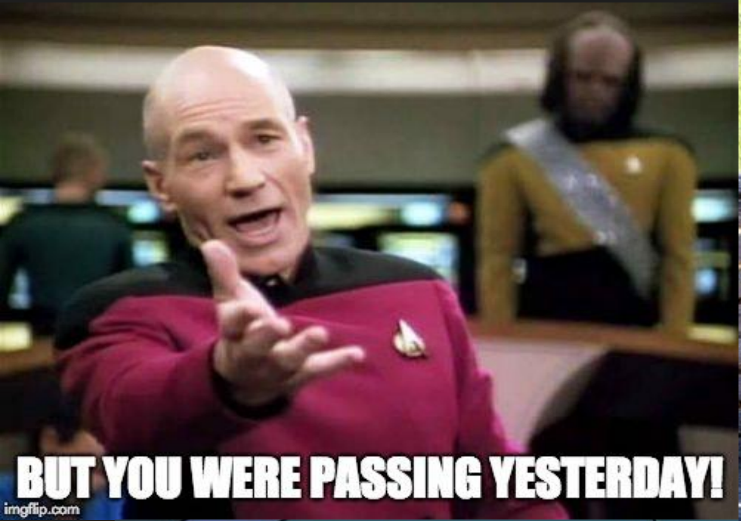
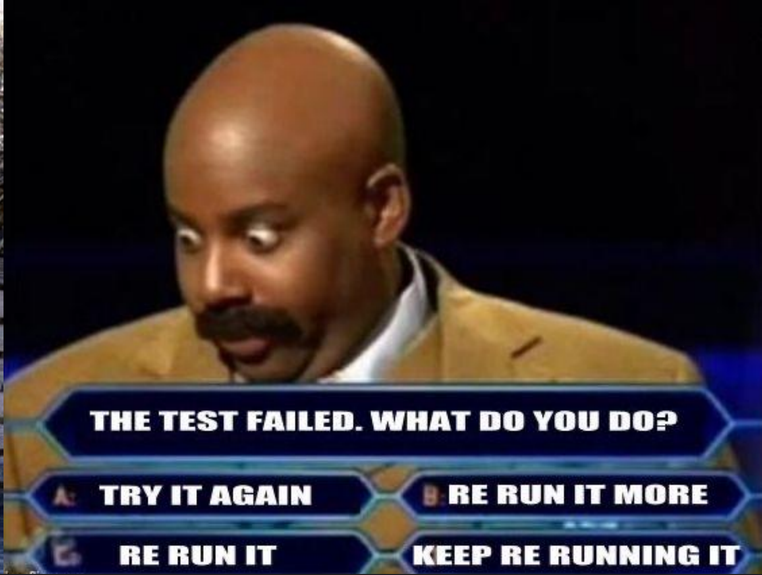
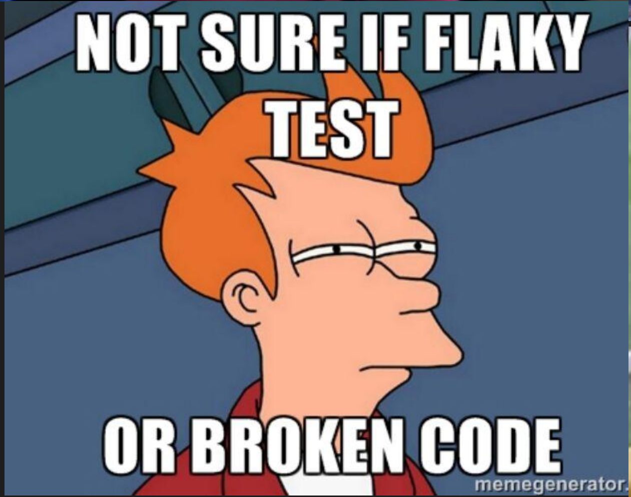
Hands on with CRUX

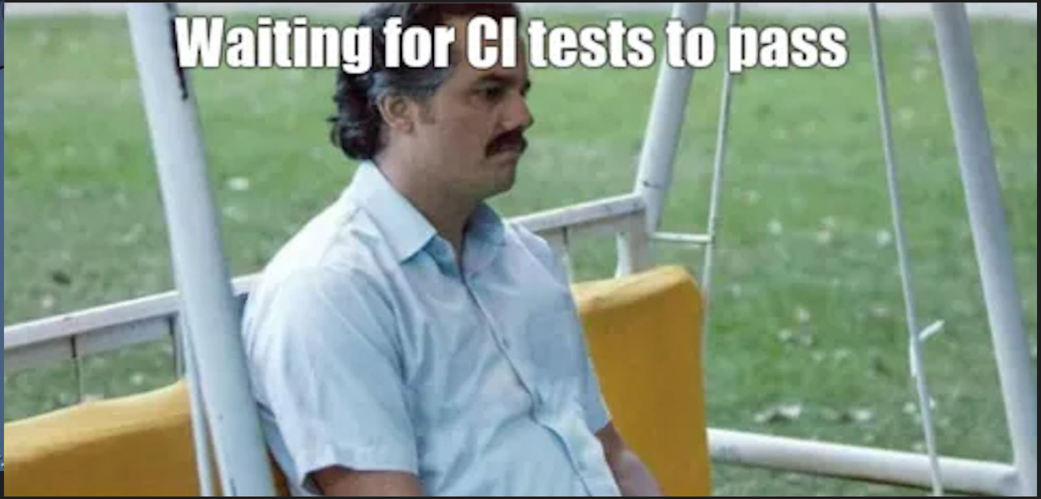
How behaviour-focused architecture enables UI testing you can love


BUT YOU WERE PASSING YESTERDAY!
imgflip.com

JUST WAITING
FOR CI TO FINISH RUNNING...
imgflip.com


THE TEST FAILED. WHAT DO YOU DO?
A: TRY IT AGAIN
B: RE RUN IT MORE
C: RE RUN IT
KEEP RE RUNNING IT


NOT SURE IF FLAKY TEST
OR BROKEN CODE
memegenerator.


Waiting for CI tests to pass

Viktor Charypar

Tech Director at Red Badger

@charypar on GitHub

@charypar on Twi… I mean 𝕏

RED BADGER

Building an app: a case study

- 2x platforms
- React Native + TypeScript
- About 30 engineers across 6 teams
- About 10 automation testers
- Several hours of E2E tests, pretty flaky
- Slows teams down, reduces confidence in testing, sucks all joy out of app dev

There must be a better way…?

What would we like?

- Minimise need for testing: pick a safer language, Rust
- Build and test most of the app once

  (but with native look and feel)
- Make testing much faster: ?

CRUX

Headless app development in Rust.

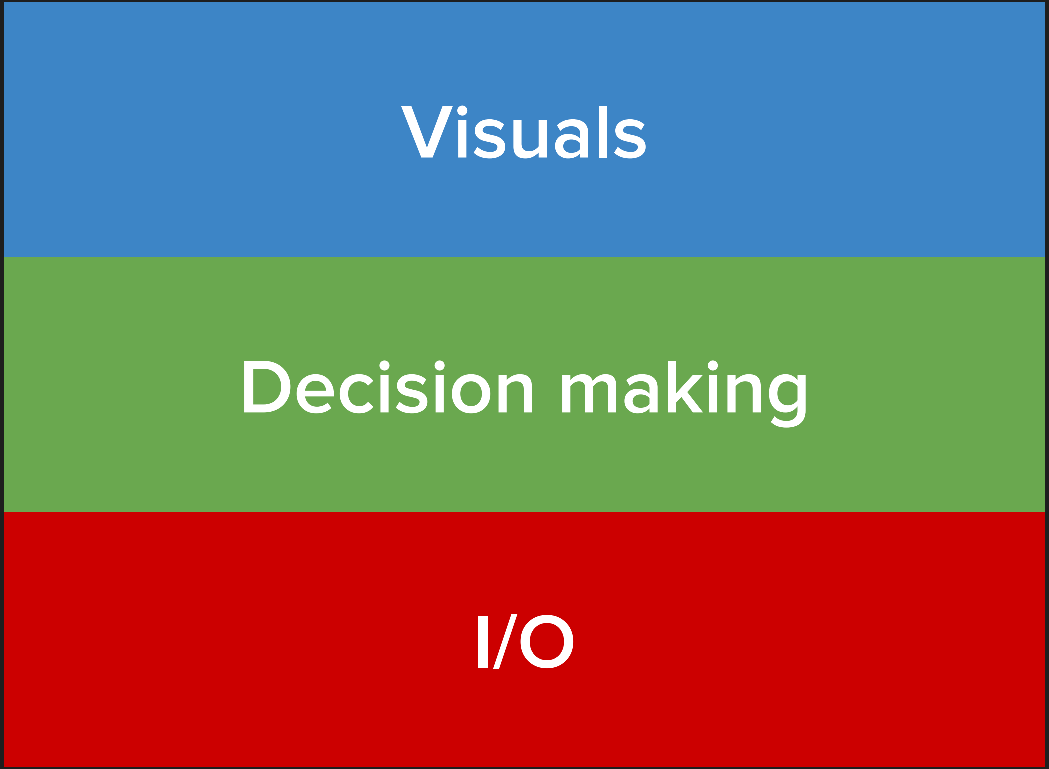
- Uses Rust for quality and performance
- High reuse of behaviour logic across platforms (iOS, Android, Web, …)
- End to end testing that runs reliably, in milliseconds

We will cover

- Why is testing GUIs difficult?
- A better approach to build more testable apps
- How Crux works
- Code walkthrough of an example Crux app
- What’s new in Crux

Why is testing UI slow and flaky?

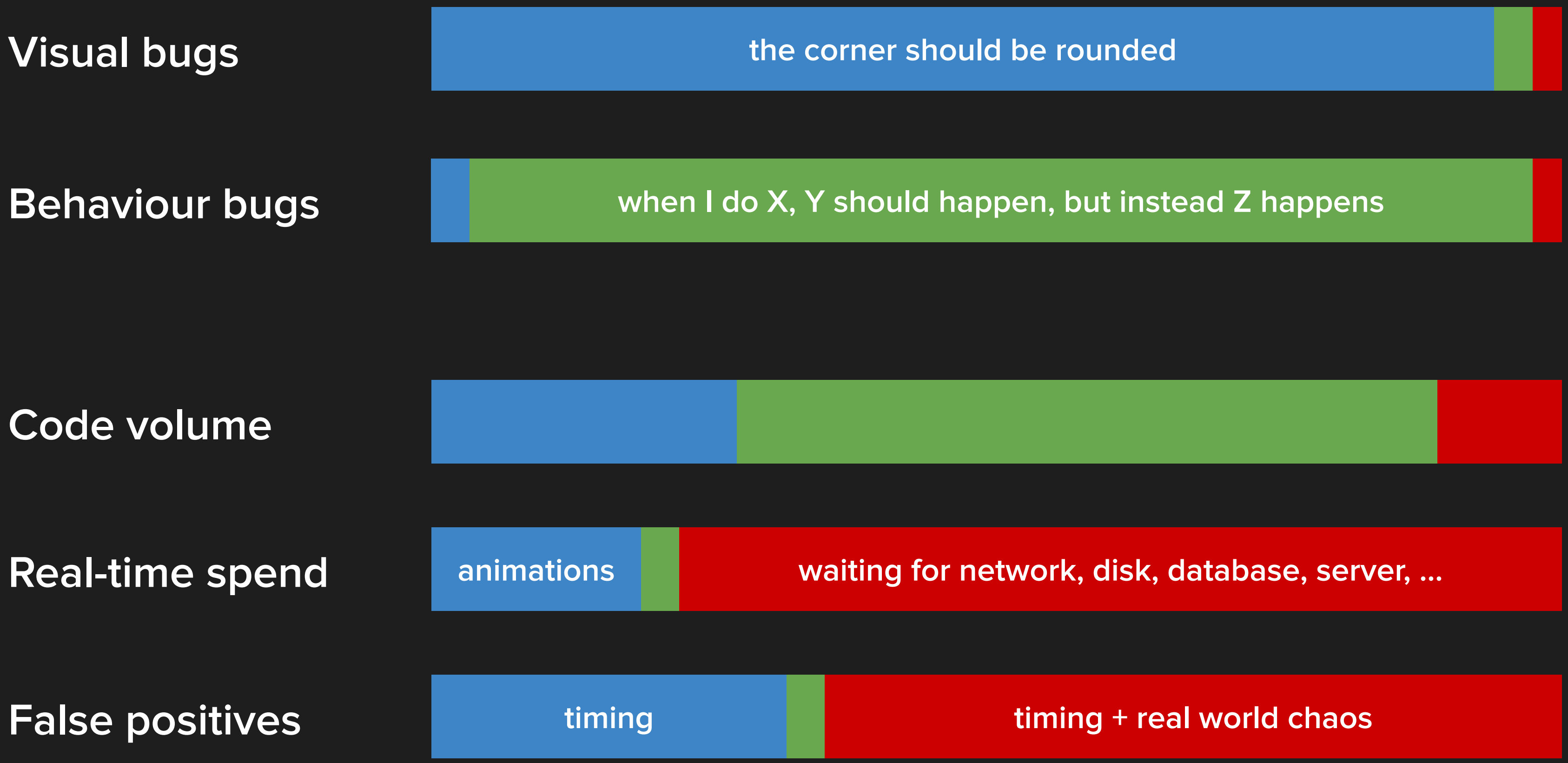
Testing overhead



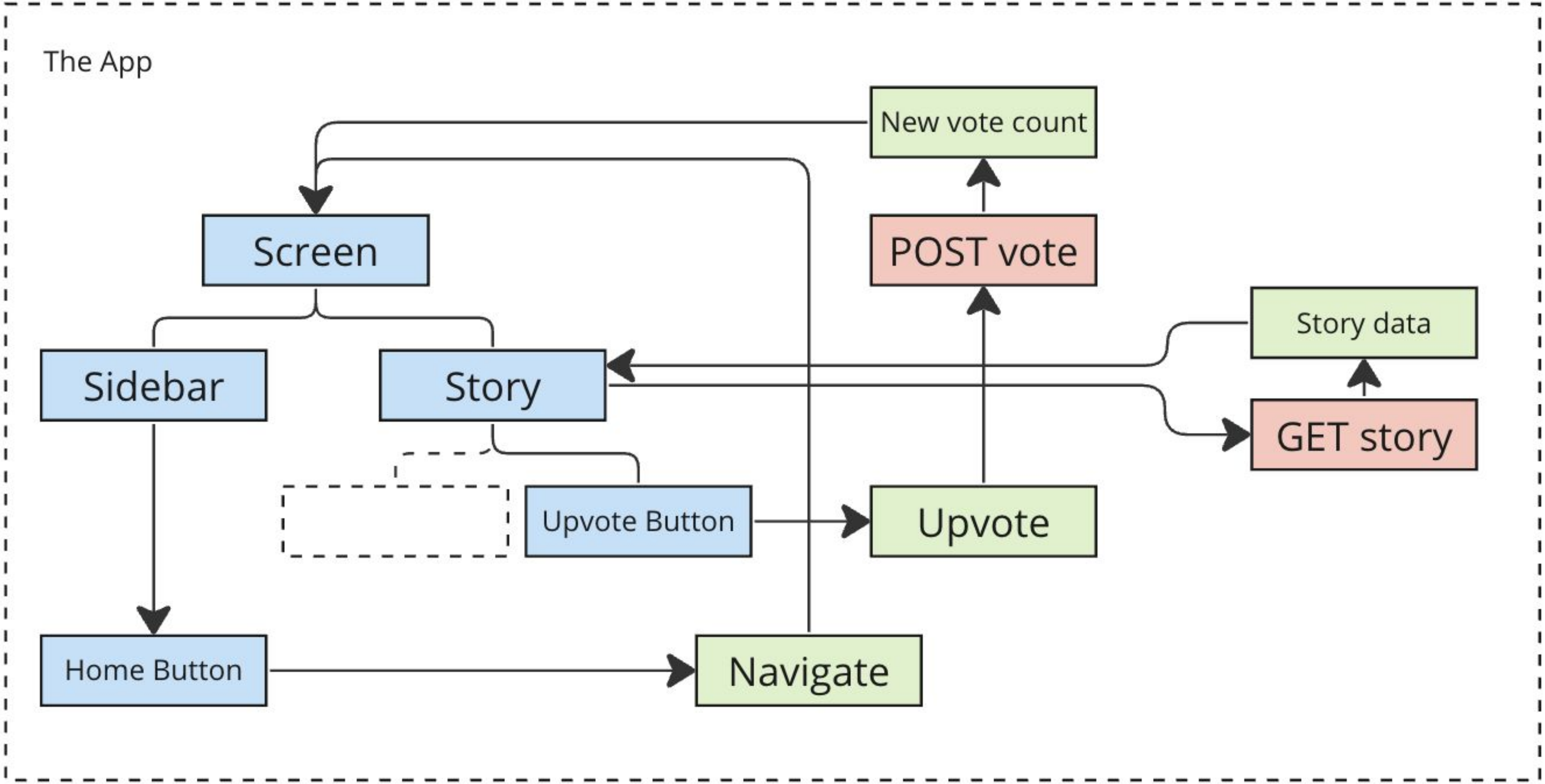
Visuals

Decision making

I/O


Visual bugs
the corner should be rounded
Behaviour bugs
when I do X, Y should happen, but instead Z happens
Code volume
Real-time spend
animations
waiting for network, disk, database, server, ...
False positives
timing
timing + real world chaos

So why do we do it?!


The App
New vote count
Screen
POST vote
Story data
Sidebar
Story
GET story
Upvote Button
Upvote
Home Button
Navigate

Two problems

- UI-centric - the UI is the organising principle of the code
    - We basically need the UI to run the code
- We layer and mix pure code with dirty* code
    - In tests we then need to swap out the dirty code

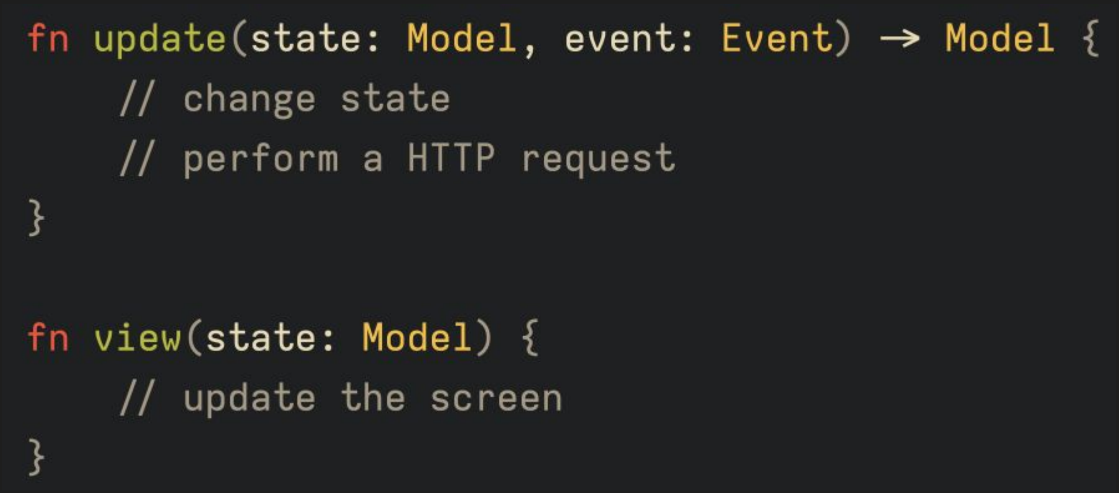
* Code with side-effects - especially I/O, date & time, randomness...

Start with the behaviour

Modeling a GUI in code

- **Behaviour**: Interaction leads to state update and some I/O

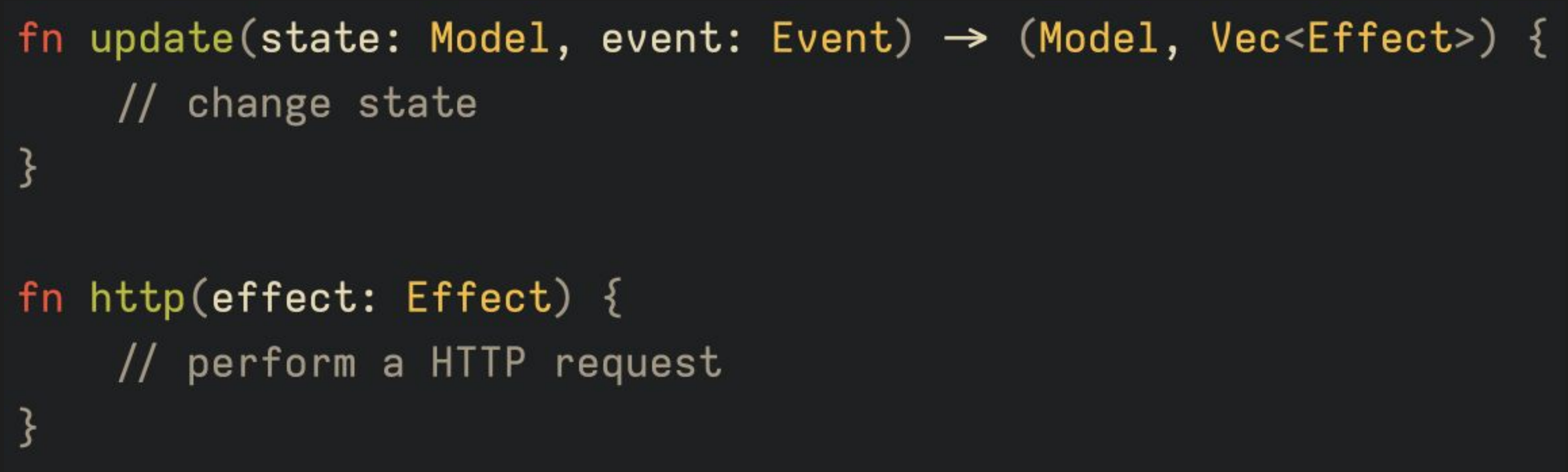
- **User interface**: New state is drawn on screen



```
fn update(state: Model, event: Event) -> Model {
    // change state
    // perform a HTTP request
}

fn view(state: Model) {
    // update the screen
}
```

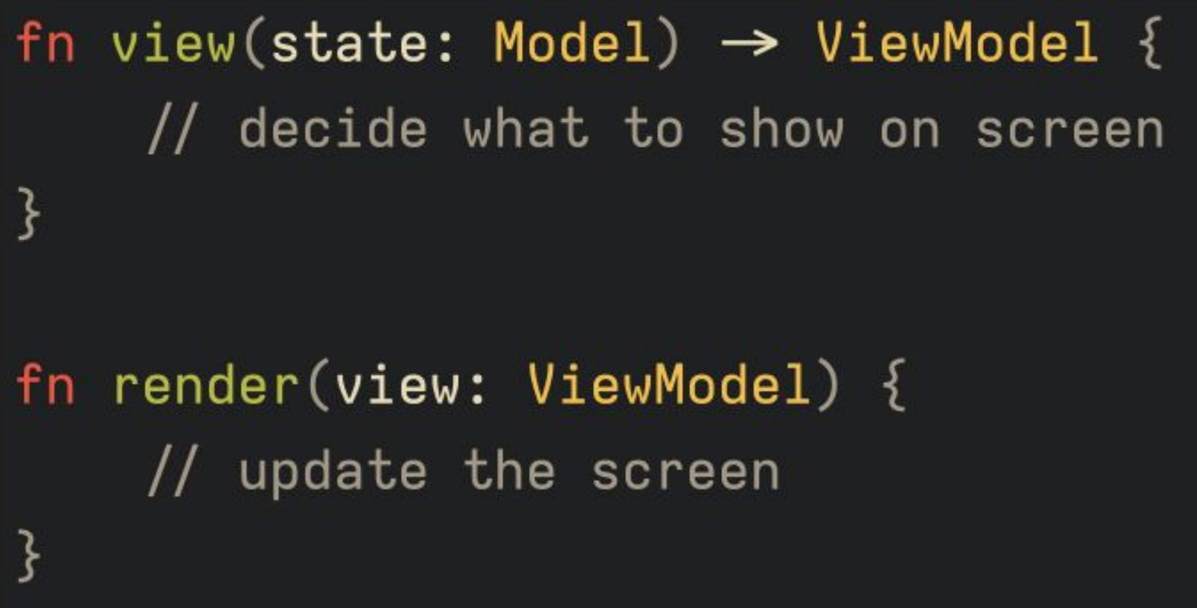
Separate pure and dirty code

Behaviour: pure and dirty

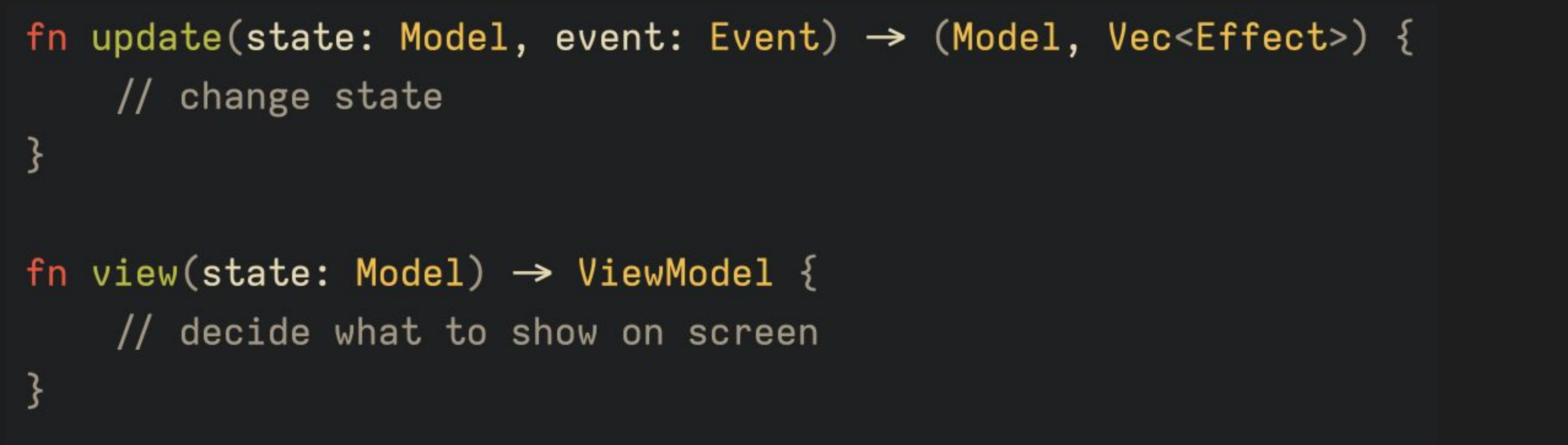


```rust
fn update(state: Model, event: Event) -> (Model, Vec<Effect>) {
    // change state
}

fn http(effect: Effect) {
    // perform a HTTP request
}
```

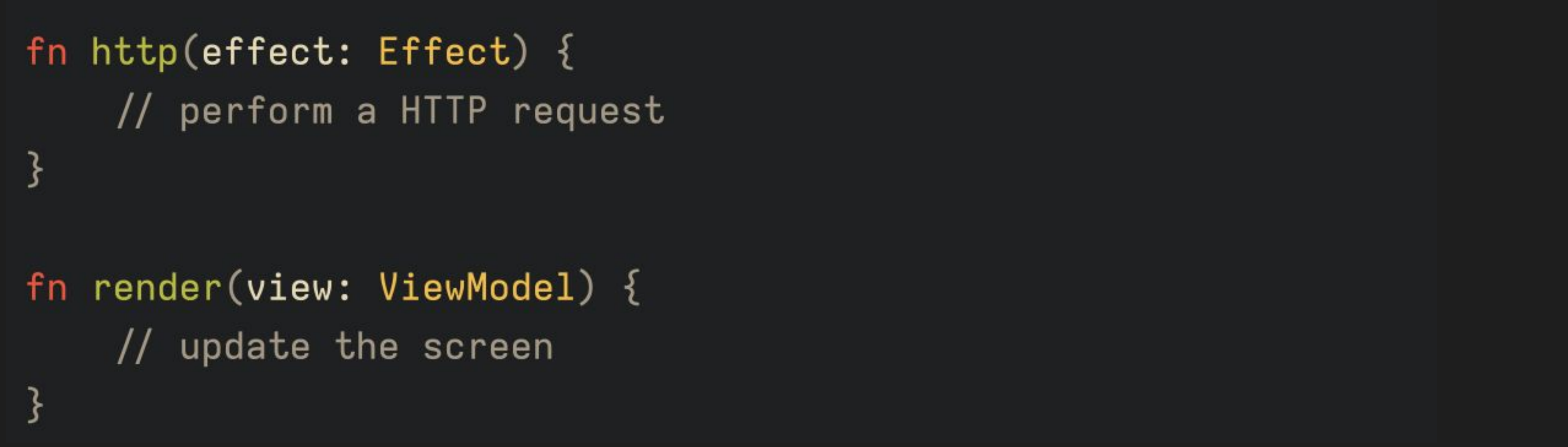
User Interface: pure and dirty



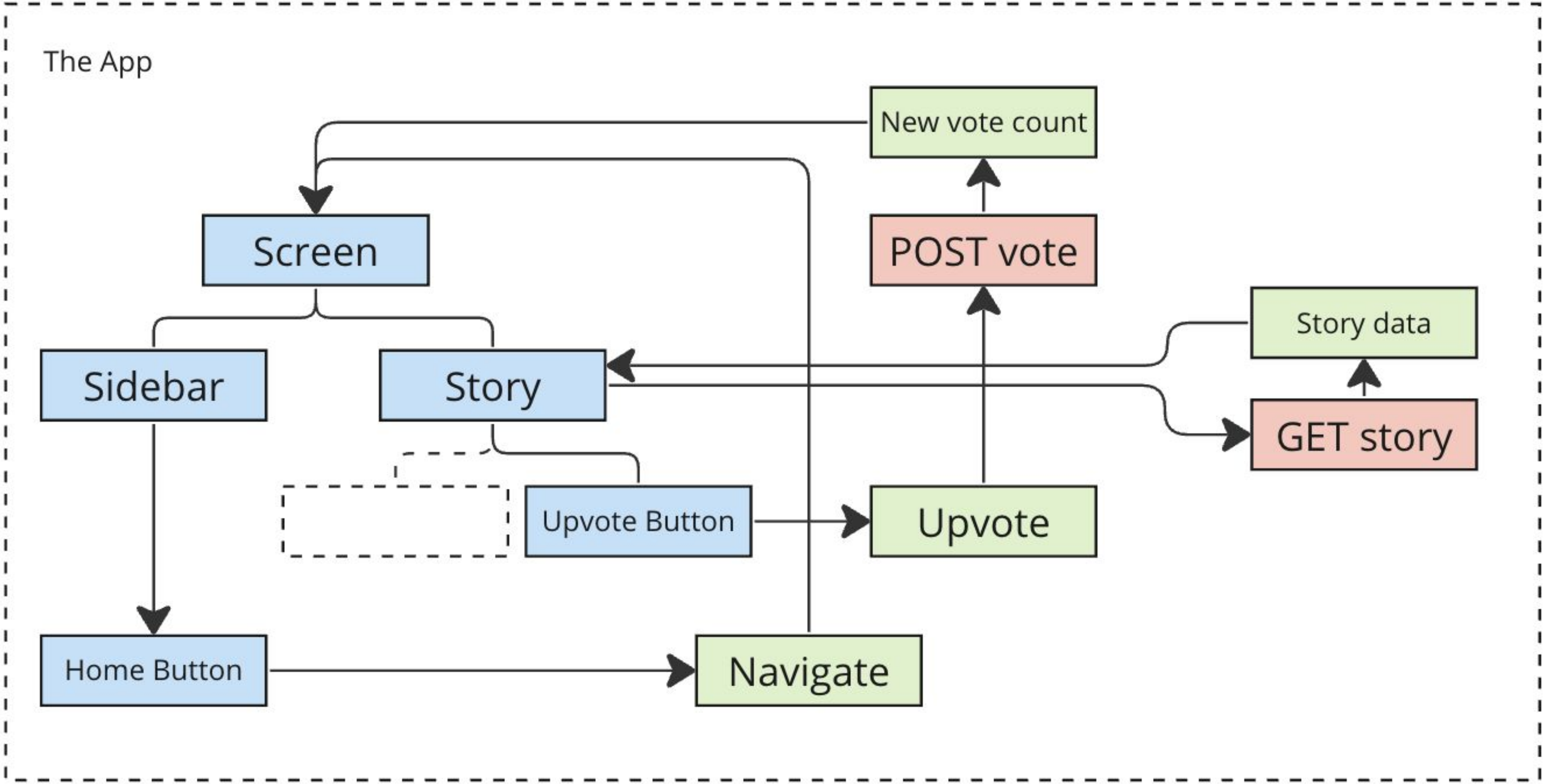
```
fn view(state: Model) -> ViewModel {
    // decide what to show on screen
}

fn render(view: ViewModel) {
    // update the screen
}
```

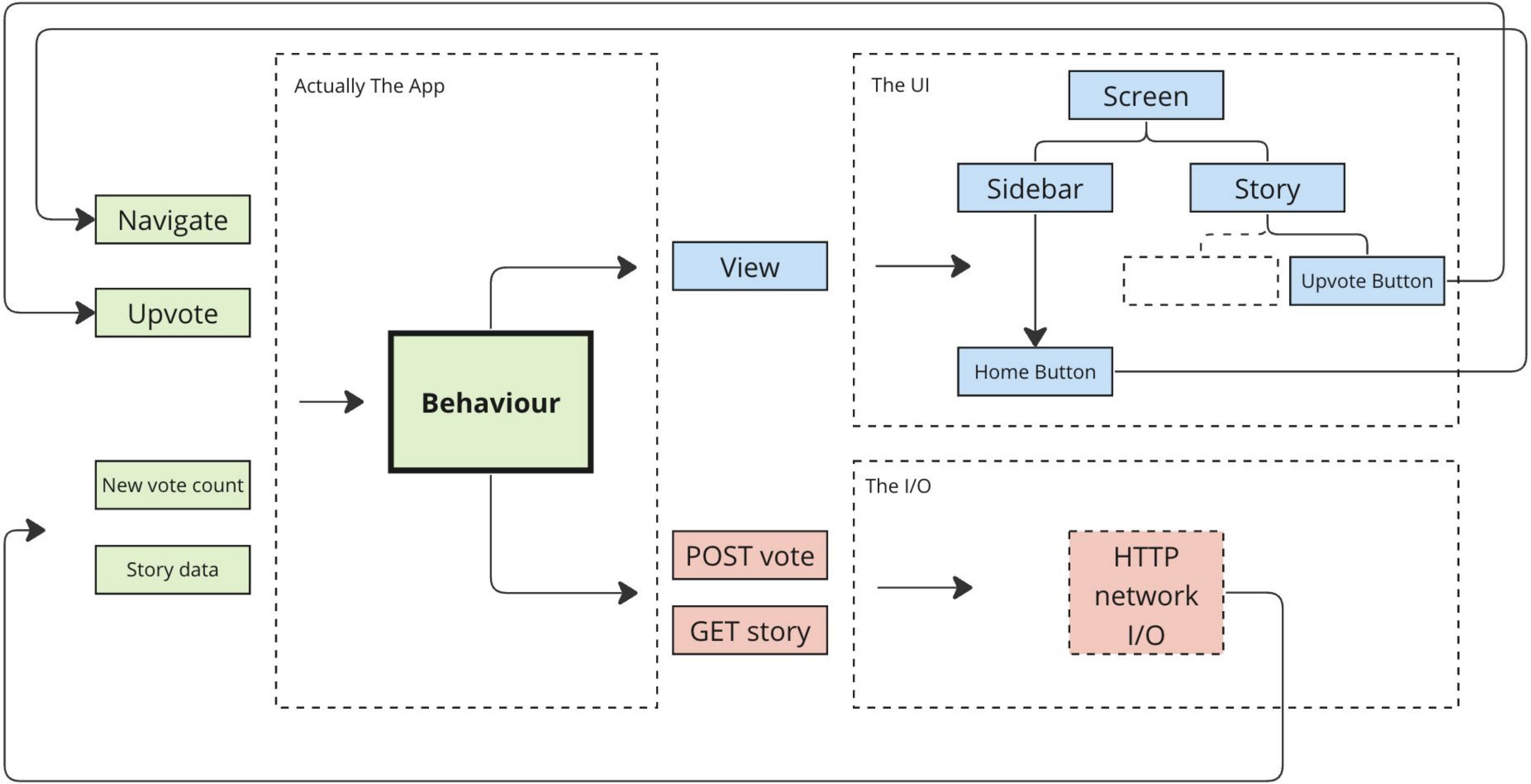


```
fn update(state: Model, event: Event) -> (Model, Vec<Effect>) {
    // change state
}

fn view(state: Model) -> ViewModel {
    // decide what to show on screen
}
```



```
fn http(effect: Effect) {
    // perform a HTTP request
}

fn render(view: ViewModel) {
    // update the screen
}
```


The App
New vote count
Screen
POST vote
Story data
Sidebar
Story
GET story
Upvote Button
Upvote
Home Button
Navigate


Actually The App
Behaviour
Navigate
Upvote
New vote count
Story data
View
POST vote
GET story
The UI
Screen
Sidebar
Story
Upvote Button
Home Button
The I/O
HTTP network I/O

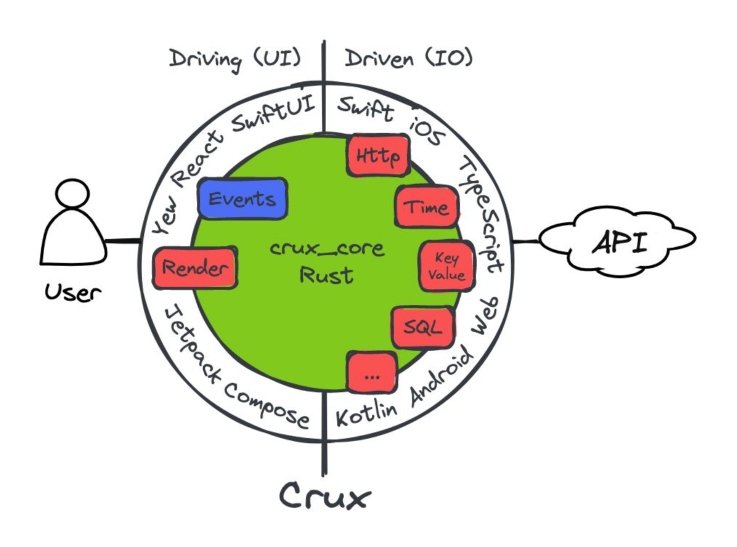
The app is now

- Headless
  Can run and be tested without the UI
- Sans I/O
  Can run and be tested with
  different I/O implementations

AKA: Hexagonal architecture, ports and adapters, other names…

CRUX

- Practical implementation of this
- Rust core (1x)
  and platform shells (Nx)
- Core is pure, shell performs
  effects and draws user interface
- Communicate over a small FFI
  with message passing
- Shell is always driving execution

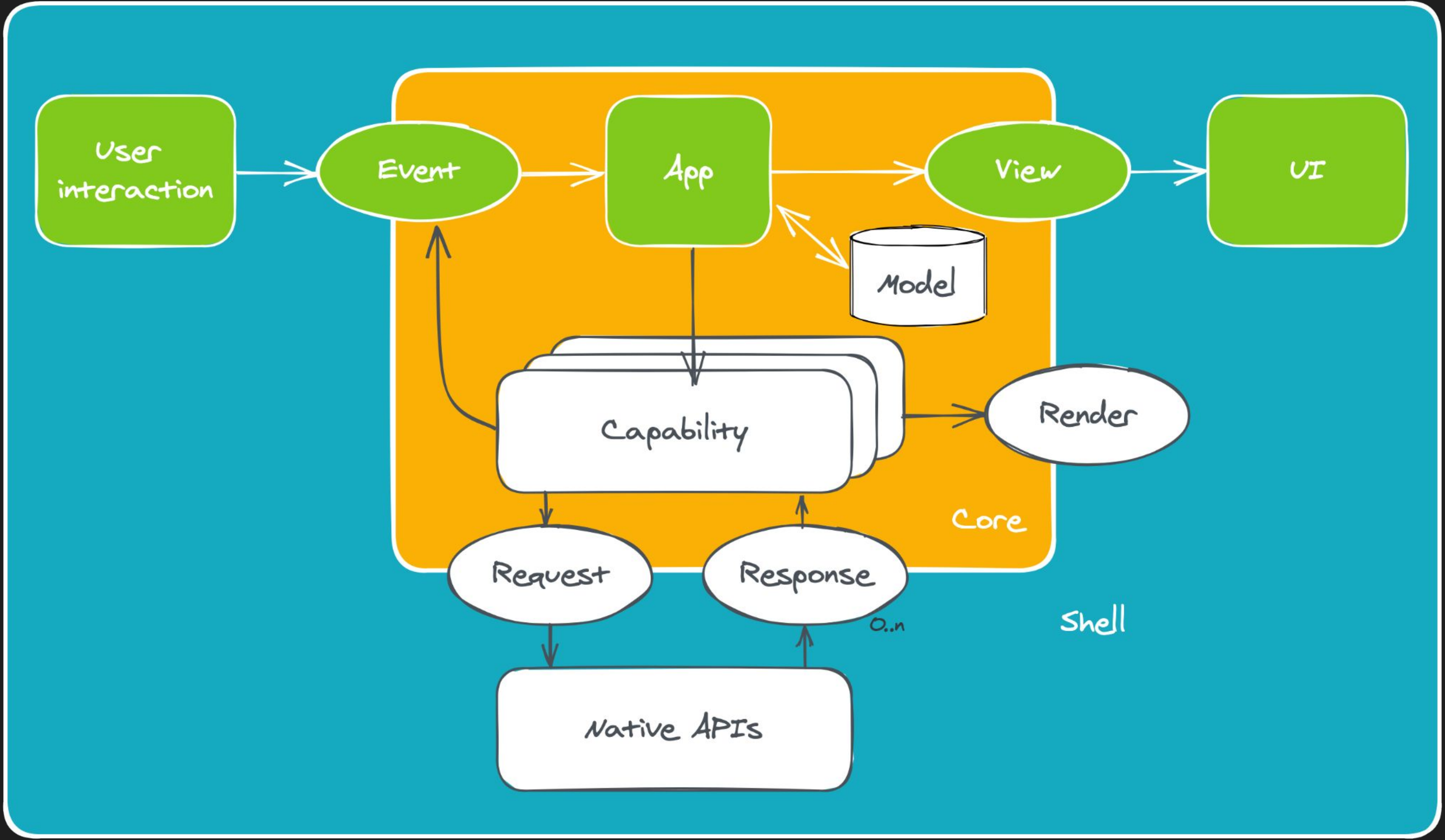


CRUX

Capabilities specify interfaces to the shell

- Notify (no response)
- Request (one response)
- Stream (many responses)

Internally access to async code


User interaction
Event
App
View
UI
Model
Capability
Render
Core
Request
Response
0..n
Shell
Native APIs

Let’s see some code!

github: charypar/rust-nation-2024-egg-timer

Key ideas to steal

- Headless :

  Build apps from inside-out, behaviour first

- Sans I/O:

  split intent and execution of side-effects

- Data oriented interfaces

New since last year

- Use from Rust without FFI
- Allow custom serialisation (no codegen)
- Capability orchestration

- Next: improve code generation

Thank you!

github: redbadger/crux

crux-community.zulipchat.com