Reducing Risk and Building Maintainable Systems with Encapsulation

18F Ed Mullen | @edmullen

Hi.

18F

Ed Mullen
18F Strategist

- Hi, I'm Ed Mullen
- I'm a strategist at 18F
- 18F is a tech consultancy within GSA
  - Fed
  - We do
  - We work with federal and state agencies
- This is our team
  - That's me in the back
  - I'm always in the back of photos
- I've worked with
  - CMS, FNS, ACF
  - several states on projects in the health and human services space
  - Prior to 18F I worked at HHS for a while
  - Lead user experience for HealthCare.gov in 2010 and again in 2012/13

We want to help people get the benefits they deserve.

- We work in this field because
- we want to help people get the benefits they deserve
- and are entitled to.

We rely on technical systems to do this.

- We rely on a variety of technical systems to ensure
  - eligible people get enrolled
  - benefits are issued
  - providers get paid for the care they provide.
- Program success is dependent on the efficacy of the underlying technical infrastructure.

We want these systems to work well.

- Compliance is not the goal.
- We want these systems to work well.
- We've seen the impact these programs can have.
- We know they save lives.
- Maybe we were once enrolled in these programs, or members of our families.

But none of these systems are perfect.

- But none of these systems are perfect.
- If we’re honest, many are really struggling
- Everyone has either outdated legacy systems
- or expensive newer systems we barely understand
- Few can quickly deploy new functionality or tools.

Modernizing these systems is hard.

- Improvement efforts are often problematic
- I don’t need to tell you all, or give examples

13%

of major government software projects succeed.

The Standish Group

Why?

- Waterfall development methods
- Very large projects
- Up-front requirements gathering
- Diminished technical capacity within government
- Long, high-dollar vendor contracts

- There are many reasons
- Some include READ

Our new report speaks to these issues.

Robin Carnahan

- We recently released a new report on how to avoid these problems.
- One of the authors is here
- It can help you set projects up for success by
    - asking the right questions,
    - identifying the right outcomes,

Basic principles of modern software design

- User-centered design
- Product ownership
- Agile software development
- DevOps
- Modular contracting
- Building with loosely-coupled parts

The report details the basic principles of modern software design.

- **User-centered design -** Ongoing research with users and prioritizing around their needs
- **Product -** Understanding where you're headed, balancing user and business needs, course correcting as needed
- **Agile** - Developing working software delivered in short cycles, always re-evaluating as you go
- **DevOps** - Sometimes DevSecOps - Powering that continuous delivery to users through deployment automations
- **Modular contracting** - Contracting methodology that supports agile, smaller agreements, focused on objectives not deliverables.
- **Loosely-coupled** -
    - Technical architecture approach
    - discrete components
    - communicating through APIs,
    - as opposed to a tightly-coupled monolithic architecture.

But how do we get there?

- That’s all fine, but how do we get there?

As always, bit by bit.

- As always, bit by bit
- Incremental work most fundamental thread in 18F's work
- Core to modern software development

Basic principles of modern software design

- User-centered design
- Product ownership
- Agile software development
- DevOps
- Modular contracting
- Building with loosely-coupled parts

- Each concept is an area of practice worth growing
- These concepts are reinforcing
- For example,
  - hard to follow user needs when contract has fixed functional deliverables
  - Not getting benefits of practicing Agile if you can't continuously deploy to users

Basic principles of modern software design

- User-centered design
- Product ownership
- Agile software development
- DevOps
- Modular contracting
- **Building with loosely-coupled parts**

- focus on this aspect
- can feel like the most overwhelming aspect.
- Especially for our context
  - struggling with either legacy systems
  - black-box system that was built for us that we don't fully understand
- If you have a large, monolithic, or mission-critical system, modernizing the system is daunting
- **How can you start to unwind such a system?**

We just have to rebuild the whole thing, right?

-

Hold on now, please don’t do that!

-

13%

of major government software projects succeed.

The Standish Group

- Remember what we said before

Strangler (fig) pattern

or

Encapsulation

or

Encasement strategy

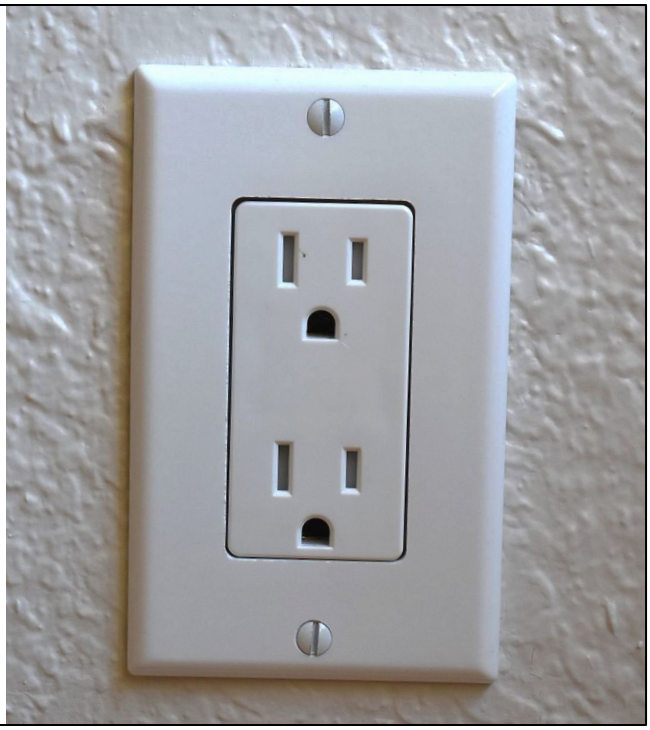
- We’re going to look at a technical strategy that addresses just this situation.
- different names, same concept
-

Strangler (fig) pattern

Martin Fowler

- Sometimes it's called The strangler pattern comes from Martin Fowler
- refers to the Strangler Fig, which starts in the upper branches of a host tree
- Grows and grows and eventually overtakes the host tree, strangling it
- But "strangler" is a pretty violent term, and I avoid it for that reason.
- Martin Fowler does too

Encapsulation



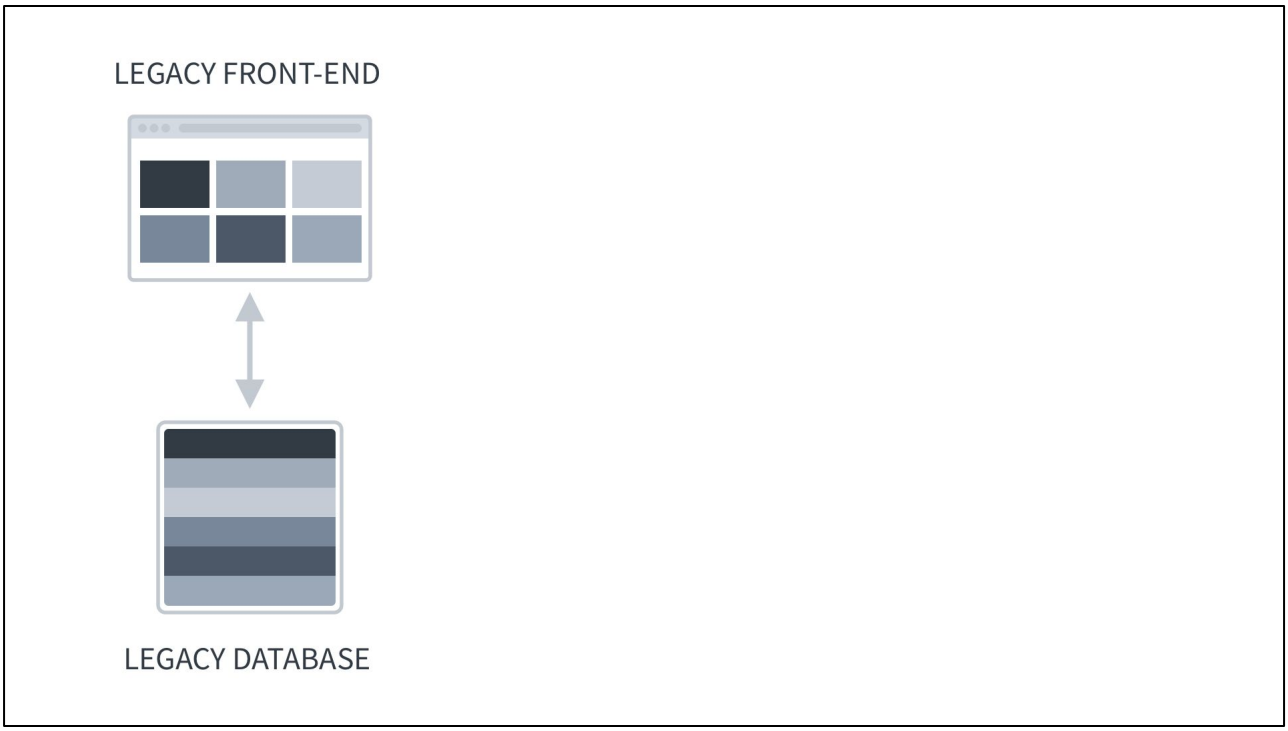
- Encapsulation is very similar
- It’s the idea that you shouldn’t need to know how a system works to use it.

Encasement strategy

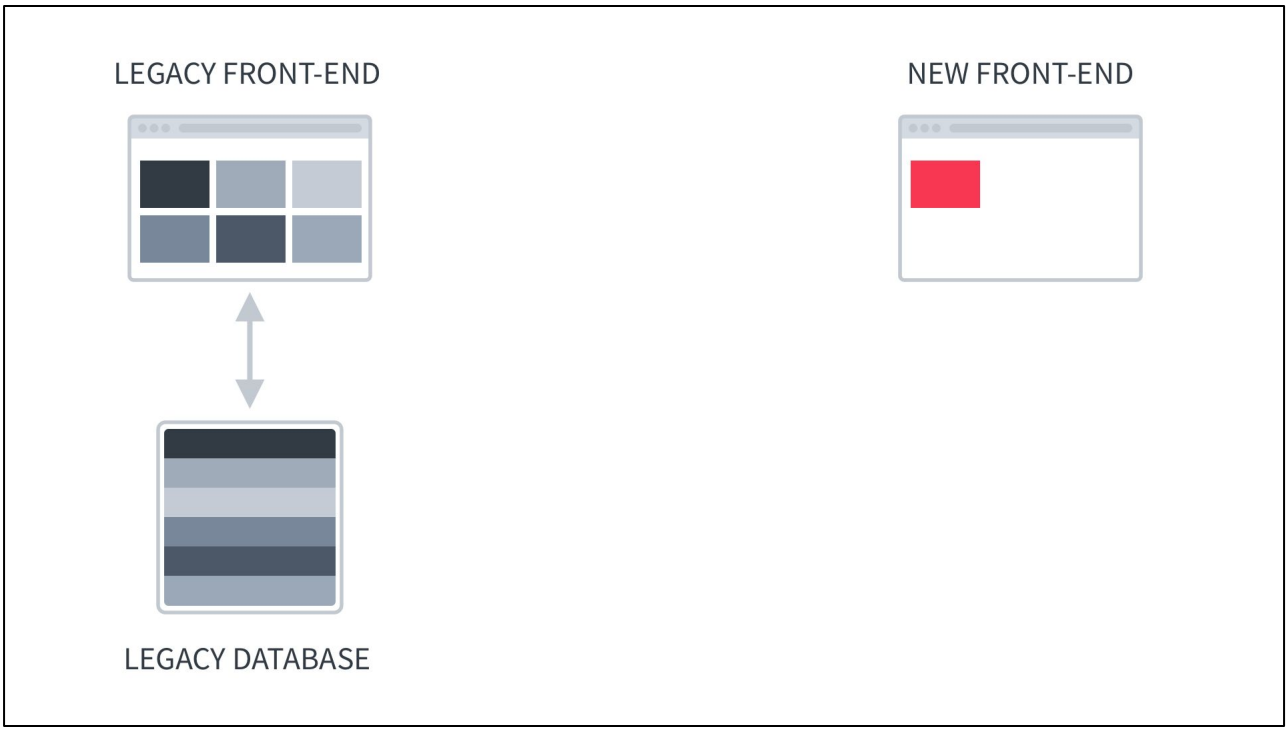
- The encasement strategy is a term we've used at 18F for a while
- It relates fragile legacy systems to a radioactive nuclear site like Chernobyl
- both are things that are scary to touch
- It's a bit hyperbolic, but this is the term I tend to use

The basic idea

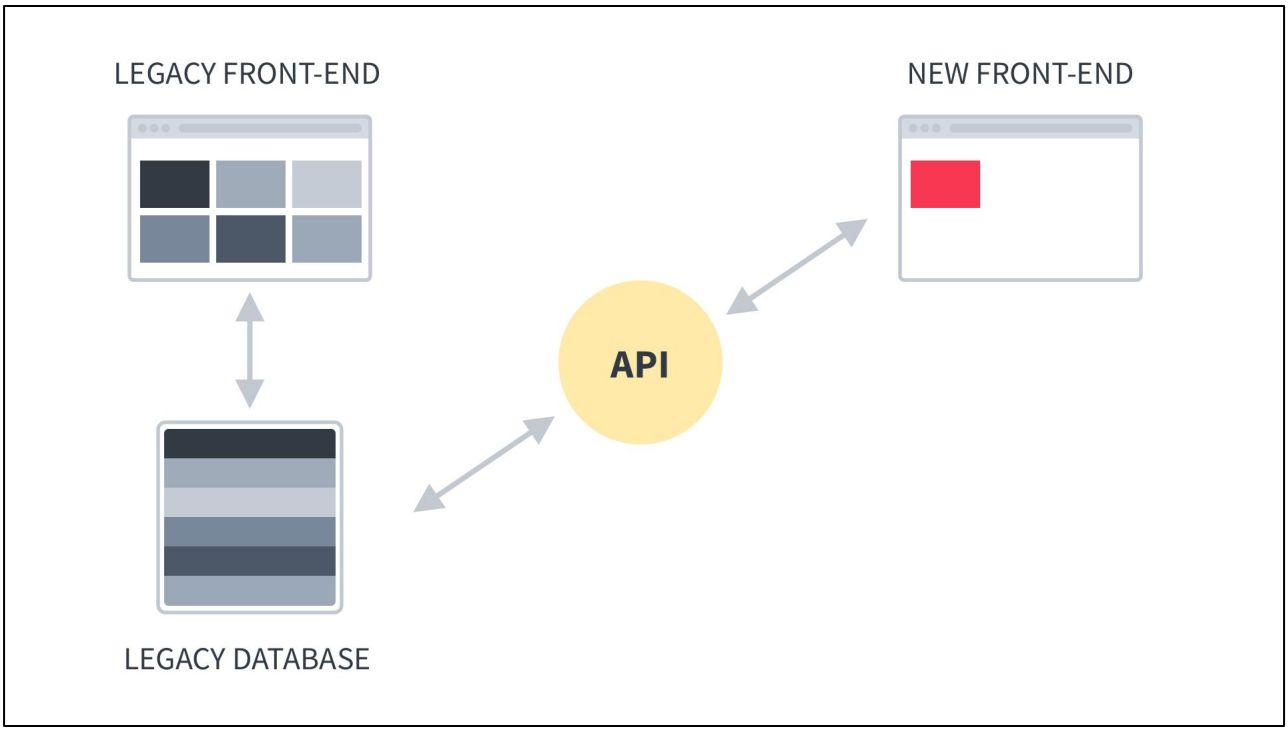
- So here’s the basic idea



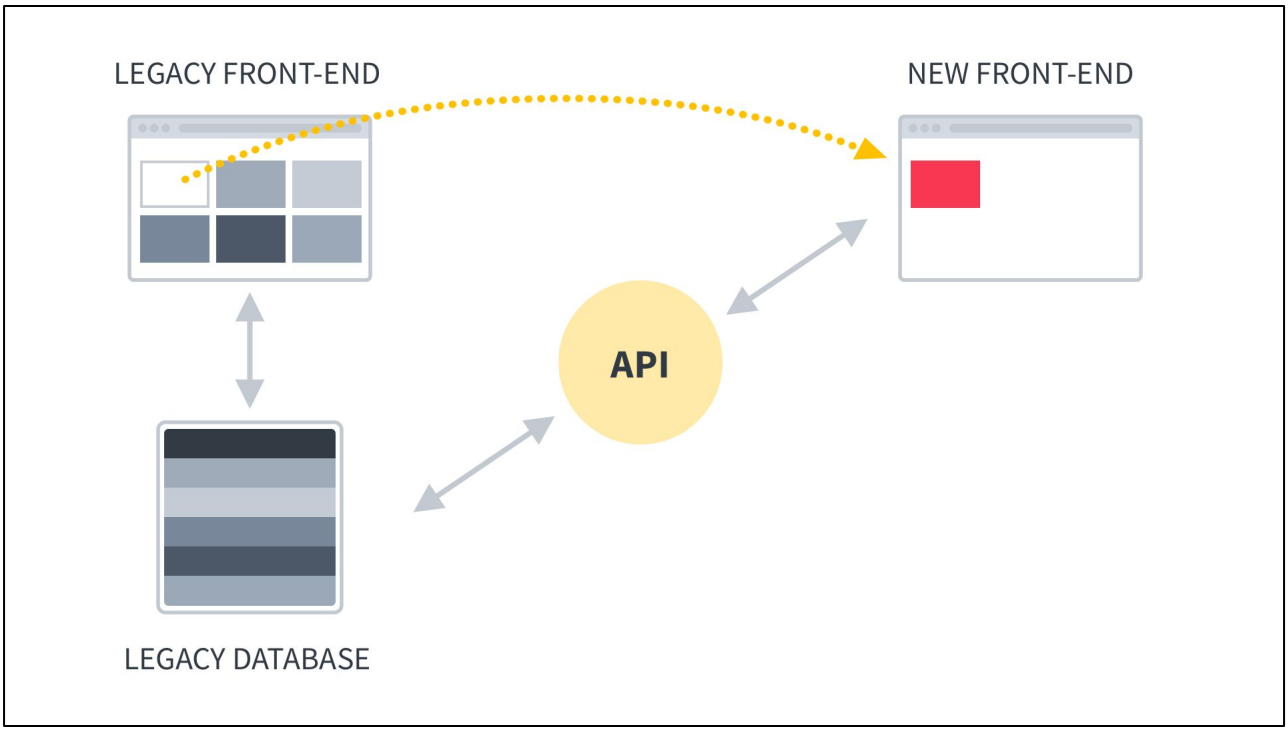
- Here’s the basic idea
- You’ve got your existing system, a front-end interface and the backend components
- They’ve been around for a long time, have a bunch of brittle pieces, and people are afraid to change it too much because everyone relies on it and no one is sure how it all works



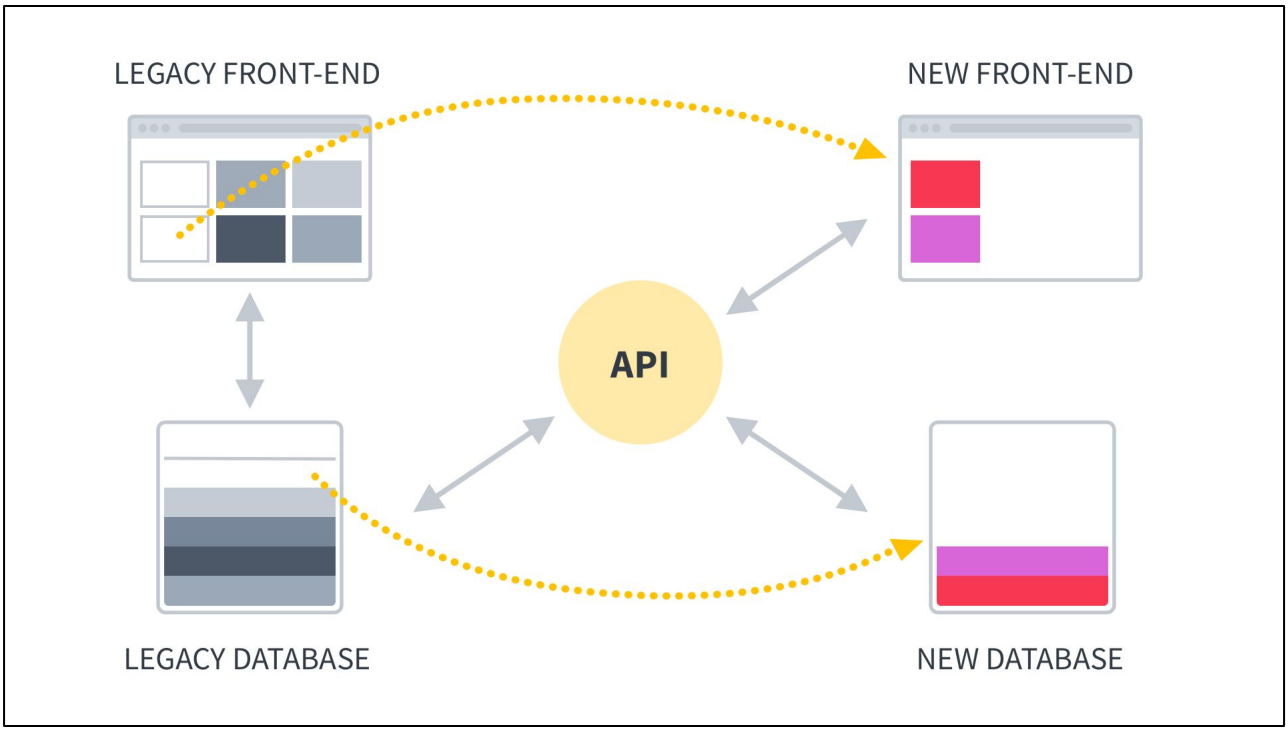
- Using the Encasement strategy
- If you want to build a new component, you build new functionality next to the old system, without really touching it too much



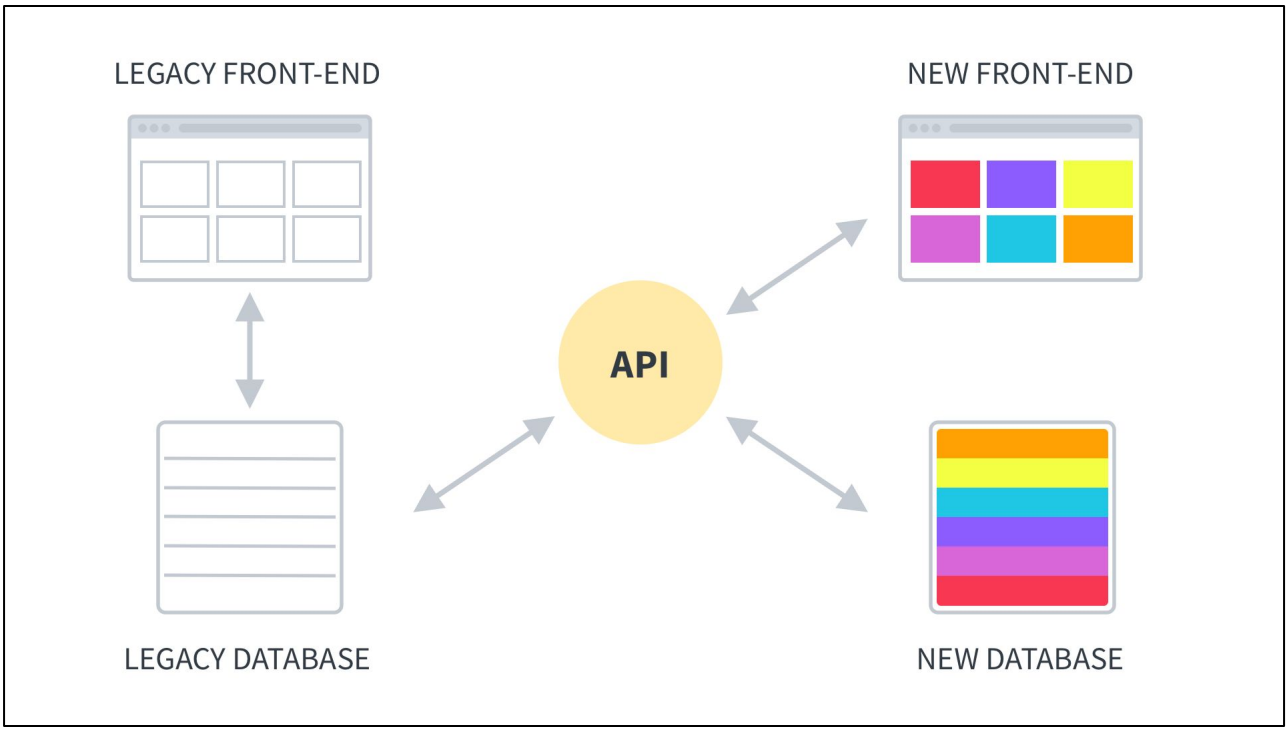
- In order to use data or functions from the legacy system
- An API layer is built.
- API means Application Programming Interface, and in general is a way for one application/program/system to interact with another
- The API here is sort of translation layer between systems, allowing them to communicate regardless of the technology used
- It allows the new system to send and receive the necessary data without having to worry about what's going on inside the old system
- It also acts as a pivot point, loosely coupling the new system to the legacy system



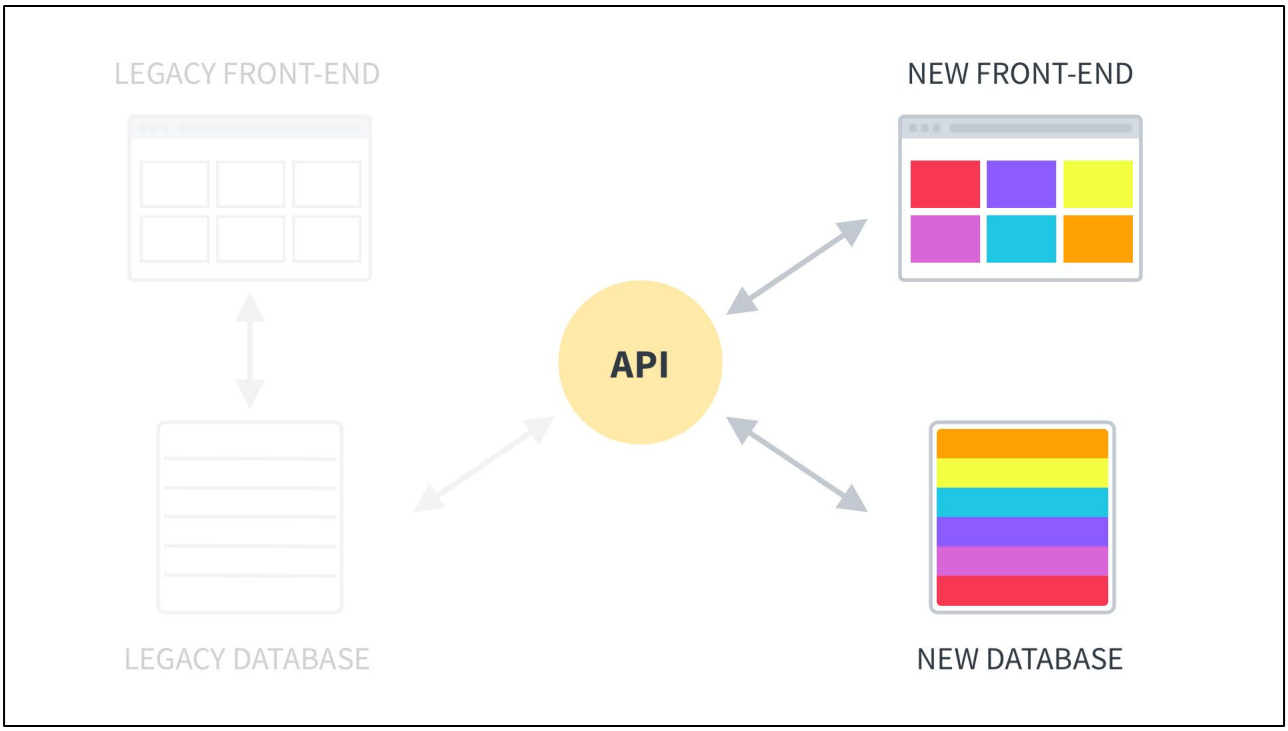
- New functionality gains adoption
- You turning off old functionality as it is replaced



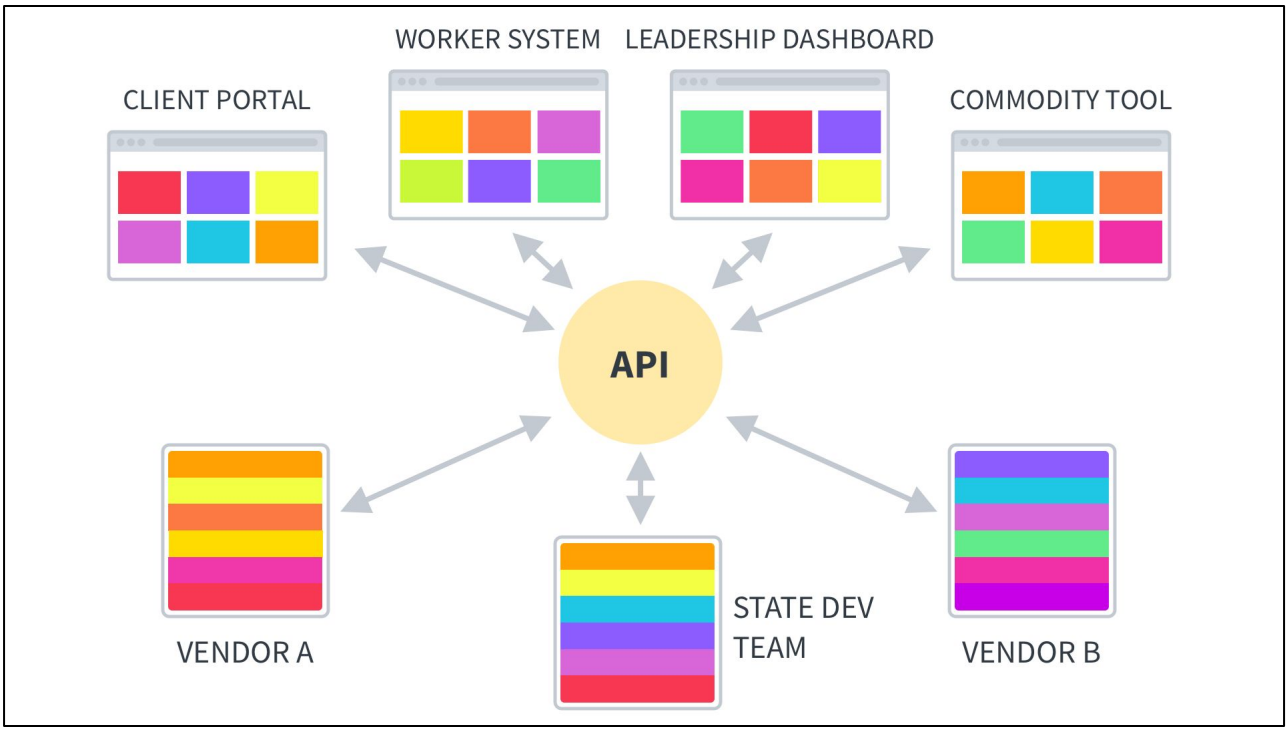
- You do the same thing with the backend systems
- Gradually shifting responsibilities from the old backend to new backend



- Once you've shifted all the front and back end functionality over to the new system,



- You can shut down the old system
- That means you can shut down that on-prem data center, having moved everything to the cloud
- Or dedicate ongoing resources to run all the manual processes
- (other modernization goal)
- There was no big bang launch here, just gradually rolling out new features, turning off old parts, until you finally have everything modernized
- But really, this image is a bit of a lie



- Developing this API layer opens up a wide range of possibilities
- The ability to build separate, smaller, more focused tools
- Involve multiple vendors in smaller procurements
- Leverage in-house talent to build components
- This is what we mean by “loosely-coupled architecture or systems
- Different components, interacting with each other through well-defined APIs
- each system doing its own thing, blissfully unaware of the other systems
- but getting everything they need through the APIs

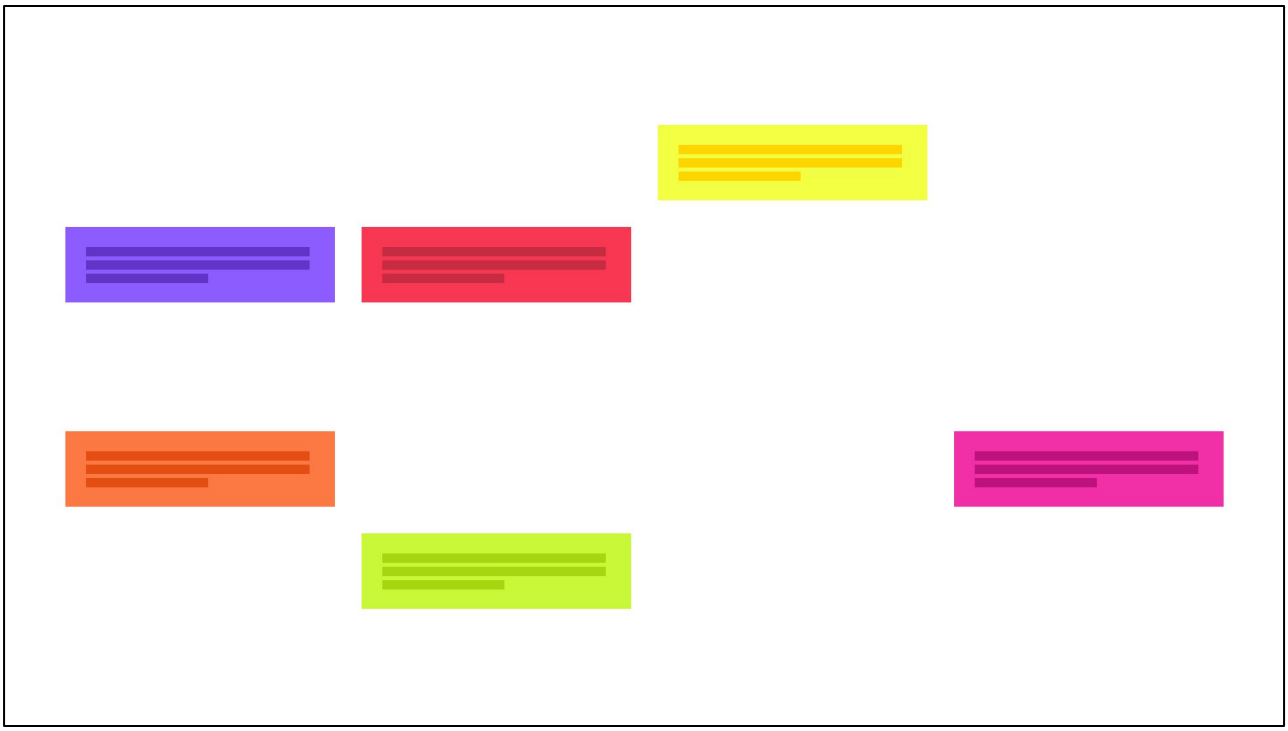
Looking a bit deeper

- So now let's look at how the encasement strategy can actually transform an application people use

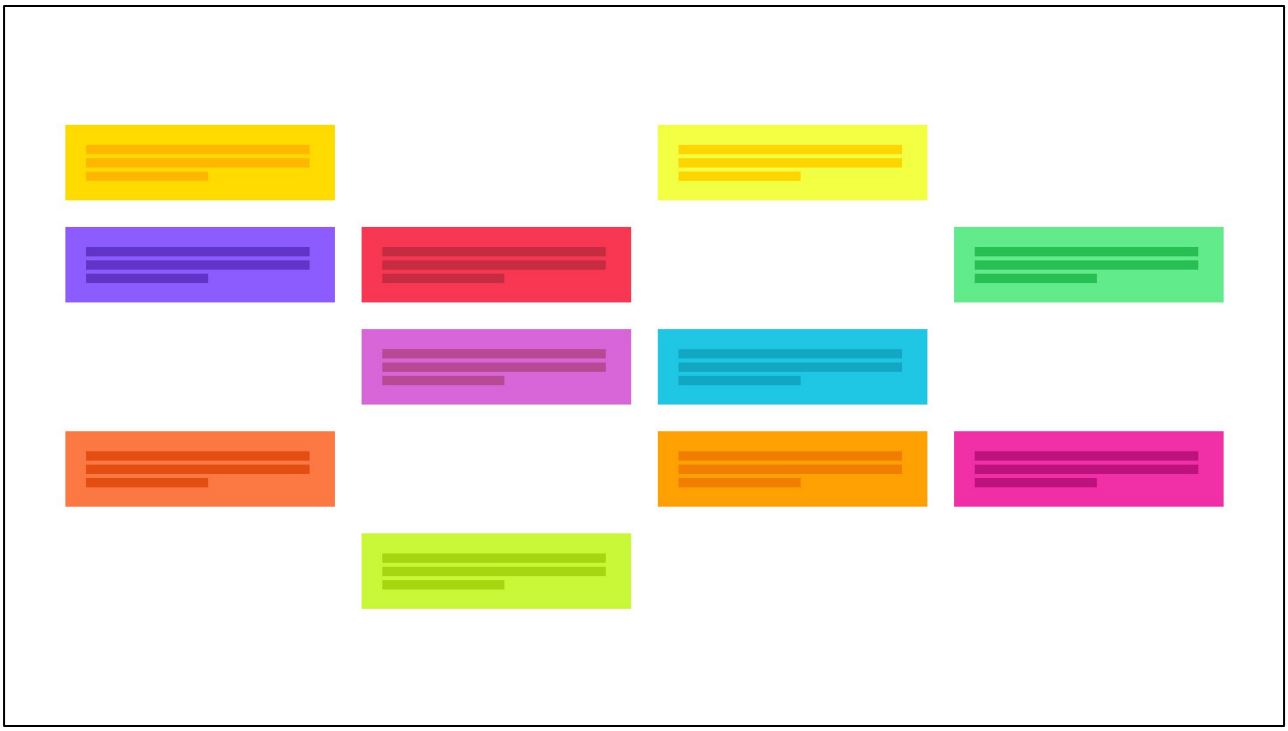
- So here is our legacy system
- Let’s say this is an application eligibility workers use to do their work

- We’ve got a homepage or start screen.

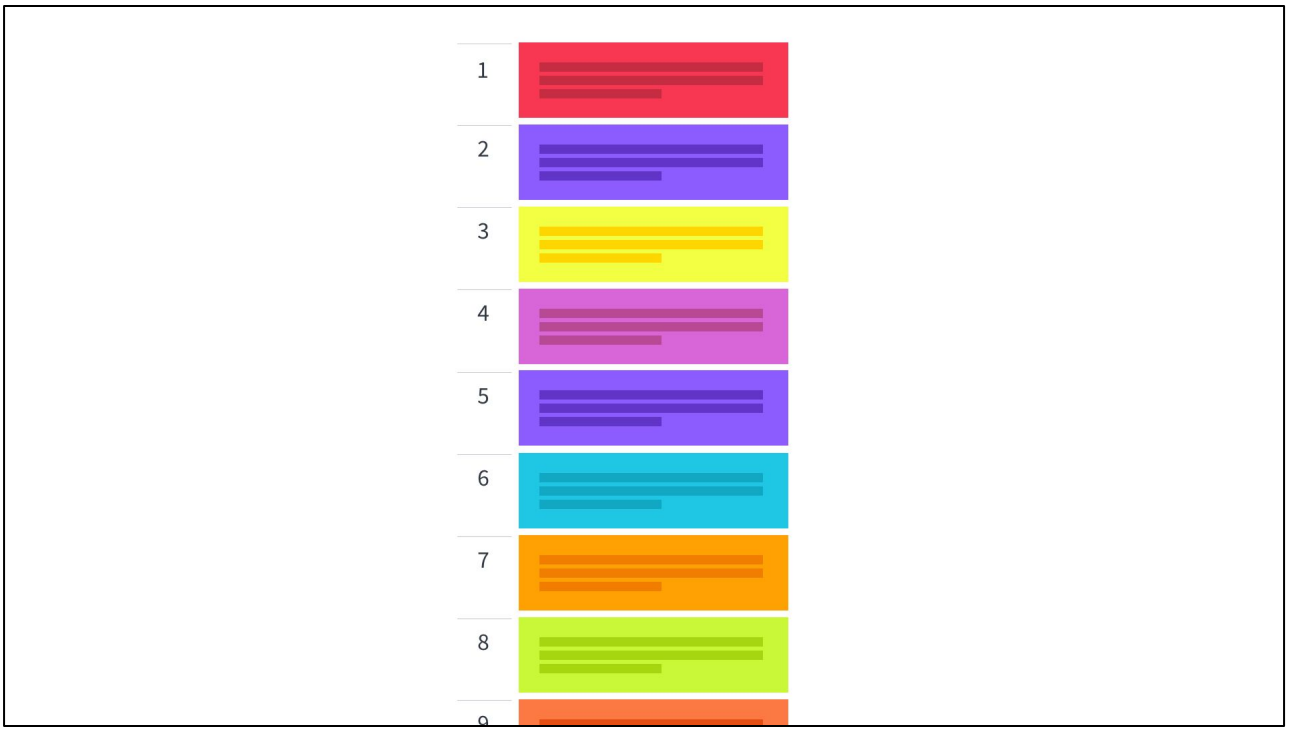
- We’ve got four tasks, or complete series of steps that a user needs to take to accomplish a piece of work.



- And then we've got work we know about
- things we know we need to improve



- And things we learn through research with or users



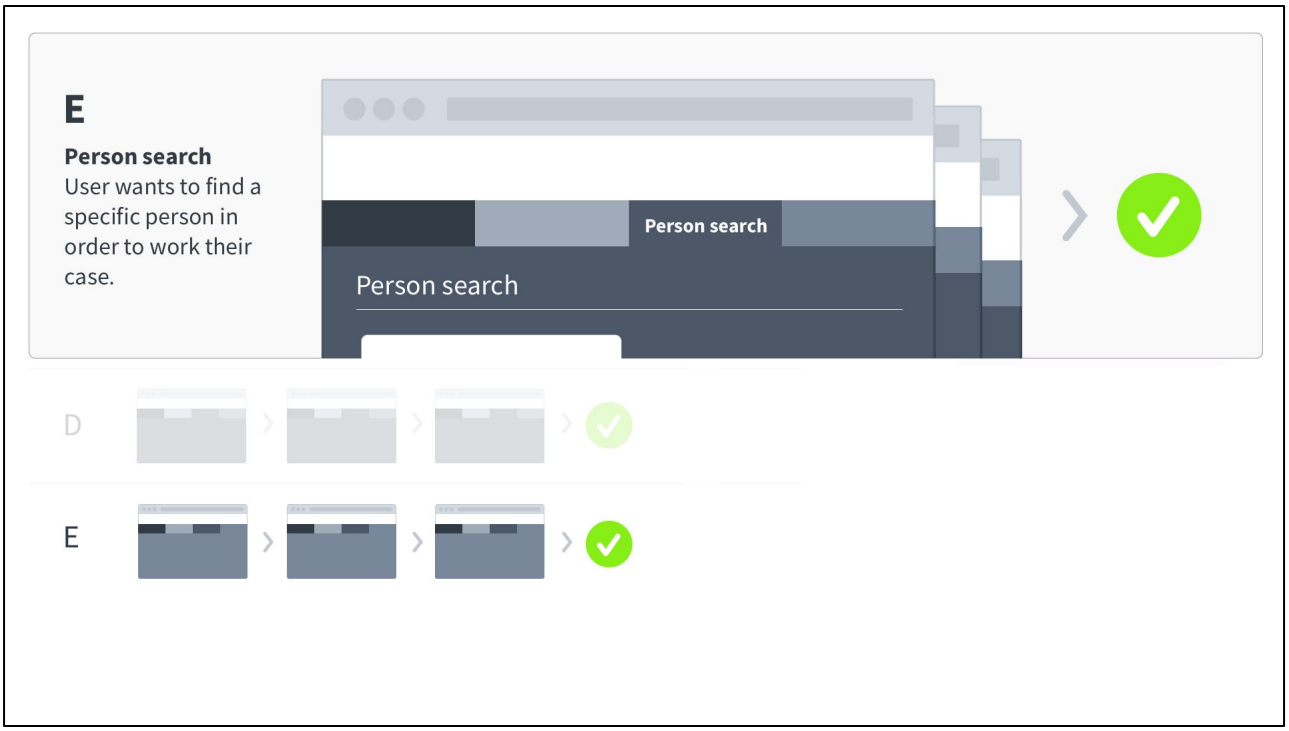
- The first thing we have to do is prioritize our work
  - This is part of adopting a “product” mentality
- This is about more than technology
  - can’t do everything at once
  - Need to direct our work
  - We create a prioritized backlog of work
  - constantly re-evaluate
  - Not a sequential order burning fires

At first, focus on learning and don’t bite off too much

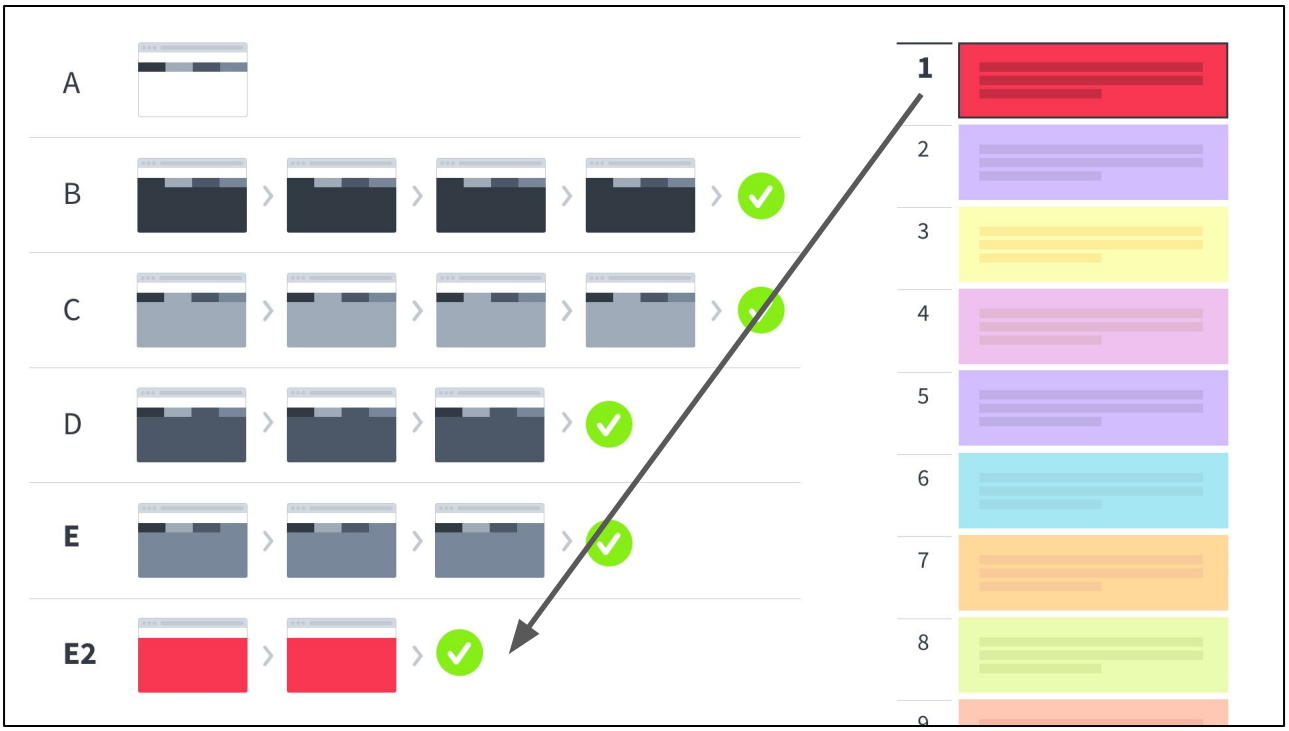
- Prioritize meaningful new pieces that aren’t *too* hard
- Build confidence in developing new APIs and building this way
- Stick with read-only functionality until you’re comfortable

- Early on, we’ll want to prioritize learning
  - existing data
  - our old backend systems
  - building with agile methods
  - new technologies such as cloud environment,
  - deployment methods
  - security requirements
- Pick
  - meaningful things
  - deliver real value to users
  - don’t bite off too much
- Also pick things that help you gain confidence
  - building your new APIs
  - knowing you can deliver this way
  - focus on reading data from your old systems, maybe hold off on write activities

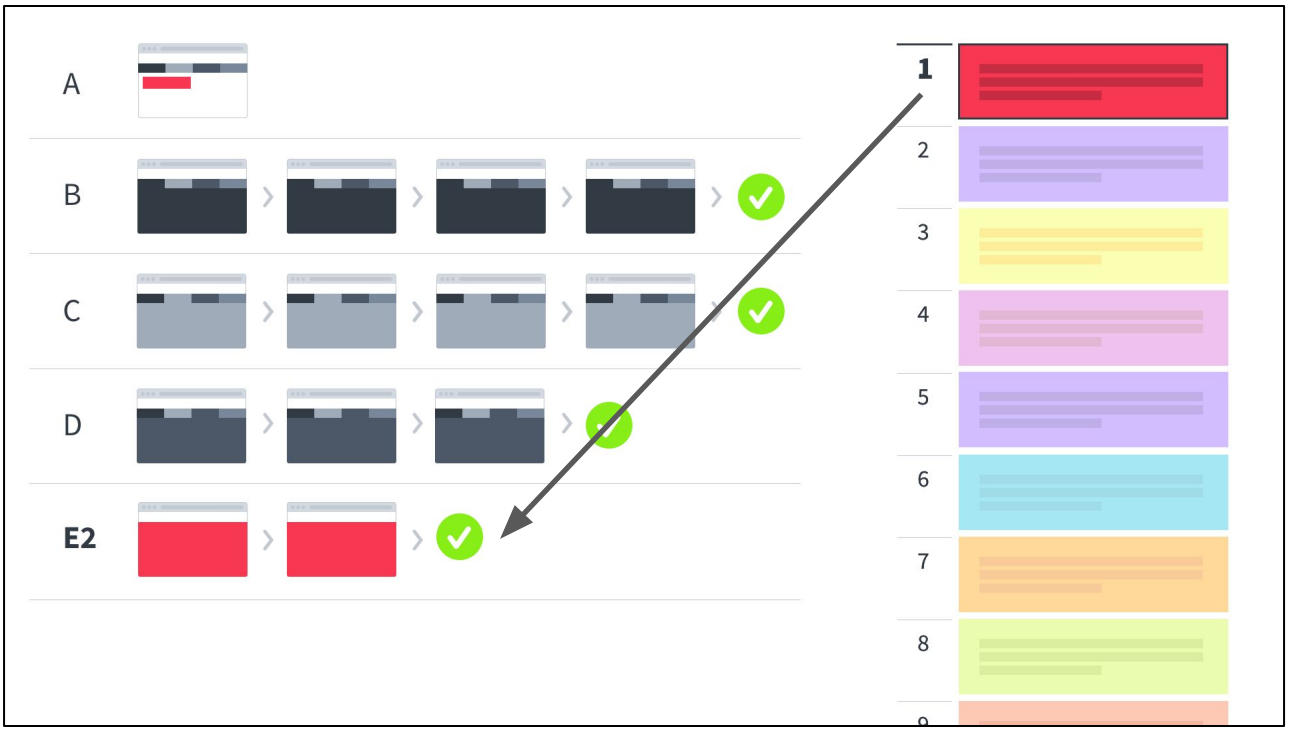
- old application and our prioritized work
- red box = new task/module/procurement
- in the top priority slot.
- We'll work on that first.



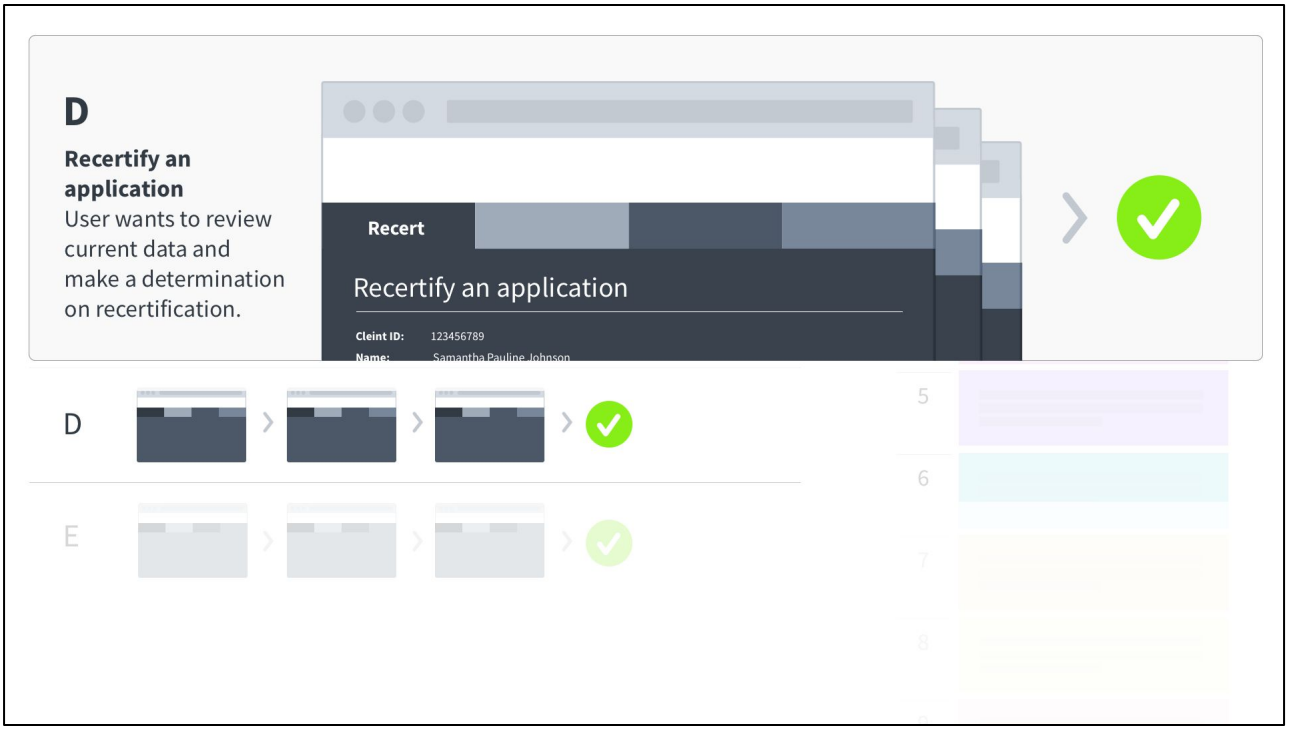
- searching for a person
- where a user looks for a specific person to work their case.



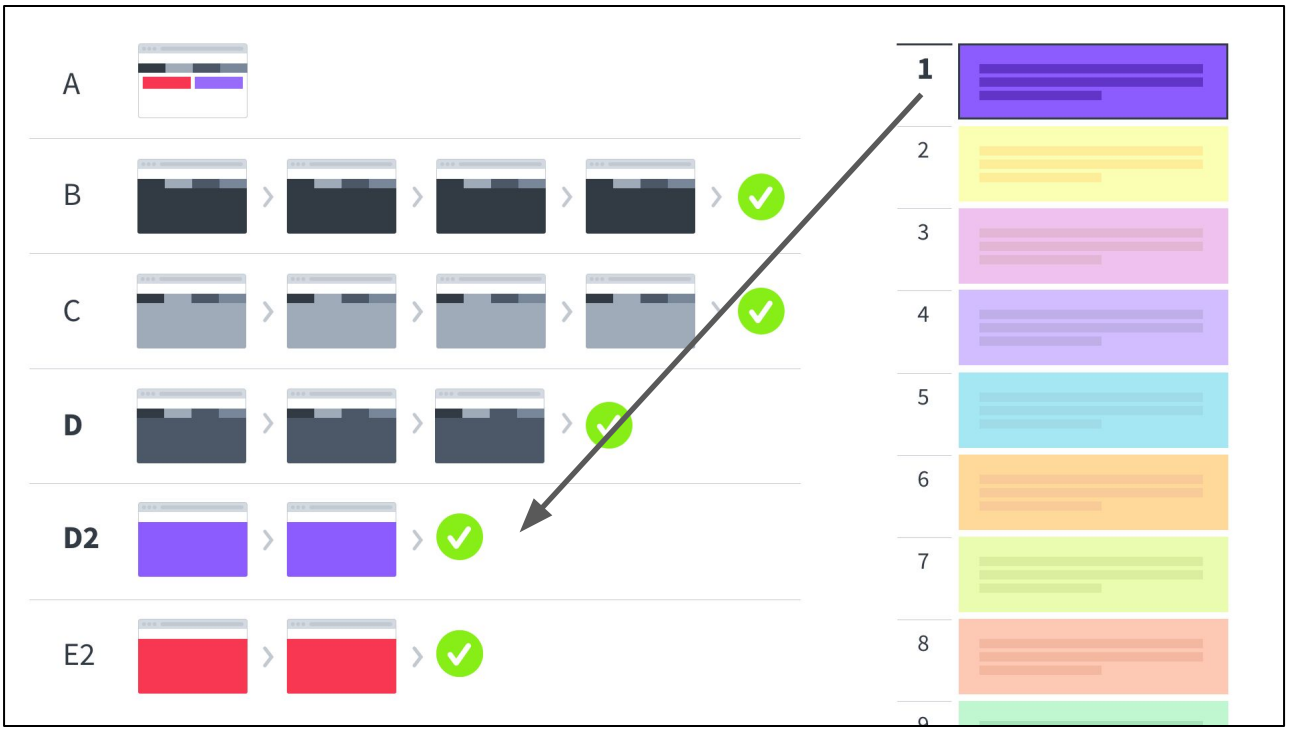
- We introduce the new module to our system.
- We'll call it E2.
- It's an alternate experience that focused on the same task as the legacy E task.
- We launch it alongside the existing workflow.
- Users can use either the old method or the new
-



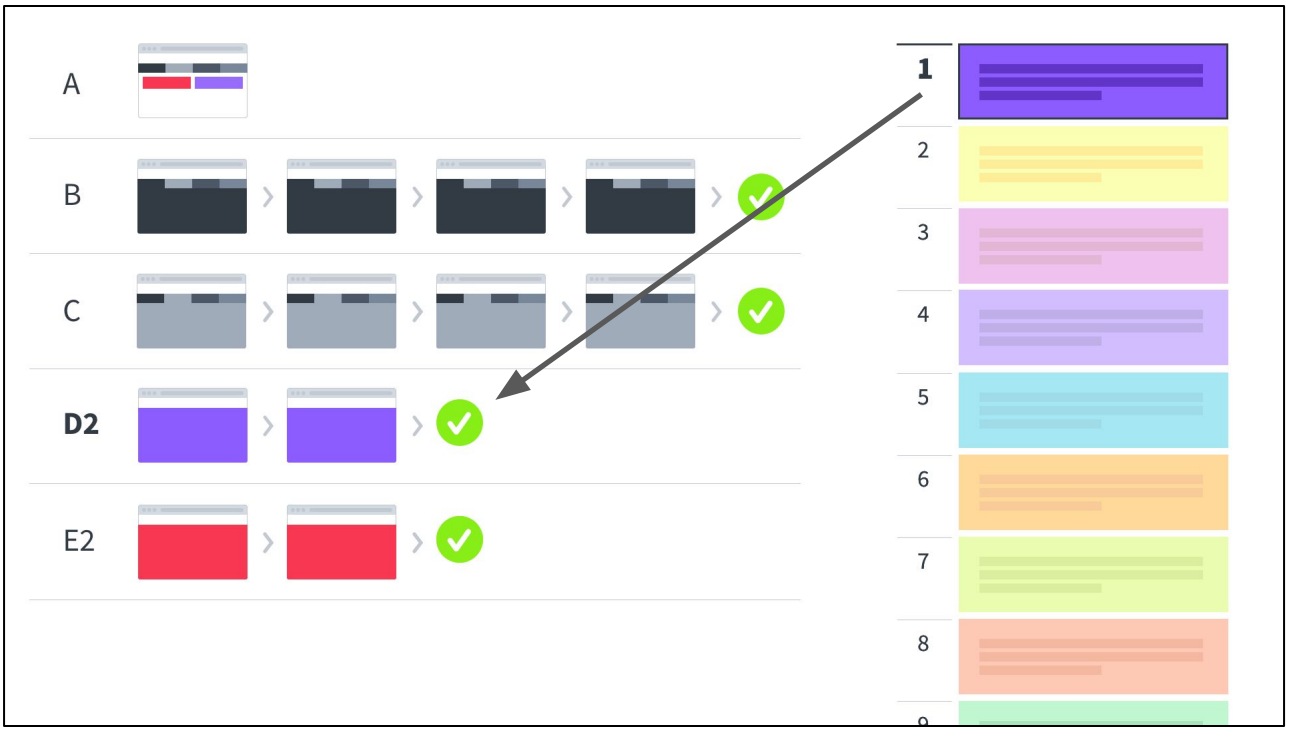
- Once E2 has been adopted, the legacy E experience can be hidden.
-



- Maybe our next task is recertifying eligibility

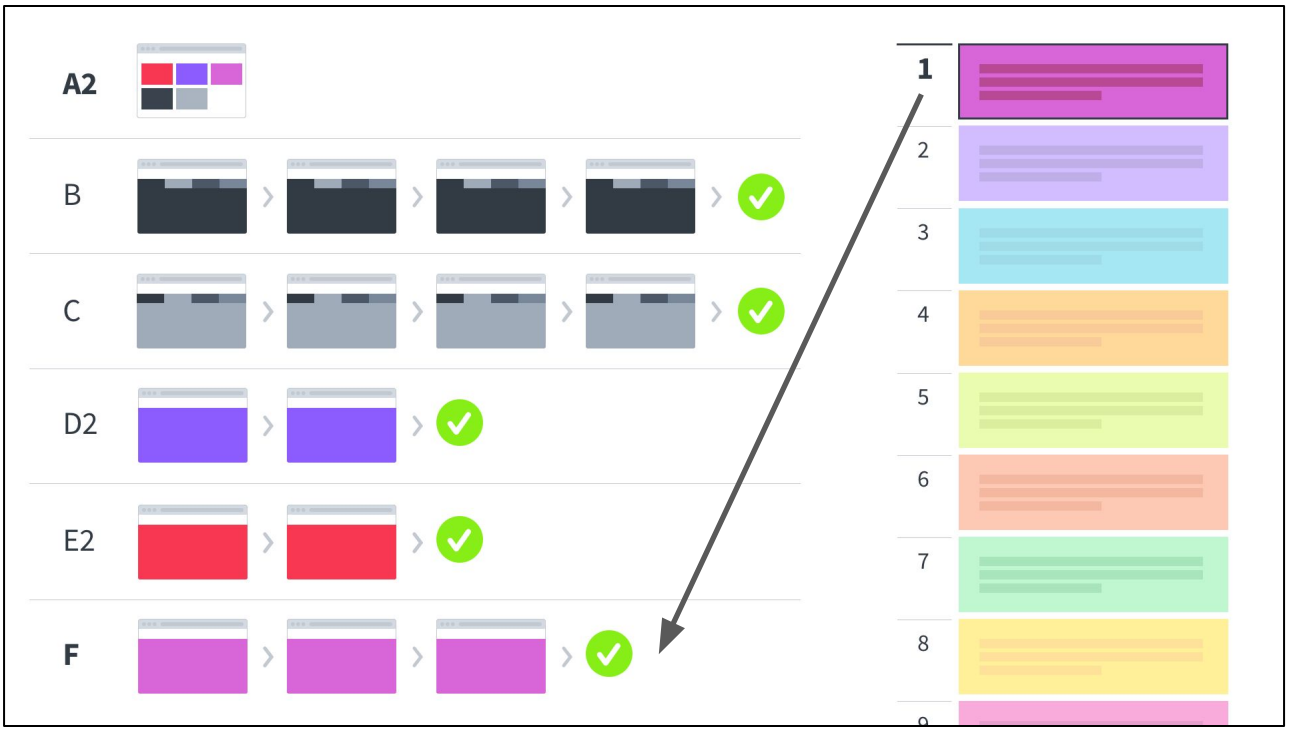


- We introduce the recertification workflow
- Shown here as D2.
-
-
-

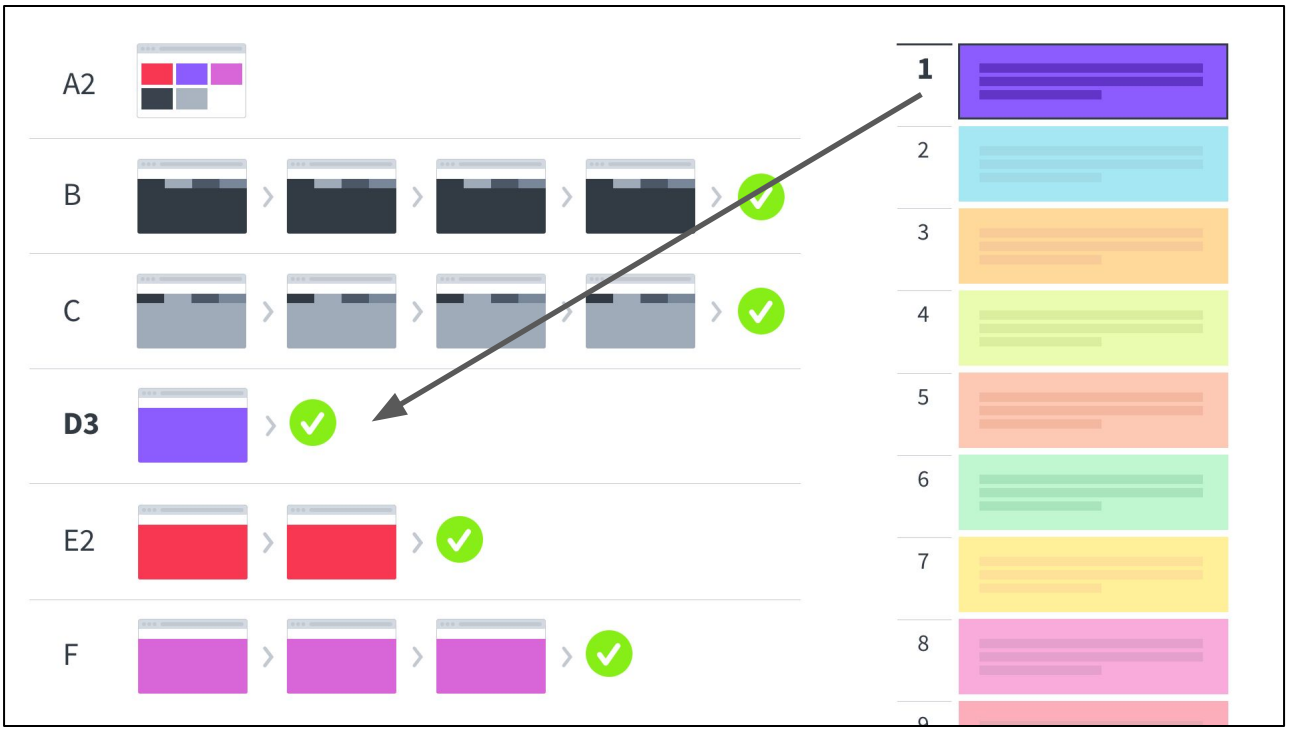


- With adoption of the new recertification, we hide the legacy D experience.
- Note how we're shifting responsibilities from the old system to the new one.
- And we're running them alongside existing legacy experiences

- New priority is focused on a new homepage
- The new homepage module provides navigation to both legacy and new experiences.



- Maybe we add new functionality not in legacy



- Them maybe we improve one of our new tasks further
-

- And then we replace another workflow

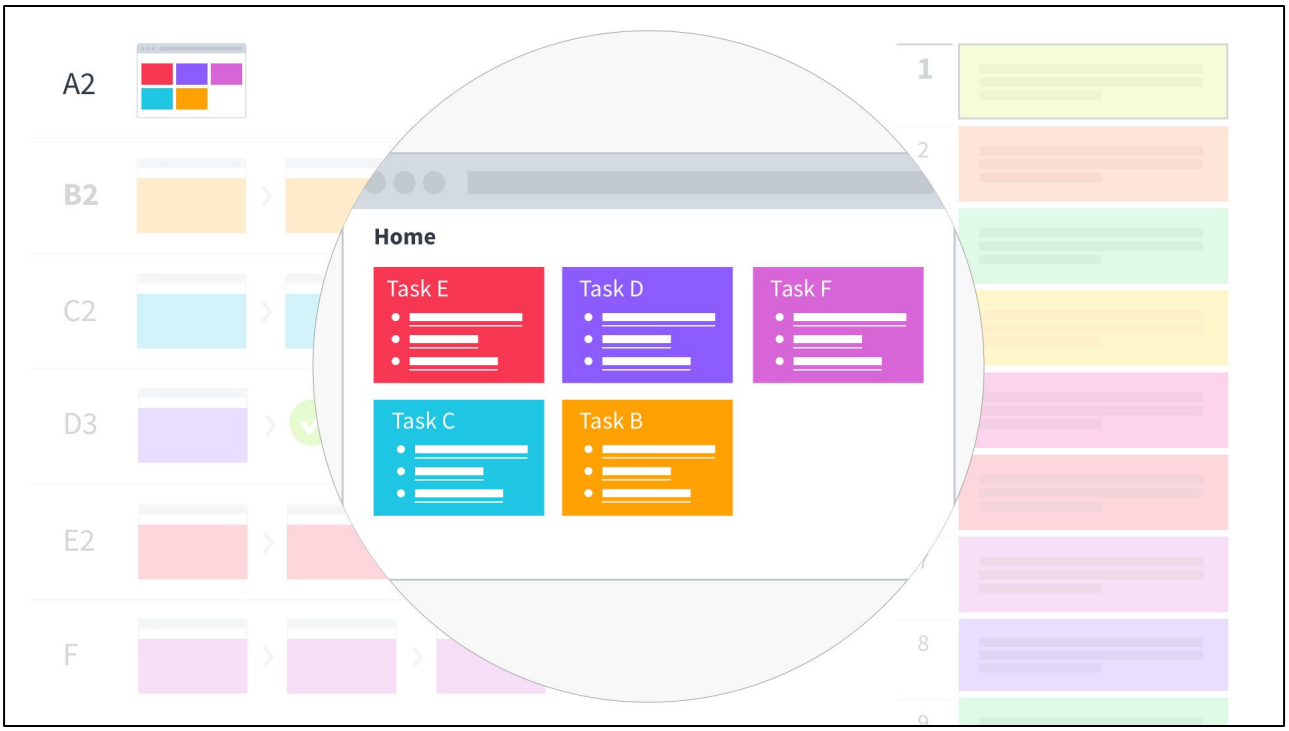
- And another

- After launching a number of modules,
- we have a system that's been iteratively transformed.
- We maintain a roadmap
- continue making incremental improvements.

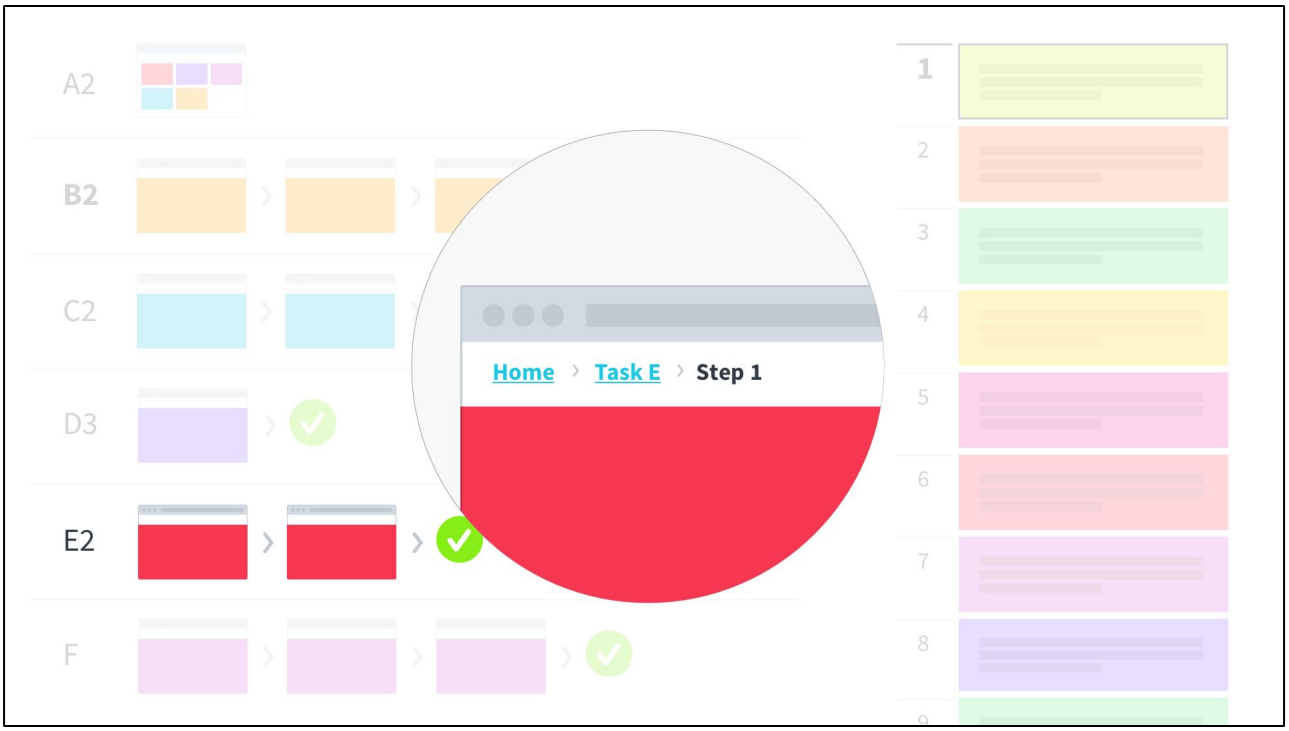
But won’t this lead to a disjointed experience during the transition?

- Yup.

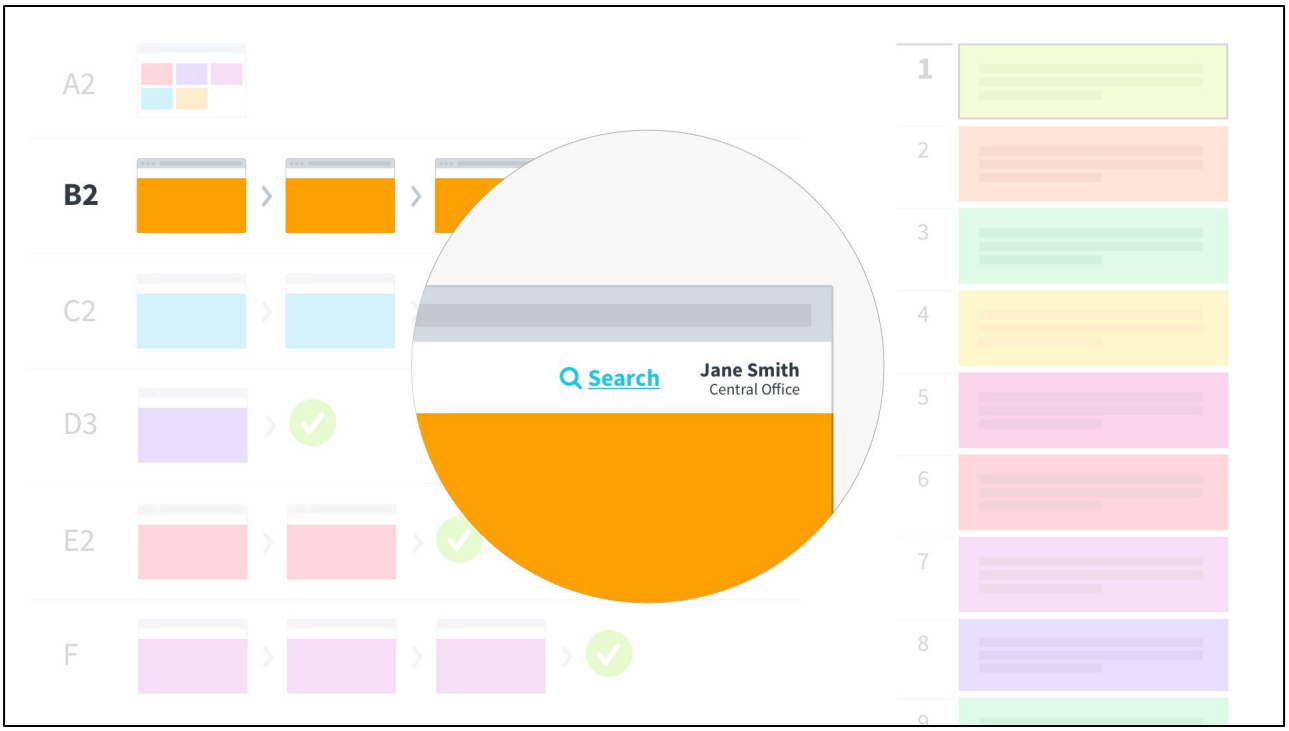
- Will experience be disjointed?
- some tasks in old way and some in new?
- Yup. that’s fine.
- Forward motion is better than perfect consistency.
- Here are a few tips that make it manageable



- Use the homepage for navigation
- facilitate global navigation
- aggregating links to anything necessary up front
- You can avoid updating all you global navs by teaching folks to navigate from the home
-



- Navigation is always difficult.
- Once set, it can be difficult to change.
- Users get used to it. Stakeholders fight over it.
- Adopting a “hub-and-spoke” approach to global navigation.
- Teams building a module don’t need to think about global nav,
- just link back home.
-



- Some services will need to be exposed across pages:
- Each module needs to be aware of the user's identity.
- Use search is a first class means of navigation
- accessible in each workflow.
-

This will look different if your legacy system is a mainframe green screen or desktop application.

- will look different if the application interface you are updating is not a web-based application
- If you have a green screen system or software that is installed on each eligibility worker's computer
- You'll do it a bit differently
- Users can't just follow links between old pages and new
- Context switching is real
- Still, follow a similar process
- Identify meaningful tasks that can be built anew
- fully be completed in the new system to avoid context switching.

What are some of the implications?

-

Optimized for change

- You'll be optimized for change
- Priorities change
- You should be able to respond to changing needs of your workers and clients
- Building out that API layer makes that easier.
- Optimized for change.

Deliver value faster

- Deliver value faster
- As pieces are developed, they are quickly rolled out
- You don’t have to wait until everything you’ve dreamed of is finished
- Roll em’out and test them with users
- Deliver value faster

Reduced risk

- Reduced risk
- You’re building smaller pieces
- Run concurrently for a short time
- Make switchover when proven
- Risk is scoped the the smaller workflow, rather than the broad system
- Reduced risk

Less vendor dependency

- Less vendor dependency
- With APIs, vendors don’t need to understand the whole system
- If one vendor doesn’t work out, you can hire another one with less startup costs
- Or multiple vendors can be working on different pieces without coordinating
- Your well-documented APIs allow them to get started fast
- Less vendor dependency

Become open to unanticipated uses

- Become open to unanticipated uses
- With API-layer in front of your backend systems
- you’re able to explore new opportunities you hadn’t planned
- Say commodity workflow management system - optimize application processing
- Plug into APIs quickly
- “commodity” = truly ready to use out of the box,
- Connect the workflow system through your APIs and start using the product instead of building one yourself
- Become open to unanticipated uses

Use the APIs of others

- Use the APIs of others
- When loosely-coupled, and pieces are talking to each other through your APIs, you also become prepared to use APIs of others
- Maybe it’s an authentication service
- Sending SMS
- Some of you may remember MAGI-in-the-Cloud
- That’s one example

Eligibility APIs Initiative

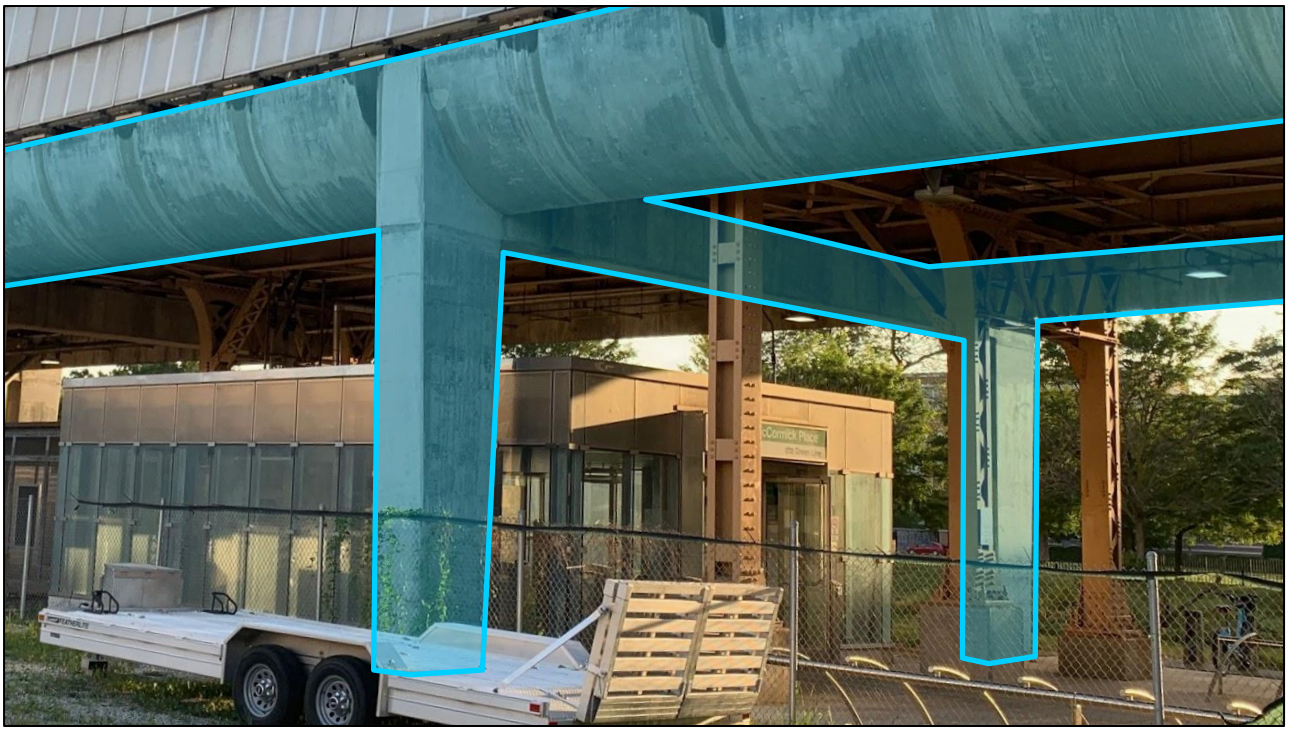
Exploring the idea of sharing eligibility criteria across multiple states via APIs to reduce the work states need to build in their own systems.

- Another example: Eligibility APIs Initiative
- We're exploring the idea of sharing eligibility criteria
  - across multiple states via APIs
  - reduce the work states need to build in their own systems.

Modernizing these systems is hard.

- As I said before, modernizing these systems is hard
- We've had success using this approach, but it's a long road
- Hopefully this provides some food for thought.

- Public transit - another complex system in need of modernization
- Cermak-McCormick Place stop on the Green line
- Taken a similar approach
- engineered and constructed around existing structure
- while maintaining full transit service



- They adding new structure alongside old
- They maintained service
- Not sure if they intend to remove the original structure
- Important to state building software is not like constructing physical structures
- That’s often a problematic metaphor
- But they are thinking about how they can make it work
- We all need to
-

18F

Eligibility APIs Initiative

18F Human Services Portfolio

Talk to me

Talk to Robin

github.com/18F/MESC2019

- I'll leave this up here
- We've got a number of people to talk to
- Portfolio
- Me,
- Robin
- Link
- happy to answer questions
- Greg Walker here to help with some of the meatier technical questions you may have.

Thanks to the following colleagues whose ideas I've pulled from in this presentation:

- Steven Reilly
- Jeremy Zilar
- Uchenna Moka-Solana
- Dr. Robert Read