Mithril Blueprint

"It's dangerous to go alone! Take this"

What is Mithril?

- A Mix Template
- Code Organization Conventions
- A Foundation for Elixir AppMachine

Original Proposal

https://hackmd.io/s/SIuNQ5ekM

Highlights

- Umbrella app
- Business logic is separate from frameworks
- Business logic is organized into **domains**
- **Kitchen sink** of options, including:
  - User Accounts
  - GraphQL API

Umbrella App

```
$ mix gen mithril armor --api graphql
.
├── README.md
├── apps
│   ├── armor      # Business logic ("Domains"), persistence
│   └── armor_api  # GraphQL API
│   └── armor_web  # Phoenix Endpoint
├── bin
│   ├── reset      # Resets and rebuilds the project
│   ├── setup      # Sets up all dependencies
│   ├── test       # Runs tests like they would run on CI
│   └── update     # Updates the project (for mobile devs)
├── config
├── mix.exs
└── mix.lock
```

What is a Domain?

- A module with a public interface
- Any *private* submodules it needs
- Each domain manages its own persistence
- Domains are designed around *features*
  - "Accounts"
  - "Shipping"
  - "Profile"

Designing Domains

1. Identify a feature or group of features that can stand "alone"
2. Define a public API
3. Finally, define persistence [1]

[1] Don't think about everything as a database table. Think behaviour first, persistence second.

Example Domain

```
Armor.Notifications.forgot_password("john@smith.com", "{reset-token}")
```

```
.
└── notifications
    ├── notifications.ex  # Public interface
    ├── email.ex          # Private module for email notifications
    ├── template.ex       # Private module to render templates
    └── templates         # Template storage
        ├── forgot_password.html.eex
        ├── forgot_password.text.eex
```

Controversial Decisions

No Global User Module

- `Accounts.User` only stores **login credentials**
- Each domain stores its own data, referencing `user_id`

No Global User Module

```
# A profiles domain would contain extra information about a user,
# for example, first/last name, date of birth, phone number
Armor.Profiles.get_profile(user_id)
# => {:ok, %Armor.Profiles.Profile{}}

# A shipping domain would contain all shipping addresses
# associated with a user.
Armor.Shipping.get_addresses(user_id)
# => {:ok, %Armor.Shipping.Address{...}}

# A billing domain would contain all a user's stored
# payment options and handle billing
Armor.Billing.get_payment_options(user_id)
```

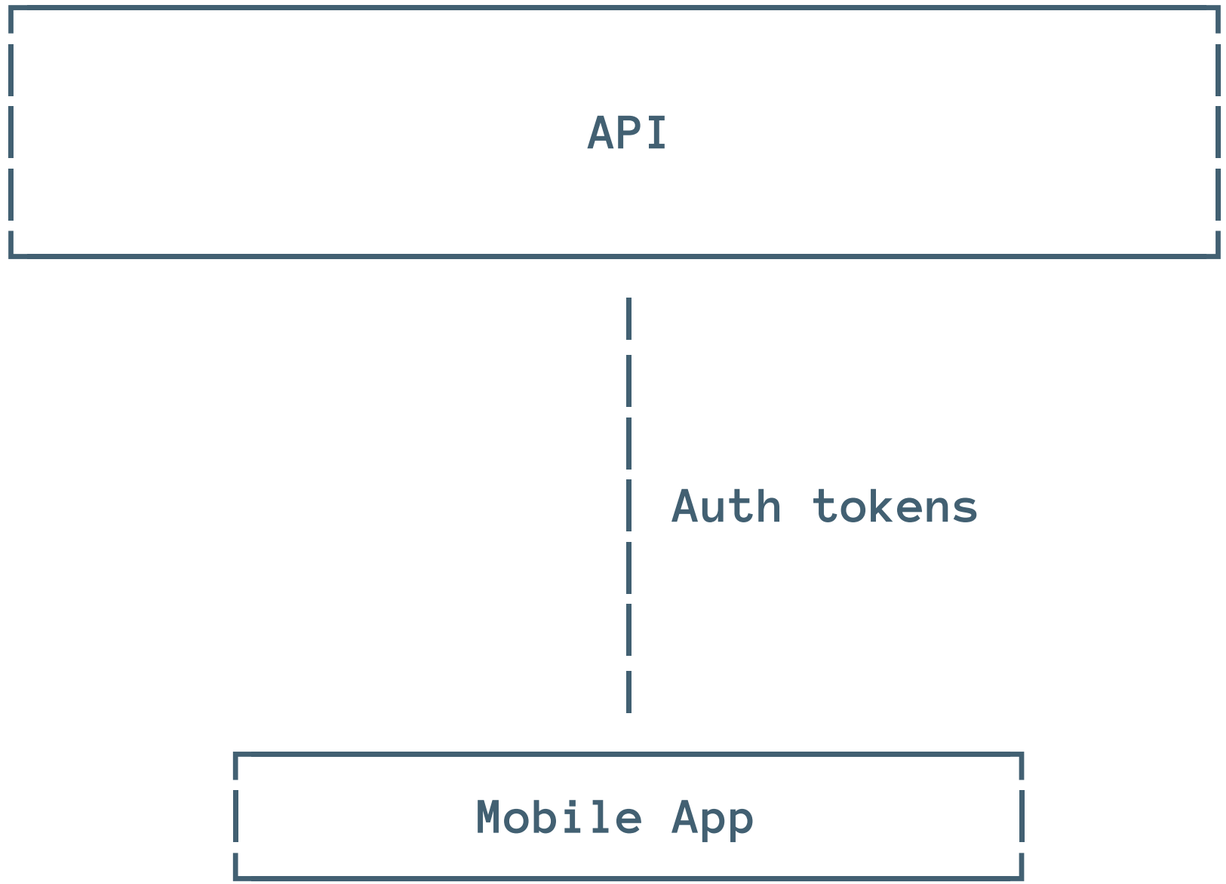
Authorization

Authorization

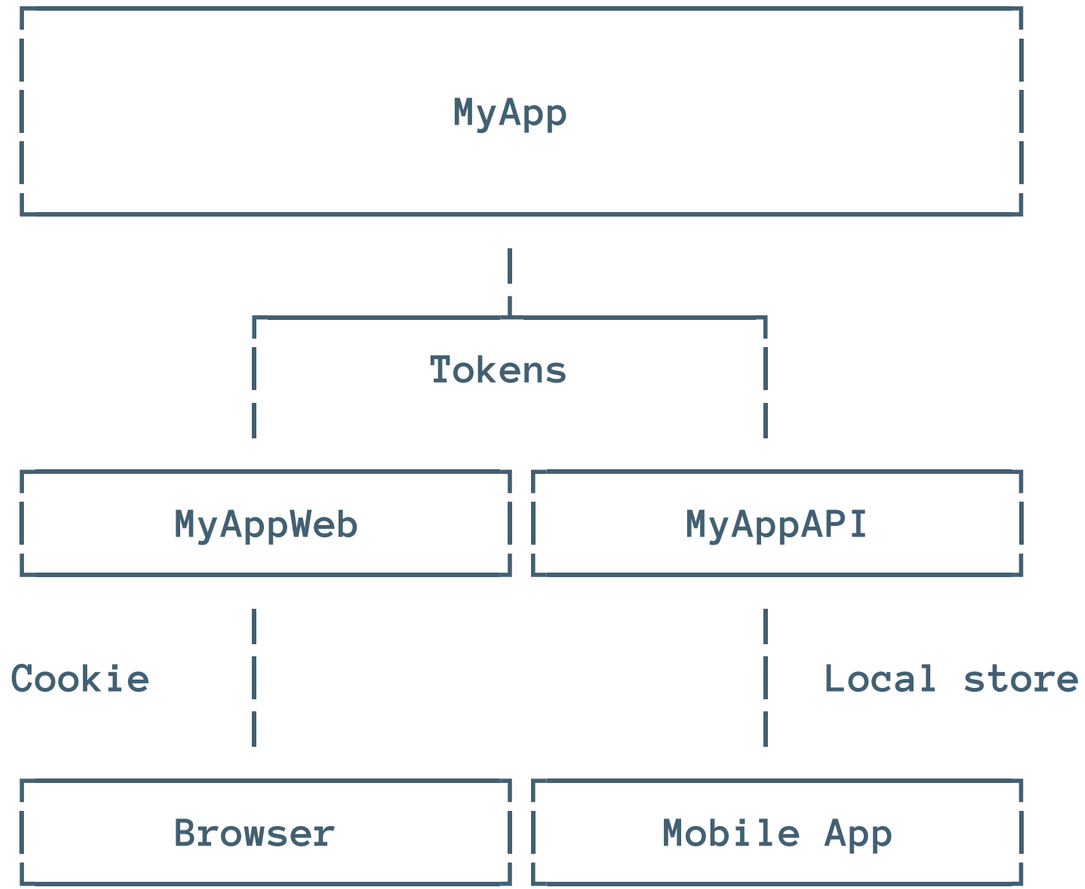
- Access control is a **core business concern**
- Specific rules **change often** and **unpredictably**
- Therefore, access control logic should go in **domains**, NOT Phoenix, plugs, or a library.[2]

[2] We need full control over it, and a pure Elixir module within our app is by far the easiest thing to control and change.

Authorization Method Already Solved!



Universal Token Authentication



Universal Token Authentication

- Each domain function requiring authorization takes a `user_token` argument, and internally calls a `Permissions` domain function to check permissions.
- `Accounts.create_login_token(email, password)` creates these tokens

Universal Token Authentication

We use the *same* authentication mechanism for our HTML and API interfaces.

- **Phoenix (HTML):** Store token in the session cookie
- **API:** Return token to the mobile client

Universal Token Authentication

```elixir
# In Phoenix controller action
# The token is stored in the session and put into assigns by a plug
def create(%{assigns: %{token: token}} = conn, params) do
  with {:ok, address} <- Shipping.create_address(token, params) do
    # ...
  end
end
```

Universal Token Authentication

```elixir
# In GraphQL resolver
# The token is parsed out of an `Authorization` header
def create_shipping_address(%{input: params}, %{token: token}) do
  with {:ok, address} <- Shipping.create_address(token, params) do
    # ...
  end
end
```

The place of

Frameworks

Frameworks are clients of the business logic

Just like mobile apps

Frameworks are Clients

- We isolate them to their own OTP app
- We use them to support external protocols
- Each client app contains only what is necessary to support their protocol

Phoenix is NOT YOUR APP

Your app is a pure Elixir app, delivered over HTTP/Websockets

Phoenix is NOT YOUR APP

If your app doesn't need HTML or Websocket support, Mithril will only generate this:

```
.
├── armor_web
│   ├── application.ex
│   ├── endpoint.ex
│   └── router.ex
└── armor_web.ex
```

GraphQL is NOT YOUR APP

Your app is a pure Elixir app, delivered over GraphQL

GraphQL is NOT YOUR APP

- Mithril won't even generate a GraphQL application unless you ask for it
- GraphQL APIs call into public domain functions exclusively, just like every other client

Benefits

Isn't this over-engineered?

Over-Engineering Defined

An abstraction whose costs outweigh its benefits.

Cost vs. Benefit

I argue that the benefits of this architecture outweigh its costs for projects of *all* sizes.

Costs

1. Complex Project Setup

Setting up this architecture manually takes some time.

However, this is completely mitigated by Mithril's project generator.

2. Naming Things is Hard

It takes more effort to define properly scoped domain modules than it does to throw all the logic together.

3. There Are More Files

- Private domain module
- Private Ecto schema or changeset function
- Public domain module & test
- Phoenix controller & test
- Phoenix view
- Phoenix template

4. Training

This architecture will require training. [3]

[3] Modeling applications the "Rails way" was also not intuitive and had to be learned.

Benefits

1. No Rewrites

Clients never want to rewrite.
We only get one chance to get it right.

Refactoring is Easy

- It's easy to upgrade a framework to the latest version, because it only contains wrapper code
- It's easy to refactor a domain, you just have to maintain its public API
- Changing a domain is unlikely to affect other domains, because domains are self-contained

New Protocols are Easy

How many projects will never need a mobile API?

- Need a GraphQL API? Just layer it on top of the existing, well-defined public API.
- A new, better web framework came out? Swap out Phoenix.
- A new, better API protocol came out? Add a new API app.

Scaling is Easy

- When the need arises, you can easily host the business logic app, API app, and web app on separate hardware.
- If you need to split things out even more, **domains** provide clear boundaries to break down the logic app into smaller apps.

2. Code Reuse

Profit through automation

Develop New Libraries

The Umbrella App architecture provides a perfect incubator for new libraries.[4]

```
$ cd apps/
$ mix new my_library
```

[4] We created at least 4 open source libraries from CrossConnect alone. Umbrella apps would have made this easier.

Reuse Domains!!

Because domains are self-contained, we can **reuse them between projects!**

Every project, no matter how small, probably has a domain we could reuse.

Reuse Domains!!

There are two ways we could reuse a domain:

1. Extract the domain to a library
2. Add the domain to `Mithril`'s project generator

Reuse Domains!!

- Accounts
- Permissions
- Shipping
- Billing
- Cart

3. Onboarding

Developers switch projects frequently

Projects Will Rhyme

- Similar domains (e.g. "Accounts")
- Predictable architecture and conventions

Effective Juniors

- They only have to learn the Public API of the logic app
- Seniors can delegate the web and API apps to the juniors and focus on the domains
- Juniors can move between projects when more capacity needed

Documentation

- Mithril generates docs and **setup scripts**
- Dividing responsibility encourages the senior to document the public API

4. BEAM

Use the full power of OTP

OTP & BEAM

When we stop building our apps around functionality instead of database tables, we will use *GenServer* and OTP more effectively.

Demo

Comments & Questions