

Building Big Lustre

Apps

Rebecca Reusch — Code BEAM Europe 2026



Pages

[README](#)[Announcements](#)[Changelog](#)[Examples](#)[Quickstart guide](#)[Managing state](#)[Side effects](#)[Server side rendering](#)[Full stack applications](#)[SPA deployments](#)[Full stack deployments](#)[For Elm developers](#)[For React developers](#)[For LiveView developers](#)

Links

[Sponsor](#)[Repository](#)[Hex](#)

Modules

[lustre](#)[lustre/attribute](#)[lustre/component](#)

01 Quickstart guide

Welcome to the Lustre quickstart guide! If you want to hit the ground running, get something on the screen, and see most of what Lustre has to offer, this is the place to start.

Lustre can be used to create HTML in many different contexts: static HTML file generation, dynamic server-side rendering, client-side single-page applications, web components, and real-time server components. This guide will focus on the most common use case: single-page applications, or SPAs.

SPAs are JavaScript applications that handle rendering in the browser rather than requesting HTML from the server. To help you build them, Lustre comes with an opinionated runtime designed to help you build robust scalable applications:

- **Declarative rendering:** Your app's ui is constructed using a declarative api that treats the current HTML as a function of your application's state. This is in contrast to more traditional imperative approaches to direct DOM mutation like jQuery.
- **State management:** If your ui is a function of your application's state, then we'll need a good way to manage that state! Lustre provides a simple message-based state management system modelled after Erlang's [gen_servers](#), Gleam's [actors](#), and the [Elm Architecture](#).
- **Managed side effects:** Managing asynchronous operations like HTTP requests or timers can be tricky when JavaScript is single-threaded. Lustre provides a runtime to manage these side effects and feed their result back into its state management system.

Lustre examples directory

Each of the examples in this directory is a self-contained, complete Gleam app that demonstrates a particular feature or concept of the library. For newcomers, we recommend looking through them in order, as each example tends to build on the previous ones. Feel free to jump in to any example that interests you, though!

01-basics

These examples cover the fundamentals of Lustre such as setting up an application or rendering elements that can handle events.

- [01-hello-world](#) is a simple example to just get something on the screen.
 - [02-attributes](#) demonstrates how to add attributes to HTML elements.
 - [03-view-functions](#) shows how to organize your view code into functions.
 - [04-keyed-elements](#) explains how to use keyed elements for optimized rendering.
 - [05-fragments](#) demonstrates how to use fragments to group elements.
 - [06-flags](#) shows how to pass initialization data to your Lustre application.
- Lustre is single-threaded. Lustre provides a runtime to manage these side effects and feed their result back into its state management system.

REA

Ann

Char

Exan

Quic

Mana

Side

Serve

Full st

SPA de

Full st

For Elm

For Rea

For Live

Links

Sponso

Reposit

Hex

Modu

lustre

lustre/at

lustre/component

W

Pages

REA

Ann

Char

Exam

Quick

Man

Side

Serve

Full st

SPA de

Full st

For Elm

For Rea

For Live

Links

Sponso

Reposit

Hex

Modu

lustre

lustre/at

lustre/componen

Lustre

Each of t

demonst

looking

to jump

01-ba

These e

render

•

•

•

•

•

•

•

•

•

•

•

•

•

•

02-inputs

Handling inputs and interactive elements is an important part of any application! These examples cover how to work with user input within the Model-View-Update architecture.

- [01-controlled-inputs](#) demonstrates the most common way to handle `<input />` elements in Lustre.
- [02-decoding-events](#) shows you how to write your own event handlers and decoders, instead of relying on the ones provided by `lustre/event`.
- [03-debouncing](#) demonstrates how to implement debouncing for user inputs.
- [04-forms](#) shows how to work with form submissions in Lustre.

03-effects

These examples demonstrate how to handle side effects in Lustre applications. Side effects are actions that read from or affect change to the outside world, such as making HTTP requests, manipulating the DOM, or working with timers.

- [01-http-requests](#) demonstrates how side effects are handled in Lustre, using HTTP requests as an example.
- [02-random](#) shows how to generate random values in a Lustre application.
- [03-timers](#) demonstrates working with timers and intervals.
- [04-local-storage](#) shows how to interact with browser local storage.
- [05-dom-effects](#) demonstrates direct DOM manipulation effects.
- [06-optimistic-requests](#) shows how to implement optimistic UI updates.

something

tion,

s, and

ngle-

uesting

ne

reats

pre

eed a

Elm

uests or timers

ne to manage

ystem.

04-applications

Examples related to building full Lustre single-page applications (SPAs), covering concepts like routing, page organisation, and state hydration.

- [01-routing](#) shows you how to set up routing and navigation between pages.
- [04-hydration](#) demonstrates client-side hydration of server-rendered Lustre apps.

05-components

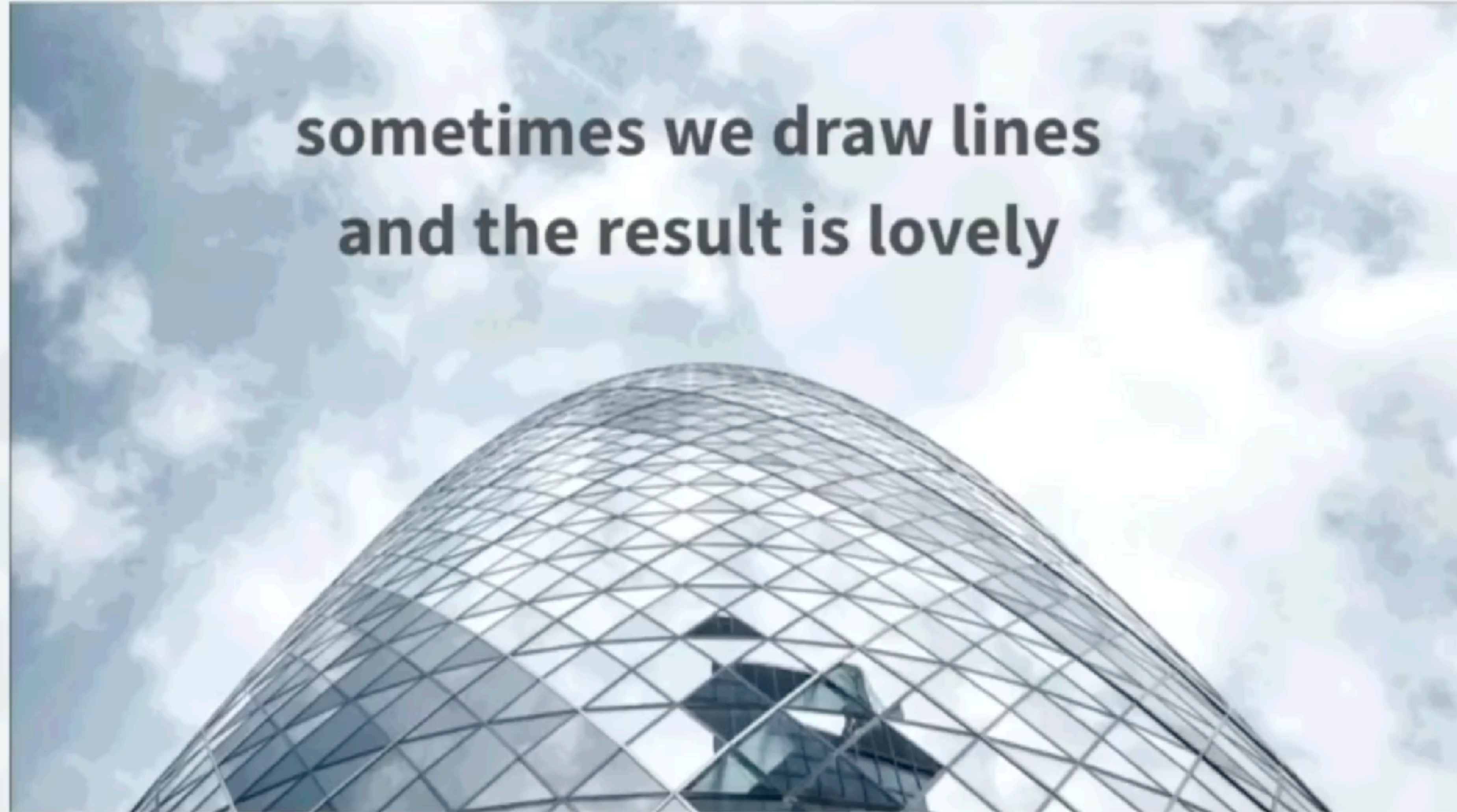
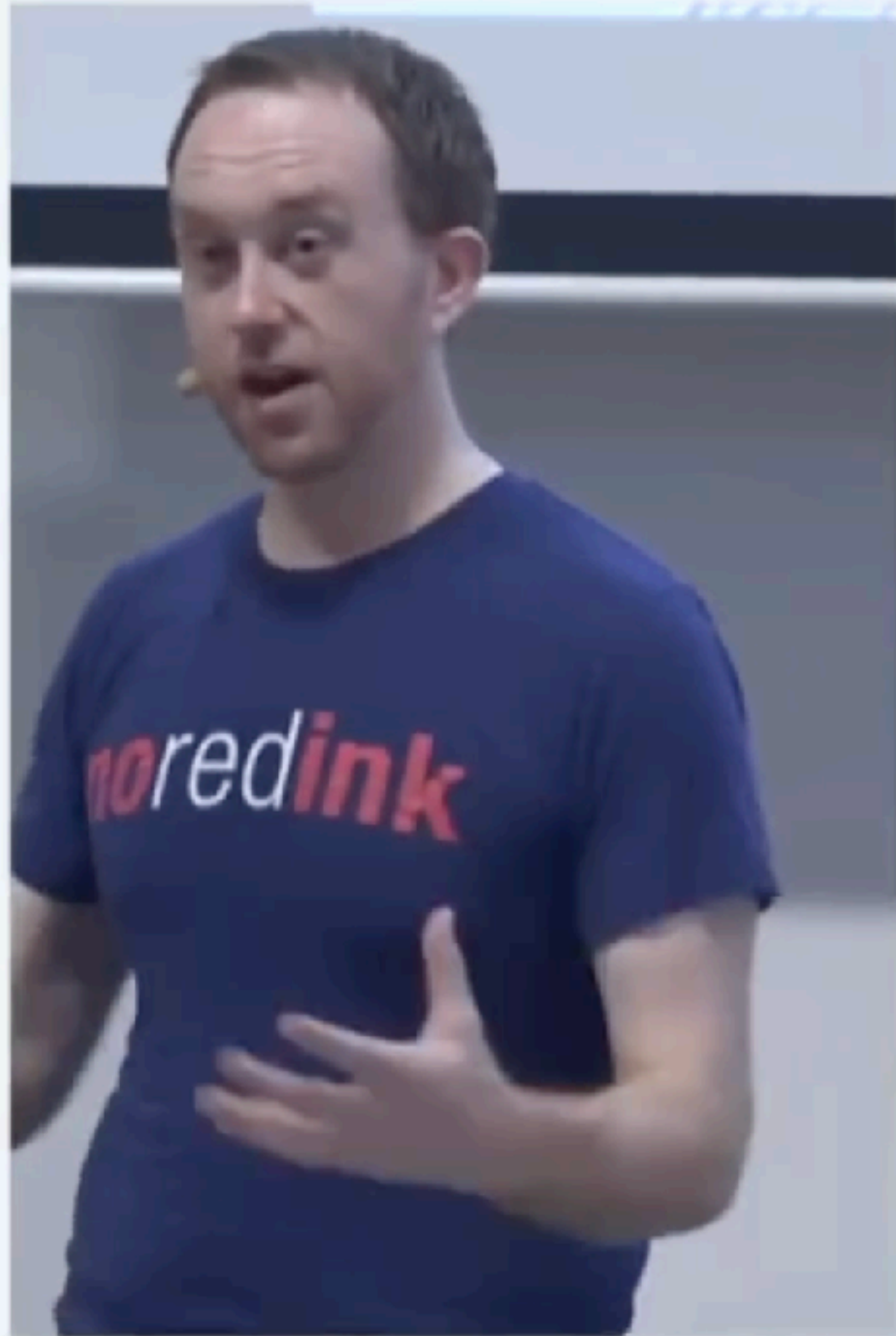
These examples demonstrate how to create reusable components in Lustre. Components are an advanced feature that let you construct real Custom Elements that have access to native features like shadow DOM, slotted content, and custom attributes and events.

- [01-basic-setup](#) introduces web components with Lustre.
- [02-attributes-and-events](#) shows how to handle attributes and events in components.
- [03-slots](#) demonstrates using slots in web components.

06-server-components

Ok but how **do** I build
real Lustre apps?





Richard Feldman, Elm Europe 2017. Scaling Elm Apps

the life of a file



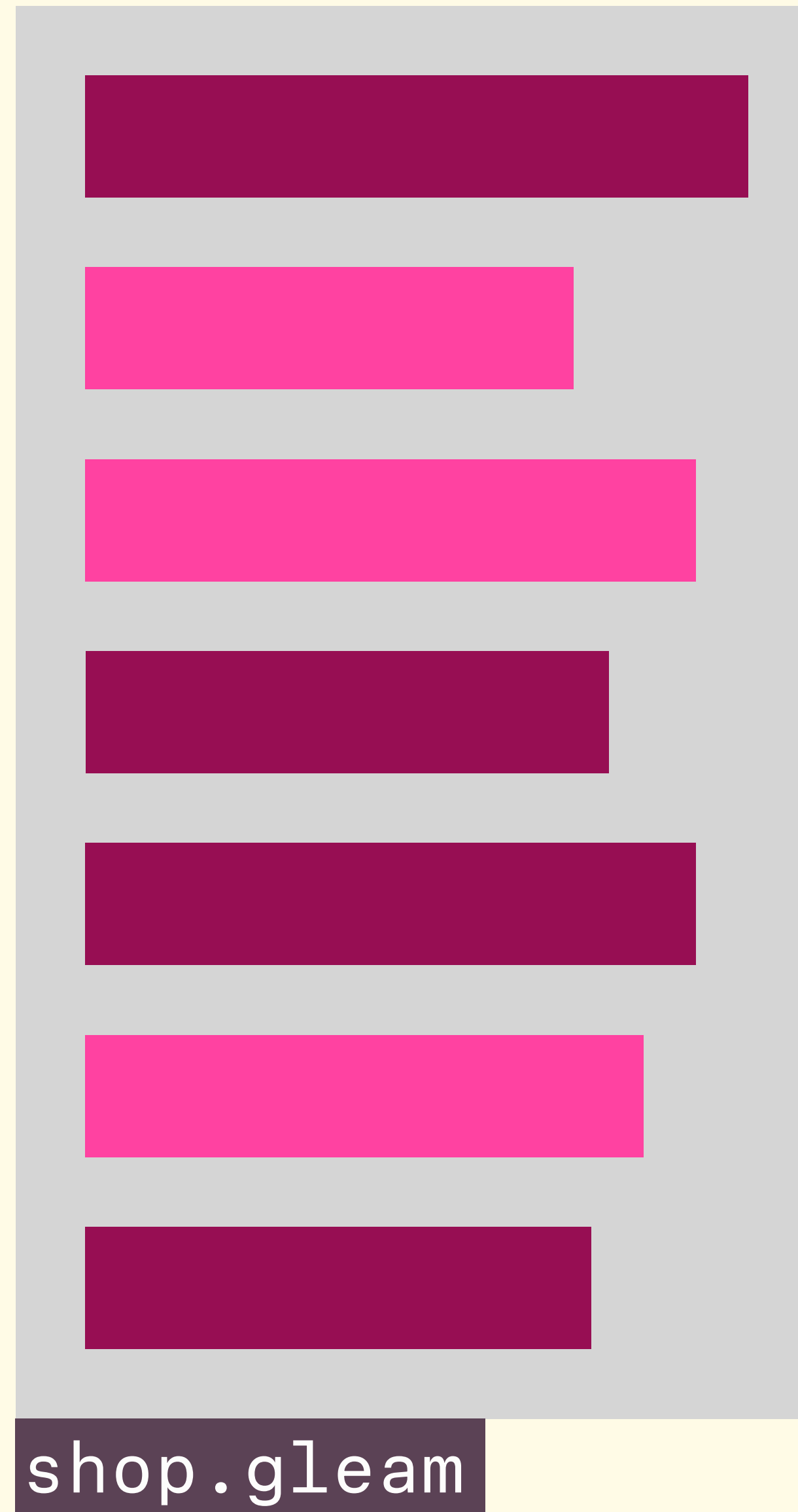
the life of a file



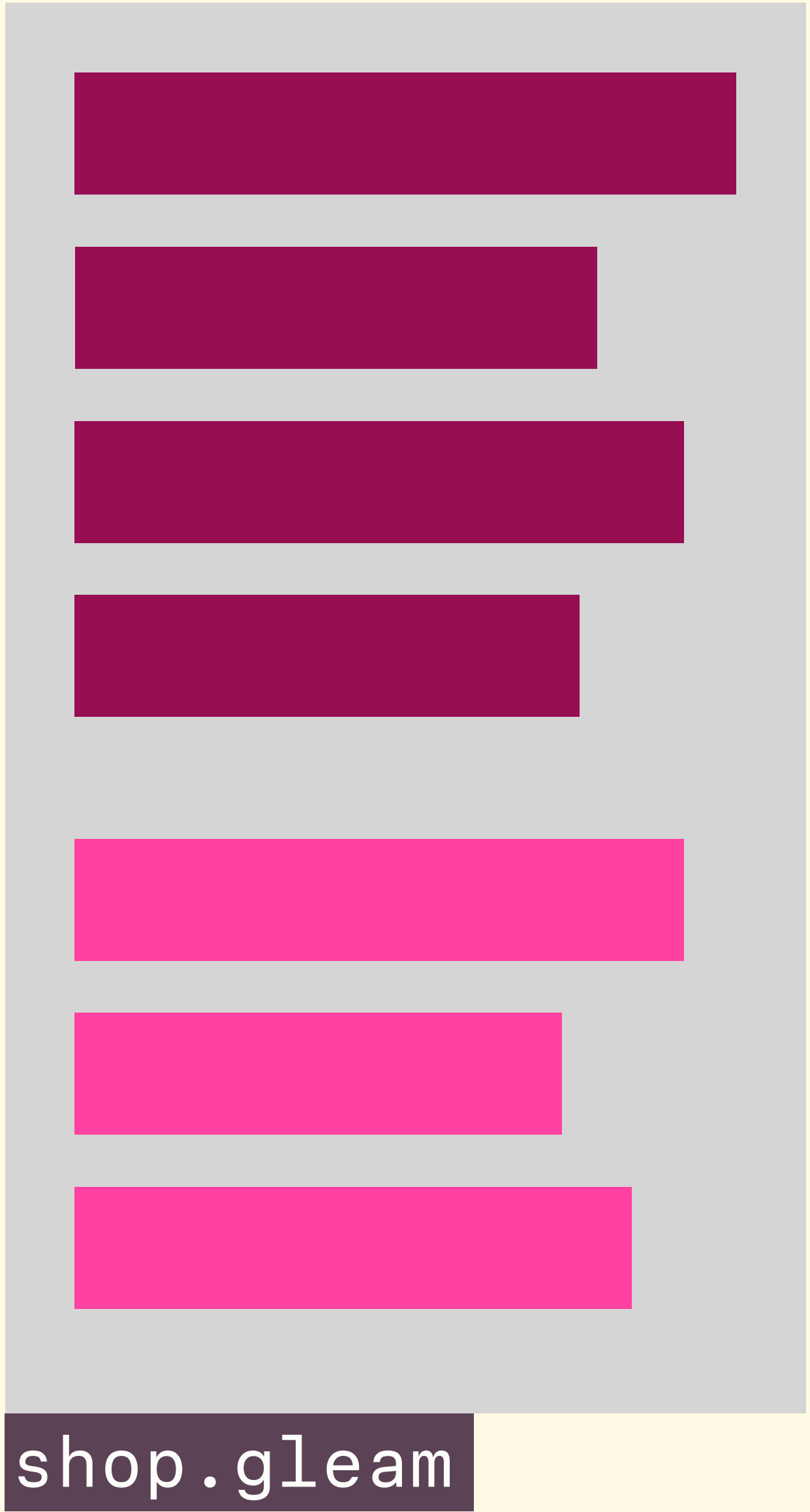
the life of a file



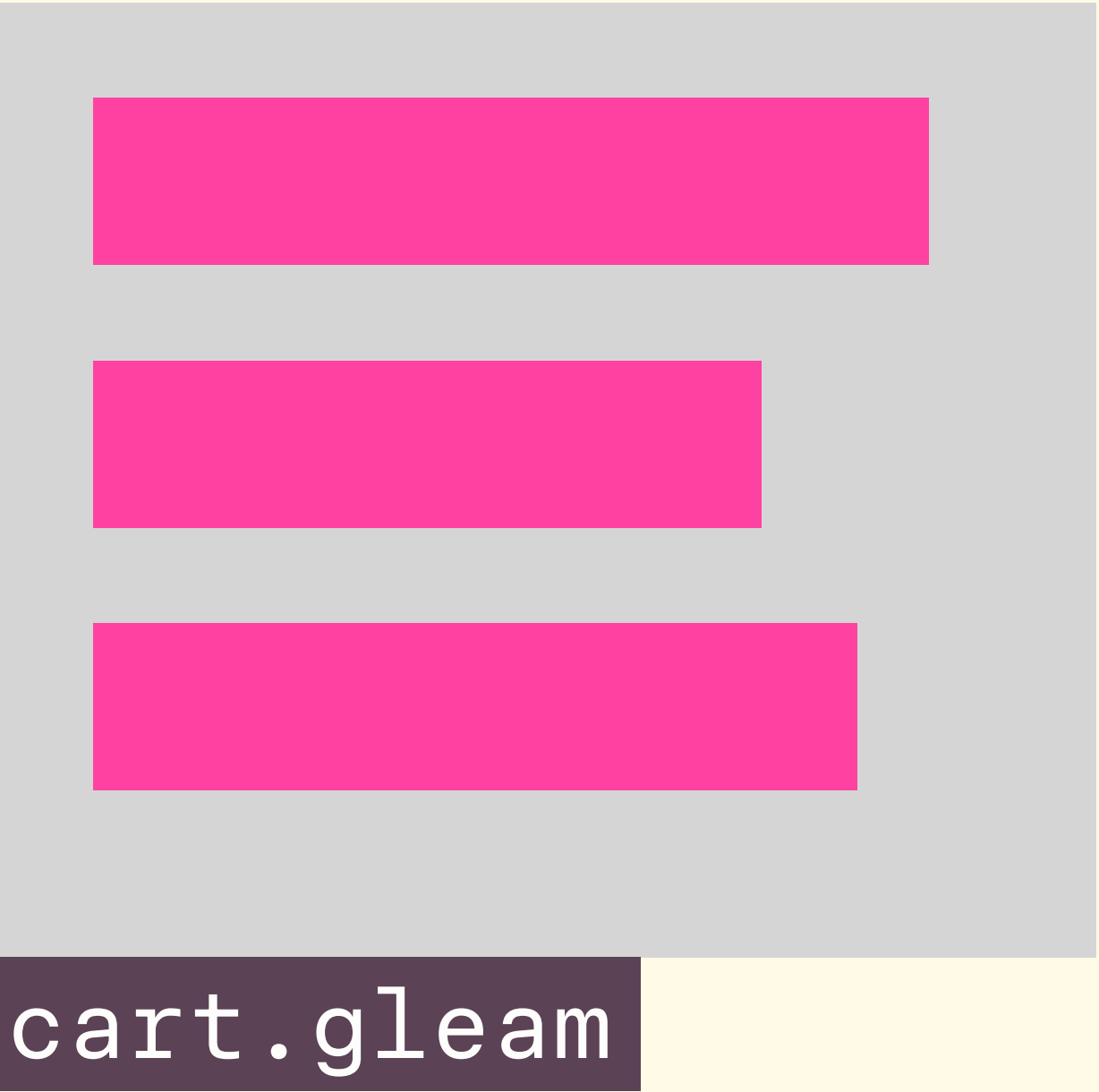
the life of a file



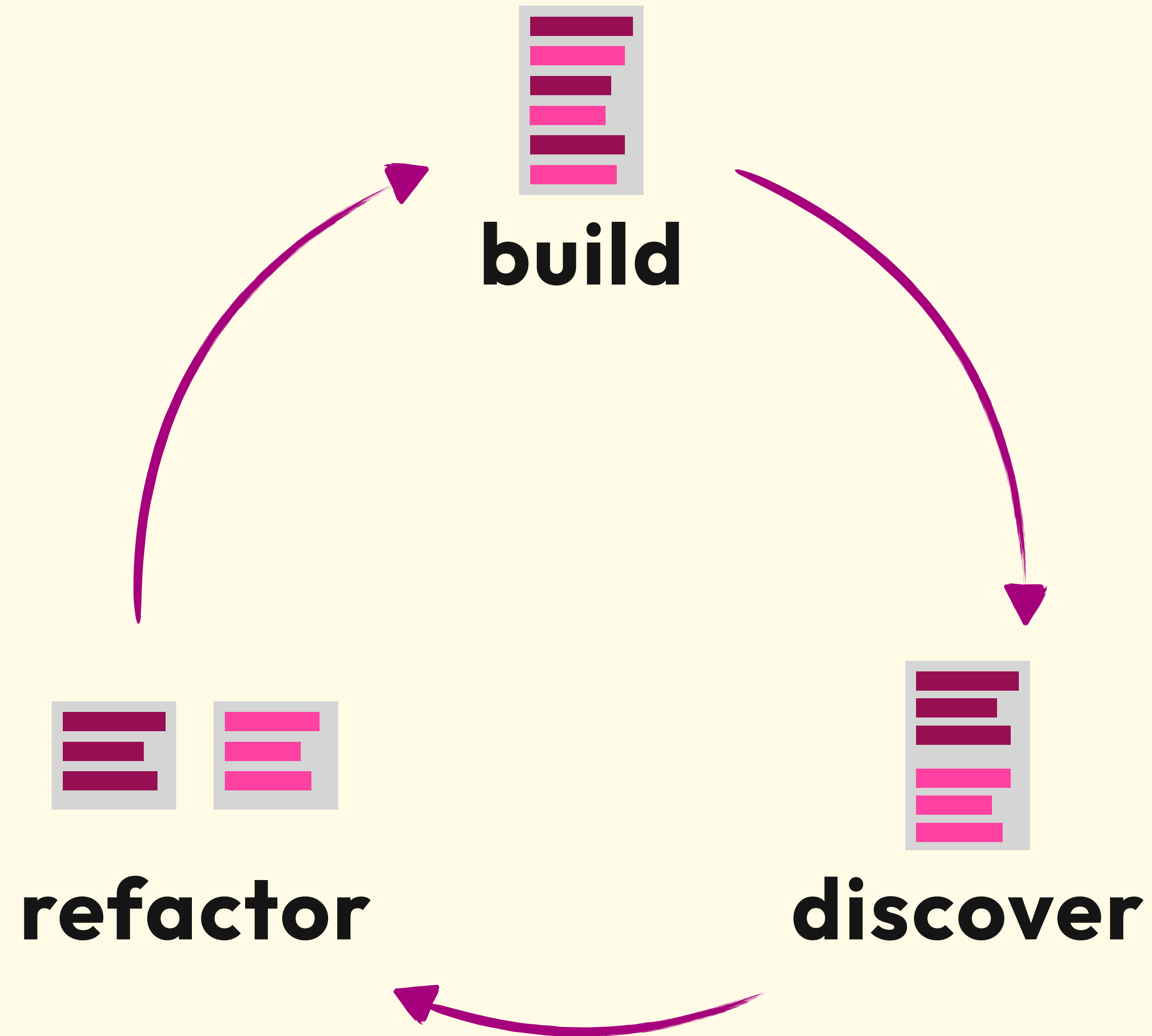
the life of a file



the life of a file

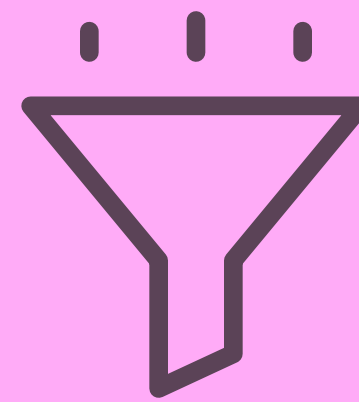


the life of a file

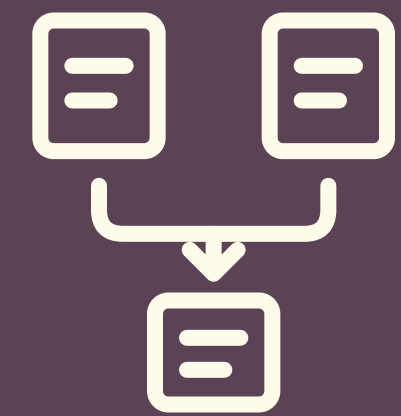




make code easier to find



reduce mental load



improve reuse

refactors are cheap

- Extract function or variable
- Rename symbol, move function
- Pattern match on argument
- Changing arguments
- Adding variants or record fields

making things easier to find

```
pub fn view(model: Model) -> Element(Message) {  
    html.div([], [  
        // ... 944 lines of code ...  
    ])  
}
```

making things easier to find

```
pub fn view(model: Model) {  
    html.div([], [  
        view_header(model),  
        view_main(model),  
        view_footer(model),  
    ])  
}  
  
pub fn view_header(model: Model) {  
    // ... 237 lines of code ...  
}  
  
pub fn view_main(model: Model) {  
    // ... 521 lines of code ...  
}  
  
pub fn view_footer(model: Model) {  
    // ... 186 lines of code ...  
}
```

reducing context

```
pub fn view_header(model: Model) -> Element(Message) {  
    // ... 237 lines of code ...  
}
```

```
pub fn view_main(model: Model) -> Element(Message) {  
    // ... 521 lines of code ...  
}
```

```
pub fn view_footer(model: Model) -> Element(Message) {  
    // ... 186 lines of code ...  
}
```

reducing context

```
pub fn view_header(user: User, menu_open: Bool) -> Element(Message) {  
    // ... 237 lines of code ...  
}
```

```
pub fn view_main(model: Model) -> Element(Message) {  
    // ... 521 lines of code ...  
}
```

```
pub fn view_footer() -> Element(message) {  
    // ... 186 lines of code ...  
}
```

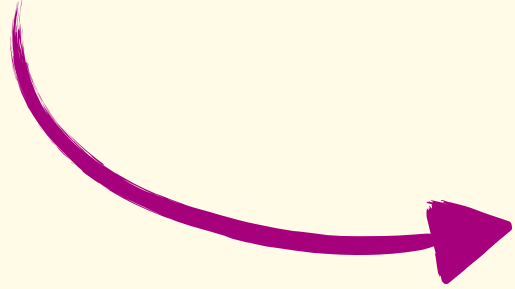
reducing context

```
pub fn view_header(
  user: User,
  menu_open: Bool
) -> Element(Message) {
  html.header([], [
    // ...
    html.button([event.on_click(UserToggledMenu)], [
      html.text("Menu")
    ])
  ])
}
```

reducing context

```
pub fn view_header(
  user: User,
  menu_open: Bool,
  handle_menu_toggle: message
) -> Element(message) {
  html.header([], [
    // ...
    html.button([event.on_click(handle_menu_toggle)], [
      html.text("Menu")
    ])
  ])
}
```

... Can only send this
specific message!



refactoring view

```
pub type Message {  
  UserSelectedTab(Tab)  
  UserChangedPage(page: Int)  
  UserClickedLogin  
  UserClickedLogout  
  AuthenticationTokenExpired  
  UserToggledMenuOpen  
  ApiReturnedArticles(Result(List(Article), Error)))  
  ApiReturnedPopularTags(Result(List(Tag), Error)))  
  UserChangedCommentDraft(String)  
  UserSubmittedComment  
  ApiCreatedComment(Result(Comment, Error))  
  // 54 more message...  
}
```

```
pub type Model {  
  Model(  
    session: Option(User),  
    route: Route,  
    tab: Tab,  
    current_page: Int,  
    total_pages: Int,  
    articles: RemoteData(List(Article)),  
    errors: List( #(String, String) ),  
    tags: RemoteData(List(Article)),  
    menu_open: Bool,  
    // 11 more fields ...  
  )  
}
```

refactoring view

```
pub type Message {  
  UserSelectedTab(Tab)  
  UserChangedPage(page: Int)  
  UserClickedLogin  
  UserClickedLogout  
  AuthenticationTokenExpired  
  UserToggledMenuOpen  
  ApiReturnedArticles(Result(List(Article), Error)))  
  ApiReturnedPopularTags(Result(List(Tag), Error)))  
  UserChangedCommentDraft(String)  
  UserSubmittedComment  
  ApiCreatedComment(Result(Comment, Error))  
  // 54 more message...  
}
```

```
pub type Model {  
  Model(  
    session: Option(User),  
    route: Route,  
    tab: Tab,  
    current_page: Int,  
    total_pages: Int,  
    articles: RemoteData(List(Article)),  
    errors: List( #(String, String) ),  
    tags: RemoteData(List(Article)),  
    menu_open: Bool,  
    // 11 more fields ...  
  )  
}
```

1. view gets too big

2. update gets too big

1. view gets too big

2. **update gets too big**

is line count important?

```
pub fn update(model: Model, message: Message) {  
    case message {  
        // ... 276 other lines ...  
  
        UserToggledMenuOpen -> {  
            let menu_open = !model.menu_open  
            #(Model(..model, menu_open:), effect.none())  
        }  
  
        // ... 545 other lines ....  
    }  
}
```

is line count important?

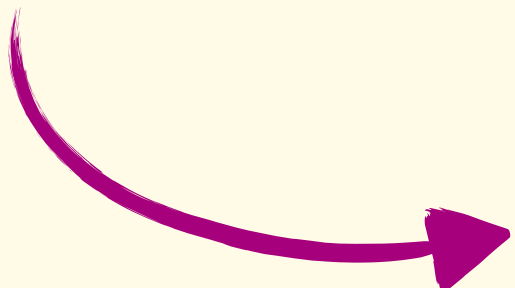
```
pub fn update(model: Model, message: Message) {  
  case message {  
    // ... 276 other lines ...
```

```
    UserToggledMenuOpen -> {  
      let menu_open = !model.menu_open  
      #(Model(..model, menu_open:), effect.none())  
    }
```

```
    // ... 545 other lines ....
```

```
  }
```

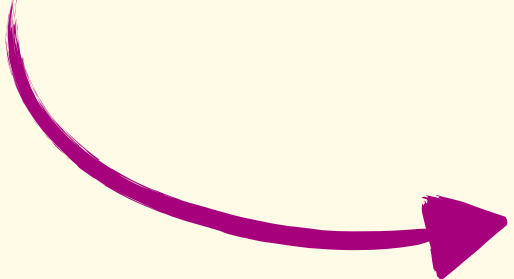
```
}
```



*... other branches cannot
influence this one!*

same strategy

```
fn toggle_menu(model: Model) -> #(Model, Effect(Message))
```



... extract update branches?

same strategy

```
fn toggle_menu(model: Model) -> #(Model, Effect(Message))
```

```
fn add_product(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

same strategy

```
fn toggle_menu(model: Model) -> #(Model, Effect(Message))
```

```
fn add_product(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

```
fn remove_product(model: Model, product: Product)  
  -> #(Model, Effect(Message))
```

same strategy

```
fn toggle_menu(model: Model) -> #(Model, Effect(Message))
```

```
fn add_product(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

```
fn remove_product(model: Model, product: Product)  
  -> #(Model, Effect(Message))
```

```
fn change_quantity(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

same strategy

```
fn toggle_menu(model: Model) -> #(Model, Effect(Message))
```

```
fn add_product(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

```
fn remove_product(model: Model, product: Product)  
  -> #(Model, Effect(Message))
```

```
fn change_quantity(model: Model, product: Product, quantity: Int)  
  -> #(Model, Effect(Message))
```

```
fn update_shipping_costs(model: Model) -> #(Model, Effect(Message))
```

```
fn compute_total(model: Model) -> #(Model, Effect(Message))
```

really an improvement?

```
pub fn update(model: Model, message: Message) {  
    case message {  
  
        UserClickedAddToCart(quantity, product) -> {  
            let #(model, effect1) = add_product(model, product, quantity)  
            let #(model, effect2) = update_shipping_costs(model)  
            let #(model, effect3) = compute_total(model)  
  
            #(model, effect.batch([effect1, effect2, effect3]))  
        }  
    }  
}
```

same strategy²

```
fn toggle_menu(model: Model) -> Model
```

```
fn add_product(items: List(LineItem), product: Product, quantity: Int)  
  -> List(LineItem)
```

```
fn remove_product(items: List(LineItem), product: Product)  
  -> List(LineItem)
```

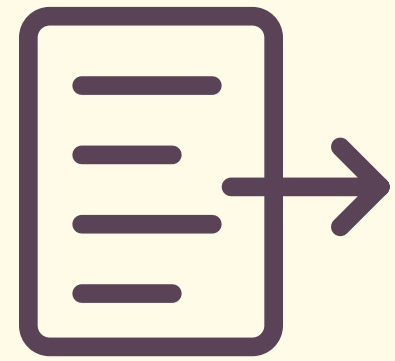
```
fn change_quantity(items: List(LineItem), product: Product, quantity: Int)  
  -> List(LineItem)
```

```
fn compute_shipping_costs(items: List(LineItem), country: Country) -> Price
```

```
fn compute_total(items: List(LineItem), shipping: Price) -> Price
```

same strategy²

```
pub fn update(model: Model, message: Message) {  
  case message {  
    // ...  
    UserClickedAddToCart(product) -> {  
      let items = add_product(model.items, product, 1)  
      let shipping = update_shipping_costs(items, model.country)  
      let total = compute_total(items, shipping)  
      let model = Model(model:, items:, shipping:, total:)  
  
      #(model, effect.none())  
    }  
  }  
}
```



awkward to extract



update is still long



no enforced guarantees

awkward to extract

```
pub type Cart {  
    Cart(  
        items: List(LineItem),  
        shipping_country: Country,  
        shipping_costs: Price,  
        total: Price  
    )  
}
```

awkward to extract

```
pub type Cart {
```

```
  Cart(
```

```
    items: List(LineItem),
```

```
    shipping_country: Country,
```

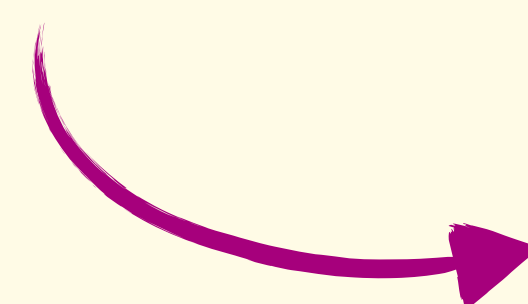
```
    shipping_costs: Price,
```

```
    total: Price
```

```
  )
```

```
}
```

... can now update all related fields together!



```
pub fn add_product(cart: Cart, product: Product, quantity: Int) -> Cart
```

```
pub fn remove_product(cart: Cart, product: Product) -> Cart
```

```
pub fn change_quantity(cart: Cart, product: Product, quantity: Int) -> Cart
```

update is still long

```
pub type CartMessage {  
    UserClickedAddToCart(product: Product)  
    UserClickedRemoveFromCart(product: Product)  
    UserChangedProductQuantity(product: Product, quantity: Int)  
}
```

update is still long

```
pub type CartMessage {  
    UserClickedAddToCart(product: Product)  
    UserClickedRemoveFromCart(product: Product)  
    UserChangedProductQuantity(product: Product, quantity: Int)  
}
```

```
pub fn update_cart(cart: Cart, message: CartMessage) -> Cart
```

enforcing guarantees

```
// -- CART -----  
pub type Cart { ... }  
pub type CartMessage { ... }
```

enforcing guarantees

```
// -- CART -----  
pub type Cart { ... }  
pub type CartMessage { ... }  
  
pub fn add_product(cart: Cart, product: Product, quantity: Int) -> Cart  
pub fn remove_product(cart: Cart, product: Product) -> Cart  
pub fn change_quantity(cart: Cart, product: Product, quantity: Int) -> Cart  
  
pub fn update_cart(cart: Cart, message: CartMessage) -> Cart
```

enforcing guarantees

```
// -- CART -----  
pub type Cart { ... }  
pub type CartMessage { ... }  
  
pub fn add_product(cart: Cart, product: Product, quantity: Int) -> Cart  
pub fn remove_product(cart: Cart, product: Product) -> Cart  
pub fn change_quantity(cart: Cart, product: Product, quantity: Int) -> Cart  
  
pub fn update_cart(cart: Cart, message: CartMessage) -> Cart  
  
pub fn view_line_items(cart: Cart) -> Element(CartMessage)  
pub fn view_summary(cart: Cart) -> Element(message)
```

enforcing guarantees

```
// cart.gleam
pub opaque type Cart { ... }
pub opaque type Message { ... }

pub fn add_product(cart: Cart, product: Product, quantity: Int) -> Cart
pub fn remove_product(cart: Cart, product: Product) -> Cart
pub fn change_quantity(cart: Cart, product: Product, quantity: Int) -> Cart

pub fn update(cart: Cart, message: Message) -> Cart

pub fn view_line_items(cart: Cart) -> Element(Message)
pub fn view_summary(cart: Cart) -> Element(message)
```

- 1. Structure and patterns are discovered**
- 2. Build modules around guarantees**

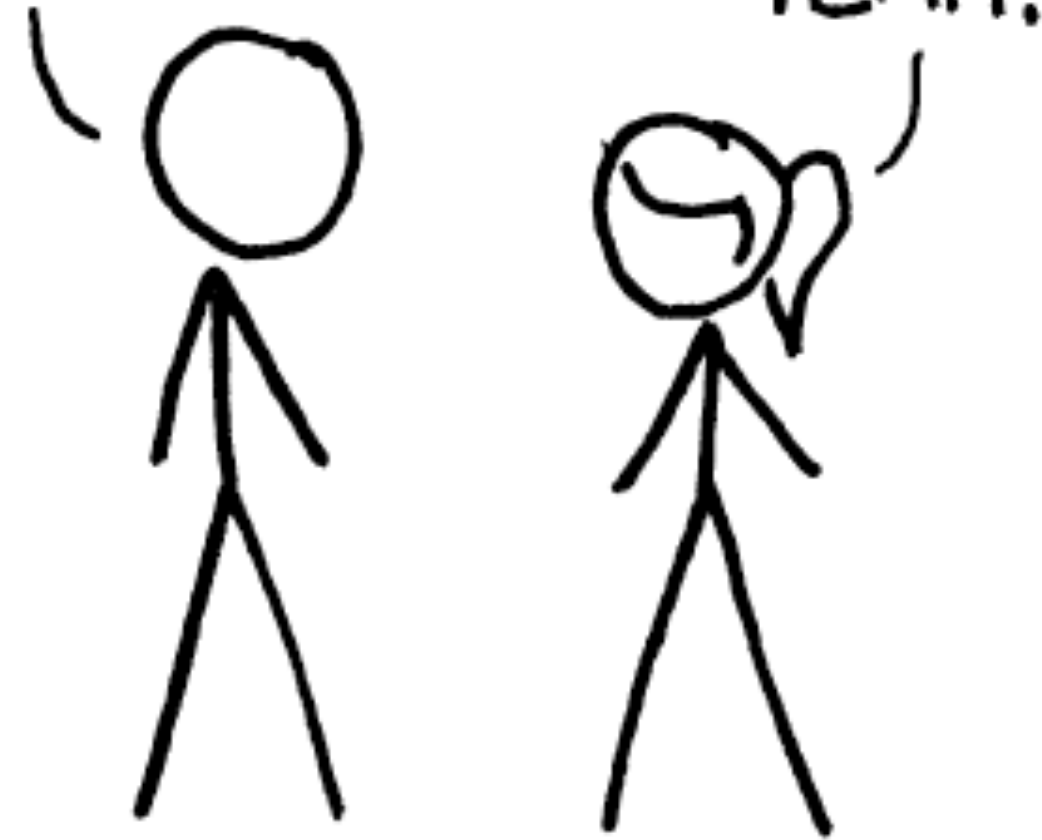
Ok but how **do** I build
real Lustre apps?



HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

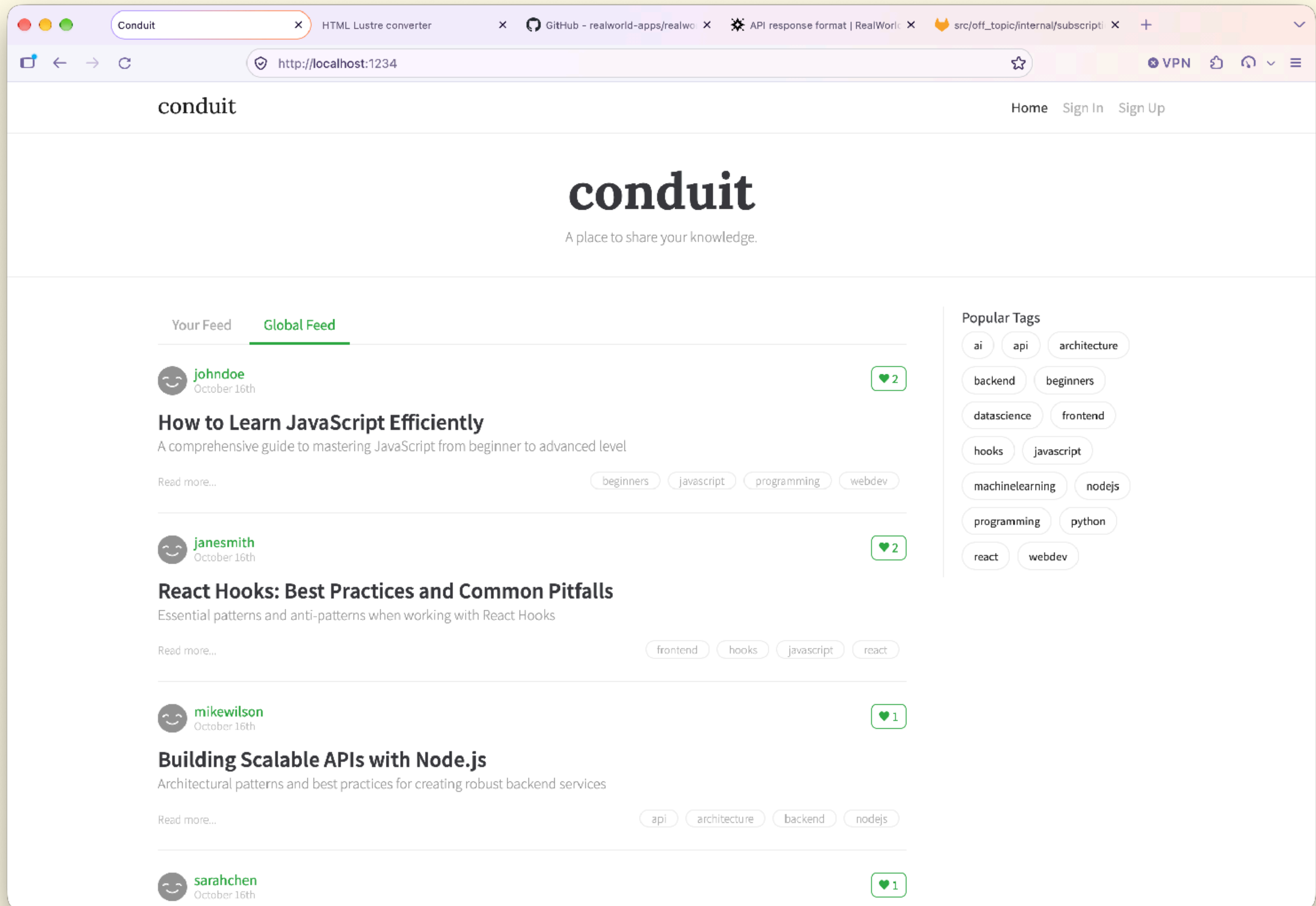
SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.



SOON:

SITUATION:
THERE ARE
15 COMPETING
STANDARDS.



directory structure

domain modules

```

📁 ./conduit/
├── ☆ api.gleam
├── ☆ article.gleam
├── ☆ comment.gleam
├── ☆ page.gleam
├── ☆ profile.gleam
├── ☆ route.gleam
├── ☆ session.gleam
├── ☆ tags.gleam
└── ☆ ui.gleam

```

pages

```

📁 ./conduit/page/
├── ☆ article.gleam
├── ☆ editor.gleam
├── ☆ home.gleam
├── ☆ login.gleam
├── ☆ profile.gleam
├── ☆ register.gleam
└── ☆ settings.gleam

```

pages are nested apps

page/home.gleam

```
// -- MODEL -----
```

```
pub opaque type Model {  
  Model(  
    tab: Tab,  
    current_page: Int,  
    feed: RemoteData(List(Article)),  
    popular_tags: RemoteData(List(Tag)),  
  )  
}
```

pages are nested apps

page/home.gleam

```
// -- MODEL -----
```

```
pub opaque type Model {  
  Model(  
    tab: Tab,  
    current_page: Int,  
    feed: RemoteData(List(Article)),  
    popular_tags: RemoteData(List(Tag)),  
  )  
}  
  
pub type RemoteData(result) {  
  NotAskedYet  
  Loading  
  Loaded(result)  
  Failed(api.Error)  
}
```

group things within one module

page/home.gleam

```
// -- IMPORTS -----
```

```
// -- MODEL -----
```

```
// -- REQUESTS -----
```

```
// -- UPDATE -----
```

```
// -- VIEW -----
```

model & init

page/home.gleam

```
// -- MODEL -----  
  
pub opaque type Model { ... }  
  
pub fn init(session: Session, tab: Tab) -> #(Model, Effect(Message))
```

model & init

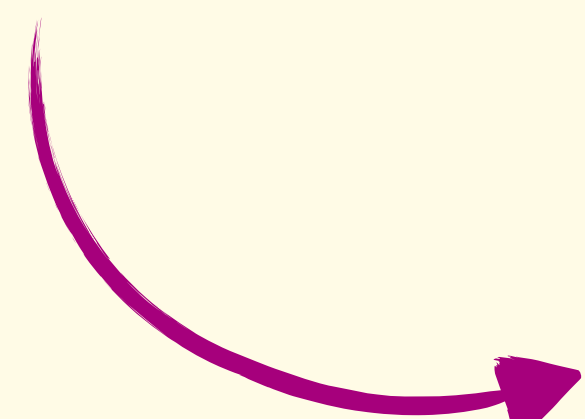
page/home.gleam

```
// -- MODEL -----
```

```
pub opaque type Model { ... }
```

```
pub fn init(session: Session, tab: Tab) -> #(Model, Effect(Message))
```

```
pub fn change_tab(model: Model, tab: Tab) -> #(Model, Effect(Message))
```



*... we can expose
the functions we need!*

expose the shape you need

page/home.gleam

```
// -- UPDATE -----
```

```
pub opaque type Message {  
  ApiReturnedArticles(Result(List(Article), Error))  
  ApiReturnedTags(Result(List(Tag), Error))  
}
```

... past-tense message names



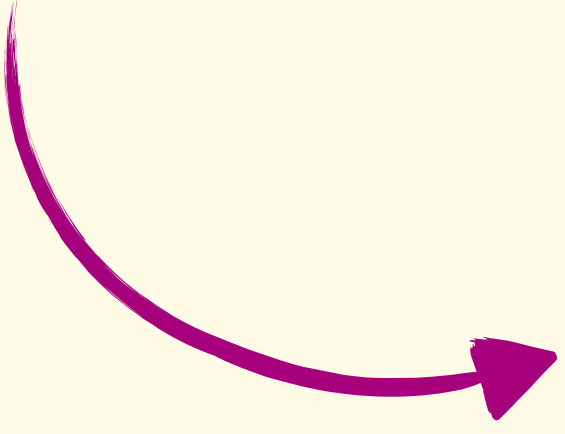
```
pub fn update(session: Session, model: Model, message: Message)  
  -> #(Model, Effect(Message))
```

multiple pages

page.gleam

```
pub opaque type Page {  
  Home(home.Model),  
  Login(login.Model)  
  Register(register.Model)  
  Article(article.Model)  
  // ...  
}
```

*... there can only be
one active page at a time*



multiple pages

page.gleam

```
pub opaque type Page {  
  Home(home.Model),  
  Login(login.Model)  
  Register(register.Model)  
  Article(article.Model)  
  // ...  
}
```

```
pub opaque type Message {  
  HomeDispatchedMessage(home.Message)  
  LoginDispatchedMessage(login.Message)  
  RegisterDispatchedMessage(register.Message)  
  ArticleDispatchedMesssage(article.Message)  
  // ...  
}
```

multiple pages

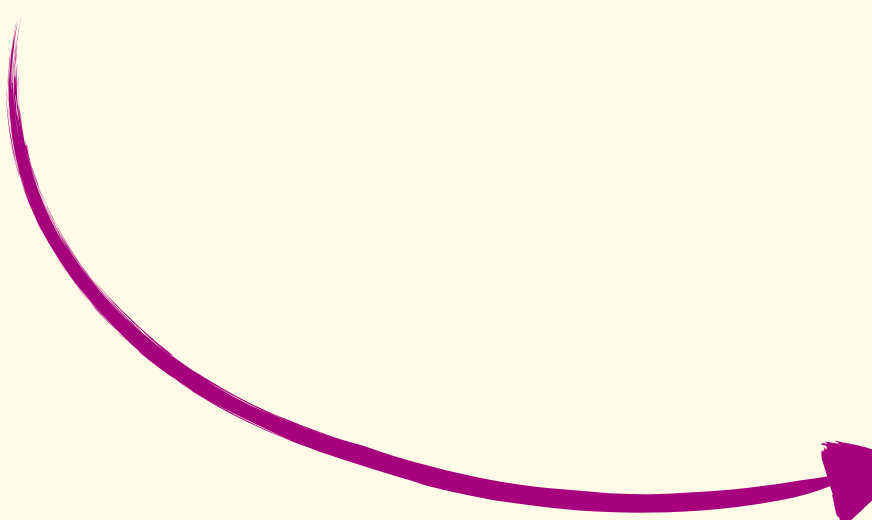
page.gleam

```
pub fn update(session: Session, page: Page, message: Message) {  
  case page, message {  
    Home(model), HomeDispatchedMessage(message) -> {  
      let #(model, effect) = home.update(session, model, message)  
      #(Home(model), effect.map(effect, HomeDispatchedMessage))  
    }  
  }  
  // ...  
}
```

route changes

page.gleam

```
pub fn handle_route_change(page: Page, route: Route) {  
  case route, page {  
    route.Index(tab:), Home(model) -> {  
      let #(model, effect) = home.change_tab(model, tab)  
      #(Home(model), effect.map(effect, HomeDispatchedMessage))  
    }  
  
    _, _ -> init(route)  
  }  
}
```



*... easy to do custom things
when needed!*

route changes

route.gleam

```
pub type Route {  
  Index(tab: HomeTab)  
  SignIn  
  SignUp  
  Article(slug: String)  
  // ...  
  NotFound  
}
```

```
pub fn from_uri(uri: Uri) -> Route {  
  case uri.path_segments(uri) {  
    [] -> Index(GlobalFeed)  
    ["sign-in"] -> SignIn  
    ["register"] -> SignUp  
    ["article", slug] -> Article(slug)  
    _ -> NotFound  
  }  
}
```

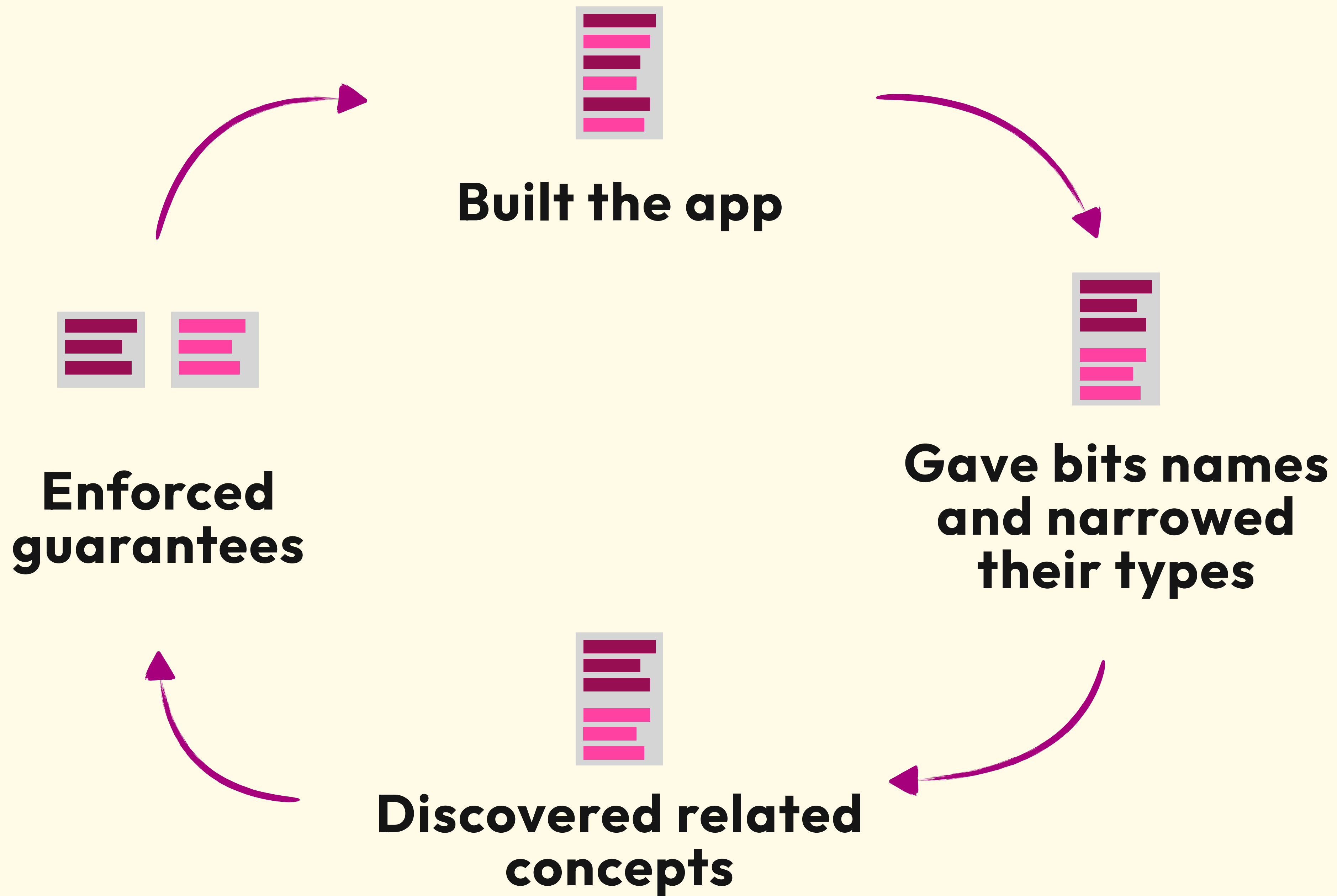
route changes

route.gleam

```
pub type Route {  
  Index(tab: HomeTab)  
  SignIn  
  SignUp  
  Article(slug: String)  
  // ...  
  NotFound  
}
```

```
pub fn from_uri(uri: Uri) -> Route {  
  case uri.path_segments(uri) {  
    [] -> Index(GlobalFeed)  
    ["sign-in"] -> SignIn  
    ["register"] -> SignUp  
    ["article", slug] -> Article(slug)  
    _ -> NotFound  
  }  
}
```

```
pub fn href(route: Route) -> Attribute(message)
```



Building Big Lustre

Apps

Rebecca Reusch — Code BEAM Europe 2026



Thank you!

