

# Building resilient frontend architecture

Monica Lent  @monicalent  
Lead Frontend Engineer at SumUp



# Why do we rewrite software?

# Why do we usually rewrite code?

1

**Inexperience**

2

**It's fun**

3

**Better solution available**

4

**Technical Debt**

Old libraries?

Code I didn't write?

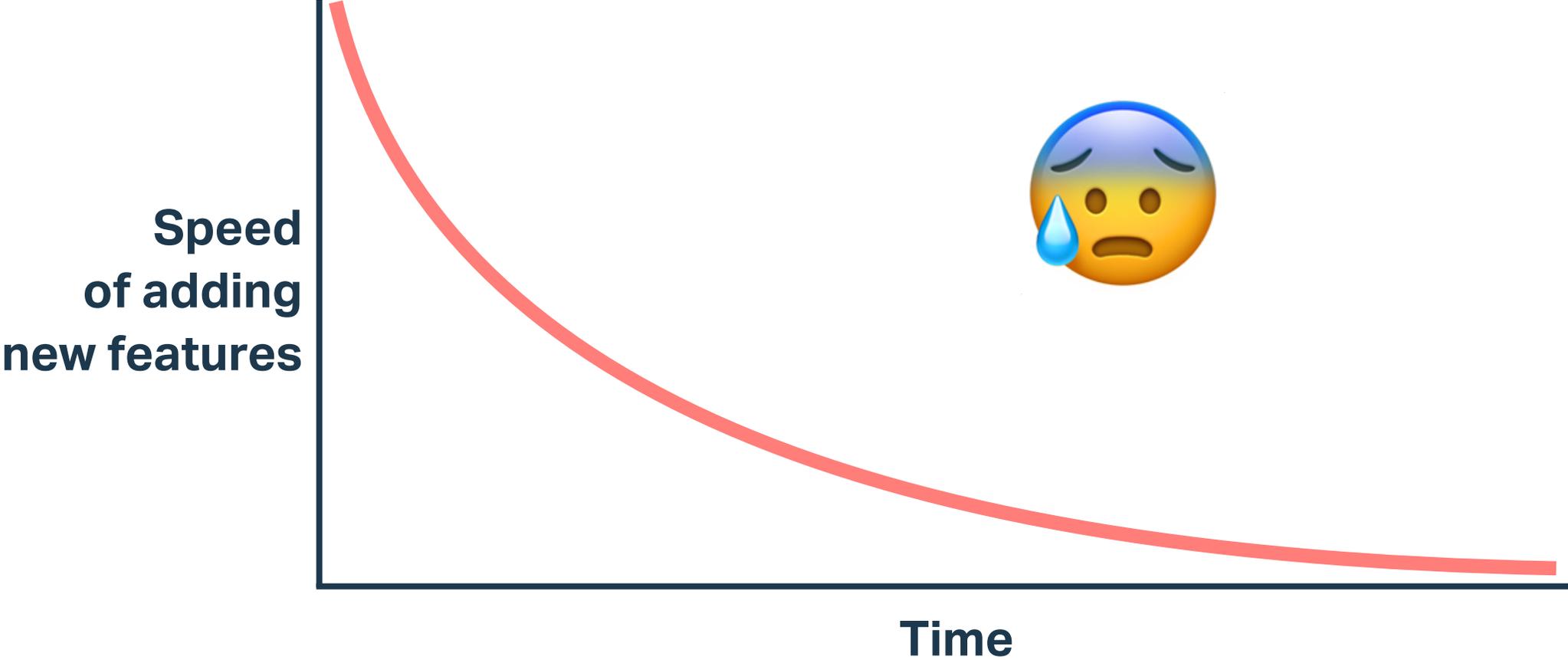
**Code that negatively  
and repeatedly affects the  
speed or quality of delivery**

# Technical debt

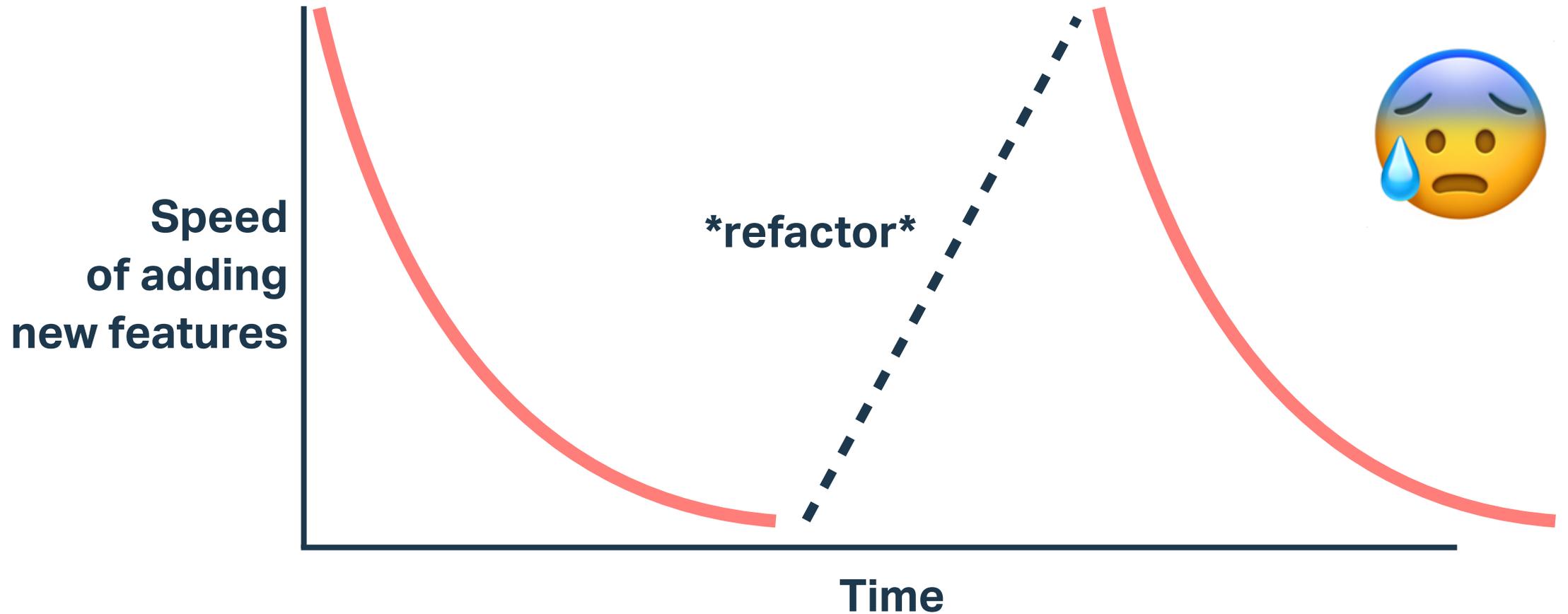
Code I wrote before I knew  
what I was doing?

Features that no one uses

# Technical debt



# Recurring technical debt



# Second system effect

The tendency of small, elegant, and successful systems to be succeeded by over-engineered, bloated systems due to inflated expectations and overconfidence.

“Legacy code” often  
differs from its suggested  
alternative by actually  
**working and scaling.”**

- Bjarne Stroustrup, Inventor of C++

**Rewrite**

Is this my destiny?



# HARD FACT

The **real cost** of software  
is not the initial development,  
but **maintenance over time**

**THE QUESTION IS NOT**

**Why do we rewrite  
software?**

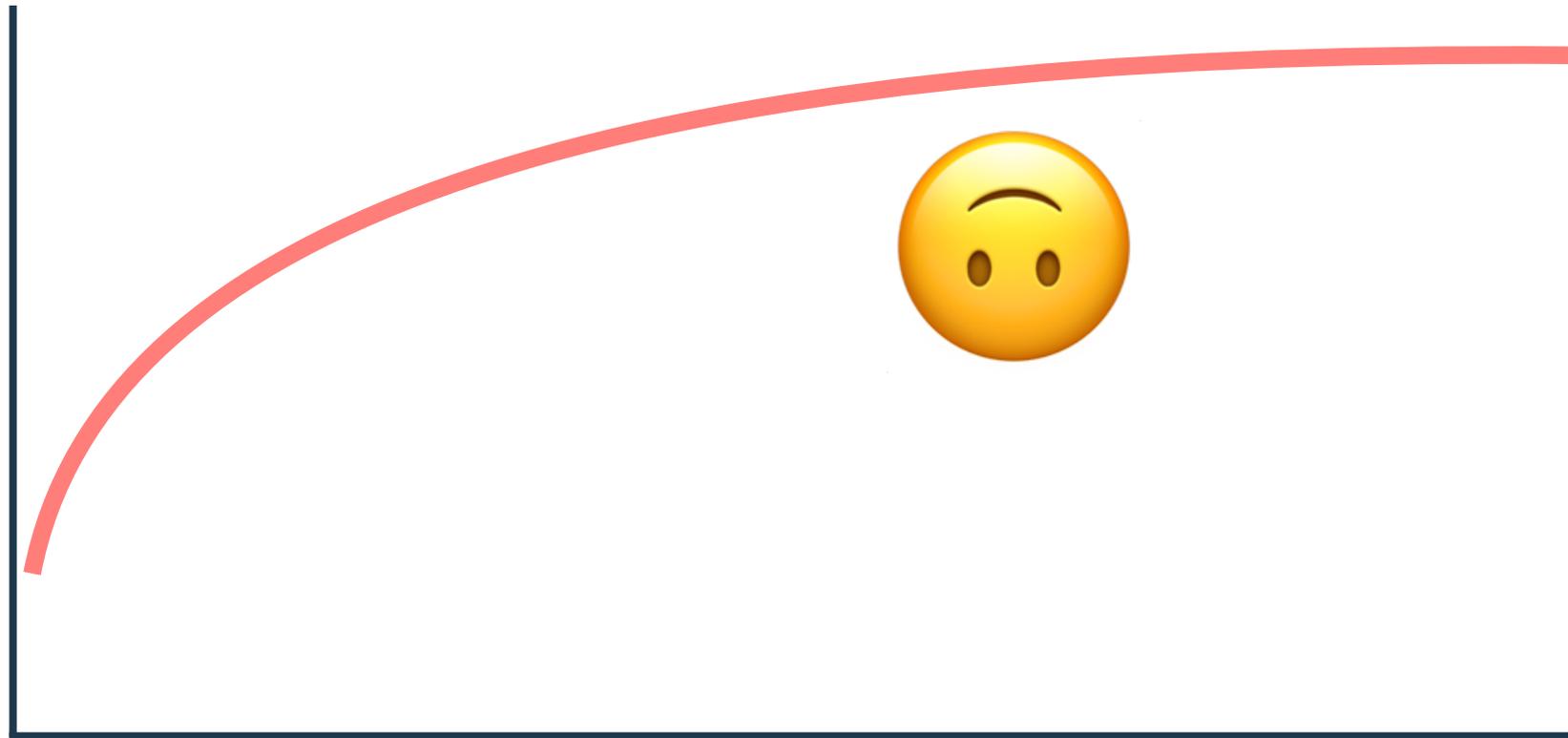
How can we  
make our systems  
more resilient to  
**inevitable change**  
?

**Speed  
of adding  
new features**



**Time**

**Speed  
of adding  
new features**



**Time**

# How do we reach this nirvana?

## "Good architecture"



**Hard to spell**

**Feels detached from daily  
problems**

**No clear definition**

**What does a software  
architect even do?**

**Sounds elite**

**“Architecture” has  
become a dirty word**

# Architecture as **enabling constraints**

**Constraints about how we use data and code that help us move faster over time**

# Enabling constraints in real life



# Enabling constraints in Programming paradigms

## Paradigm

OOP

Functional

## Constraint & Enablement

From function pointers to classes →  
**Independently deployable  
subcomponents**

From mutable to immutable data →  
**Eliminate race conditions and  
concurrency problems**

# Enabling constraints in Frontend development

## Paradigm

var → const

jQuery → React

CSS → CSS-in-JS

## Constraint & Enablement

No more reassignment →  
**Predictable data**

No more DOM manipulation →  
**Predictable UI**

No more naming / side-effects →  
**Safety and fewer global clashes**

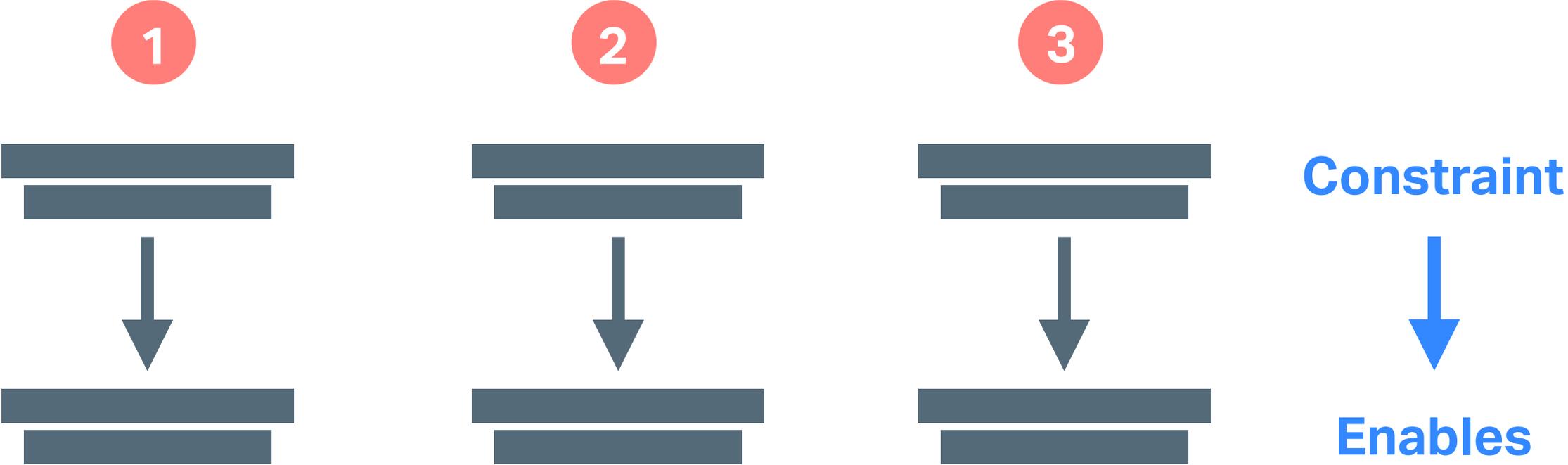
We are constraining  
ourselves **all the time**

We trade constraints for  
**safety and speed**

**NOT EXHAUSTIVE**

**3 constraints**  
**you can use today**  
for more resilient frontend  
architecture

# Constraints for more resilient frontend architecture



# Constraints for more resilient frontend architecture

1

Source code dependencies must point inward



2



3



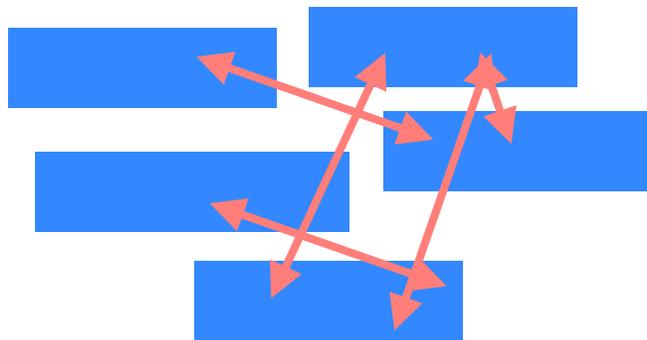
Constraint



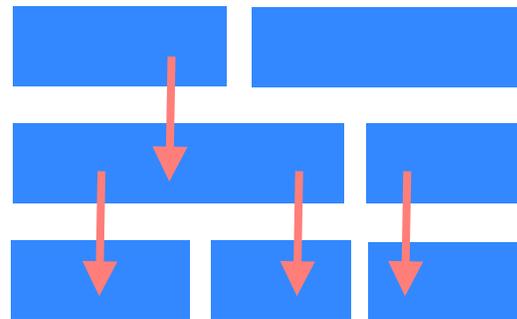
Enables

# A few ways of organizing our dependencies

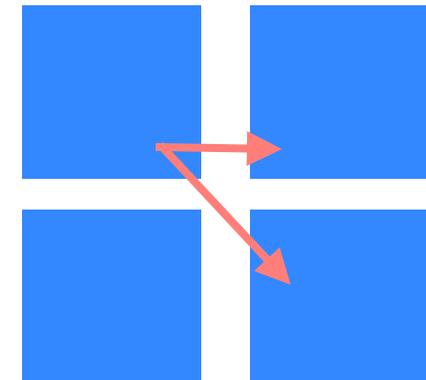
## Big Ball of Mud



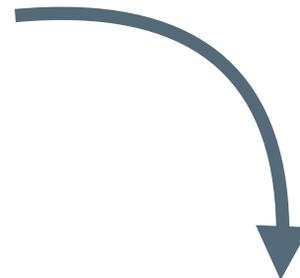
## Layered



## Modular

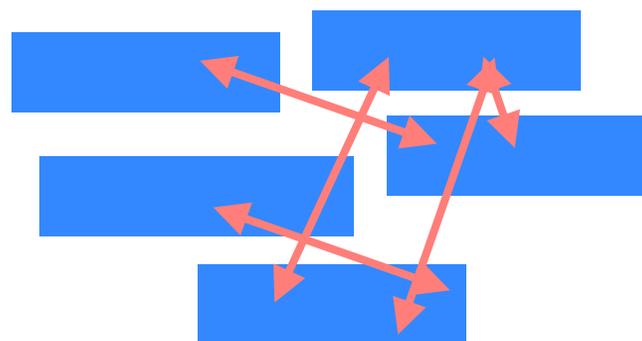


What's the difference?

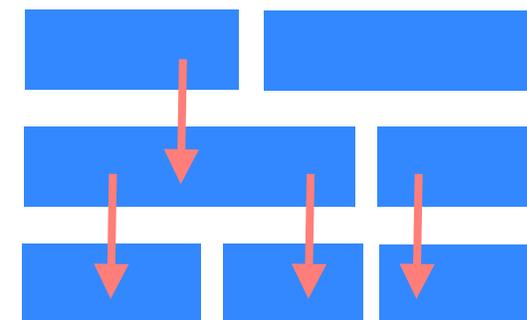


Constraint  
Source code  
dependencies  
must point inwards

Big Ball of Mud

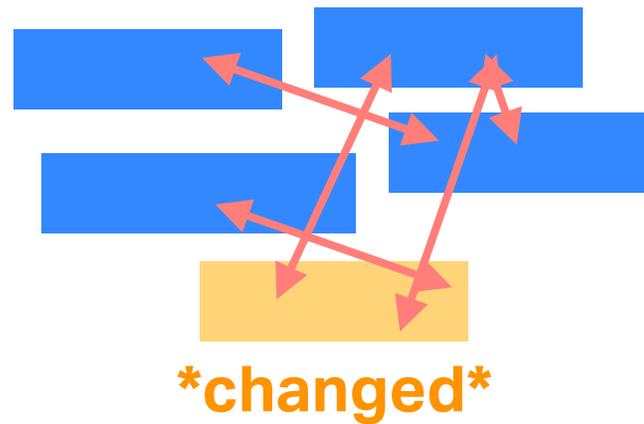


Layered

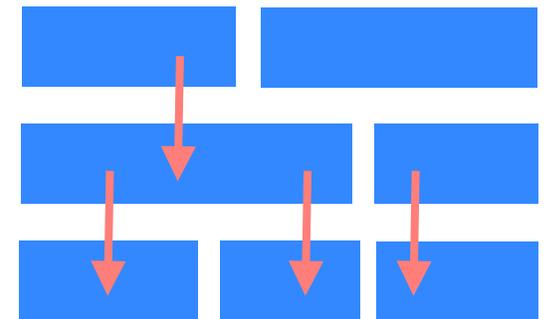


Constraint  
Source code  
dependencies  
must point inwards

## Big Ball of Mud



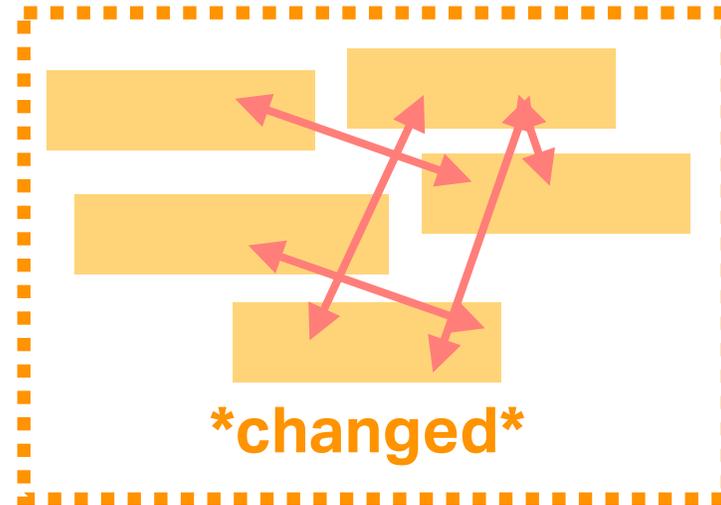
## Layered



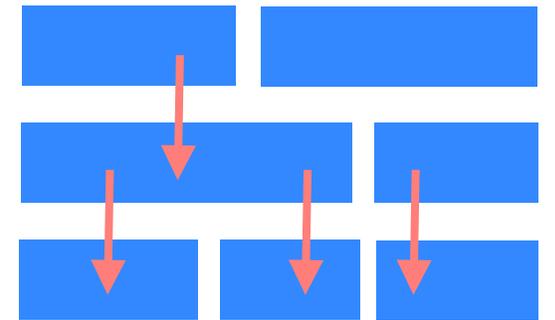
Huge or unknown regression scope  
Cross-team conflicts

Constraint  
Source code  
dependencies  
must point inwards

Big Ball of Mud

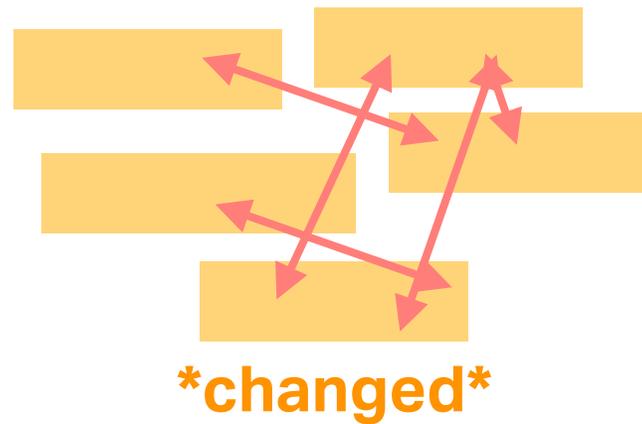


Layered

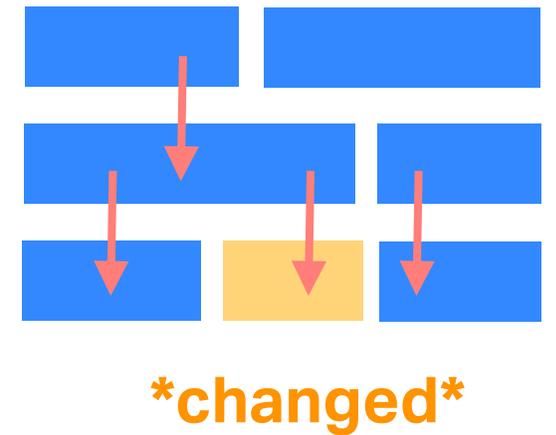


Constraint  
Source code  
dependencies  
must point inwards

Ball of Mud

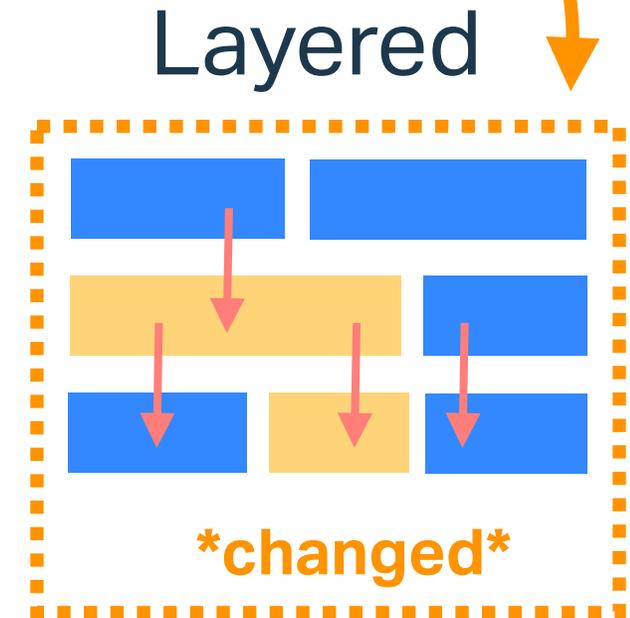
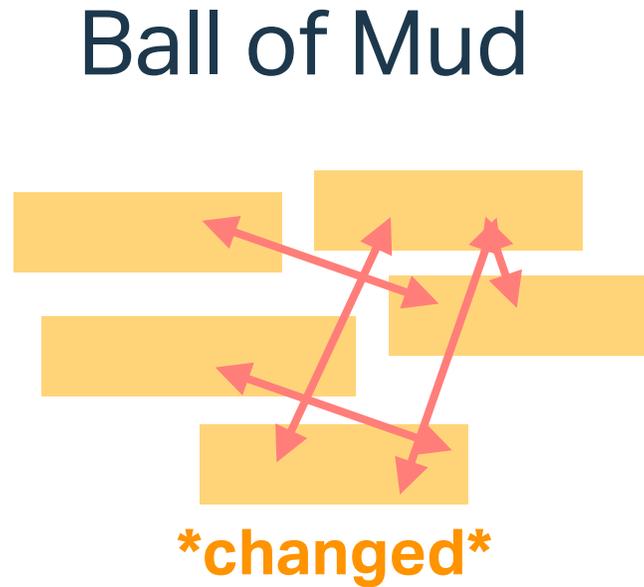


Layered



Constraint  
Source code  
dependencies  
must point inwards

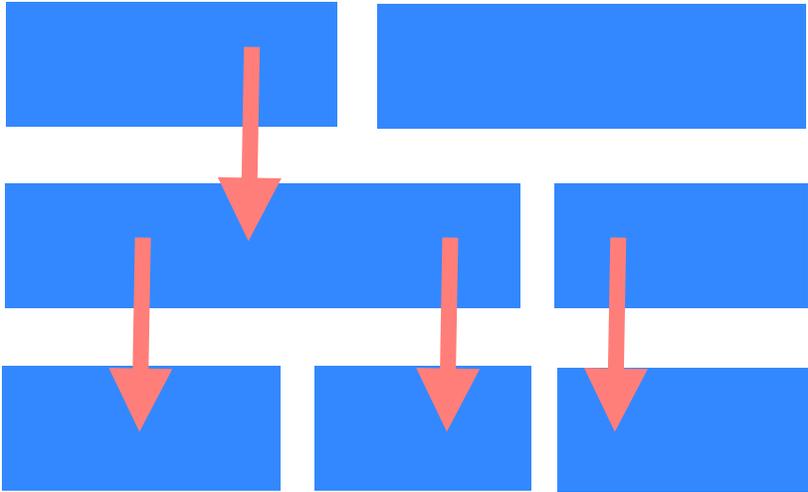
Limited regression scope  
(Usually) does not affect other teams



Key difference between a ball of mud  
and a well-organized monolith is  
**dependency organization**

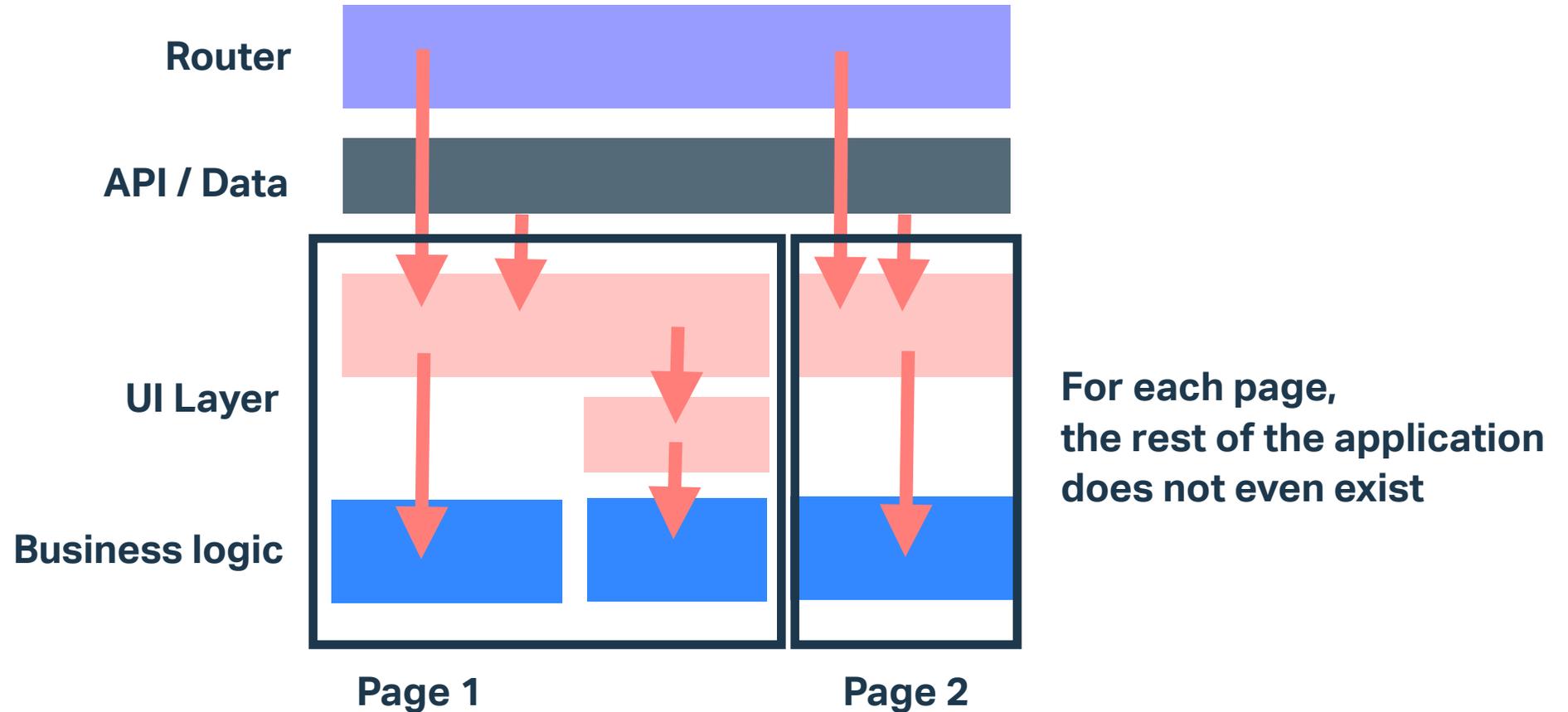
# Constraint

## Source code dependencies must point inwards



# Constraint

## Source code dependencies must point inwards



# Constraints for more resilient frontend architecture

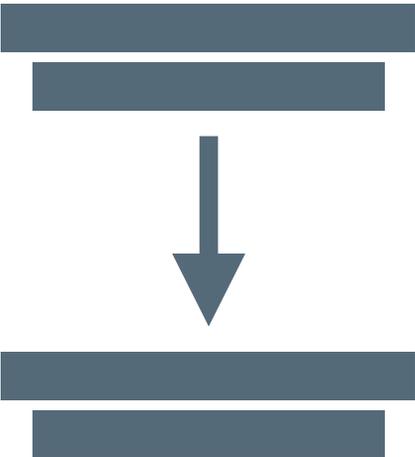
1

Source code dependencies must point inward

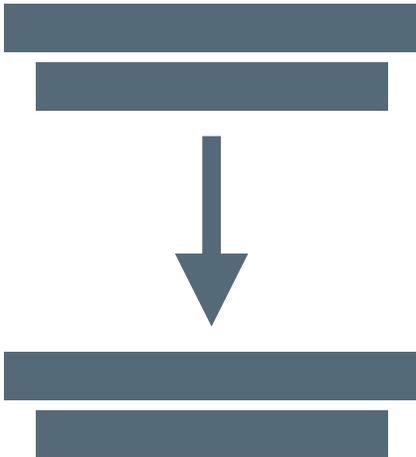


Easier to isolate impact of changes

2



3



Constraint



Enables

# What about **shared components**?

Design system 🎨 -or- copy-paste ✂️

# Constraints for more resilient frontend architecture

1

Source code dependencies must point inward



Easier to isolate impact of changes

2

Be conservative about code reuse



3



Constraint

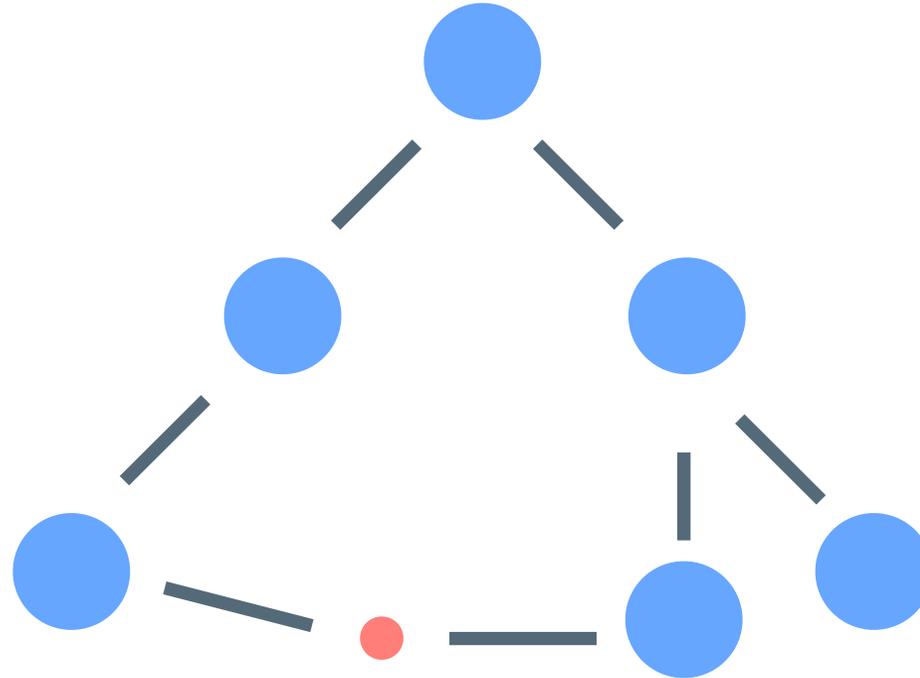


Enables

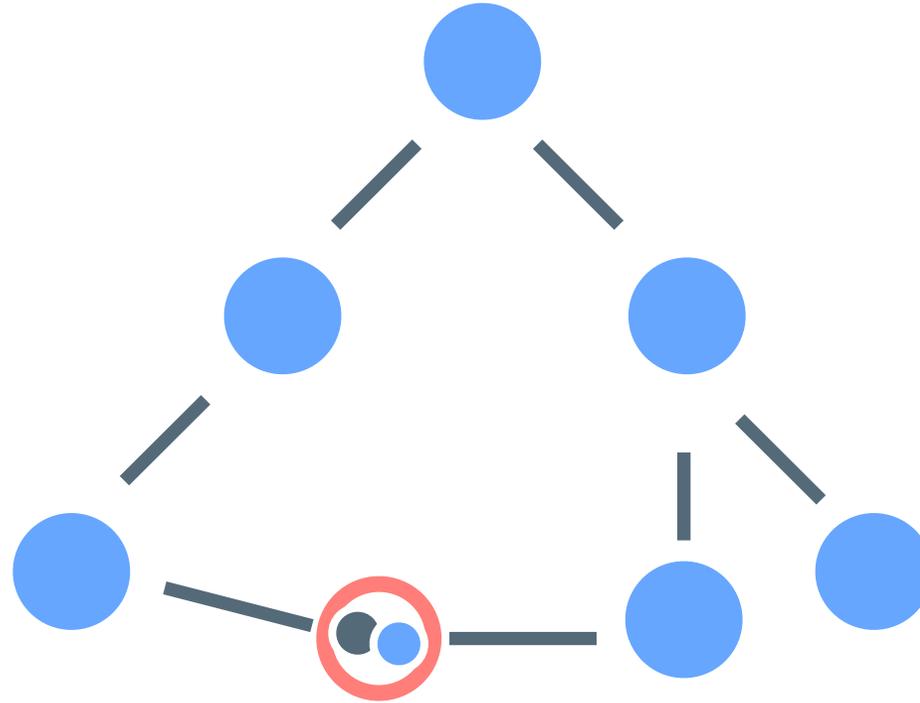
**WE**  **DRY**

The result is often **brittle** and  
**side-effect ridden** code  
in the name of **code reuse**

# Impact of time on shared code



# Impact of time on shared code



**if, context, branches...**

# DECOUPLED > DRY

Code reuse is not a goal in and of itself



**JBD**  
@rakyll

Following



A regular person sees it either full or half empty.  
An engineer sees it half full and learns in what way it makes sense to see it half full, and when to see it half empty.

**Sometimes you just need two glasses!**

12:58 AM - 25 Jan 2019

# Constraints for more resilient frontend architecture

1

Source code dependencies must point inward



Easier to isolate impact of changes

2

Be conservative about code reuse



Avoid coupling code that diverges over time

3



Constraint



Enables

# Constraints for more resilient frontend architecture

1

Source code dependencies must point inward



Easier to isolate impact of changes

2

Be conservative about code reuse



Avoid coupling code that diverges over time

3

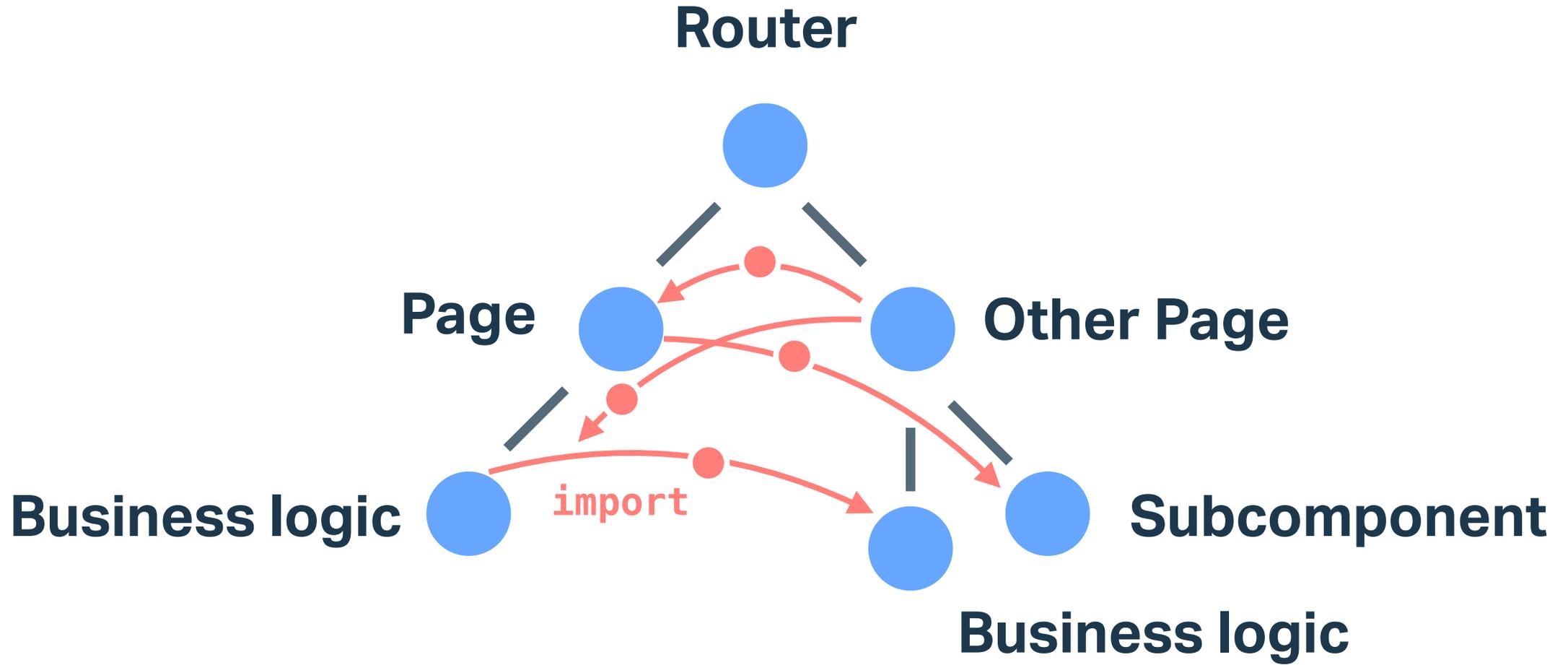
Enforce your boundaries



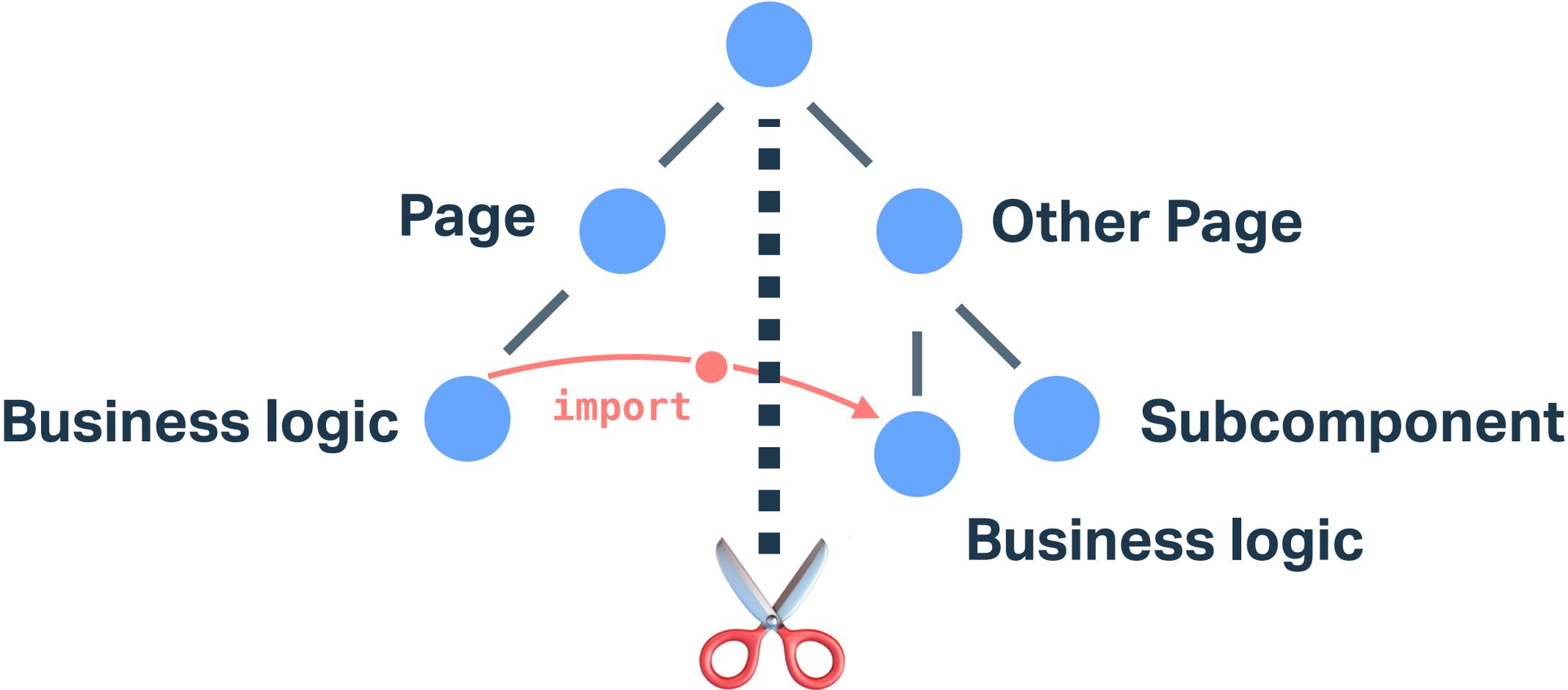
Constraint



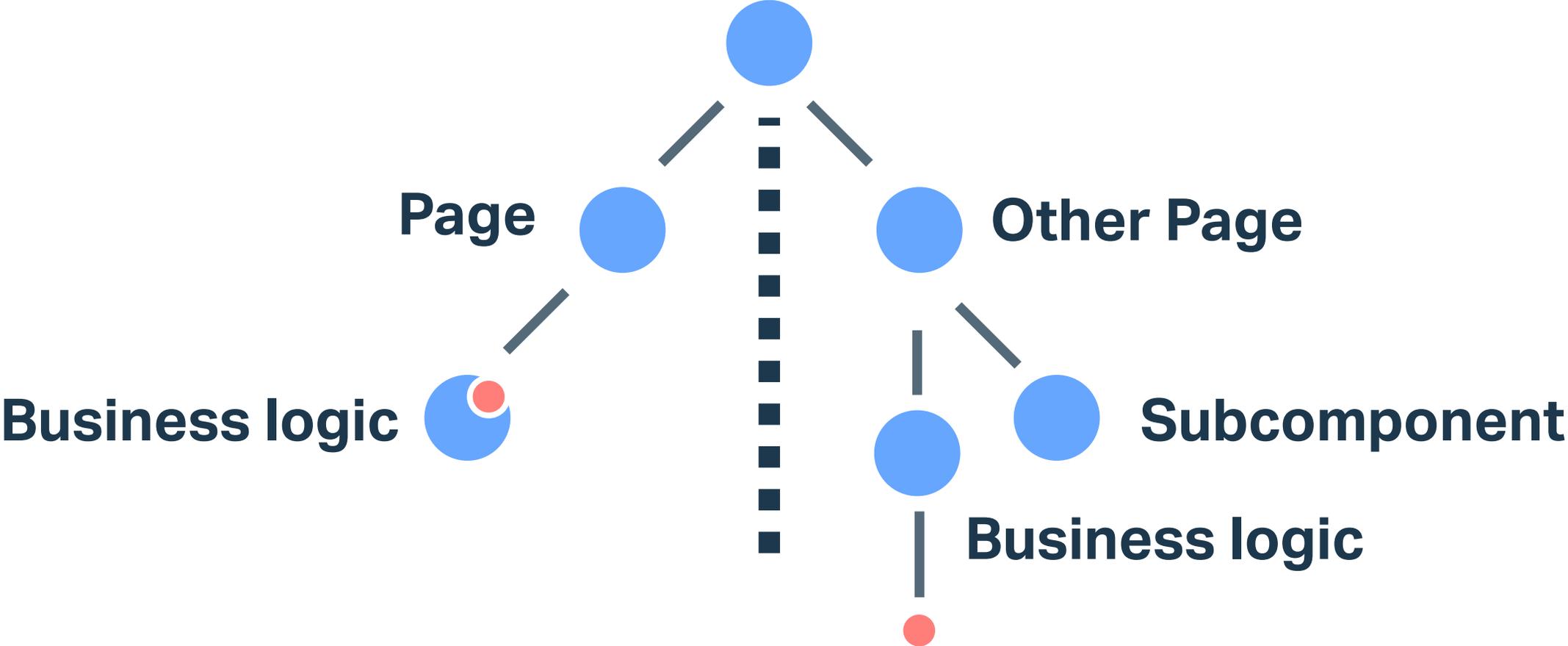
Enables



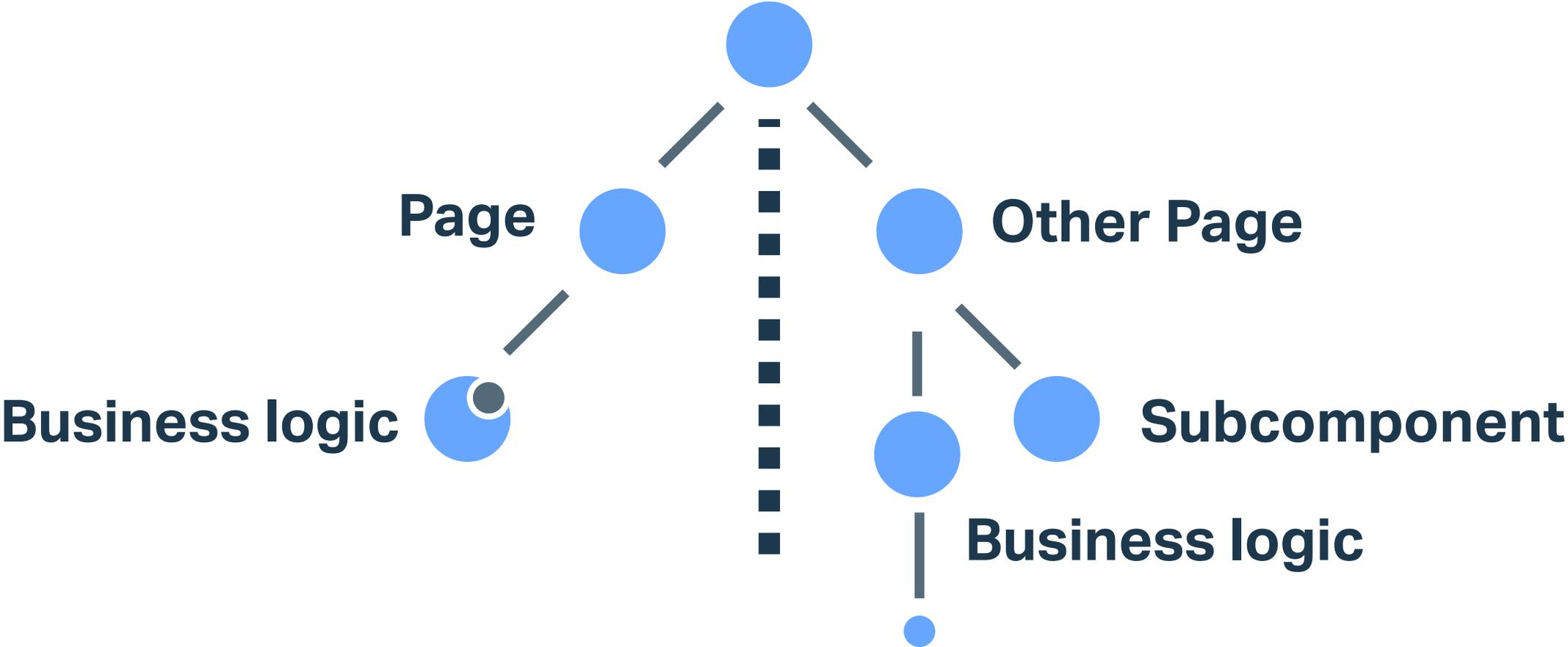
# Router



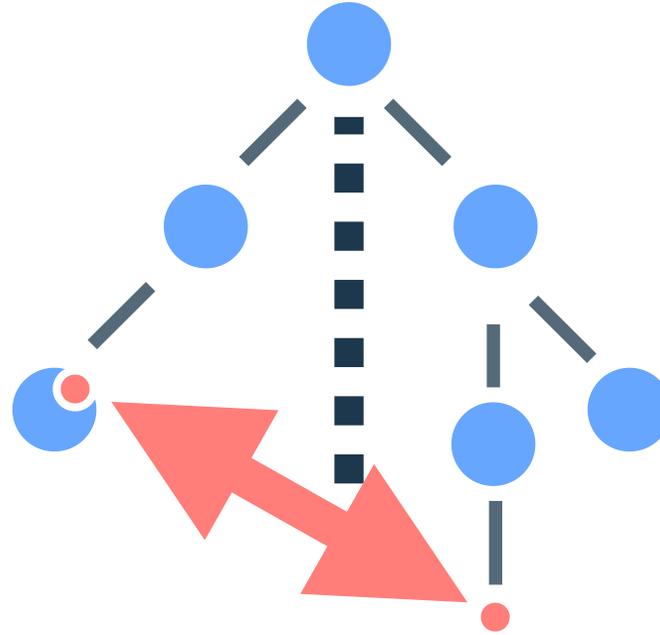
# Router



# Router



# Forbidden dependency tests



**BUILD**

**FAILING**

# Forbidden dependency tests

.dependency-cruiser.json

```
{
  "forbidden": [{
    "name": "Your Page",
    "comment": "Should not depend on other pages",
    "severity": "error",
    "from": { "pathNot": "^pages/YourPage" },
    "to": { "path": "^pages/YourPage" }
  }]
}
```

```
npm install --save-dev dependency-cruiser
```

# Constraints for more resilient frontend architecture

1

Source code dependencies must point inward



Easier to isolate impact of changes

2

Be conservative about code reuse



Avoid coupling code that diverges over time

3

Enforce your boundaries



Preserve your architecture over time

Constraint



Enables

The real cost of software is **maintenance over time**, because change is inevitable



## What we've learned



Architecture is about applying **enabling constraints** to how we use code and data



We can make **small changes** to make our projects more resilient (1. Think **directionally**, 2. Be **conservative** on reuse, 3. **Enforce** our boundaries)



Every time you write a function  
(or don't), create a new module  
(or don't), you're making an  
**architecture decision**



You don't have to derive  
architecture from  
**first principles**



Thank you!

@monicalent

