Building iOS, Android and Web Apps that share a single Rust Core

2023-09-06

Stuart Harris

Founder & Chief Scientist

Red Badger

RUST GLOBAL

WASMCON

BETTER TOGETHER

RED BADGER

1. **What** is the problem with multi-platform app development today?
2. **Rust**, **WebAssembly**, and **Ports and Adapters**
3. **Crux** — experimental, open source tooling for building **headless** apps

Stu

- Software engineer
- Founder of Red Badger

RED BADGER

1. **What** is the problem with multi-platform app development today?
2. **Rust**, **WebAssembly**, and **Ports and Adapters**
3. **Crux** — experimental, open source tooling for building **headless** apps

1

The problem

❶ Tooling and Architecture

It's too hard to "build quality in"

- Historically, a lack of good **tooling** has lead to a poor developer experience for multi-platform app development
- Bad **architectures** make applications **hard to test** and maintain

① Building a multi-platform app (don’t @ me!)

	Platform Native	Kotlin MM	React Native	Capacitor Ionic	Flutter
Native UX	✅	✅	😐	❌	❌
Web?	❌	😐	😐	✅	✅
Development	😐	✅	😐	✅	✅
Testing	😐	😐	🤯	🤯	😐
Maintenance	😐	✅	😡	😡	✅
Effort	3x	2x	2x	1.5x	1.4x

1 UI-centric architecture

- UI layout is the **primary** organising principle
- Behaviour and interaction with the outside world are **secondary**

1 UI-centric architecture

- UI layout is the **primary** organising principle
- Behaviour and interaction with the outside world are **secondary**

> “It looks like this… and does that”

1 **What** is the problem with multi-platform app development today?

2 **Rust**, **WebAssembly**, and **Ports and Adapters**

3 **Crux** — experimental, open source tooling for building **headless** apps

2

The solution

2 Better Tools and Better Architecture

- **Rust** is a **revolution**
    - everyone can now build reliable, high quality software in almost any space — perfect for multi-platform app development
- **WebAssembly** is a **revolution**
    - fast and portable — great for building apps in the languages we love
- **Ports and Adapters** is a **revolution**
    - portable and easy to test — easy to build high quality apps

❷ What if we start with behaviour?

We build a **core** that encapsulates our app's behaviour:

- updates a **model** in response to **events**
- emits **effects** — intent to perform side-effects
- is **pure** (can be easily tested)

② Behaviour

A pure update function (cf. Elm, Redux, etc.)

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

② Behaviour with side-effects

A pure update function (cf. Elm, Redux, etc.)

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

A dirty function with side-effects

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

❷ UI

Imagine the UI as a projection of state (cf. early React)

```rust
fn view(state: Model) {/*update UI*/}
```

❷ UI with platform independence

Imagine the UI as a projection of state (cf. early React)

```
fn view(state: Model) -> ViewModel
```

Introduce a view model

```
fn render(view: ViewModel) {/*update UI*/}
```

② Before

Behavior

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

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

UI

```
fn view(state: Model) -> ViewModel
```

```
fn render(view: ViewModel) {/*update UI*/}
```

② After

Core

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

```
fn view(state: Model) -> ViewModel
```

Shell

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

```
fn render(view: ViewModel) {/*update UI*/}
```

2 Behaviour-centric architecture

- Behaviour is the **primary** organising principle
- Interaction with the outside world is **secondary**
- UI is also a side-effect

② Behaviour-centric architecture

- Behaviour is the **primary** organising principle
- Interaction with the outside world is **secondary**
- UI is also a side-effect

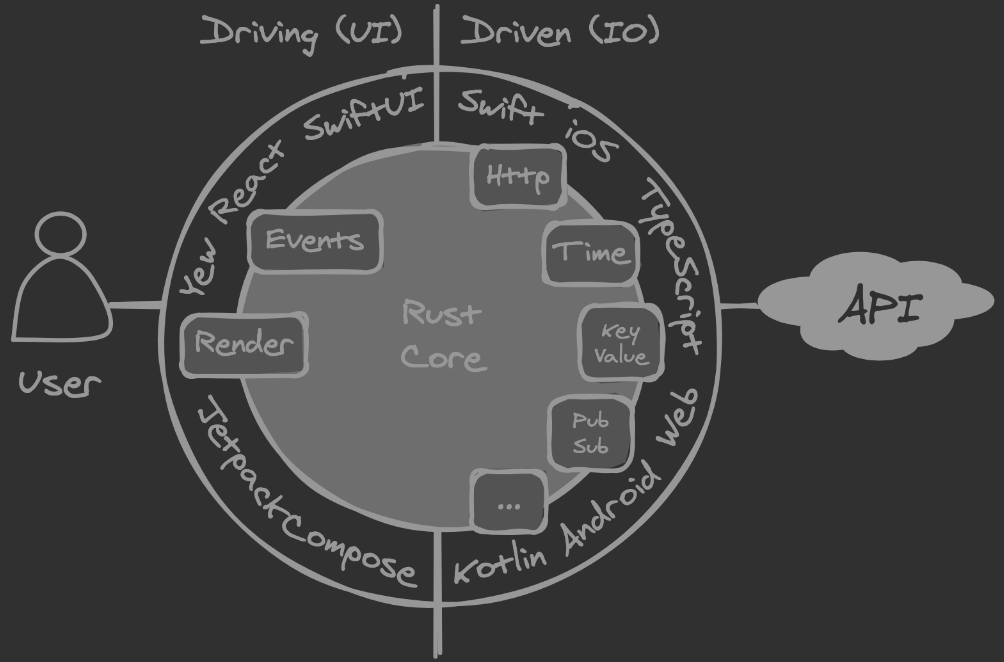
> “It does this… and looks like that!”

❷ Ports and adapters

> Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

Alistair Cockburn, 2005

Hexagonal Architecture



② Ports and adapters

> The application can be deployed in "headless" mode, so only the API is available, and other programs can make use of its functionality

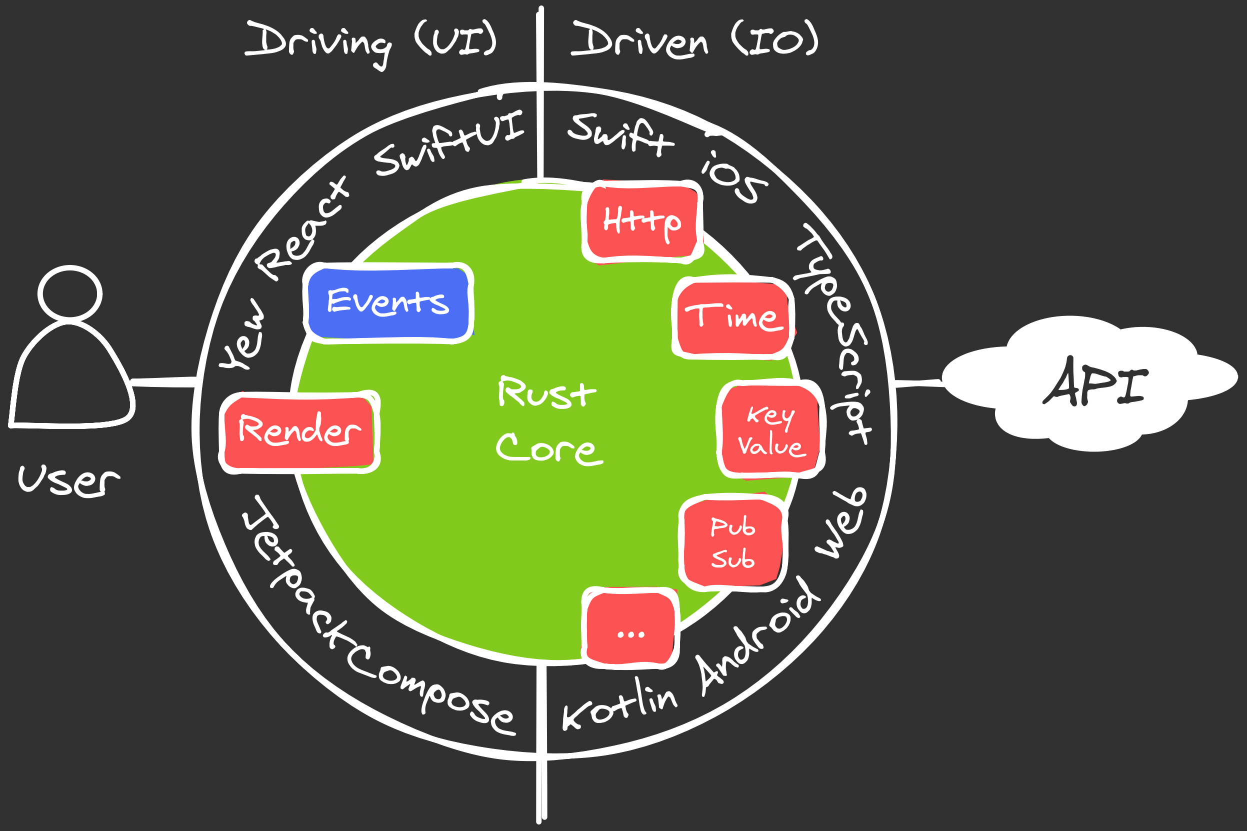
Alistair Cockburn, 2005

Hexagonal Architecture

1 **What** is the problem with multi-platform app development today?

2 **Rust**, **WebAssembly**, and **Ports and Adapters**

3 **Crux** — experimental, open source tooling for building **headless** apps



3

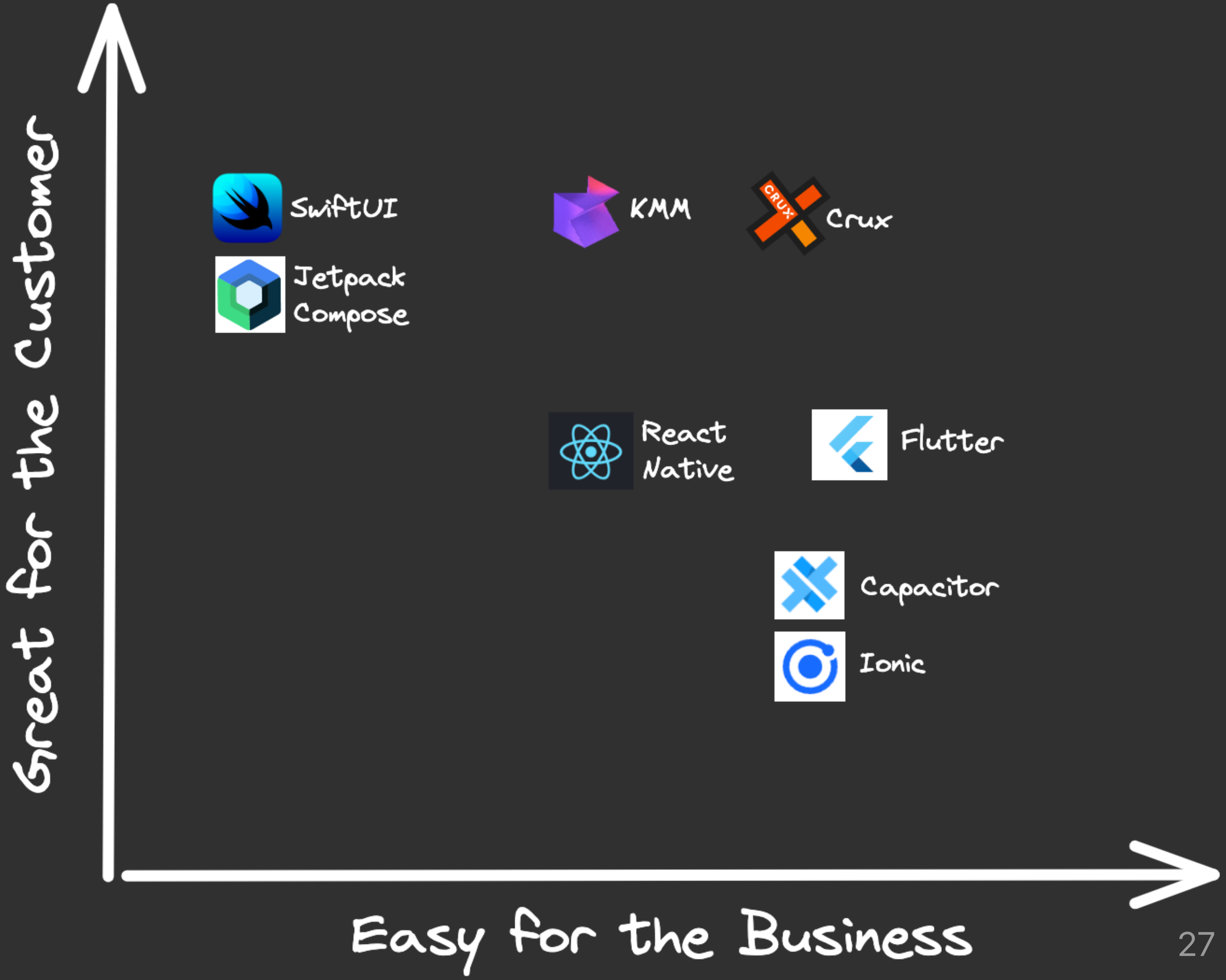
CRUX

- Shared **behaviour**
- Capabilities
- Platform **native** UX
- **Rust** yay! 🦀

❸ Building a multi-platform app (don’t @ me!)

	Platform Native	Kotlin MM	React Native	Capacitor Ionic	Flutter	Crux
Native UX	✅	✅	😐	❌	❌	✅
Web?	❌	😐	😐	✅	✅	✅
Development	😐	✅	😐	✅	✅	✅
Testing	😐	😐	🤯	🤯	😐	🤩
Maintenance	😐	✅	😡	😡	✅	✅
Effort	3x	2x	2x	1.5x	1.4x	1.4x

3 Who benefits?

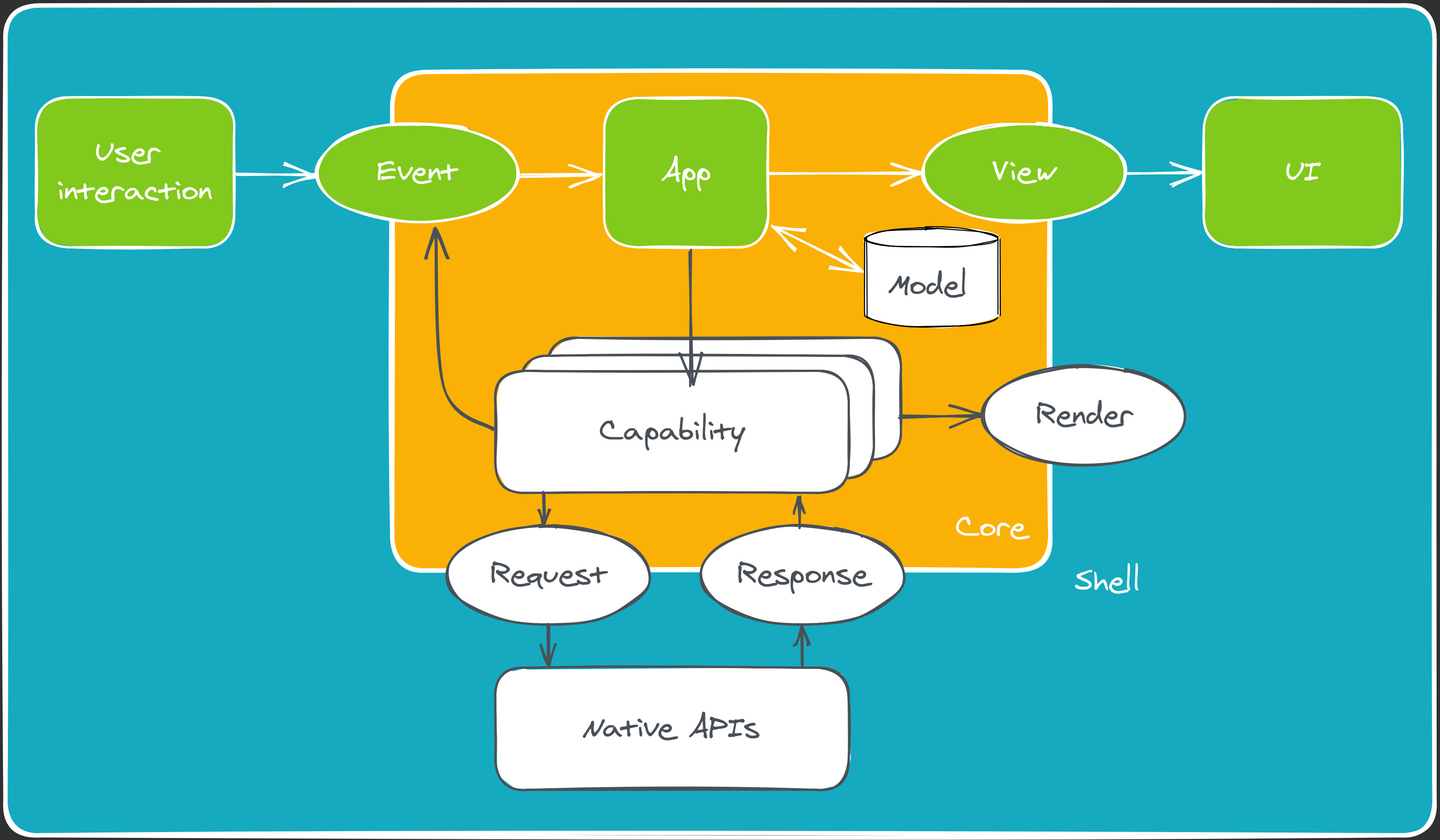


③ Any client

platform	language	UI	library	lib name	FFI
iOS	Swift	SwiftUI	static	`libshared.a`	`uniffi-bindgen`
Android	Kotlin	Compose	dynamic	`libshared.so`	`uniffi-bindgen`
Web	TypeScript	Remix	wasm	`shared.wasm`	`wasm-bindgen`
Web	Rust	Leptos	crate		
CLI	Rust		crate		

Type generation with `serde-generate`

3


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

3 Capabilities

Fire and forget

```
caps.render.render();
```

Request/response

```
caps.http.get(API_URL).expect_json().send(Event::Set);
```

Streaming

```
caps.sse.get_json(API_URL, Event::Update);
```

3 Capabilities

- Built-in (`Render`)
- `crux_*` crates (`Http`, `KeyValue`, `Platform`, `Time`)
- Custom
    - `ServerSentEvents` in the Counter example
    - `Delay` example in the book
    - `Timer` and `PubSub` in the Notes example
- Community contributed

3 What does a Crux app look like?

```
#[derive(Default)]
pub struct App;

impl crux_core::App for App {
    type Event = Event;
    type Model = Model;
    type ViewModel = ViewModel;
    type Capabilities = Capabilities;

    fn update(&self, event: Self::Event, model: &mut Self::Model, caps: &Self::Capabilities) {
        match event {
            Event::Increment => model.count += 1,
            Event::Decrement => model.count -= 1,
            Event::Reset => model.count = 0,
        };

        caps.render.render();
    }

    fn view(&self, model: &Self::Model) -> Self::ViewModel {
        ViewModel {
            count: format!("Count is: {}", model.count),
        }
    }
}
```

3 What does a test look like?

```
#[cfg(test)]
mod test {
    use super::*;
    use crux_core::testing::AppTester;

    #[test]
    fn increments_count() {
        let app = AppTester::<Hello, _>::default();
        let mut model = Model::default();

        let update = app.update(Event::Increment, &mut model);

        // Check the app asked us to `Render`
        assert_effect!(update, Effect::Render(_));

        // Check view model is correct
        let actual_view = app.view(&model).count;
        let expected_view = "Count is: 1";
        assert_eq!(actual_view, expected_view);
    }
}
```

3 Demo

Headless app development in Rust

CRUX

3 The crux of Crux

- a lightweight runtime
  - for headless, multi-platform, composable apps with shared **behaviour**
  - for better **testability**
  - for higher **quality**
  - for better **reliability**, safety, and security
  - and more **joy** from better tools

Thank you!

- https://github.com/redbadger/crux
- https://redbadger.github.io/crux/
- https://red-badger.com/crux
- https://docs.rs/crux_core/latest/crux_core/
- https://www.youtube.com/watch?v=cWCZms92-1g&t=5s