

# NORMAL UI

How Non-Designers Can Make  
Their Web Apps Easier To Use

*TONY ALICEA*

Introduction.....	1
Normalization .....	4
Workflows, Screens, and Frames.....	6
How Do You Normalize? .....	10
Metrics .....	12
Low-frequency/High-complexity - You Should Normalize.....	16
High-frequency/High-complexity - You Should Denormalize.....	25
Low-frequency/Low-complexity - You Might Normalize.....	37
High-frequency/Low-complexity - You Should Denormalize.....	45
The Four Quadrants.....	51
Normalization In The Real World .....	53
Keep Workflows Small-ish .....	54
Determine Complexity on a Napkin .....	56
Normalize Dangerous Workflows .....	58
Workflows Form the App Skeleton.....	61
Normalized UIs Have Dev Benefits.....	62
Handle Scope Creep Through Normalization .....	64
Provide Context on Normalized Screens .....	65
Make Clear Calls-to-Action.....	67
Help Users Recover From Errors.....	69
Tell Users They Were Successful, and Help Them Keep Going.....	72
How To Talk About Normal UI .....	74
Recommended Reading .....	75
Conclusion .....	76

# Introduction

How do you make a web application easier to use? Note that I said “easier”, not “easy”.

Making a web application “easy-to-use” involves multidisciplinary expertise in areas like usability and web performance. Making a web application “*easier* to use” just means doing something that *improves* the experience.

That’s what this book teaches – **a repeatable technique** you can use to make any web application easier to use.

If you’re a developer who builds web applications (e-commerce, SaaS, intranet apps, or anything else you’re building), this book is for you. Designers who know more about making web apps look good, but less about making them work well, will also benefit.

Your web apps need to be easier to use. Usable software is successful software.

Where did this technique come from? Well, I’ve had an interesting career. For decades, I’ve been a developer and database architect. I’ve built many web applications that run large businesses. I’ve taught over 350,000 students to build web apps in my online courses.

I’ve also spent much of that time as a user experience designer. I’ve built design systems, run countless usability tests, designed interfaces used by thousands, and trained others in cognitive science, usability, and more.

Working heavily in both code and UX is considered rare in the industry. It’s given me, I think, a unique perspective.

This book is a result of my years of straddling the dev/UX fence. After 25 years of both building software and watching people use that software (and software

others designed and built) I can say with confidence that using this technique will make your software easier to use.

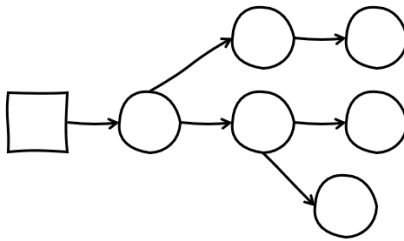
That doesn't mean I'm not standing on the shoulders of others. This technique sits on top of the massive body of work of usability engineers, cognitive scientists, and more.

I've refined and used this approach for decades, resulting in software that usability tested fantastically well, is easy for devs to implement, attracted more and happier users, and satisfied clients and stakeholders.

You don't need to be a designer to use these ideas. The technique is simple enough to be applied to any web application.

I call it **Normal UI**.

# The Technique



# Normalization

When I use the word “normal”, I’m referring to the term as it’s used in databases. A quick understanding of that will help you understand Normal UI.

In a “relational” database you might have a table (also called a “relation”) that looks like this:

OrderID	CustomerID	CustomerName	CustomerAddress	ProductID	ProductName	ProductPrice	OrderDate
1	1001	John Doe	123 Elm St.	2001	Widget A	\$10	2023-07-01
2	1002	Jane Smith	456 Oak St.	2002	Gadget B	\$15	2023-07-02
3	1001	John Doe	123 Elm St.	2003	Gizmo C	\$20	2023-07-03

A denormalized table, in all its glory

This table is “denormalized”. Each row contains a lot of information about the customer, the product they ordered, and the order itself. Each row of the table is doing a lot of work.

This kind of unfocused table results in data (like the customer’s name and address) being repeated, which means it’s more work when the customer moves and needs to update their personal information.

When you normalize a table like this, you split the information into multiple tables that might look like this:

CustomerID	CustomerName	CustomerAddress
1001	John Doe	123 Elm St.
1002	Jane Smith	456 Oak St.

Customers

ProductID	ProductName	ProductPrice
2001	Widget A	\$10
2002	Gadget B	\$15
2003	Gizmo C	\$20

Products

OrderID	CustomerID	OrderDate
1	1001	2023-07-01
2	1002	2023-07-02
3	1001	2023-07-03

Orders

OrderID	ProductName
1	Widget A
2	Gadget B
3	Gizmo C

Order Details

Each table focuses on one type of thing (Customers, Products, Orders, Order Details). Information is only stored in one place and referenced as needed.

When it comes to avoiding duplication and keeping each table easy to reason on, you want normalized data. But there are times (like doing a task that requires seeing all the data), when you need to pull it all together, and use a denormalized table.

In database land that's usually done via something called a SQL query, which provides a way to temporarily combine multiple tables into a single denormalized table. But that's beyond this book.

Database normalization provides the inspiration for Normal UI. In Normal UI, we think of workflows and UI screens as normalized or denormalized. For the context of this book, then, let's specify a definition of "to normalize":

**Normalize (*verb*):**

1. *In database design*: to organize data so that each table is focused on a single topic or theme, thereby eliminating redundancy and ensuring data integrity.
2. *In user interface (UI) design*: to split software workflows across different screens, so that each screen is focused on a particular task, simplifying the user experience and minimizing confusion.

For the rest of this book, we'll be focused on definition #2. How do you normalize an interface? When do you want a denormalized one? How will that make your web application easier to use?

# Workflows, Screens, and Frames

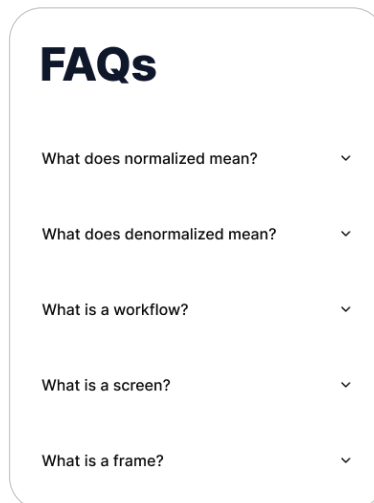
If you design or build software, you may be used to thinking in terms of “features”. But that isn’t helpful here. Instead, we are going to think in terms of “workflows”, “screens”, and “frames”. In Normal UI we define these terms as follows:

A **screen** is *a page or a modal* in a web application.

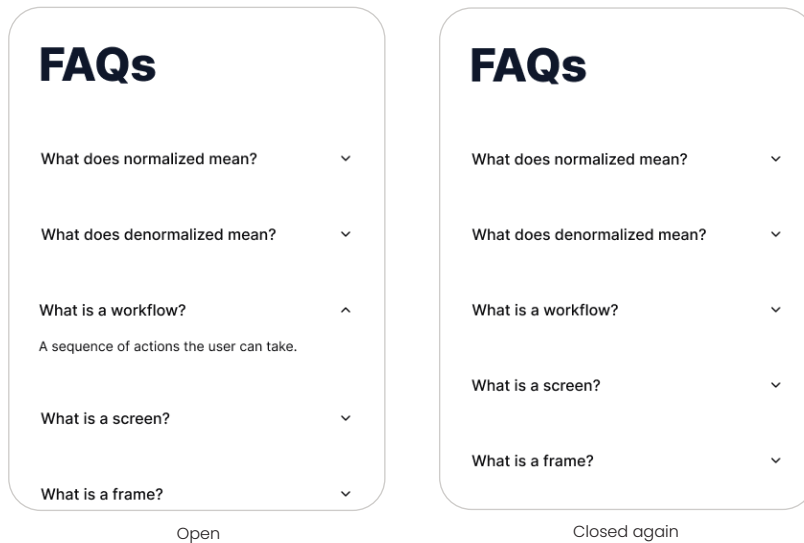
A **frame** is the set of UI elements that can all be accessed simultaneously by the user at a point in time. It’s *everything the user can interact with, at a single moment, without having to reveal it in some way*. It represents a subset of the screen’s potential content.

A **workflow** is *a sequence of actions that the user can take*. An action may or may not take you to a new screen but will almost always take you to a new frame.

A simple example is a set of accordions. We have a screen on its starting frame.



Let's say a possible **workflow** is to click on an accordion to reveal its contents, then click again to hide its contents. Each click is an action in the workflow, and results in a new frame.

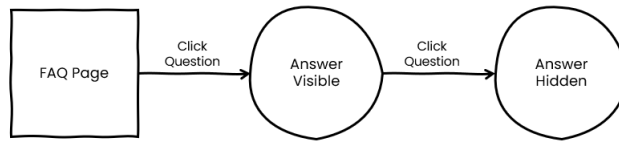


Remember I said you didn't need to be a designer to use Normal UI? All you need to be able to do is draw basic squares, circles, and arrows.

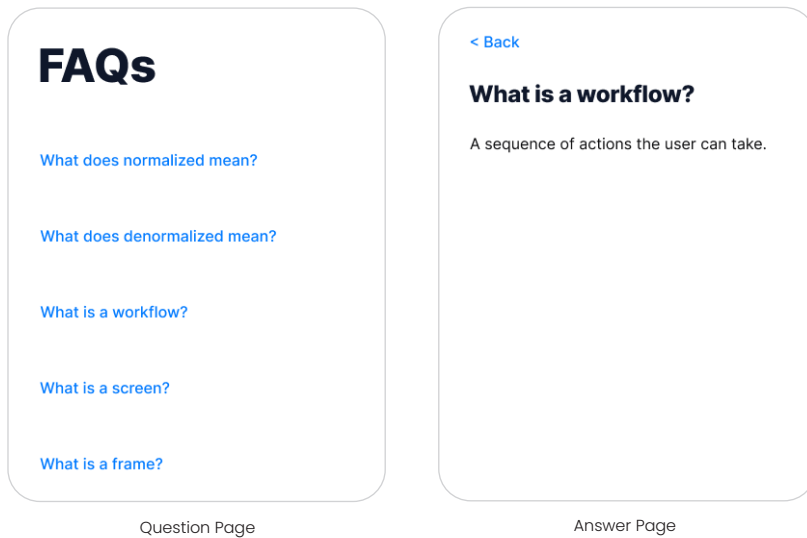
What we draw is a **workflow diagram**.

If you draw a square, it represents transitioning to a new screen. If you draw a circle, it represents transitioning to a new frame, while staying on the same screen. Arrows are the action the user takes.

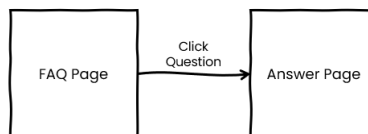
A workflow diagram for opening and closing a single accordion looks like this:



If, instead, we moved the contents of the accordion to its own page or a modal popup, so that clicking the question took you to a new screen, then the app might look like this:



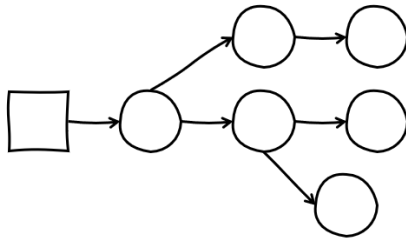
and the workflow diagram would look like this:



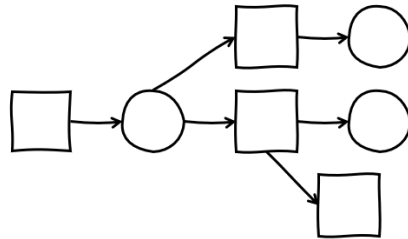
I tend to put the name of the screen in the squares, and a description of the state of the screen (the frame) in the circles.

Here we already arrive at an easy way to distinguish between normalized and denormalized interfaces. The diagram for a normalized interface has a lot of squares (screens). The diagram for a denormalized interface has a lot of circles (frames).

As you get used to thinking in these terms, you may not even need to draw everything out. But it can be very helpful when you're first starting out.



A denormalized interface  
(few screens, lots of frames)



A normalized interface  
(more screens, less frames)

# How Do You Normalize?

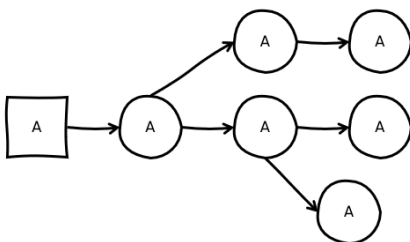
You may have noticed we've used the verb "normalize" and the adjective "normalized". The verb "normalize" essentially means to adjust from a more denormalized to a more normalized state.

So, how do you normalize a workflow exactly? By adding screens. You either: 1) **convert frames to screens**, or 2) **split one screen into multiple screens**.

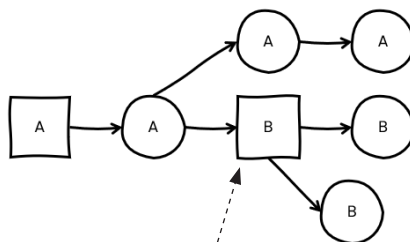
For frames, you take an action the user does in the software, and instead of that action keeping them on the same screen, it takes them to a new screen or modal.

Other times you take a portion of a page, and move it into its own page or modal.

In the workflow diagram, you might take a circle and make it a square. Or you might just add more squares.



In this workflow the user stays on screen A no matter what action they take.



We normalize an action in the workflow, taking the user to a new screen B.

If you had built our previous FAQ example, and you changed what happened when the user clicked a question from “open the accordion” to “go to an answer page”, you would have just *normalized* (verb) that “click the question” action. You would have made that workflow more *normalized* (noun) overall.

There are two major benefits to normalization, and each has even more positive side effects: 1) You reduce the user’s cognitive load at that point in the workflow and 2) You give the screen space to make that point in the workflow more understandable and guessable.

The word “cognition” means the mental action of acquiring knowledge and understanding. In other words, cognition means the work your brain does. By “cognitive load” we mean how much the user must think about and comprehend. When you split an action into a separate screen, you let the user *focus on just that action*, reducing their cognitive load.

By splitting an action into a separate screen, you also get the benefit of increased real estate. You can focus the UI elements of a separate page or modal on just what is needed to accomplish an action. You can place a few big, obvious buttons on the screen, instead of lots of small hard-to-notice ones.

We will get deeper into the implications and benefits of normalization in just a bit. But don’t think that this book is going to tell you to split your software across lots of screens. In fact, sometimes a denormalized workflow will be easier for your users to use!

What we really need, then, is a way to determine whether it’s better to build a normalized workflow or a denormalized one. We need a way to measure a workflow in our software and use that measurement to make decisions.

There’s a word for “a method of measuring something”: it’s called a “metric”. Surprisingly, I’ve found you only need two metrics to decide between normalization and denormalization: frequency-of-use and complexity.

# Metrics

Making an app truly easy-to-use involves a lot of research and metrics. Usability testing, performance measurements, A/B testing, etc. It also requires the expertise to analyze those metrics and come up with solutions to the problems that are found.

With Normal UI, you can make an app *easier* to use by utilizing only two metrics. One of which you can determine for yourself with no research.

The two metrics are:

1. **Frequency-of-use**

How often do users perform a workflow?

2. **Complexity**

The maximum number of frames that can be reached from any screen.

## Frequency-of-use

To understand this metric, you will have to have some understanding of your users. When it comes to making things easier to use, that's simply unavoidable. But you should want to know your users! They're the ones who have to use what you build, after all.

To implement Normal UI the only thing you need to know about users is *how often they use a workflow*. You have a few possible research options. You can use analytics to see what areas of an app are most used. But the absolute best approach is to simply talk to your users.

You may not be in a position in your company to talk directly to users. But *someone* is. You need to interface with that someone.

Talking to users is best because analytics may show you what screens are most used, but not necessarily which frames (unless you setup your app to track interactions in some way). Users will be able to tell you what they do most often.

A terrific help is user observation. Watching over someone's shoulder (in-person or via screenshare) as they use the app for a while. User observation combined with user interviews will give you a lot of great information on the usability of your app and where the problems lie.

The "frequency-of-use" metric sounds subjective. But in practice I find that if a **user performs a workflow at least once every few days**, you can consider that high-frequency usage. Anything less is low-frequency. That's just a rule of thumb, though.

If you are designing brand-new software, so there's nothing to observe or analyze, then you need to understand how your users work currently, in whatever physical or digital process they are using. That will give you a clue, at least, on what they will do often in your app.

## **Complexity**

The second metric is one you can determine for yourself. What is the complexity of the workflow? That is, how many different actions can the user take while never leaving the same screen?

The less focused a single screen is on a single action, the more complex that screen becomes. You may have multiple screens in the workflow, and it is still complex because so many actions can be taken on each screen.

Here's a rule of thumb: if you have any screen that leads to more than 10 frames, that part of the workflow is definitely complex, so we judge the whole workflow to be complex. More than 5 and it might be.

That rule of thumb may sound overly restrictive, but remember when we drew the workflow diagram of our FAQ page, we didn't draw a separate action for each question. We drew a single "click a question" action.

We don't have to necessarily count every individual possible click on a page as an action and new frame. We're more interested in the different *types* of things a user can do on a screen, because those are distinct things the user has to think through.

A workflow is complex when you have lots of "pathways of thought" stuffed onto a few screens.

Don't get overly wrapped up in the exact number, though. You'll be able to "feel" if a workflow is complex once you start thinking this way.

You can also judge complexity with a simple question. Ask yourself "**what is the purpose of this screen?**" If you can't describe the intent of the screen in one or two bullet points, then it's probably complex.

At this point you may be asking yourself: what if I'm starting from scratch and there's no existing screens? Or what if I'm dealing with existing software that's already normalized?

Good questions! We will get into that in the second half of the book. For now, just focus on the definition of complexity in Normal UI. It's an important and powerful metric that will help you make your web apps more usable.

## **Classification**

These two metrics (frequency and complexity) make it easy to classify a workflow. A workflow is either:

1. Low-frequency/high-complexity,
2. High-frequency/high-complexity,
3. Low-frequency/low-complexity or
4. High-frequency/low-complexity.

It turns out these four classifications give us exactly what we need to decide if an interface should be more normalized or denormalized!

We'll now look at each classification individually, along with some examples, which will enable you to use the Normal UI technique right away.

# Low-frequency/High-complexity - You Should Normalize

If a workflow is complex and isn't used by users very often, you should normalize it. Yup. A straightforward recommendation with no caveats.

If users use a complex workflow at low-frequency, they'll never gain expertise at performing that workflow. That means their cognitive load is very high, as every time they perform it they have to remember how to take every action and fully process every frame presented to them.

By normalizing the workflow, reducing the amount of thinking the user has to do on each screen, and hand-holding them through a process via multiple screens, you will make the workflow easier to use.

Let's look at a concrete example, so you can see how normalization improves usability in this case.

## Example: Registration

The lowest of low-frequency usage is "only once". If your users need to register to use your web app, they (hopefully) only ever do it once. They will never become practiced at signing up for your app, except for the parts that are similar to other things they've signed up for.

Every registration will require the cognitive effort of the first time you do something. The first time is always the hardest.

While registration is a low-frequency workflow, you might think that it isn't very complex. But once you start to diagram it in the Normal UI way, your opinion may change.

Let's suppose your web app registration page looks like this:

**Register**

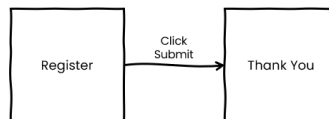
**Name**

**Email**

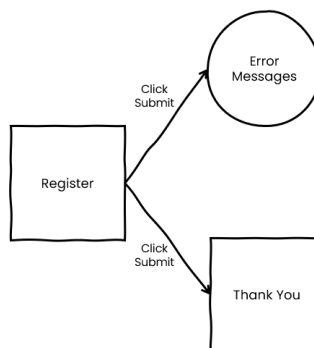
**Username**

**Password**

Seems simple! The user enters their name, email, username, and password. They click “Submit” and go to a thank you page. So far, the workflow diagram looks like this:



But the user may try to submit invalid or missing information. In which case a web app typically shows an error summary and/or error messages next to the form fields. So, the workflow is actually:



If there are errors the user stays on the registration page, so it's a new frame. If there are no errors the user goes to the thank you page.

But let's suppose there are other requirements for registration. If the username is already taken the error message should say so and give the user some options to choose from for a username.

An error frame might look like this:

**Register**

Name  
Tony Alicea

Email  
hey@tonyalicea.dev

Username  
Username is already taken. You might try:  
[tony105](#)  
[tonyalicea283](#)

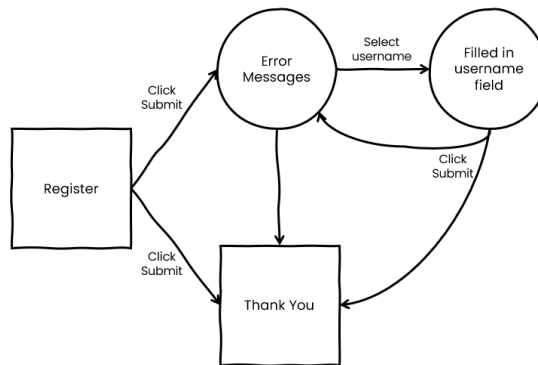
tony

Password  
Password

Submit

The user is given some options for a valid username. They can click a username to fill in the username field, and then try submitting again.

So, the workflow diagram looks like this:

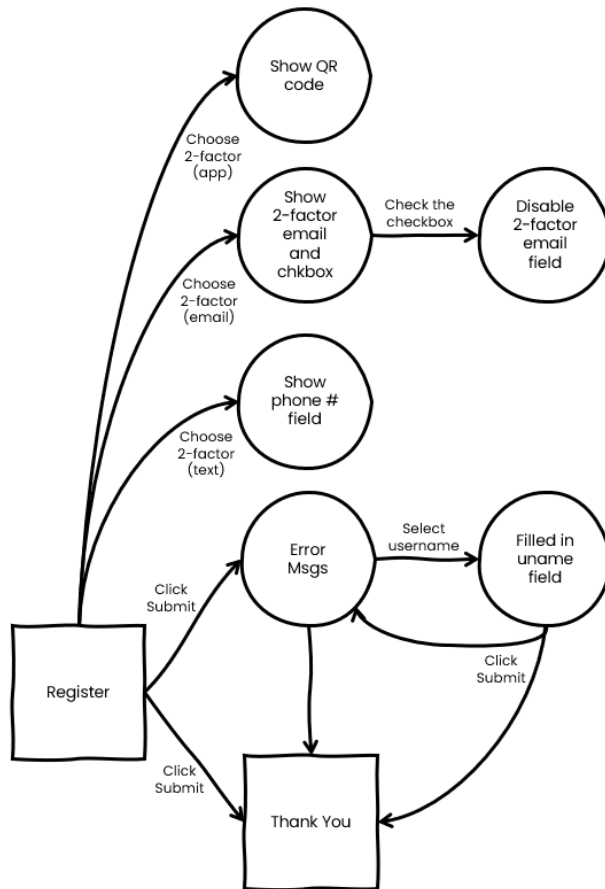


A bit more complex, but not too bad. There are only two frames.

But now let's suppose that a new security requirement is added. The user needs to decide what kind of two-factor authentication they want: text, email, or an authenticator app.

If they choose to receive a security code over text, then we show a phone number field. If they choose email, we show an email field with a checkbox to use the email they already entered. If they choose an authenticator app, we show a QR code so they can set that up during registration.

The workflow diagram now looks like this:



Uh-oh. The register screen already has six possible frames it leads to. We have definitely entered the realm of high-complexity.

Note that we didn't put a frame for every possible individual error message. Each possible missing required field doesn't need a frame. Just the different *kinds* of actions the user might have to take to complete the task.

Also, I didn't necessarily draw arrows back through every possible path (like to the "Thank You" page). The goal isn't a complete map, but just to get an idea of the amount of screens and frames.

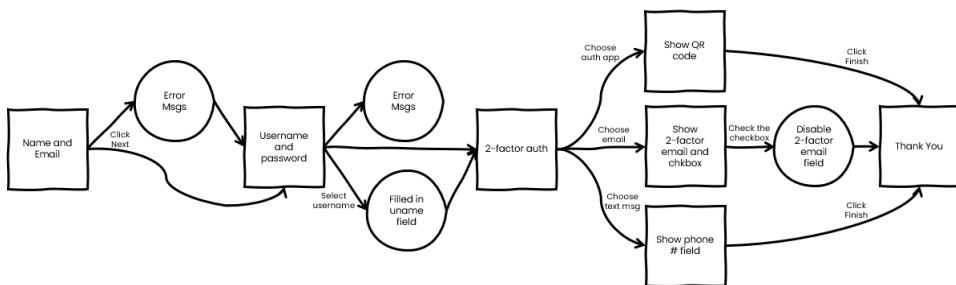
If you were to build out this entire form and watch an average user (not a developer) try to use it, you'd likely watch them slow down as they try to understand all the things being asked of them at once.

They might find things confusing that would shock you. It's hard to put yourself in the shoes of people who don't know how software works.

Since we've identified this workflow as low-frequency and high-complexity, we should normalize it!

This means we need to identify circles (frames) that we can turn into squares (screens) or squares that can be split into multiple squares. We need to normalize at least a portion of the workflow onto new screens. You may also need to move the frames around or create new frames as part of the process.

What actions would you normalize? Here's one way:



We used both methods of normalization.

1) We split one screen into a series of multiple screens.

Instead of a single "Register" screen, we now have three screens. The first asking the name and email, the second the username and password, and the third on two-factor authentication.

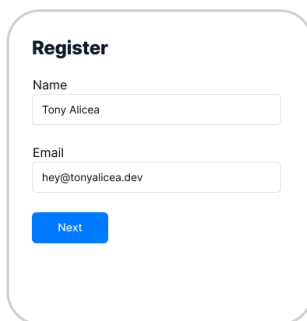
and

2) We converted frames to screens.

When you choose the type of 2-factor authentication you want (email, text message, or auth app), you are taken to a new screen (a different one for each type) instead of staying on the same screen.

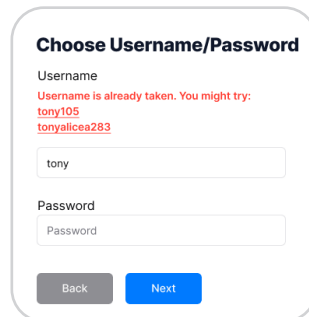
Normalization has reduced the number of frames in the workflow and increased the number of screens. Does this really result in an easier-to-use workflow for the infrequent user?

Well, the screens now might look like this:



A mockup of a 'Register' screen. It has a title 'Register' at the top. Below it are two input fields: 'Name' with the text 'Tony Alicea' and 'Email' with the text 'hey@tonyalicea.dev'. At the bottom is a blue 'Next' button.

The first step of registration is now unintimidating.



A mockup of a 'Choose Username/Password' screen. It has a title 'Choose Username/Password' at the top. Below it is a 'Username' section with a message 'Username is already taken. You might try:' followed by two suggestions: 'tony105' and 'tonyalicea283'. Below the suggestions is an input field with the text 'tony'. Below that is a 'Password' section with an input field labeled 'Password'. At the bottom are two buttons: a grey 'Back' button and a blue 'Next' button.

The only thing to think about is choosing a good username and password.

**2-factor Authentication**

This helps keep your account safe. You will be sent a security code when you log in. How would you like to receive the code?

Send me a code via...

Text Message

Back

Next

2-factor authentication isn't understood by all users. Now we have a chance to explain.

**2-factor Authentication**

You will receive a text with the security code to this phone number.

Phone Number

555-555-1111

Back

Finish

A different, focused screen for each choice the user might make.

**2-factor Authentication**

You will receive an email with the security code. It can be different from your account email.

Email Address

☒ Use Account Email Address

Back

Finish

We can explain the difference between the user's account email and the 2-factor one.

**2-factor Authentication**

Use this QR code in your authenticator app like [Google Authenticator](#) or [Microsoft Authenticator](#).



Back

Finish

Authenticator app setup requires its own thought process. We can be more helpful to the user with more screen real estate.

Normalization doesn't always result in a series of 'Back/Next' steps. But, in this case, it did and it helps make registration a much clearer process that we can coach the user through.

Each screen has the breathing room to make each action more obvious, and thus easier to use. The user is never overwhelmed, and I can say with confidence this series of screens tests well with users. You also have room to grow for future additional features.

Notice that we didn't even bother building out the denormalized version of the workflow. It would have resulted in a big form with multiple UI elements that changed based on user input. The user may have suffered from **change blindness** while using it.

Change blindness is the tendency to not notice when something right in front of you has changed, because so there are many things in front of you to focus on. If you've never heard of change blindness, I highly recommend searching the web for the "selective attention test" or "the invisible gorilla". It's fascinating.

When testing forms where things appear and disappear, it can be surprising what the average user doesn't notice. Normalizing away elements on the screen reduces the likelihood of change blindness, because there isn't as much information to focus on.

## Takeaways

As requirements increase, the likelihood of needing to normalize grows, especially if the workflow is infrequently used.

You can apply Normal UI when working to improve existing software or when designing new software. When you're designing a new app, you can save yourself a lot of work by looking at the workflows and deciding on normalization before implementation even begins.

If a workflow is used at low-frequency, and is high-complexity, it should be normalized.

You're more likely to have a usable app that makes users happy.

# High-frequency/High-complexity - You Should Denormalize

If a workflow is used by users very frequently, and it is complex, you should denormalize it. Another straightforward recommendation!

When users have to use a complex workflow frequently, they become more concerned with speed than being hand-held through it. Bouncing around more screens than necessary tends to annoy them as they feel it is slowing them down.

A caution, though. Sometimes a confusing, but high-frequency, workflow is justified by saying that users will “get used to it”. When something doesn’t make sense, it adds cognitive load every time you use it, no matter how often you do it. You still want high-frequency workflows to make sense.

Let’s look at another concrete example, so you can see how to balance normalization and denormalization in this case.

## Example: Filters

Imagine a complex list filtering feature within your web app. Users have a wide range of filters to choose from. Each filter takes up its own tiny piece of the interface. Users use this *all the time*.

If choosing and running filters is a complex, high-frequency workflow, then it should be denormalized. That means fewer separate screens, more frames, where you stay on the same screen after taking an action.

Let’s suppose that we’re working on an existing web app, where users are complaining that the filter functionality is slowing them down. The current screens look like this:

Invoices

Add Filters

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

An invoices list screen provides a button to add filters.

Invoices

Add Filters

Add Filters

Choose Filter

Person

Name

Location

Payments

Amount

Date

Run Filters

Clicking 'Add Filters' opens a modal where you select from a list of filters to add.

The screenshot shows the 'Invoices' page with a modal titled 'Payments: Amount' open. The modal has a close button (X) in the top right corner. Below the title, it says 'Enter a range'. There are two input fields: 'From' with the value 'Low' and 'To' with the value 'High'. A blue 'Add Filter' button is at the bottom right. In the background, a table of invoices is visible with columns for invoice number, status, payment method, and amount.

Invoice	Status	Payment Method	Amount
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

Once you select a filter, the modal loads a unique form for that filter, where the user can enter the details and add the filter.

The screenshot shows the 'Invoices' page with a modal titled 'Add Filters' open. The modal has a close button (X) in the top right corner. Below the title, there is a dropdown menu labeled 'Choose Filter'. Below this is a table with two columns: 'Filter' and 'Detail'. The table has one row with the filter 'Payments (Amount)' and the detail 'From: \$50 / To: \$100'. A blue 'Run Filters' button is at the bottom right. In the background, the same invoice table as in the previous screenshot is visible.

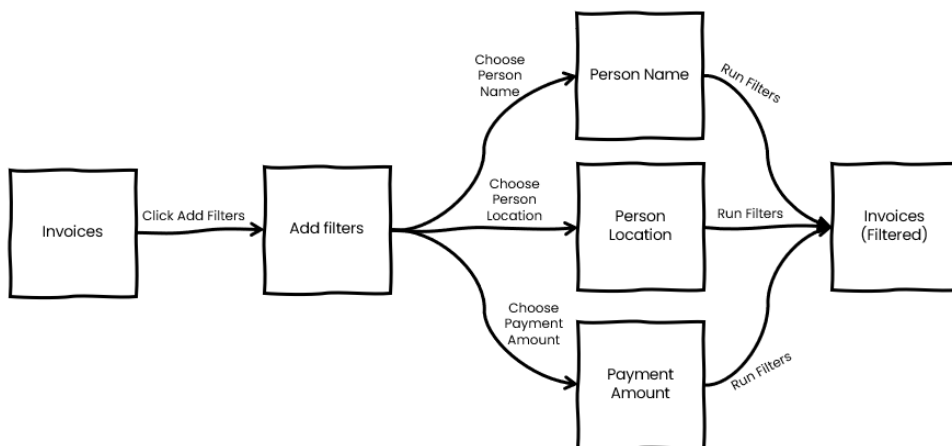
Filter	Detail
Payments (Amount)	From: \$50 / To: \$100

After adding the filter, the user is taken back to 'Add Filters' where a summary of all chosen filters is shown. The user can either add more filters or click 'Run Filters'.

Invoices			
<button>Add Filters</button>		Payments (Amount): From: \$50 / To: \$100	
Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

Clicking 'Run Filters' closes the modal and applies the filters, updating the list of invoices.

The workflow diagram looks like this:



Is this a bad design? It depends!

Despite all the functionality, the design has kept the complexity of the workflow low. There are very few frames (if any) that can be reached from any screen. Each screen is tightly focused on a particular action.

Remember modals and pages both count as screens, because they let the user focus on just the UI elements inside of them.

If users very rarely added filters, this highly normalized workflow would probably work great! Users wouldn't get confused or lost and would successfully be able to filter the invoices every time.

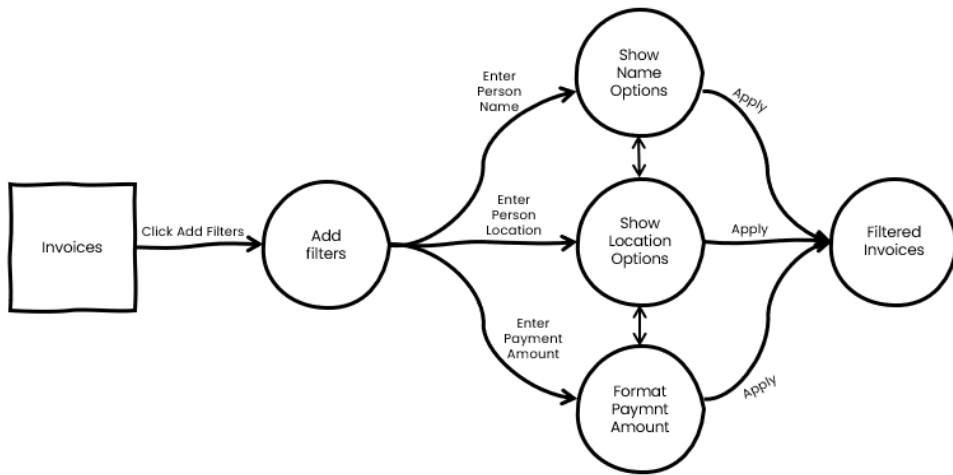
However, normalized workflows also tend to be slower. That is, slower for the practiced user. So, if users are using this workflow at high-frequency they'll have a usability problem. It won't be that the workflow is hard to understand. Rather, for high-frequency users, it's inefficient.

This workflow is high-frequency, and all those squares in the workflow mean if we denormalized it (turning them into circles), the workflow would become complex. Those two things tell us this workflow should be denormalized.

If we were starting from scratch, rather than redesigning, we would have seen how many possible actions there are and leaned towards a more denormalized workflow due to the high-frequency usage.

What might a denormalized workflow look like? Denormalization is the opposite of normalization. In the workflow diagram, we **convert squares to circles and combine multiple squares into one square**. We take things that are on separate screens (pages or modals) and **keep them on one screen**.

Here's a possible denormalization of this workflow:



Well, that looks familiar. It's basically the same! But now there's only one screen and the rest are frames. Because of this we also have drawn arrows between the individual filter frames (name, location, amount). It's conceivable to add all your filters before running them, bouncing back-and-forth between them.

We imagine that as you use the filters other things will appear, like typeahead lists, thus the three frames near the middle.

What would an implementation of this look like? There are lots of possibilities. Normal UI isn't about UI patterns. It's about determining when we should keep things on the same screen, and when we should split them across separate screens.

A workflow diagram gives us design guidance, not the design. But we are much more likely to implement a design that meets the users' needs.

Here's one possible implementation:

The screenshot shows a web application titled "Invoices". At the top left, there is a blue button labeled "Add Filters". Below this, a sidebar panel titled "Filters" is open, containing several input fields for filtering invoices: "Person Name", "Location", "Amount From" and "To", and "Date Start" and "End". An "Apply" button is at the bottom of the sidebar. The main area of the application displays a table of invoices with the following data:

Invoice	Status	Method
INV001	Paid	Credit Card
INV002	Pending	PayPal
INV003	Unpaid	Bank Transfer
INV004	Paid	Credit Card
INV005	Paid	PayPal
INV006	Pending	Bank Transfer
INV007	Unpaid	Credit Card

Clicking 'Add Filters' now goes to a frame instead of a screen. That is, the result of the action keeps the user on the same page, and doesn't open a modal.

Here we implemented a side panel that slides out and contains all the possible filters. The user can bounce between each filter and then click 'Apply' to run the filters.

Even though we have the filters in their own panel, it doesn't count as a new screen because the user still sees and can interact with all the other UI elements. Their cognitive load is still high.

The workflow is now more complex, because there are many things the user can do on a single screen. But we know the users use this workflow frequently. So, if the interface makes sense, they will only get faster at using it.

Denormalization results in a faster workflow for our frequent users! Does that mean *everything* should be on one screen? No. There can be paths through the workflow that are used more often than others. We can still improve the usability by limiting what's on the screen to the **most used paths** through the workflow.

For example, instead of showing all the filters immediately, we could show the most commonly used filters first. Or, if what is common differs per user, we could add user settings where the user can choose which filters they want to see by default.

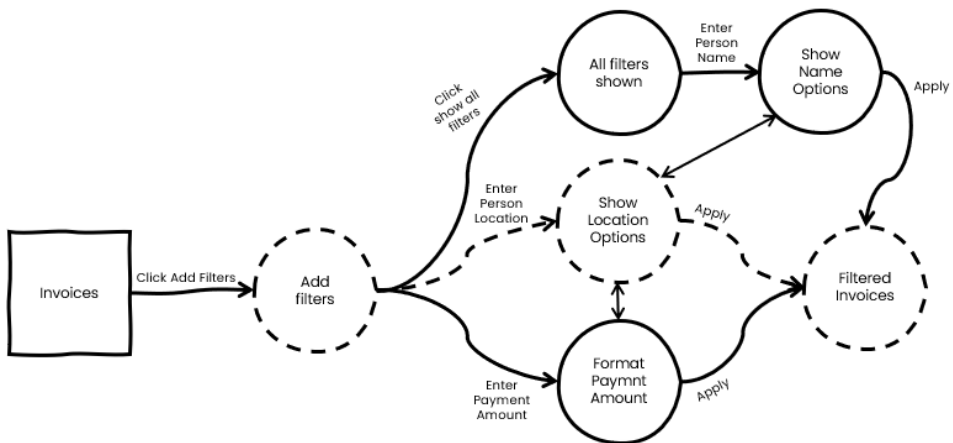
Then we can hide the rest of the filters in a new frame, limiting the amount of information on screen for the most used pathway. The screen might look like this:

The screenshot shows a web application titled "Invoices". At the top left is a blue button labeled "Add Filters". Below it is a sidebar titled "Filters" with a close button (X) in the top right corner. The sidebar contains two sections: "Person" with a "Name" label and an input field, and "Payments" with a "Date" label, "Start" and "End" sub-labels, and two input fields. Below these is a link "Show All Filters..." and a blue "Apply" button at the bottom right. To the right of the sidebar is a table with three columns: "Invoice", "Status", and "Method". The table contains seven rows of invoice data.

Invoice	Status	Method
INV001	Paid	Credit Card
INV002	Pending	PayPal
INV003	Unpaid	Bank Transfer
INV004	Paid	Credit Card
INV005	Paid	PayPal
INV006	Pending	Bank Transfer
INV007	Unpaid	Credit Card

The most common (or user-chosen) filters appear immediately, and the rest or hidden behind a 'show all filters' option.

If they click the option, the rest of the filters are shown. The workflow diagram now looks like this.



I've marked one of the "most used paths" with dotted lines. That path has less frames than the "show name options" one, which is hidden by the "show all filters" option.

Thus, even within a denormalized workflow, you can still optimize by making the most used paths the shortest ones. This even further increases efficiency for high-frequency users, and reduces cognitive load for the most common tasks.

## Should We Do Both?

What if some users are high-frequency and some aren't? Should we ever implement both normalized and denormalized workflows for the same task?

It certainly is possible to support both types of workflows at the same time. For example:

### Invoices

[Add Filters](#)

**Filters** ×

**Person**  
Name

**Payments**  
Date Start  End

[More Filters...](#)

[Apply](#)

Invoice	Status	Method
INV001	Paid	Credit Card
INV002	Pending	PayPal
INV003	Unpaid	Bank Transfer
INV004	Paid	Credit Card
INV005	Paid	PayPal
INV006	Pending	Bank Transfer
INV007	Unpaid	Credit Card

Keep the most used or user-chosen filters and add a “More Filters” option.

### Invoices

[Add Filters](#)

**Add Filters** ×

Choose Filter ▼

**Person**  
Name  
Location  
**Payments**  
Amount  
Date

[Apply](#)

INV006	Pending	Bank Transfer
INV007	Unpaid	Credit Card

When that option is clicked...it takes us to the modal from the original normalized version!

In these screens we kept the high-frequency users happy with common or user-chosen filters in a denormalized workflow but kept all the other filters in a highly normalized workflow. This could potentially work for both kinds of users.

However, implementing multiple workflow approaches for a single task should be rare. It can be tempting to simply always design a workflow for both cases.

But actually building both usually means extra development work, more maintenance, and more places where bugs can creep in. The technical debt is higher, and you can miss out on big usability wins.

For example, in this case, the high-frequency user still has to go through the more inefficient process for less used filters.

It is better to decide what workflow is most needed by the majority of your users and make your normalization decisions to optimize for them.

## **Mission-Critical Workflows**

An important note: sometimes a workflow is mission critical. You have to make sure users never get it wrong, probably because it can't be undone (like sending a mass email).

In that case a normalized workflow is better, even if users use it frequently. Normalized workflows are slower, but clearer and harder for the user to get wrong.

We'll talk more about mission-critical workflows later in the book, but it's worth mentioning here. The critical or permanent nature of a workflow overrides any other normalization considerations.

## Takeaways

When a workflow is frequently used, and speed is important, a denormalized workflow is ideal, unless the workflow is mission-critical or can't be undone.

You can still reduce the cognitive load of a workflow by splitting less used paths into their own frames or even screens.

Users who use the features in your app often will appreciate how fast they can get things done.

# Low-frequency/Low-complexity - You Might Normalize

If a workflow isn't used very often and isn't very complex, you may or may not need to normalize. Since it's low-complexity, users probably will be able to figure it out.

However, if you want to have the best user experience (and the most successful web app you can) you may want to normalize since the workflow is low-frequency. The easier a web app is to use, the better.

But how would you normalize a low-complexity workflow? It may simply have to do with the amount of data on the screen.

## Example: Settings

The number of user settings in an app tends to grow over time. The more features there are, the more settings the user may need to set.

But settings are rarely complicated, and rarely used. So, should they be normalized or denormalized?

It depends. If our settings have grown, we can simply implement good information architecture. That means how we choose the words and organize the information on the screen.

Once enough UI elements (words, forms, etc.) are on one screen, normalization can help deal with information overload.

For example:

## Settings

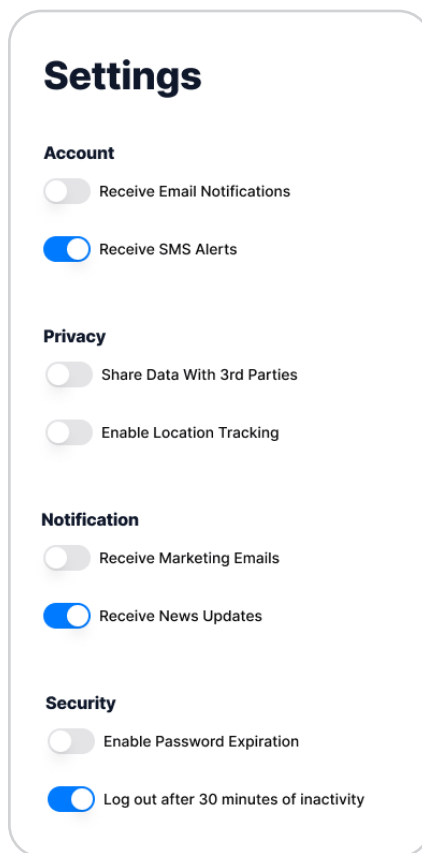
- ☐ Receive Email Notifications
- ☒ Receive SMS Alerts
- ☐ Share Data With 3rd Parties
- ☐ Enable Location Tracking
- ☐ Receive Marketing Emails
- ☒ Receive News Updates
- ☐ Enable Password Expiration
- ☒ Log out after 30 minutes of inactivity

Here we have a set of user settings in a web app. These will likely only grow as new app features are added.

This is a low-complexity screen. It is also low-frequency, users likely aren't coming to this screen often to update their settings.

You could likely get away with leaving the screen denormalized. All the settings are on one page. However, as **information density** (that is, the amount of information on the page) increases, it becomes harder for the user to process the screen.

We can start fixing this with good information architecture.



By sub-dividing the screen into sections, we make it easier for the user to parse the information.

This isn't perfect though, as settings grow the user will have to scroll around the screen a lot more and hunt for the right information.

For example, if they are looking to change their password expiration, the word "security" might not be immediately visible.

People generally don't read screens, they scan them. When there's a lot of information on a screen you want it to be immediately scannable.

We can take a more drastic step and fully normalize the workflow of updating settings.

You can think of good information architecture as a sort of steppingstone to normalization. When you divide information on screen with headers you are helping the user to focus on only one set of information at a time.

The words that you choose for those headers also help the user when scanning. They need to be words that match the words the user is looking for. If they are looking to change something about their password, then the words “password” and “security” will draw their attention while scanning.

This applies when normalizing the workflow as well. Links work similarly to headers. They draw the attention of a user scanning the page.

Moving from subheaders on the screen to full normalization has other benefits beyond scannability though.

Here’s what a fully normalized version might look like:

## Settings

[Account](#)

[Privacy](#)

[Notification](#)

[Security](#)

Dense information becomes  
scannable links.

[< Back](#)

## Security Settings

### Enable Password Expiration

Password expires every 60 days



### Logout after 30 minutes

Inactivity means you don't click  
anywhere. Mouse movements don't  
count as activity.



A normalized screen per type of  
setting leaves room for explanations  
and lowers cognitive load.

Normalizing a dense screen means splitting it into multiple screens. The entry point to the workflow now becomes a scannable set of links, rather than a screen the user has to scroll around to see everything.

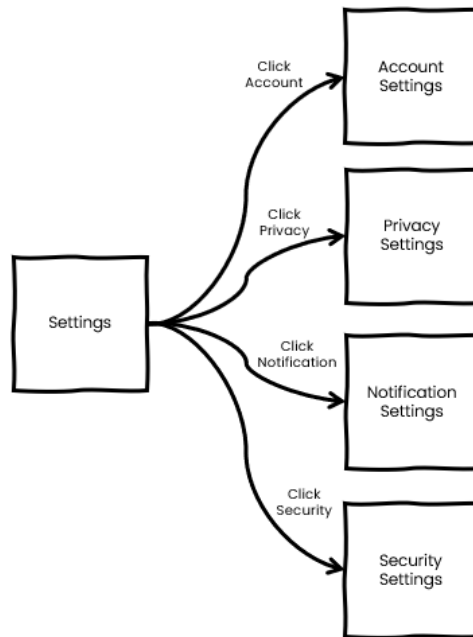
Once a user clicks on a link, we have a normalized, focused screen where we have room to breathe. For example, we can add explanations for each setting. On the denormalized version explanations would have only made the scrolling and scanning situation worse.

In a workflow diagram, sometimes I mark that a screen is particularly dense, either with an extra thick border or a double border.



A single, information-dense screen.

Then, after normalizing, I get rid of the special border to show that the density has reduced.



A normalized set of multiple screens.

Even though the workflow is low-complexity, high information density means the usability of the workflow benefits from normalization.

This is especially true when the workflow is low-frequency.

## Takeaways

When a workflow is low-frequency and low-complexity, you may not need to normalize it.

However, when screens have high information density (the amount of information on the page), normalization improves usability, especially if the workflow is low-frequency.

After normalization, screens are easier to scan, there is breathing room to help users through tasks, cognitive load is lessened, and there is room for the app to grow.

## High-frequency/Low-complexity - You Should Denormalize

If a workflow is used often and isn't very complex, you should keep it denormalized.

It isn't as urgent to do so as the high-frequency/high-complexity scenario, because speed won't be as hampered if there aren't many steps the user has to take.

There's one more consideration we add here: standards. If the effort to denormalize results in a lot of custom development work to build non-standard, not-built-into-the-browser (non-native) UI elements, that is a clue that you may need to normalize, most especially if the workflow is low-complexity.

### **Example: Sorting**

Let's suppose our list of invoices gets sorting functionality. The workflow to sort by a column is unnecessarily normalized into a modal.

**Invoices**

Sort

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

Data is sorted by column first by clicking the "Sort" button.

**Invoices**

Sort

**Sort** ×

Choose Column ▼

Status

Method ▼

Sort

INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

This opens a new modal to choose which column to sort by. After choosing the modal closes and the data is sorted.

If a user had to sort this data often, the extra steps of opening the modal, choosing a sort, and closing the modal is inefficient and gets annoying for the user.

So, we denormalize the workflow:

Invoices			
Invoice	Status ↑↓	Method ↑↓	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

The sort workflow is denormalized to simply clicking on the column header to sort the data by that column.

We denormalize using the simple, standard approach of making the column headers clickable.

The implementation has been condensed to interactions that stay on the same screen, and we see that it results in a low-complexity workflow.

Denormalization is the best choice in this case as it keeps the workflow fast for high-frequency users, and there still isn't a lot of cognitive load on them.

## **Difficult Requirements**

Sometimes new requirements send designers and developers down unnecessarily complicated roads.

Let's suppose a new requirement was added to be able to sub-sort the data. First sort by one column, then another. What would the UI look like?

There is a tendency in these cases to begin implementing non-standard interfaces. Custom components that need a lot of work and testing to be usable. Maintenance, accessibility, and usability problems abound.

Rather than coming up with some custom sort system for the data, we can choose instead to re-normalize the workflow. Normalized screens tend to give the space needed to stick to simple, standard UI elements and features that already exist in the browser (like standard form elements).

So, normalizing the workflow after getting this new requirement might look like this:

### Invoices

Sort

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

We go back to the data being sorted by first clicking on a "Sort" button.

### Invoices

Sort

#### Sort

×

First sort by:  

Status

Then sort by:  

Method

Sort

INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

This opens a new modal to choose which column to first sort by, and the secondary sort as well. We can adjust the options in the second dropdown based on what is chosen in the first.

By normalizing, we deal with difficult requirements without resorting to a lot of extra design and development work building unusual UI elements.

Normalized screens lend themselves to keeping interfaces standard and simple. Standard, simple user interfaces are more usable. When the user interface elements for a denormalized interface would require a lot of custom dev work, a normalized interface is best even for high-frequency scenarios.

## **Takeaways**

When a workflow is high-frequency and low-complexity, a denormalized workflow is generally best.

An exception is when keeping things denormalized requires extra dev work to build custom UI elements. In those cases, the usability benefits of normalizing and keeping the interface simple and standard outweigh the concerns of slowing down the user.

# The Four Quadrants

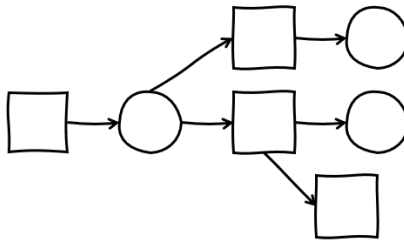
To summarize what we've discussed so far, you can think in terms of the four quadrants formed by our metrics.

	Low-frequency	High-frequency
High-complexity	Normalize	Denormalize
Low-complexity	Normalize (if it's dense)	Denormalize (unless it's difficult)

If the workflow is complex, then frequency determines whether it should be normalized or not.

If the workflow is low-complexity, then you should normalize if it is also low-frequency, a screen has high information density, or it's technically difficult to implement a denormalized solution.

# Normalization In Practice



# Normalization In The Real World

Normal UI is about making your apps easier to use. But never forget that the most important judges of what works well are your users. That doesn't mean they are designers though.

Users know what doesn't work for them, but that's different from knowing how to design what *will* work.

Don't make design a war of opinions. Make it a scientific process. We hypothesize that a particular interface works well. Then we test it with users.

Watching your users use your apps (without helping them) is the best way to know what works or doesn't.

That said, Normal UI gives you an extremely strong foundation for your design choices. You're much more likely to build something that tests well.

You'll find, if you watch and interview your users, that the concept of normalization often encapsulates what works and what doesn't.

So, now that we have the technique spelled out, we'll spend the rest of this book going through some practical tips and guidance for utilizing Normal UI in your real-world web app development work.

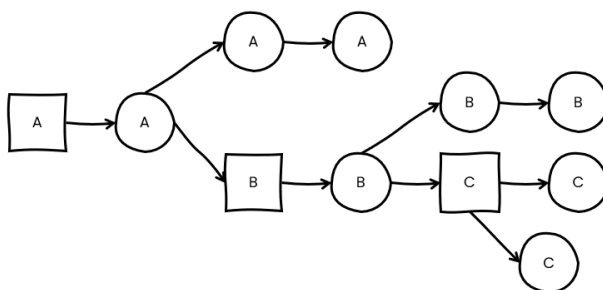
## Keep Workflows Small-ish

When you start using Normal UI, you may find that drawing out workflow diagrams is cumbersome. This will likely be because you are drawing workflows that are too big.

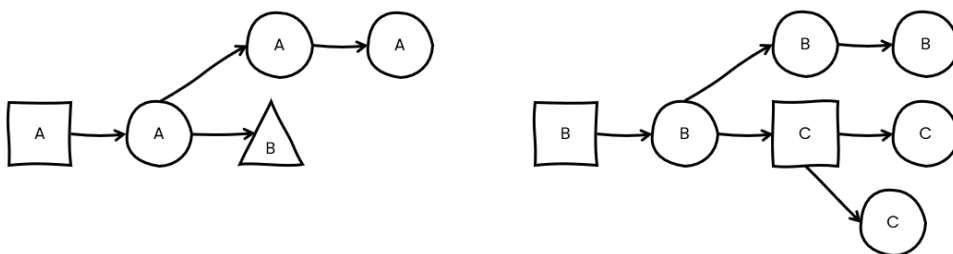
Remember that a workflow isn't a feature. It's a specific series of actions the user takes to accomplish a task. A feature is likely made up of many workflows. Take workflows that feel large and break them up into smaller ones.

You may be drawing out what really are separate, related tasks as if they were one workflow.

Sometimes, though, you really do have a very large workflow for one task that the user will carry out from start to finish. To keep that from becoming unwieldy to diagram, you can connect separately drawn workflows. Here's how I do it:



Suppose you have a workflow with one particular branch that is quite large.



You can split it into multiple workflows. The triangle marks a “portal” from one workflow to the other.

The key to splitting a large workflow diagram into smaller diagrams is keeping track of how they link to each other.

Here we took a branch of the workflow and pulled it out as its own diagram. Then, where that branch would go, we used a triangle to mark a kind of “portal” to the other diagram.

The label on the triangle is the same as the name of the label at the beginning of the other workflow (in this case “B”).

This is also useful because sometimes multiple paths lead to the same screen. You can use these “portal” markers in more than one workflow, and just reuse the workflow they point to.

The broader point is to keep your workflows small-ish. You’ll have to decide how big is too big for you. As soon as it starts to feel like a chore to keep track of what’s going on, split it so you can draw on a different whiteboard or piece of paper or digital page.

Drawing workflow diagrams should not become a required, unhappy task. They’re just a tool. Use them as much as you think you need to. As you use Normal UI more and more, the less you’ll need to rely on them.

## Determine Complexity on a Napkin

We've talked a lot about complexity as an important metric in determining whether to normalize or not.

But how do you use the complexity metric to make decisions if you're starting from scratch, or if you're working with existing software that is highly normalized?

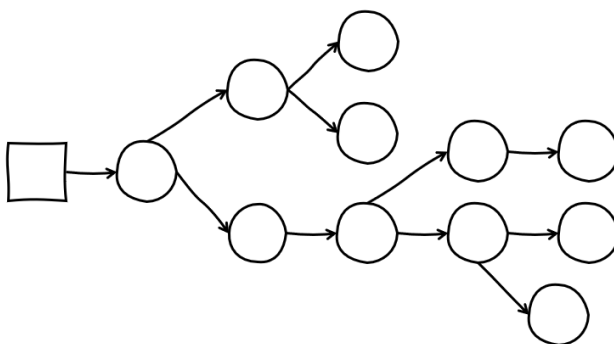
Remember we said we'd get into those questions in the second half of the book? Well, here we are!

The answer is actually quite simple. *Just start sketching what a denormalized version of the workflow would look like.*

You can either draw a workflow diagram, or just start sketching screens. They don't need to look good. Think more like a sketch on a napkin.

You don't need to finish them either! You'll reach a point where you'll realize whether the denormalized version would be complex or not. You can then decide where the workflow fits in the four quadrants.

Let's say, for example, that you're designing a new feature. You decide to draw a possible workflow in that feature. It ends up looking like this:



That's complex! You now know one of the metrics.

Now let's suppose you realize the workflow will be used infrequently. The combination of low-frequency and high-complexity means you should normalize! So you build a normalized version of this workflow, reducing the complexity for the user.

Conversely, if the workflow was going to be used at high-frequency, you'd keep a more denormalized one.

Run the same process if you're dealing with existing software that's already normalized. Just start sketching out what the denormalized version would look like. That will give you the sense of complexity that you need to make a decision using the four quadrants.

Of course, if the workflow is dangerous or a denormalized version is technically difficult, you don't even need to go this far, since you know it the workflow should be normalized.

I've found looking at the denormalized version of any workflow gives a reliable measurement of complexity that can be used to determine what really should be done. You don't need to draw the final version, the exercise itself will reveal complexity quite quickly, and that's what's important.

To recap: when starting from scratch or working on an existing normalized workflow, "determine complexity" means "start sketching a denormalized version". That sketch, even if incomplete, gives you a quick measurement of complexity, which lets you determine a course of action.

This should not be a long, arduous process. Determine complexity on a napkin, or the digital equivalent of one.

# Normalize Dangerous Workflows

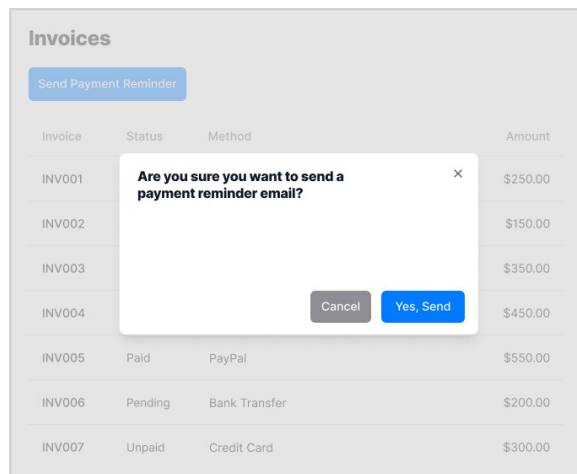
We previously mentioned that you should normalize “mission-critical” workflows. I also like to call these “dangerous” workflows.

Any workflow that ends in something that is vitally important the user gets right or cannot be easily undone you should consider as “dangerous”, and you should heavily normalize it.

For example, let’s suppose a button sends a mass email, which then requests confirmation from the user.

Invoices			
<button>Send Payment Reminder</button>			
Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

The “Send Payment Reminder” button will initiate the workflow to send automated emails to customers who haven’t paid.



A confirmation dialog/modal.

For dangerous workflows, don't consider confirmation popups or modals as a form of normalization.

The more popup confirmation boxes there are in your app, the more people become numb to them. They will tend to click through them without reading.

Instead, try normalizing the process into either a sequence of screens or a separate page instead of a modal or popup.

For example:

### Send Payment Reminder Email

You are about to send 134 payment reminder emails. **This cannot be undone.**

Are you sure you want to send these emails?

No, Go BackYes, Send 134 Emails Now

A separate page allows for space to give further details about the impact of the action and ensure the user doesn't blindly click through confirmations.

As we've seen before, normalization provides breathing room to make the user interface clearer or draw more attention.

In this case we tell the user exactly how many emails they are about to send and can make the impact of the action scarier.

We give the user reason to pause and prevent them from simply clicking through a confirmation popup without thinking.

If we were more worried, we could first show a summary of the number of emails about to be sent, then a 'Next' button, then a final confirmation.

Normalization works wonders for dangerous workflows. Less accidental but impactful actions mean happier users, and less time for devs trying to help undo problems your users are having.

It's a win for everyone.

## Workflows Form the App Skeleton

Thinking in terms of workflows and normalization also means you've done the work to help structure the app itself at the file level.

In front-end development you may be thinking in terms of pages, files, routes, components, or some combination of such when you're initially scaffolding your app.

Thinking through what steps in the workflow will be new screens can translate directly to getting your app code started, even if you don't know every detail of every screen yet.

Sometimes in real-world work you need to get some coding work started even while prototyping and design work is going on. That may not be ideal, but in those cases you can save future refactoring work by at least thinking through your workflow and normalizations before starting any code.

# Normalized UIs Have Dev Benefits

Normalized interfaces tend to have concrete technical benefits for developers. If you are building out a normalized workflow, be sure to take advantage of them!

## You Can Stick to Native Controls

We already mentioned that a benefit of splitting work across multiple, less dense screens is that you tend to be able to stick to native controls.

It's worth mentioning why native controls (UI elements built into the browser) are better.

Using native browser controls, like inputs, dropdowns, and dialogs, means less code for the user's device to download and execute.

It means the experience will be optimized for different types of devices. For example, native dropdowns have an optimized experience on mobile browsers.

Native controls are also accessible out-of-the-box. For example, they are usable for individuals using assistive devices like screen readers.

Ultimately native controls mean less technical debt. Thus, it's worth repeating: if you find yourself building custom UI elements with JavaScript, consider normalizing that technical complexity away.

## Lazy Loading and Prefetching

Browsers provide functionality so that, in code, you can load portions of a page after the page initially loads. That's lazy loading. You can also load things you will need later before you need them. That's prefetching.

When you normalize your interface, you are already making it friendly to lazy

loading and prefetching. When you split workflows into separate screens, you are also splitting code into separate files.

Separate files mean not everything has to be loaded at once. JavaScript frameworks and techniques that implement lazy loading and prefetching will be able to take advantage of that to speed up the app and reduce how taxing it is on the user's device and internet connection.

The point is that normalization can lead to performance benefits, if a dev takes advantage of it.

## **Smaller Components**

Building web apps often involves building components (individual, reusable pieces of UI and code).

By normalizing workflows across separate screens, you also tend to split work across components. You end up with more components, but smaller ones.

Smaller components mean less code per component, making the component easier to reason on and easier to debug. You tend to have less bugs-per-component overall.

That doesn't mean you can't split a denormalized interface into many small components, but you have to be aware of how they all interact with each other.

With a normalized interface, you end up with less to think about without even trying.

# Handle Scope Creep Through Normalization

Scope creep is a reality of software development. Good process can help minimize it, but that isn't what this book is about.

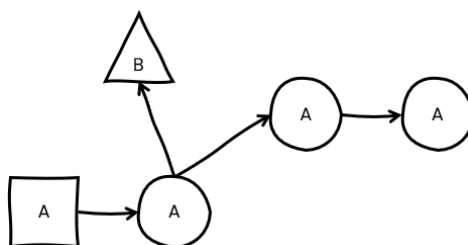
What normalization can do is help prevent scope creep from intensifying the complexity of your code.

If requirements begin adding complexity to a screen, consider normalizing that complexity away onto separate screens.

You keep the code to handle that feature isolated, and you have to a way to yank that feature out of your codebase if it's realized that the scope creep has gone too far.

This is especially true if features being added won't be used often.

When a new requirement comes through late in development, sketch out what the workflow would be for a normalized version. If it isn't prohibitively inefficient, then keep that branch of the workflow diagram on its own screen.



Scope creep adds new workflow branches, so we isolate them onto their own new screen B.

# Provide Context on Normalized Screens

A downside of normalization is that the user loses context when they move to a separate screen.

They can no longer see the information on the previous screen, which might be needed to understand what they now see or the decisions they now need to make.

Fix that by determining what information is needed, and *only* what information is needed, from the previous screen and repeat it on the new screen.

For example, remember the normalized filters modals we worked with earlier? There's a potential loss-of-context problem.

The screenshot shows a web application interface for 'Invoices'. At the top, there's a header 'Invoices' and a blue button labeled 'Add Filters'. Below this, a modal window titled 'Payments: Amount' is open. The modal has a close button (X) in the top right corner. Inside the modal, it says 'Enter a range' and has two input fields: 'From' with the value 'Low' and 'To' with the value 'High'. A blue button labeled 'Add Filter' is at the bottom right of the modal. In the background, a table of invoices is visible, showing columns for invoice ID, status, payment method, and amount.

Invoice ID	Status	Payment Method	Amount
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

When you go into a particular filter, you no longer see on screen what *other* filters you've already chosen.

People's memories are faulty, and they may get distracted by other things while in the middle of a workflow.

Providing context prevents the user from having to remember what they were just doing.

For example:

Invoices

**Payments: Date** X

Enter a range

Start  
Low

End  
High

Add Filter

Payments (Amount): From: \$50 / To: \$100

INV007 Unpaid Credit Card \$300.00

Here we've added at the bottom of the modal the list of filters that have already been chosen. Now the user doesn't have to remember information from previous screens when making decisions on the current screen.

We reduce cognitive load on the user, while still keeping the amount of information on normalized screens to a minimum.

Whenever you normalize, don't forget to add the needed context.

# Make Clear Calls-to-Action

For every action in your workflow, make sure there's a clear call-to-action on the screen. That usually means a button or link, along the words you choose within them.

Consider what words the user is thinking of when they are trying to take that action, and make sure the button or link says one of those words or an obvious synonym.

In the payments email example, the user is thinking about sending people a reminder that they need to submit a payment. So:

Invoices			
<a href="#">Send Payment Reminder</a>			
Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00

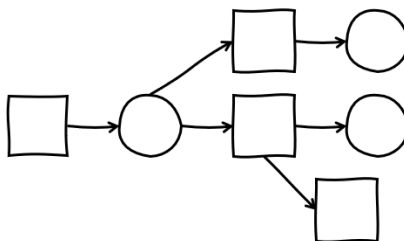
The button “Send Payment Reminder” is a clear call-to-action. When the user is scanning the screen, those words will draw their attention.

If the button just said “Send Email” it would be less clear. Send email for what?

If the button said “Automated Messaging” that would be even worse.

Devs tend to name things according to database columns and under-the-hood processes. Instead, be certain calls-to-action match the user’s vocabulary.

Check that each action in your workflow has a clear call-to-action and your app’s usability will improve dramatically.



Make sure arrows (actions) in the workflow diagram have clear calls-to-action in the real app.

# Help Users Recover From Errors

Errors are hard on users. The more difficult it is for a user to recover from an error, the worse their experience will be in your app.

Even if your app has terrific usability in other areas, there is a phenomenon called “negativity bias” where people will focus more on the bad than the good.

So, if your error recovery experience is bad, users will be more likely to feel your app is poorly designed in general.

You don’t want to let users do things that they shouldn’t, and you have to respect business rules, but you can consider dealing with errors as a step in the workflow, and thus something that can be normalized.

For example, suppose in our invoices app we were dealing with some old data, so not every customer has an email address in the system.

After running the payment reminders email workflow, we might see a message like this:

### Invoices

Send Payment Reminder

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	B:	<b>Payment Reminder Errors</b> 3 recipients missing email addresses.
INV007	Unpaid	C:	

This isn't bad, in that it informs the user of the error. But it doesn't help the user *recover* from the error.

One way to jumpstart error recovery is to normalize the error process itself. So, if there are errors, you might take the user to a separate error screen.

### Could Not Send to Some Recipients

3 customers do not have email addresses in the system.

**What would you like to do?**

[Go to recipients without email addresses](#)

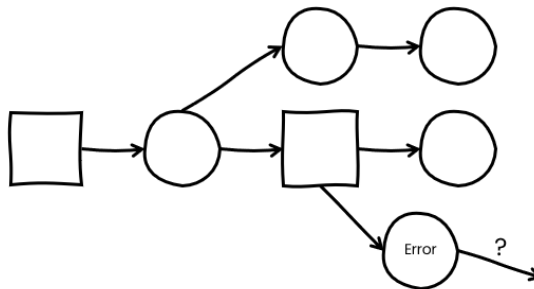
[Print reminders for mailing](#)

[Go back to the invoices pages](#)

Now we get the user's attention (no change blindness), inform them of the problem, and give them links to possible avenues to fix the problem or complete the task in a different way.

Here's a way to think about error recovery: if an error state is a frame in your workflow diagram, *are there any actions that take the user from that error frame to somewhere that helps them resolve the error?*

Error recovery is another usability superpower that normalization can give you. And, since errors hopefully don't happen that often, they are likely a low-frequency part of the workflow that is fine to normalize.



The user can reach an error frame in the workflow.  
Then what? What if the frame was a screen?

# Tell Users They Were Successful, and Help Them Keep Going

If you choose to normalize, don't forget that "success" is also a step in the workflow. Giving the success step some attention can yield usability benefits.

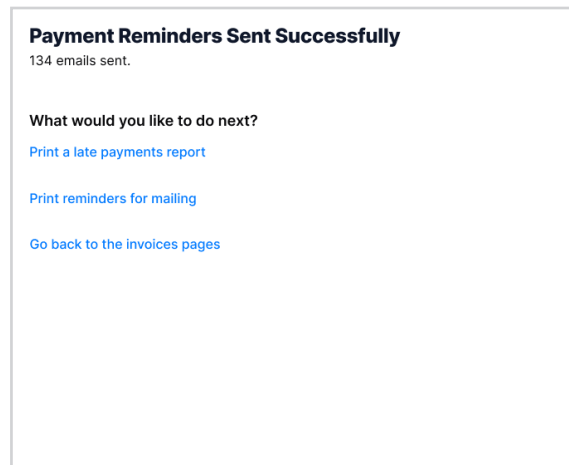
When something works in the app, it's common to inform the user with a new frame, not a new screen. Like a toast popup.

The screenshot shows a web interface titled "Invoices". At the top left is a blue button labeled "Send Payment Reminder". Below it is a table with four columns: "Invoice", "Status", "Method", and "Amount". The table contains seven rows of invoice data. A white toast message box is overlaid on the bottom right of the table, displaying a green checkmark icon, the word "Success", and the text "134 emails sent.".

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bi	
INV007	Unpaid	C	

Again, we've informed the user of their success, which is great. But, for workflows that you're normalizing anyway, there is opportunity to smooth the path for the user towards the next thing they may want to do.

You can normalize a success message over to its own page.



With normalization we can give the user more information, and calls-to-action to help them move on to other related workflows that they may want to carry out.

Here we have a callback to our database inspiration for Normal UI. Normalized tables in a database help you organize related data.

Normalized screens help you connect users to *related workflows*.

You should consider how a normalized success screen could help your users. Remember, this is for normalized workflows, not high-frequency workflows where this might slow the user down.

On the other hand, in some cases, the links to other workflows might speed the user up!

Normalization and denormalization are tools in your toolbox. Use what works for the situation you're dealing with.

The easier it is for your users to accomplish their tasks, the happier they will be.

# How To Talk About Normal UI

Whenever you learn a technique, it's good for everyone on the team to learn the terms so everyone is speaking the same language.

But how can you talk about Normal UI in your team? Here's some ways to refer to this process:

1. You can say you're "implementing Normal UI".
2. You can use the verb and adjectives forms of "normalize" and "denormalize". Like "we should normalize this workflow" or "I think we need a denormalized workflow".
3. If you want to sound fancy and technical, you can use the term "interface normalization". That's what I called this technique when I first decided to formally write it down years ago.

Any of these terms work. As long as you and your team members all understand them.

Clear communication between team members is vital to successful web development. If you and your team members speak the same technical language, your meetings get shorter and work gets easier.

# Recommended Reading

If you have read this book, then you are interested in making your web apps easier to use. That's great! Here are my recommendations for other books on the subject.

I've recommended these books to many students over the years, and they always have a big and useful impact.

## **Don't Make Me Think (by Steve Krug)**

This book is a clear and easy read, and gives you concrete tips on how to make your websites and web apps more usable.

## **Rocket Surgery Made Easy (by Steve Krug)**

A guide on how to run usability tests inexpensively and on your own.

## **The Design of Everyday Things (by Don Norman)**

A modern classic that ties software usability to the usability of everyday objects. You'll never see doors the same way again.

## **The Psychology of Human-Computer Interaction (by Card, Moran, and Newell)**

This huge book was released in 1989. Doesn't matter. If you want to get deep (and I mean deep) into theory, this is still my recommendation.

Fair warning: if you read all these don't be surprised if you become a usability nerd and see usability problems everywhere you look.

# Conclusion

You'll find as you gain practice at implementing Normal UI, it will become second nature to look at an app or prototype and start normalizing and denormalizing in your head.

That's one of the usability superpowers that comes with Normal UI.

I hope as you use it, you come to appreciate how asking yourself a few simple questions, and drawing a few simple drawings, can help you make your web apps much easier to use.

And I hope to see you again! There's a companion video series to this book you can find at [normalui.com](https://normalui.com), where I walk through normalizing and denormalizing common web app features. Plus, I'm in the video forums answering questions.

You can also find me, my social media links, and my courses on my website: [tonyalicea.dev](https://tonyalicea.dev). Or drop me a line at [hey@tonyalicea.dev](mailto:hey@tonyalicea.dev).

Happy normalizing!

- Tony Alicea