checks once.

If all your internal distributed services are behind a protected network, then they can trust that the API gateway has already vetted the client. This removes the need for a service to communicate with the auth system to ensure the client is authorized, leading to less work for the authentication system and for the service.



API gateway authenticating HTTP requests up front.

Third-Party Services

Many organizations choose to leverage third-party auth services like FusionAuth, Okta, or Auth0. Building your own SSO, user role management, two-factor authentication, token signing, performance optimizations, and so on is tedious, time-consuming, and potentially dangerous. By leveraging auth services created by experts, you don't have to worry about many of these scalability issues. You also get the peace of mind knowing that your authentication is done properly.

You can combine some of the scalability techniques covered here with third-party services, too. For example, FusionAuth can generate JWT tokens and still allow you to [verify tokens in your server code](#)⁴⁸.

⁴⁸<https://github.com/fusionauth/fusionauth-jwt#verify-and-decode-a-jwt-using-hmac>

Conclusion

If you think that authentication and authorization are hard, it's because they are!

Considering that broken authentication is the No. 2 OWASP security risk for web applications and APIs, you want to make sure that you do authentication well and that it can scale as your application grows.

Make sure your engineering team is willing and able to spend the time to ensure that your authentication system doesn't become a bottleneck; alternatively invest in a third party system which can offload these concerns from your team.

James Hickey is a Microsoft MVP with a background in fintech and insurance industries building web and mobile applications. He's the author of "Refactoring TypeScript" and the creator of open-source tools for .NET called Coravel. He lives in eastern Canada and is a ten-minute drive from the highest tides in the world.

When to Self-Host Critical Application Architecture

By Matthew Fuller

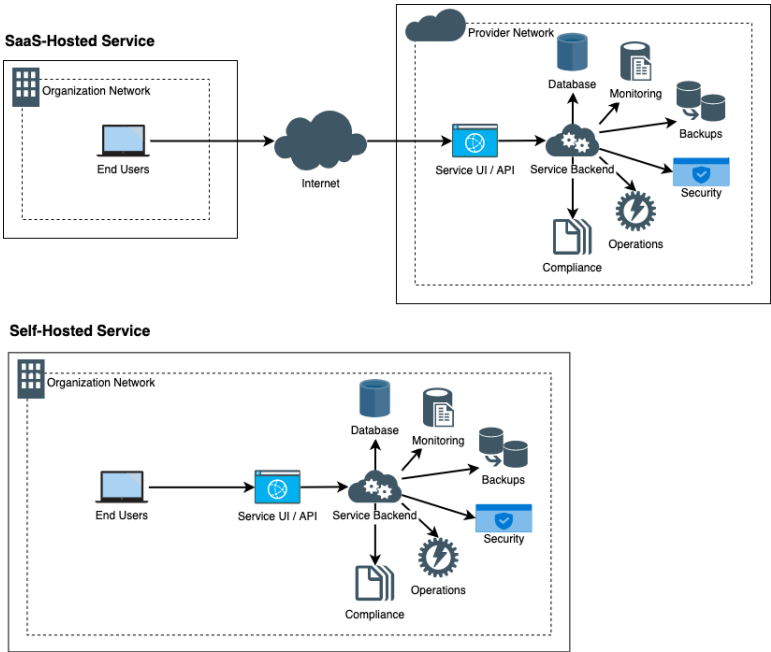
In April of 2021, Auth0, an identity provider powering authentication for hundreds of websites, experienced an [hours-long outage](#)⁴⁹. During the outage, users could not access their authentication portals and many of their websites were rendered unusable due to the broken authentication flows.

Auth0 joins Microsoft, Amazon, GitHub, and a growing list of cloud and SaaS providers who have experienced major outages in recent years. If you and your operations teams have found yourselves feeling helpless during these outages, you may wonder whether self-hosting the service could provide better reliability for your organization.

Although self-hosting may seem like a simple decision at first, especially in the midst of a major outage, there are a considerable number of factors at play that may complicate your choice. It's important to consider the financial, operational, security, and legal implications before deciding to host a service yourself instead of paying for a managed SaaS counterpart. Choosing to self-host a critical application will impact many departments of your organization and even affect how your organization scales and grows its teams in the future.

⁴⁹<https://news.ycombinator.com/item?id=26880147>

Evolution Of Self-Hosting



A diagram comparing self-hosted to SaaS deployment models.

Self-hosting is not a new concept; in fact, it was the default choice prior to the more recent proliferation of SaaS. As software companies discovered the benefits of reliable and recurring revenue from subscription services, and organizations discovered the benefits of consuming specialized software produced by focused teams, the industry shifted from large, release-driven, self-hosted software to continuous releases delivered as a service (SaaS is an acronym for “software as a service”).

This trend was especially profound in the consumer software business but quickly spread to business software as well. Today, many services are “SaaS first,” with only a few offering a self-hosted op-

tion that is usually limited to certain customers and plans. [GitHub](#)⁵⁰ and [Bitbucket](#)⁵¹, for example, only offer self-hosted versions of their software as part of their Enterprise or “Datacenter” plans, which are significantly more expensive than their commercial SaaS offerings.

Taking a different approach, some vendors, such as database providers MongoDB and Elastic, produce both open-source and commercial versions of their software. The open-source versions are provided “as-is” but can be self-hosted and run without licensing costs. These same platforms can alternatively be used as subscription-based SaaS services, in which the company manages the underlying infrastructure and handles all other aspects of operating it in exchange for a recurring fee.

The Benefits Of Self-Hosting

When considering which option is best for your individual organization, it’s important to weigh the pros and cons of each. Self-hosting comes with its fair share of benefits, chief among them compliance friendliness and data ownership.

Self-Hosting To Maintain Compliance

While many organizations will have the choice between SaaS and self-hosted options, others may not have this luxury. Due to regulatory, compliance, or legal factors, self-hosting may be the only available option. These situations are especially prevalent in heavily regulated environments such as banking, finance, law, and healthcare.

Even if your organization can use a SaaS, there may be more restrictive internal policies set by your legal or risk management

⁵⁰<https://github.com/enterprise>

⁵¹<https://www.atlassian.com/software/bitbucket/enterprise/data-center>

departments that dictate whether a service must be self-hosted. For example, you may be permitted to use a SaaS to track IT department tickets but must use self-hosted solutions when managing employee payroll.

Before continuing to evaluate SaaS options, consider the following questions:

- Will this application handle sensitive data such as the personal information of employees or customers, financial records, payment information, or health information?
- Is your organization subject to any compliance or regulatory requirements such as PCI, HIPAA, or FedRamp?
- Do your customers require that all systems handling their data be located within a specific geographic region?

While answering any of these questions affirmatively does not preclude you from using a SaaS version of an application, it will place additional constraints on which application is used. For example, while a company such as Atlassian, the creators of Jira, may be large enough to provide a SaaS solution that is EU compliant by storing its data entirely within the European Union, smaller companies may not be able to afford the additional infrastructure.

Even if you do not have specific compliance obligations, using a [vendor security checklist](#)⁵² can help your security and risk management teams decide if the SaaS version of an application is suitable for your environment.

Owning Your Data

Data ownership is an increasingly complex topic as companies outsource various parts of their development, deployment, and runtime environments to third parties, who in turn outsource the

⁵²<https://securityscorecard.com/blog/vendor-risk-management-questionnaire-template>

same components to additional vendors. It's turtles all the way down. For example, your customers' data may live in a database hosted in Amazon Web Services, operated by MongoDB, which out-sources metrics to Datadog who in turn hosts their infrastructure on Microsoft Azure.

For organizations operating in industries with stringent data requirements, or who serve customers in those industries, self-hosting may be a feasible option to reduce this sprawl of data ownership. Self-hosting ensures that you have complete control over each piece of the application stack that processes your customers' data.

Performance and Availability

This control can also help when you have strict performance requirements. Gaming and financial services companies, for example, may find the control of self-hosting their data gives them a performance edge their customers demand.

As mentioned initially, using a SaaS provider such as Auth0 can result in significant impact to your application availability, especially if it is in the critical path for your users. While such major outages are rare, they are extremely frustrating for everyone involved. Self-hosting critical components can empower your team to address availability challenges, whether through process, automation, or infrastructure improvements.

You can also control the upgrade cycle when you self-host. If you need a feature in the latest release, you can upgrade. If you'd rather remain a few versions back from the bleeding edge, you have the control to slow-roll upgrades.

Controlling Costs

Self-hosting also ensures greater control over the costs of each component in the application and its supporting infrastructure. Unlike SaaS services, which coalesce the costs of infrastructure, data transfer, backups, and other operational management activities into one bill, self-hosting exposes you directly to these costs. In doing so, it gives you greater flexibility to negotiate costs with your hardware and network vendors, sign long-term contracts, or distribute costs across multiple applications through chargebacks.

While this may not result in favorable rates for smaller organizations, it can result in massive savings for larger enterprises who are willing to commit to longer timeframes and larger network or data utilization contracts.

The Downsides Of Self-Hosting

While self-hosting can provide you with frictionless data ownership and keep your company compliant, there are some downsides to consider. The cost of hosting every needed application component can really start to add up. There is also the operational excellence of SaaS providers that internal teams might not be able to match.

The Hidden Costs

The direct price of a software solution is a primary driver of its adoption. Yet in the context of choosing a self-hosted option, the cost goes far beyond the advertised upfront price paid to a vendor. Depending on the vendor's business model, the cost could be impacted by many factors exceeding the base subscription price, including support contracts, overage fees, and solution architecture or consulting rates.

Some applications, especially those produced by open-source or [open-core](https://en.wikipedia.org/wiki/Open-core_model)⁵³ vendors, may not have any upfront licensing costs. In these cases, you should evaluate whether your organization possesses the operational expertise and hardware needed to run the application according to your uptime and availability requirements. In particular, hiring and retaining operational expertise can be expensive, both in terms of dollars out the door and opportunity costs. If your team is spending time improving the availability of your self hosted service, what are they unable to work on?

You could consider a hybrid approach of self-hosting the software but paying for a support contract to ensure expertise is on-hand at all times to assist with installing, managing, upgrading, and troubleshooting the application.

In general, the costs for SaaS applications are not directly comparable to their self-hosted equivalents. This is primarily because SaaS pricing tends to amortize higher costs over periodic payments. For example, it may cost the SaaS provider \$1,000 in hardware and support staff to onboard a customer to an application that is billed at \$500 per month. The upfront monthly payment can be more approachable for smaller companies, despite costing more than self-hosting in the long-term.

These costs can be difficult to compare directly because they are influenced by scale, staffing, existing contracts, and long-term business projections.

Operational Excellence

The popularity of SaaS services has been driven in part by their ease of use. For most deployments, there is no hardware to deploy and no network to configure. Accounts can often be provisioned in minutes, for free or with a business credit card, and configured quickly afterward. The SaaS provider handles the security of the service,

⁵³https://en.wikipedia.org/wiki/Open-core_model

upgrading and maintaining the underlying infrastructure, taking backups, complying with regulations, monitoring the service, and restoring access in the event of downtime. As the developers of the software powering the SaaS application, the SaaS provider also has the in-house expertise, the focus and the incentive to quickly resolve bugs and implement feature requests.

Leaders of organizations attempting to self-host an application with these same expectations must ask themselves whether these responsibilities can truly be handled in-house. Consider some of the following questions to determine your organization's operational readiness:

- Will the application need to be available 24/7 with an uptime SLA?
- If so, do you have a global DevOps or SRE team prepared to respond to outages?
- Do your teams have the experience to install, manage, upgrade, and troubleshoot the application?
- Do you have the ability to scale the application using your network and infrastructure?
- What monitoring will you put in place to observe the application and alert on potential issues?
- Are you familiar enough with the performance metrics of the application that need to be monitored to ensure you'll be able to manage it when usage grows?

If you aren't certain about the answers in the hypothetical, ask the same questions about any currently self-hosted critical internal application.

Operational excellence is an ongoing, and potentially expensive, requisite for operating self-hosted software. You will need to factor salaries, training, documentation, and internal support into any staffing plans for managing the service, including increases for potential upticks in internal adoption and larger-scale operations as that adoption grows.

Legal Considerations

While the legal considerations of deploying SaaS and self-hosted services may feel nuanced in comparison to larger operational issues, understanding their implications is paramount to a successful deployment. When self-hosting, there will be legal terms dictating what you can and cannot do with the software. It's crucial to have your legal department review the licensing terms and contracts prior to implementation.

In some rare cases, the terms or licenses for a service may change in future releases, potentially impacting your ability to continue using it. Recently, Elastic, the makers of the popular Elasticsearch database, [underwent a licensing change](#)⁵⁴ to limit the ability for competitors to use future versions of their software. While this specific change may not have a direct impact on your self-hosting capabilities, it does demonstrate how your company may be at the mercy of the vendor developing the software.

A Non-Permanent Choice

Planning a SaaS or self-hosted deployment can feel like a decision set in stone, especially when multiple departments within the company are involved in its rollout. But it's important to remember that many offerings may support hybrid or evolutionary approaches to deployment. For example, this may allow you to move from a SaaS offering to a self-hosted option over time with your data and settings intact.

Conversely, you may also proceed in the opposite direction and begin by self-hosting an open-source or free version of the software before realizing that the operational costs are too significant and

⁵⁴<https://www.zdnet.com/article/elastic-changes-open-source-license-to-monetize-cloud-service-use/>

moving to a SaaS solution. While you may have no plans to use the alternative options at the time of deployment, it is crucial to work with the vendor to understand what your options are in the future should you decide to change models.

Conclusion

Deciding between a SaaS or self-hosted version of an application may seem like a daunting choice for any company. However, with a bit of planning, cost modeling, and legal analysis, you can consider all of the factors important to your organization and make the choice that will allow you to deploy quickly, scale seamlessly, and operate within your expected budget.

As your organization and its usage of the service evolves, continuing to evaluate these options will ensure reliable operation for years to come. And remember, if you decide later on that you've made the wrong choice, you can always change your mind and switch.

Matthew Fuller is an accomplished founder and entrepreneur in the cloud security space. He began his career developing, deploying, and supporting cloud-native applications for startups and enterprises, during which he saw the difficulties in developing truly secure workloads in complex cloud environments. This led him to found CloudSploit, an open source and commercial cloud security configuration monitoring service, which was acquired in 2019 by Aqua Security.

Migration of Auth Data

By Dan Moore

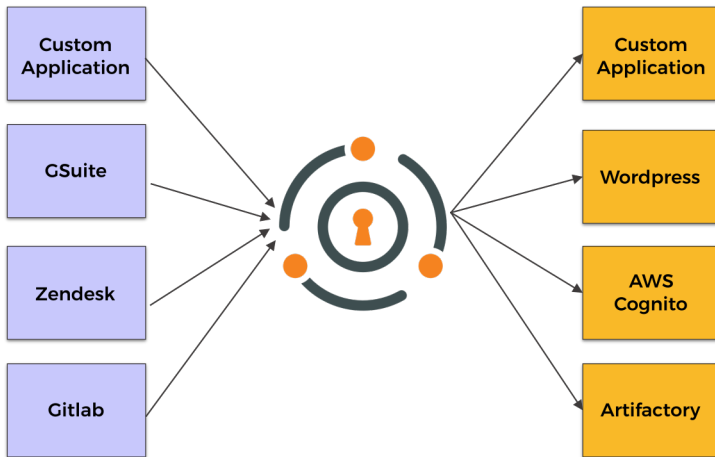
Migrating data used for auth is a pain. People have to login to your application, and maintaining access is critical. User data is often siloed and changes at the whim of your users. Why would you endure the data migration pain? There are many reasons why you might decide to migrate your user data.

Why Migrate User Data

There are many reasons to migrate user data. You might:

- Have outgrown a homegrown auth system.
- Need a single view into customers', users' or employees' profile data across all applications.
- Have an outdated user management system with an impending license renewal and are looking for a better cost structure.
- Be looking to integrate both COTS applications and home grown internal apps and want a centralized auth system which supports standards.

This last one has many benefits, and is referred to as the bottleneck architecture:



A bottleneck architecture enabled by a modern auth system.

Types Of Migration

There are three approaches to user data migration. Every migration involves a cutover for a user, where they authenticate with the new system and not with the old one. While each approach differs in implementation details, a good way to consider which is right for you is to look at how many cutovers you want to handle.

You can:

- Migrate everyone at once, also known as a “big bang” migration. With this approach, you have one cutover.
- Segment your users and migrate each segment. With this method, you have multiple cutovers, each with a natural chunk of users.
- Migrate when a user authenticates, also known as a “slow migration”. With this choice, there are two cutover points.

The first is the application cutover, which happens when you direct users to the new system for authentication. Then, at each user's login, the data is migrated and the user's system of record changes. Therefore there are many data cutover events.

Each of these approaches migrates user and other account data into the new system from one or more other systems of record. Pick the one which works best for your situation after ensuring your new auth system supports it. Let's examine each approach in more detail.

Note that this guide focuses on migrating users. Don't forget to plan to migrate anything else required by your application. These could include:

- Groups
- Roles
- Application configuration
- Permissions
- Miscellaneous configuration

Plan to spend time mapping between the current system's definitions and the new system's for these other entities, as well as for user profile data.

The Big Bang Migration

With a big bang migration, you are moving all your users at one time. The exact duration varies, but there is a single cutover period. The basic steps are:

- Map user attributes from the old system to the new system.
- Build a set of migration scripts or programs.

- Test it well. Ensure that migration accuracy and duration meet your needs.
- Plan to modify your applications to point to the new system.
- When you are ready to migrate, bring your systems down or to a mode where authentication is degraded (read-only or disallowed).
- Run the migration scripts or programs.
- Perform the cutover and flip the system of record for all your users from the old system to the new.

This approach has strengths:

- If you manage the timing of auth unavailability, the migration can have minimal impact on users.
- It has a fixed timeframe. When you have completed the migration, you're done and can shortly shut down the original system.
- If you have to decommission the old system by a certain deadline, perhaps due to an upcoming license renewal or other external factors, you can plan to migrate before the deadline.
- You only have to run two production user auth systems for a short period of time; typically you'll run the original system after the cutover in case you need to roll back.
- Employees or contractors accessing user data, such as customer service reps, only need to switch their working routines after the migration is performed.

The big bang approach has some challenges, though.

- It is common to miss issues during testing because this is a unique procedure. Production systems are often different in subtle ways from testing environments.

- Any problems with the migration impact many users, since all are migrated.
- The big bang requires you to write code which you'll test intensely, use once and then throw away.
- The new auth system must be compatible with the old system's password hashing algorithm for the migration to be transparent to the end user. (An alternative is to force all users to reset their password.)

In short, this is a high risk, low outage duration, high reward solution.

Segment By Segment Migration

Segment by segment migration is the second alternative. It can be thought of as a series of "little bang" migrations. With this approach, you split your user accounts into segments and migrate each segment. Natural division points could be the type of user, source of user data, or applications used.

Such a migration lets you test your processes in production by migrating less critical, or more understanding, sets of users first. The engineering team will be more understanding of any migration issues than paying customers, for instance. You will probably be able to reuse code in the different segments migration scripts. This approach works well when you have more than one old system from which you are migrating users. In general, this approach decreases risk when compared to a big bang migration.

However, this approach is not without its issues:

- You have multiple projects, downtime periods and cutovers to manage, not just one.
- There may be no natural divisions in your user base.

- If most of the users are in one segment, this approach may not be worth the extra effort. For example, if you have one popular application and a couple nascent apps, the extra work to migrate in phases may not be useful. You won't get a real test of the migration process until you do the popular application, which is where all the risk is as well.
- This will take longer to complete, requiring you to run both old and new systems for longer.
- You'll need to consider how to handle the cutover from the old system to the new system. Depending on how you segment your users, this could be complicated and require additional development. For example, if you divide your users by type and migrate the admin user segment first, you will need some kind of proxy in front of your auth systems to send admin users to the new system and normal users to the old one.

Segment by segment migration decreases cutover risk, but in exchange requires a longer cutover timeline.

Slow Migration

This approach is a logical extension of segment by segment migration. Here, each segment is a single user. With a slow migration:

- Map user attributes from the old system to the new system.
- Set up a connection between the original auth system and the new one
- Modify your application or applications to point to the new system. This is the application cutover point, which may require some downtime.
- The new system receives all auth requests, but delegates the first such request for each user to the original user management system.

- The old system returns the information and the new one creates a new user. This is the data “cutover” point for this user.
- For this user’s subsequent authentication requests, the new auth system is now the system of record. The user has been migrated.

To implement a slow migration, the new auth system typically needs to pass the user’s auth credentials to the old system and expects the user information which is being migrated. You also need to modify applications to point to the new system before any migration starts. A slow migration has the following benefits:

- Since you are only doing a user migration at the time a user authenticates, the blast radius of a mistake is smaller; it’s limited to whoever is logging in.
- You can upgrade your password hash algorithms transparently without requiring anyone to reset their password.
- You don’t have to migrate inactive users; this lets you scrub your user base.
- You can use this opportunity to contact any dormant application users and encourage them to log in.
- There’s less downtime during the application cutover because you aren’t moving any data, only switching where users authenticate.
- You don’t have to understand all the moving pieces of the old auth system. You don’t have to understand all the business logic which goes into authentication in the old system.

However, a slow migration isn’t the right solution for every application. Issues to be aware of:

- You are passing a user’s plaintext password from the new auth system to the old one. Take special care to secure this data in transit. If possible, keep it from travelling over the internet.

- The old user management solution must be extensible or support a standard like LDAP. You may need to extend it to add an auth API and you need to understand the user account attributes.
- You have to run both systems for the duration of the migration. Depending on the state of the old user auth management software, this may be painful.
- Customer service and other internal users may need to access two systems to find a user during the migration period.
- Rollback from a phased migration is more complex if there are issues, because there are two systems of record, one for migrated users and one for users in the old system.

A slow migration is, in short, a lower risk, long duration choice.

Planning and Mapping

The first step to any successful data migration is planning. You need to know where all your data sources are, how to connect to them, and what the data looks like.

Consider edge cases. What fields are required and optional in the old system? What about the new system?

Is there a clean mapping between the data fields? The answer is almost certainly no, so think about how you are going to handle irregularities.

Let's examine a trivial example. Suppose an old auth system has a user data model with these attribute names and data types:

- `fname` - `string`
- `lname` - `string`
- `birthdate` - `string`
- `phone_num` - `string`

Assume the new system has a user object with these attributes and data types:

- `first_name` - string
- `last_name` - string
- `date_of_birth` - date
- `area_code` - string
- `phone_number` - string

When you are moving between them, you'll face three challenges.

The first is converting from `fname` to `first_name` and `lname` to `last_name`. This is pretty easy.

The second is parsing the `birthdate` field into a date format to place in the `date_of_birth` field. Depending on how clean the original data is, this could be trivial or it could be painful.

The last issue would be splitting the `phone_num` field into an `area_code` and a `phone_number`.

As this example shows, getting ready for a migration consists of many tiny choices and design decisions. Make sure you understand the data model for your users before you start the migration.

If you have the option of storing arbitrary key value data in the new system, serialize the user object from the old system and store it there. The old user model may be helpful in the future because if there was mistranslated data, you'll have the original copy.

As mentioned above, think about relationships between users and other objects. Groups, roles, historical data, and anything else the old system has tied to users. Make sure you know where this data is coming from, if it can be discarded, and if not, where it will end up. Perhaps it will be migrated to the new system, perhaps to a different datastore.

One special complexity which will require more planning is if you have user data in multiple datastores and are planning to merge the

data. Ensure you map both of the old data models into the new user data model before you write any migration code. Another is if you need to ensure a no-downtime migration, and thus put applications in a degraded read-only mode where users can log in but not update their profile data.

Fields Worth Extra Attention

There are two field types worth commenting on in more detail. The first is user ids. These are often referenced by other systems and are used for auditing, analytics or other historical purposes. If you can preserve user ids, do so.

If you cannot, plan accordingly. You may want to keep that field in the new system in an `old_user_id` field, accept the loss of this data, or build a system to map from old user ids to new user ids, to be used by any external party which depended on the old user id.

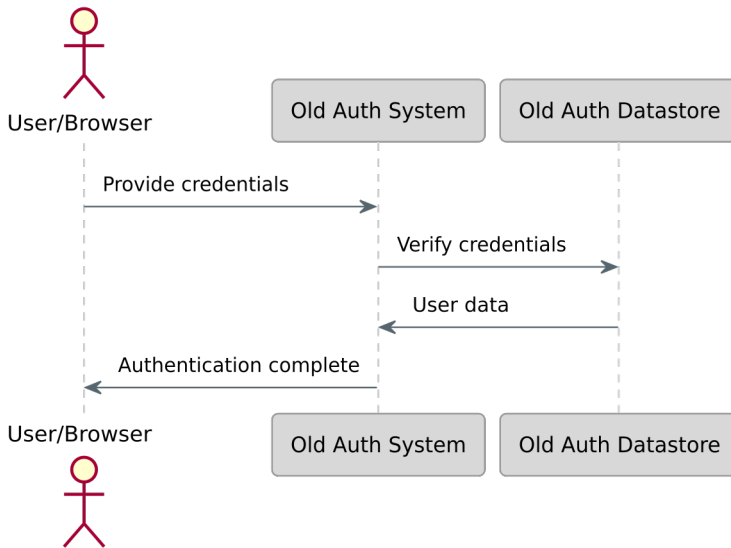
The other notable attribute is the password, and any related fields such as a salt. Passwords don't have to be migrated in a slow migration. The user will provide the password during authentication, and it can be stored in the new auth system during that process.

Big Bang Implementation

Below, find out how to migrate all your user data into an auth system with one cutover. But first, let's look at how the migration affects your users.

User Authentication Over the Migration Project

Before a big bang migration project begins, the old datastore is the system of truth:

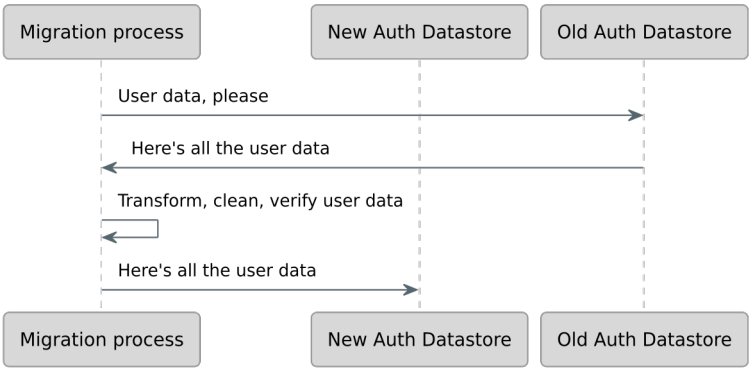


The user auth process before the big bang migration.

Users are happily authenticating against the old auth system, wherever it may be.

Then, you perform the migration. No authentication is allowed during the migration, so plan for downtime.

If downtime isn't an option for your system, you may use a read-only version of the old datastore to allow for degraded login functionality. This could be set as a feature flag or configuration option. Such a change may require releasing your application or applications, so if this is a requirement, spend some time planning for any application changes.



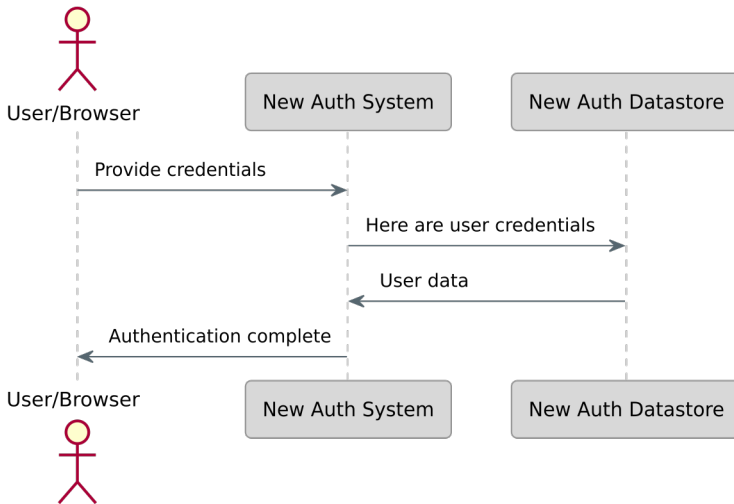
The migration process during the big bang migration.

After the migration, all users authenticate against the new system, and the old system can be decommissioned after you ensure there's no need to roll back to the old system due to functionality issues or data that was incorrectly migrated.

Keep on eye on customer support tickets or calls to understand if there were any such issues missed during the big bang migration.

Below is a diagram of the user login experience after the big bang migration is completed successfully.

The new user datastore is now the system of truth, and the old auth system is no longer in the picture:



The user auth process after the big bang migration.

These diagrams illuminate to the strength and the weaknesses of the big bang migration, as mentioned above. Conceptually it is simple, but in reality there are a lot of moving pieces to change your system from the first diagram above to the last. Let's take a look at some of them.

Test Import Performance

Because you have downtime with this approach, you're going to want to import users quickly. Review auth system documentation for performance enhancements. This may involve degraded functionality in some instances, but as long as that functionality can be re-enabled after import, that's ok. You're looking for speed here, both because you want the production import to happen quickly and also because your team will be importing the dataset multiple times during the test phase.

Steps you can take vary depending on the system, but common options include:

- Size your system with plenty of headroom.
- Take advantage of any bulk import APIs, rather than single user creation API endpoints. The latter are great for ongoing management, but for import performance, bulk options are the way to go.
- Disable any webhooks or other integration points.
- Set the HTTP timeout to a large value on API requests.
- Deduplicate any user data you can. No sense in importing users multiple times if it is avoidable.
- Stage your data if possible. This may mean exporting user data to flat files, and then importing that. This will make debugging easier, since you can load one file at a time, and you can repeat a data load if there are issues. It will also be more performant than loading data from a database or databases.

Building the Migration Scripts

To actually move the data, you'll build out a series of scripts and programs. The exact steps here are highly implementation dependent, but make sure you treat these scripts like any other software project.

- Document them. You don't need excessive documentation, but ensure that team members can pick up the scripts if needed.
- Keep them under version control.
- You don't need to set up a continuous integration system, but ensure you can build them.
- Don't worry about making them fast until you determine they are the bottleneck. It's far more likely the auth system or password hashing (if applicable) will be the slowest part of the system.

You can write the migration scripts in shell, or any language which is supported by the client libraries of your authentication provider. The basic flow for migrating user data is:

- Iterate over all the users in the old system or systems. Make sure you capture all the user information you need, including both profile data and password hash data.
- Build the intermediate files to assist with debugging and performance.
- Import the files into the new system using the scripts.

You want this process to be automated as much as possible because the last thing you want during downtime is to forget a critical step and delay the cutover. As an additional benefit, this process will be done over and over again during testing, so automating it will relieve toil.

Make sure you create all your groups, tenants, applications and other configuration before you import your users. The best option is to create these in an automated fashion so you can tear down your environment and rebuild with minimal effort. Scripts or tools like Terraform are helpful here.

Password Hashes

Hopefully your current system hashes user passwords. If it encrypts them or stores them in plaintext, moving to a modern auth system will improve your security posture. If this is your situation, configure the new auth provider to hash passwords on import.

If, on the other hand, the original system hashes passwords, find out which algorithm was used. If it is one supported by the new system, that's great. If the old system uses an algorithm other than one supported by the new auth system, check if the new one supports custom hash algorithms.

If the new auth system doesn't, you have two options:

- Force all users to reset their password.
- Use the slow migration option discussed elsewhere.

If your system does allow for custom password hashing algorithms, check if you can rehash passwords using a stronger or more standard hashing algorithm on login.

Refresh Tokens

Determine if you need to import refresh tokens, if you are migrating from an OAuth based auth system.

Doing an import will allow your end users to continue to login without interruption, as when their access tokens expire, the refresh token stored on their device will continue to work. This is especially important if your users have logged in on devices such as TVs or appliances, where re-authentication can be painful.

Testing Data Migration Correctness

Test this import process with as large of a dataset as possible. If you can, use your entire user dataset, but be aware of PII concerns. Treat the data carefully.

Testing with a realistic sized load lets you know how long your import will take. Real world data will reveal edge cases, such as duplicate emails or incorrectly formatted user attributes.

Given you want to test with as many users as possible, plan to be resetting the auth system often. This is usually the quickest way to get back to a “clean slate” when an import script crashes or your assumptions about the data are incorrect.

Test that your applications can integrate with the new auth system. You probably did some of this during a proof of concept while you were testing this migration provider, but now you can test with production or close to production data.

Performing the Migration

When your scripts work in your testbed environment, prepare to do a production migration. Inform all the internal stakeholders. Plan for downtime unless you can run your application with the original user store in a read only mode. How much downtime? You should know based on your testing of the import.

Run the migration on your production dataset, moving the data from the original system to the new one. When the migration is finished, release your application changes. All applications should point to the new system for authentication requests and related user flows, including, but not limited to:

- Log in
- Registration, if applicable
- Forgot password
- Password changes

Should you need to rollback, revert the changes to your application, and have it point to the old system. If users have updated profile data in the new system, you'll need to port those changes back to your legacy system.

A script using an API and searching for users with recent updates will be a good starting point, though the exact data rollback will be application dependent.

Segment By Segment Implementation

A segment by segment migration is similar to the above big ban migration, except that you are going to split your user data into segments and migrate each segment. Logical user database

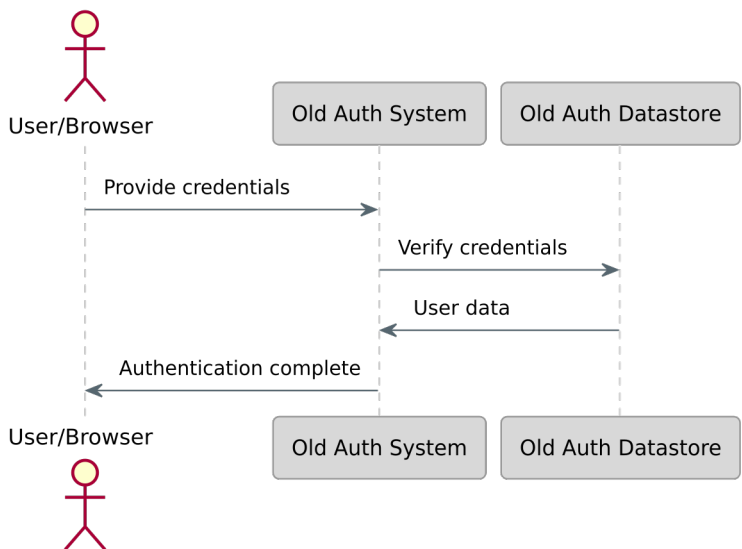
segmentation points include by application or by role. However, the planning, mapping and execution are similar, just done for smaller chunks of users and multiple times.

However, the application cutover process with this approach is not as simple. You can't simply send all your users to the new auth system when you haven't migrated all of them.

Which users are sent to which system depends on how you created your segments. If you split on application usage, then update one application to send any authenticating users to the new system. If you split your users based on other attributes, build logic in your application to determine where to send a user when they log in.

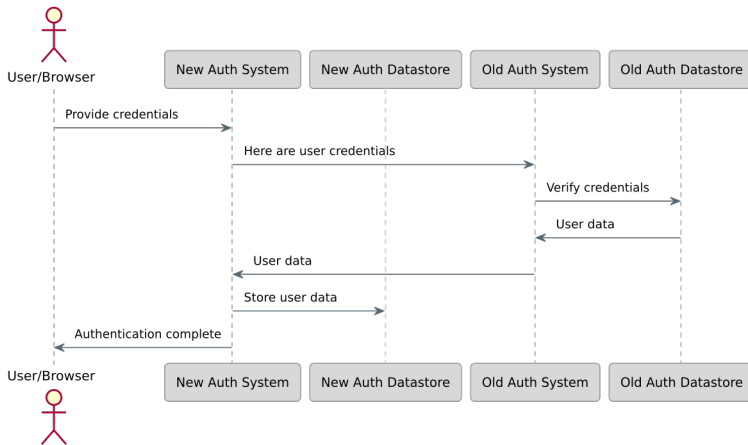
Slow Migration Implementation

At a high level, a slow migration happens in four phases. Each user proceeds through the phases independently of other users. Here's how the data flows before any changes to the auth system are made:



The user auth process before the slow migration.”

In the second phase, you’d stand up the new system, connect it to the old system, and route all authentication requests from applications to the new system. When the new auth system proxies the old auth system, the latter is consulted the first time a user authenticates:



The auth process during an initial authentication in a slow migration.

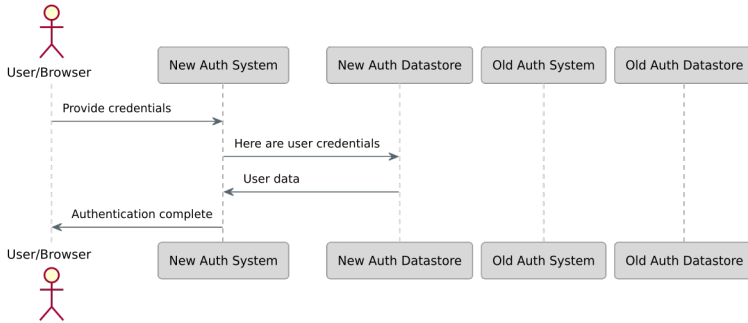
When this initial login is successful, any user data to be migrated is returned. That data is stored in the new system. The old system is the system of record for the first login of this user, but not after.

This migration is also an excellent time to clean user data up as it is transferred to the new system, as long as you can do it quickly; the user is signing in, after all. For example, you can convert addresses to a standard format.

You can also upgrade the user's password. If the old system stored the password as an md5 hash, on migration you can use a more modern hashing algorithm, such as bcrypt. Since you have the user's password in plaintext, the normal difficulty of changing a password hash is avoided.

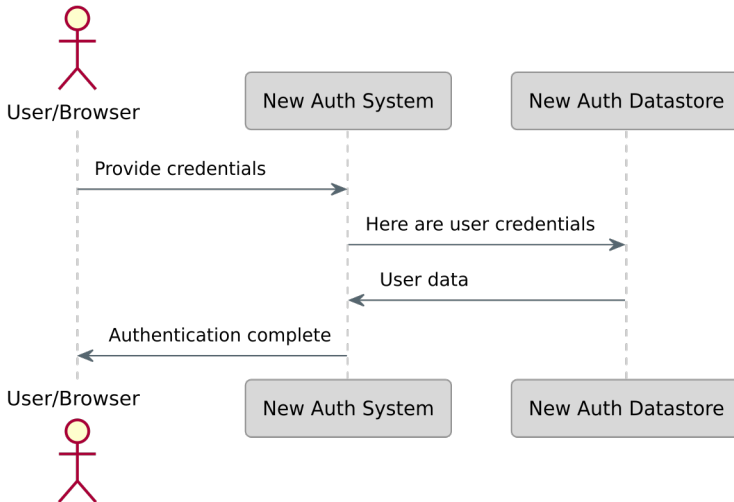
For subsequent logins, as mentioned, the user's data has been migrated. For this user, there's no longer any need to delegate to the old auth system. However, it continues to run because there are other users who have not yet logged in, and therefore have not yet been migrated.

Here's the auth process for a user who has been migrated to the new system:



The auth process during a subsequent authentication in a slow migration.

After a period of time, most user data has been migrated. There's no need to consult the old auth system and it can be safely shut down.



The auth process after the migration is completed.

Slow migration is similar to the [strangler pattern](https://martinfowler.com/bliki/StranglerFigApplication.html)⁵⁵, first documented by Martin Fowler. Let's walk through the steps to successfully undertake a phased migration.

⁵⁵<https://martinfowler.com/bliki/StranglerFigApplication.html>

What Does Done Mean

A key part of planning for a slow migration is setting a completion goal. Because slow migrations move users one at a time, it is unlikely you'll migrate 100% of your users in this manner. Some people will use your software rarely, others may have abandoned their account. No matter how long the migration, some people may not log in during that time.

Decide what “done” means to you. When thinking about this, consider:

- How often do people log in, on average? Is there a significant long tail of users who visit the application less frequently than the average user?
- What are the ramifications of a user being locked out of their account? Are there business, compliance, legal, or security repercussions? Is loss of timely access to data an annoyance or a disaster for the user?
- What will you do with unmigrated users?
- How hard or painful is it to keep both systems running?
- What is the value in a customer who has not logged in to the system in six months? A year? Three years?

It's hard to give any blanket guidance as all systems are different, but you should definitely set a goal of percentage of users migrated, migration time elapsed, or both. Otherwise you may be in for a frustrating situation, where you don't know when to cut over. You'll want to query new and old systems and determine how many accounts have been migrated to determine progress.

Proxy To the Old Auth System

Once the planning and data mapping is done, you need to connect the two systems.

The new auth system could connect to the old system's datastore, but it's better to create an API in the old system. Doing so allows you to leverage any business logic the old system performs to authenticate the user or assemble their data. This API may also call any other APIs or systems needed to construct the full user object for the new system.

This code should mark the user as migrated in the old auth system datastore, if possible. This will be helpful in tracking progress and determining the system of record for each user.

If you can't modify the old system to add an API, because it is proprietary or hard to change, you can either reach directly into the database, or put a network proxy in front of the old system to perform required data transformations. In a worst case scenario, you could mimic whatever user agent the old system expects, such as a browser, and convert the corresponding result into data to feed into the new system.

Make sure to lock down this API. At a minimum, use TLS and some form of authorization to ensure that no malicious party can call this endpoint. You don't want someone to be able to arbitrarily try out authentication credentials. Use basic authentication or a shared header value, and discard any requests which are not expected. You could also lock access to a given IP range, if the new system is only connecting from a certain set of IPs.

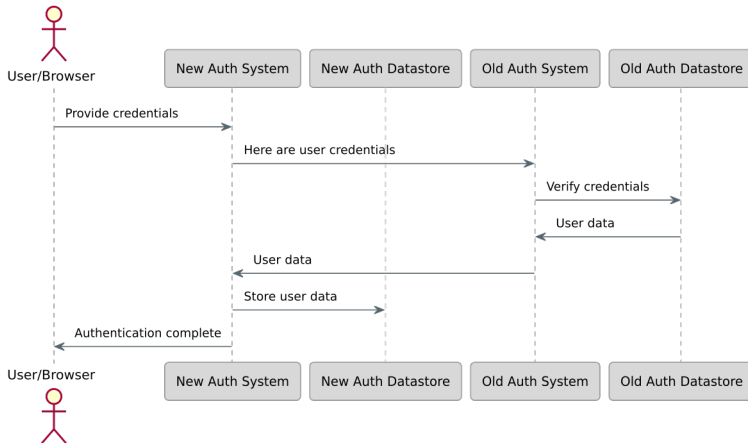
You should build an automated test against this API, ensuring that if there are any changes to the old auth system, you learn about them before your users do.

When you have tested that this proxy returns the correct user data for a given set of authentication credentials, cut your applications over to the new system.

No data needs to be migrated, but new users should register with the new auth system. It should receive all authentication requests first. It will, of course, defer such requests to the old auth system.

The Authentication Process

When a user signs in, the new auth system passes on the credentials to the old auth system, and receives the user data in response. Here's the diagram from above:



The auth process during an initial authentication in a slow migration.

After initial authentication, the data should be migrated to the new auth datastore. Then, this user can be completely managed by it.

During authentication, add a migration success marker to the user in the new auth system. This is the inverse of the migration marker in the old system, mentioned above. Having this data will let you know how many users have been migrated successfully. If troubleshooting the new auth system, it is useful to know if users with issues are newly registered or migrated.

Remove the Proxy

After running reports on the numbers of migrations, you'll know when you've reached your goal of migrated users. Prepare to shut down the old auth system. Clean up any code or configuration

in the new system which was used to communicate with the old system.

At this point, you need to determine what to do with all the users who haven't migrated. You considered this in the planning section, but now you need to execute on the plan, or adjust it. You have the following options:

- Notify the users to encourage them to migrate (“if you don't login, your account will be deleted on DATE”).
- Archive or delete them from the old system. This will force users to re-register, and may mean lost data.
- Move them to the new auth system with a big bang migration.
- Continue the slow migration and extend the time running both systems.

Let's examine each of these options.

Encourage Users To Log In

You can reach out to the users who remain in the old auth system and remind them of the value of your application. Encourage them to login, which will migrate their data.

Who among us hasn't received a notice stating “inactive accounts will be deleted on DATE. If you want to keep your account, please sign in.” If you send out a notice like this, provide ample lead time.

Force Users To Re-Register

If the accounts don't have valuable associated data and it's easy to sign up for a new account, you may want to archive or delete them from the old system. Once you've done that, anyone who only had an account on the old system will receive an authentication error. They can, of course, register in the new auth system.

In this case, make sure to stop billing any unmigrated users. Charging people for an application to which they no longer have access is a great way to annoy them.

Migrate All Remaining Accounts

If the accounts are valuable, migrate them with a big bang. This will be less risky than if you were trying to migrate everyone, because you are migrating fewer, less active users.

If you don't have access to the password hash logic, you could force all these accounts to reset their password. While this is a disastrous path if you are migrating *every* user in a system, this subset of users is less active and smaller. They haven't accessed your system in a while, and may have forgotten your application even exists.

Extend the Migration

If the reason you are doing a slow migration is because the big bang is too painful, and the prospect continues to be worrisome even with a smaller set of users, then you can extend the slow migration period. Simply keep running both systems.

You can mix and match these approaches. For example, you could migrate all paying customers with a big bang, while archiving free customers who may have been kicking the tires on your application a year ago.

Conclusion

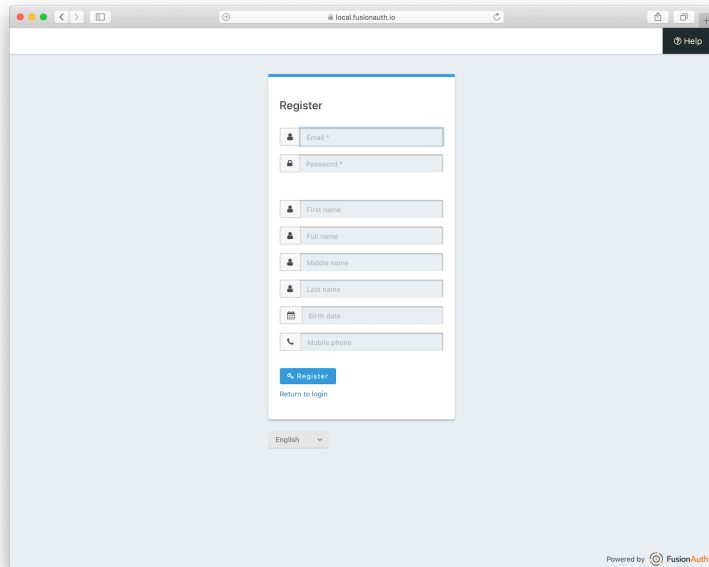
Data migrations are never easy, and auth system migrations are even tougher. The auth system is the front door to your application, and you don't want it locked any time. Consider the tradeoffs of these different types of migrations and pick the one best suited to your situation.

Dan Moore is FusionAuth's head of Developer Relations.

Best Practices for Registration Forms

By Dan Moore

Signing up for accounts is something we're all familiar with. It's a gateway to applications we want and need. But it's not really fun. Or even pleasant. After all, we're signing up to access the application, not because we want to set up another username and password and enter personal data into yet another system:



The image shows a web browser window with the address bar displaying "local.fusionauth.io". The page features a "Register" form with the following fields: "Email *", "Password *", "First name", "Full name", "Middle name", "Last name", "Birth date", and "Mobile phone". Below the form is a blue "Register" button and a "Return to login" link. At the bottom of the page, there is a language selector set to "English" and a footer that says "Powered by FusionAuth".

A long registration form. Honestly? You need my full name and my middle name?

You can make the registration form process simpler. And you should.

Is a Registration Form Needed At All?

Before you start, ask yourself a fundamental question: does your application require registration?

While user registration to create an account is commonplace, every step required of a user before seeing the value of your application affects sign up rates. What can you do to avoid a typical registration process?

Let Them Try It First

Potential users are trying to kick the tires on your application. From a 2020 masters thesis, [A User-Centered Approach to Landing Page Optimization in a Software-as-a-Service Business \(PDF\)](https://aaltodoc.aalto.fi/bitstream/handle/123456789/44930/master_Meissner_Mai_2020.pdf?sequence=1&isAllowed=y)⁵⁶:

“Visitors have several different needs before deciding to sign up and test the service. The most prominent need is understanding what the service offers and whether it could be a suitable solution for their problem.”

Is there functionality, degraded or not, which you can offer to visitors? For example, if you are building a drawing application, can you let a user create and download a picture without an account? This degraded functionality will allow a user to see the value of your application with minimal investment.

⁵⁶https://aaltodoc.aalto.fi/bitstream/handle/123456789/44930/master_Meissner_Mai_2020.pdf?sequence=1&isAllowed=y

If you are building a site which displays information based on a search, allow users to see a few results without signing up. While teasers may be frustrating to some, they also reveal the results' value.

In a [2013 study \(PDF\)](#)⁵⁷, Microsoft researches found the source of a sign up impacted conversion:

The more valuable services are more likely to be considered worth the privacy and effort cost of disclosing personal data, while the least valuable will not.

Allowing visitors to use your application, even in a limited fashion, helps them assess if a signup is worth their time.

Social Or Third Party Login

You can also rely on a third party for account management. Would social login make for a smoother experience? If you are targeting enterprise users, can you integrate with internal identity providers such as ActiveDirectory?

Different types of users will expect different third party auth providers. If you have a consumer focused application, almost everyone has a Facebook account, so offer that social identity provider. One less password for your potential users to remember and one less obstacle to them signing up.

If you are targeting enterprise customers, integrate with ActiveDirectory or similar corporate directory. If developers are your target market, GitHub authentication makes for a simple registration process and signals that you understand their needs.

⁵⁷http://cups.cs.cmu.edu/soups/2013/trustbusters2013/Sign_up_or_Give_up_Malheiros.pdf

Passwordless

Passwordless login allows a user to authenticate with something they have (access to a phone or email account) rather than something they know (a password).

While passwordless authentication requires providing some level of contact information, users do not have to create and remember yet another password.

All the above options provide an application with less data than the typical registration process. That's the cost. The benefit is less signup friction.

Ease the Pain Of Registration

If you've decided you need a registration form, remember that it is a form first and foremost. You should follow known best practices.

A Form Is a Form Is a Form

The Nielsen Norman Group discusses improving forms in their 2016 article, [Website Forms Usability: Top 10 Recommendations](https://www.nngroup.com/articles/web-form-design/)⁵⁸. The number one suggestion is:

“Keep it short. ... Eliminating unnecessary fields requires more time [to decide what data is worth asking for], but the reduced user effort and increased completion rates make it worthwhile. Remove fields which collect information that can be (a) derived in some other way, (b) collected more conveniently at a later date, or (c) simply omitted.”

⁵⁸<https://www.nngroup.com/articles/web-form-design/>

Carefully consider the information you are asking for. The fewer fields the better. While admittedly in a different context, [Imagescape more than doubled a contact form conversion rate](#)⁵⁹ by decreasing the number of fields from 11 to 4. There may be data needed only for certain features of your application; ask for it when that feature is first accessed, rather than at signup.

Make sure your form is mobile friendly; test at various screen sizes. The tediousness of data entry on a mobile device and the prevalence of their usage are another reason to have as few signup form fields as possible.

If a form field is optional, clearly mark it so. Even better, don't ask for optional data on the initial user registration. Request that information later, when the user is more engaged and has discovered the value of your offering.

Provide clear error messages when data fails to validate. Use both client side validation, which is faster, and server side validation, which is tamper proof. On the topic of tampering, ensure any form is submitted over TLS. You want to keep submitted information confidential and secure.

Make use of the full suite of HTML elements. Dropdowns and radio buttons are powerful, but number and email input fields leverage browsers' built-in validation and should be used as well. If you aren't sure what's supported, use tools like [caniuse.com](#)⁶⁰ to verify compatibility.

Registration Forms Are Unique

But registration forms aren't just another form. They are the gateway to your full application or site.

What causes angst when a user is signing up? This [2012 paper](#)

⁵⁹<https://unbounce.com/conversion-rate-optimization/how-to-optimize-contact-forms/>

⁶⁰<https://caniuse.com/>

(PDF)⁶¹ examined registration for government services. It defined sign up friction as “the imbalance between the business process (user goals) and [required] security behaviour” around signing up. This study found friction was best explained by the following attributes of a signup process:

- The number of new credentials required
- Any delay in the process, such as waiting for an activation email
- Whether registration requires an interruption of a user’s routine
- The frequency of legally obligated use of the service

Obviously you can’t control the last aspect, but minimize the number of new credentials, delays and interruptions in your registration process.

Most registration forms ask for a username and password. Make it clear what are valid values. If a username is an email address, allow all valid email addresses, including aliases.

Avoid complicated password validation rules. Allow users to use a password manager. NIST recommends a focus on avoiding passwords known to be insecure.

“[A]nalyzes of breached password databases reveal that the benefit of [password complexity] rules is not nearly as significant as initially thought, although the impact on usability and memorability is severe.” - [Appendix A—Strength of Memorized Secrets, NIST Special Publication 800-63B](#)⁶²

Obtain proper user consent. What this is depends on your plans for the requested information and what regulatory regime applies.

⁶¹https://discovery.ucl.ac.uk/id/eprint/1378346/1/ewic_hci12_diss_paper7.pdf

⁶²<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf>

Different levels of informed consent may be required. For example, if you are in the USA and are dealing with a child's personal information, you'll want to make sure you get parental consent because of COPPA.

Ensure new users enter sensitive data correctly. If there are any critical fields, such as a government ID, provide a confirmation field asking for the data to be re-entered to ensure the data is typed correctly. This is an exception to the rule of asking for less data. If the stakes of incorrect data entry are high, the additional work is worth the inconvenience.

If you need more than a few pieces of information on registration, consider splitting the sign up form into steps and using a registration stepper, also known as a multi-step form or a wizard. A stepper is a user interface element which shows how many steps are required to complete an action:

“Steppers display progress through a sequence by breaking it up into multiple logical and numbered steps.” - [Google's material design reference](#)⁶³.

Multi-Step Registration

Splitting up a registration form allows you to ask for more data, but avoid imposing a high initial cognitive cost on a potential user. It also allows you to track registration progress. Rather than a registration being an all or nothing proposition, you can see where people fall out of the registration funnel: is it because of step two or step three? It also may increase the conversion rate: Instapage saw an [18% increase in conversion rate](#)⁶⁴ when they split their registration form into multiple steps.

⁶³<https://material.io/archive/guidelines/components/steppers.html>

⁶⁴<https://instapage.com/blog/multi-step-form-part-2>

The image shows a web browser window with the address bar displaying 'local.fusionauth.io'. The main content area features a registration form titled 'Register'. The form is a white box with a blue border. It contains four input fields, each with a label and a small icon: 'Your first name' (person icon), 'Email' (envelope icon), 'Password' (lock icon), and 'Your mobile phone number' (phone icon). Below these fields is a blue 'Next' button with a right arrow. To the right of the button, it says 'Step 1 of 2'. At the bottom left of the form area is a language selector showing 'English'. At the bottom right of the browser window, it says 'Powered by FusionAuth' with the FusionAuth logo.

A multi-step registration form.

When creating the pages, group fields logically, as per the Nielsen Norman Group recommendations which suggest grouping “related labels and fields.” Separate pages allow you to provide a contextual explanation of how providing the data will be useful to the visitor at each step. If you can’t come up with a reasonable one, consider removing the fields.

Ask for as little as possible on the first registration step. Once they take that first step, they’ll be more committed to finishing, thanks to our [love of consistency](#)⁶⁵.

Ensure you are clear about the number of steps the registration process will take. Doing so lets the user assess the effort involved.

⁶⁵<http://changingminds.org/techniques/general/cialdini/consistency.htm>

Maintaining State

Splitting a form up into multiple steps requires maintaining state across submissions. If you are using a framework, investigate helper libraries, such as [wicked](https://github.com/zombocom/wicked)⁶⁶ for Ruby on Rails or [lavavel-wizard](https://github.com/ycs77/laravel-wizard)⁶⁷ for Laravel.

If you are rolling your own solution, you can maintain state in hidden form parameters or in the user's session. Either way, you'll want to serialize entered and validated data and store it. Then, when the form is ready to submit, you can deserialize this value and process the entire form, typically saving it to a datastore.

Another option is to save registration data in the datastore and progressively add to the user's profile as they work through the steps. This approach has downsides, however. Either any required fields must be submitted on the first step, limiting flexibility, or you'll have to have a staging registration table, where partial registrations are stored until completed.

Sensitive Information and Multi-Page Registration

Typically an application collects sensitive information in a registration form. Passwords are the canonical example, but other types of data may need to be treated carefully, such as a social security number or other government identifier.

When collecting sensitive data, keep it secure throughout the multi-page process. You have a couple of options:

- Don't ask for this information until the last step. Since the registration form will be sent over TLS, the sensitive data is then secure.

⁶⁶<https://github.com/zombocom/wicked>

⁶⁷<https://github.com/ycs77/laravel-wizard>

- Once submitted on any step, keep it in the server side session until the form is complete.
- Encrypt and store the form data in localStorage.
- After sensitive data submission, store it in a hidden form field, but encrypt it first.

Each of these options has tradeoffs. If you wait to ask for any sensitive information until the last step, you are limiting the flexibility of your form design. If more than one piece of sensitive data is needed, you're forced to put all of them on the last page.

Keeping the value in the session means increasing the size of your user sessions. You will also be forced to use sticky sessions, hindering the scalability of your application.

Storing the data in localStorage is risky, due to malicious code running in the browser. If you encrypt it, make sure the encryption happens on the server, otherwise the same code may be able to decrypt the data.

Encrypting the data and storing it in a hidden form field means additional processing. If you choose this, use a symmetric key stored only on the server side. Since you'll be decrypting the form field in the security of your server environment, the browser shouldn't need to read the data after the step on which it is submitted.

Dan Moore is Head of Developer Relations for FusionAuth.

Final Thoughts

Choosing an auth system is critical to the success of your application.

Done right, it provides needed, undifferentiated functionality that gates access to your application, secures your users' data, and doesn't lock you in to software costly in terms of money or team time.

Done wrong, an auth system can become a chokepoint, which inhibits your application and business flexibility, or a sinkhole, where you are spending team time and money to maintain functionality that, while important, your users don't really care about.

This guide has focused on evaluation, risk management and implementation concerns. I hope this guide has helped you. Please feel free to contact the FusionAuth team at <https://fusionauth.io/> with feedback about this book, or questions about how an auth system can help you.