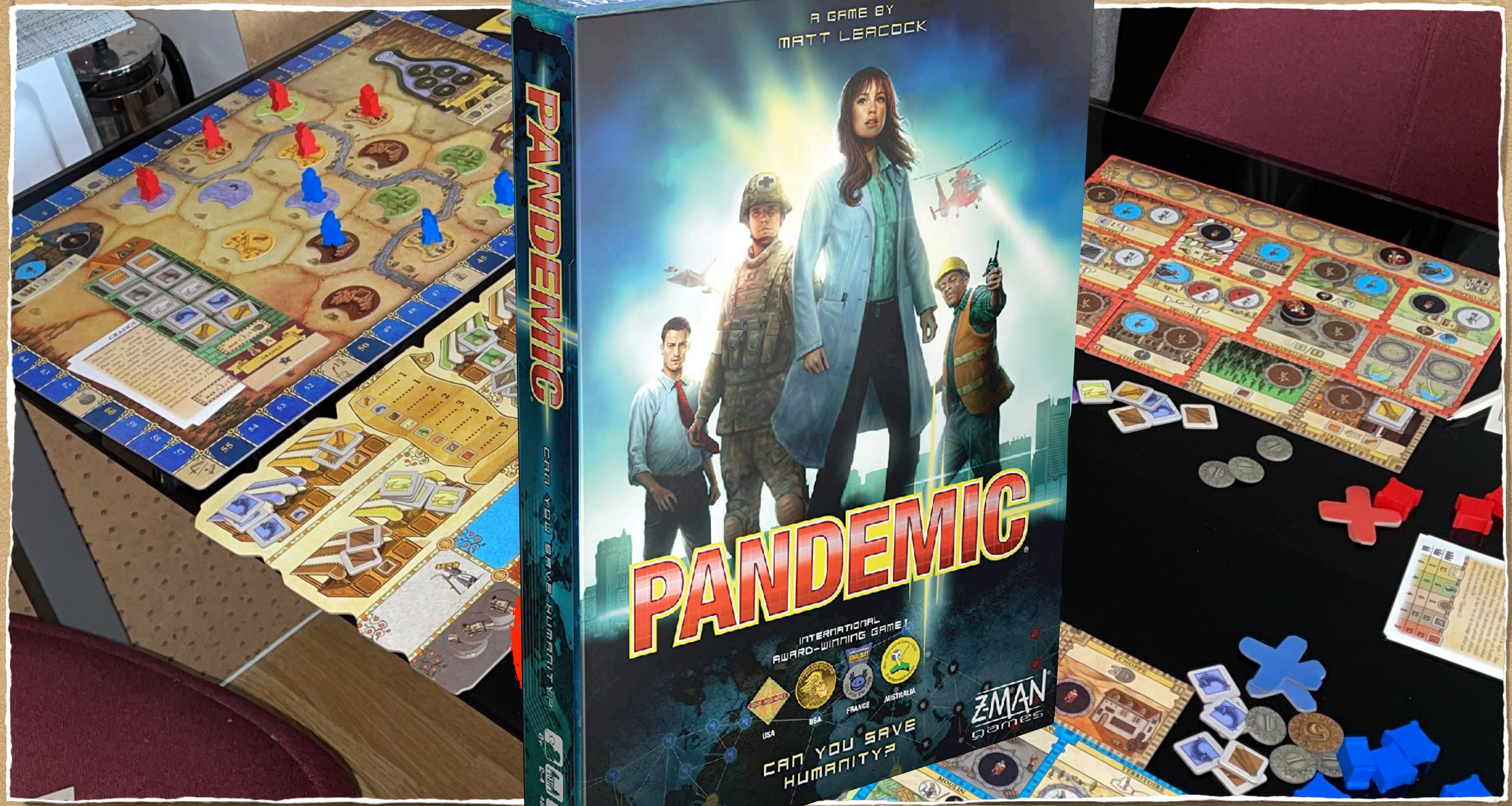
Playing board games with Scala.js


A GAME BY
MATT LEACOCK
PANDEMIC
INTERNATIONAL
AWARD-WINNING GAME!
USA
USA
FRANCE
AUSTRALIA
CAN YOU SAVE
HUMANITY?
Z-MAN
games
PANDEMIC
CAN YOU SAVE HUMANITY?

Replacing 7 wonders score sheets

A web app that feels like
a native mobile app

with a PWA, implemented using Scala 3,

Scala.js, and scalajs-react

A compiler that emits Javascript
from Scala code

A library to build UIs
with a declarative approach
and reusable components


21:36
5
Famileo
PhotoSync
Board games
14
22

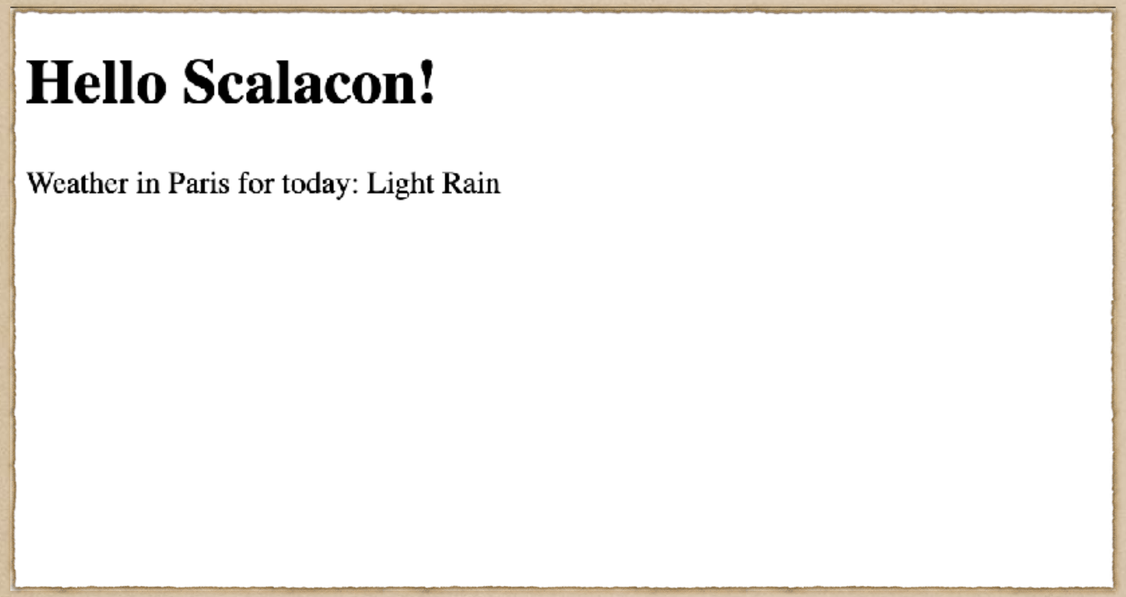
So, why Scala.js ?

- You can't over-sell a great type system
- TypeScript is unsound by design
- Javascript lacks a decent standard library
- I already know and love Scala

Getting started is easy

```
// In project/plugins.sbt
addSbtPlugin("org.scala-js" % "sbt-scalajs" % “1.7.0")

// In build.sbt
lazy val weatherApp = project
  .in(file("weather-app"))
  .settings(
    libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "2.0.0",
    scalaJSUseMainModuleInitializer := true
   )
  .enablePlugins(ScalaJSPlugin)
```

Let's code something simple



- Write "Hello Scalacon!" in an h1 tag and append it to the document's body
- Fetch the weather from an external API
- Write the weather in a paragraph tag and append it in the document's body

We start by defining facades

```
@js.native
trait WeatherItem extends js.Object:
  @JSName("weather_state_name")
  val name: String = js.native

@js.native
trait Response extends js.Object:
  @JSName("consolidated_weather")
  val items: js.Array[WeatherItem] = js.native
```

These let us manipulate Javascript through Scala's type system.
They have no runtime cost.

Then the rest is very much like good ol' Javascript

```
def greetEveryone(): Unit =
  val titleNode = dom.document.createElement("h1")
  titleNode.textContent = "Hello Scalacon!"
  dom.document.body.appendChild(titleNode)

def fetchWeather() =
  val url = "https://www.metaweather.com/api/location/615702/"
  dom.fetch(url).`then`(_.json().asInstanceOf[Response])

def printWeather(weather: Response) =
  val weatherNode = dom.document.createElement("p")
  weatherNode.textContent = s"Weather in Paris for today: ${weather.items.head.name}"
  dom.document.body.appendChild(weatherNode)

def main() =
  greetEveryone()
  fetchWeather().`then`(printWeather)
```

Then the rest is very much like good ol' Javascript

```
def main() =
  greetEveryone()
  fetchWeather()
    .`then`(printWeather)
```

1. 2. 3. …

This code is heavily imperative, like Javascript

```scala
def greetEveryone(): Unit =
  val titleNode = dom.document.createElement("h1")
  titleNode.textContent = "Hello Scalacon!"
  dom.document.body.appendChild(titleNode)
```

It uses mutable properties, like Javascript

```scala
def printWeather(weather: Response) =
  val weatherNode = dom.document.createElement("p")
  weatherNode.textContent =
    s"Weather in Paris for today: ${weather.items.head.name}"
  dom.document.body.appendChild(weatherNode)
```

It even has an unsafe access to a list's head

Scala has more to offer than
"just" Javascript with types

Let's take a look at

japgolly/scalajs-react

Scalajs-react provides more than static types for the DOM …

```
val header =
  <.
```

```
h1: japgolly.scalajs.react.vdom.Exports.Htm…
h2: japgolly.scalajs.react.vdom.Exports.Htm…
h3: japgolly.scalajs.react.vdom.Exports.Htm…
h4: japgolly.scalajs.react.vdom.Exports.Htm…
h5: japgolly.scalajs.react.vdom.Exports.Htm…
h6: japgolly.scalajs.react.vdom.Exports.Htm…
head: japgolly.scalajs.react.vdom.Exports.H…
header: japgolly.scalajs.react.vdom.Exports…
hr: HtmlTagOf[HR]
html: japgolly.scalajs.react.vdom.Exports.H…
hashCode: Int
```

```
: japgolly.scalajs.react.vdom.Expo ×
rts.HtmlTagOf.Tag[Heading]

Heading level 1
```

In scalajs-react, UI elements are data,

not statements

```
def greetEveryone(): Unit =
  val titleNode = dom.document.createElement("h1")
  titleNode.textContent = "Hello Scalacon!"
  dom.document.body.appendChild(titleNode)
```

```
val greetings: VdomNode =
  <.h1("Hello Scalacon!")
```

Hence, you can use pure functions to map data to UI elements (also data)

```
def printWeather(weather: Response) =
  val weatherNode = dom.document.createElement("p")
  weatherNode.textContent = s"Weather in Paris for today: ${weather.items.head.name}"
  dom.document.body.appendChild(weatherNode)
```

```
def renderWeather(weather: Response): VdomNode =
  weather.items.headOption.map(item ⇒
    <.p(s"Weather in Paris for today: ${item.name}")
  )
```

UI and behaviour are packed together in reusable components, whose lifecycles are also statically-typed

```scala
case class Props(locationId: Int, locationName: String)

// Our "view layer", a pure function
def renderWeather(props: Props, state: Option[Response]): VdomNode =
  state.flatMap(_.items.headOption).map(item ⇒
    <.p(s"Weather in ${props.locationName} for today: ${item.name}")
  )

// Our data fetching layer, wrapped in typed-effects
def fetchWeather(locationId: Int): AsyncCallback[Option[Response]] = ???

// A reusable component that can fetch and display the weather for any
// location, in any part of our app, simply by passing it a locationId
val Weather = ScalaComponent
  .builder[Props] // Props
  .initialState(Option.empty[Response])
  .render_PS(renderWeather)
  .componentDidMount($ ⇒
    fetchWeather($.props.locationId).flatMap($.setState(_).async)
  )
```

Virtual DOM
Reusable components
(and their lifecycles)
One-way data flow
Context API
Massive ecosystem
…

It's all of React, and all of Scala

Case classes
Algebraic data types (now with a new syntax)
Immutable data structures
Extension methods
Union types
Opaque types
Rich ecosystem …

Virtual DOM
Reusable components
(and their lifecycles)
One-way data flow
Context API
Massive ecosystem
…

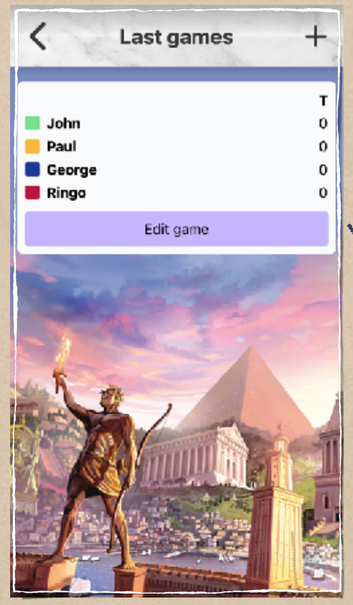
It's all of React, and all of Scala
and even a bit more

Referentially-transparent callbacks
Cats integration
Automatic performance management using type classes
Included router
…

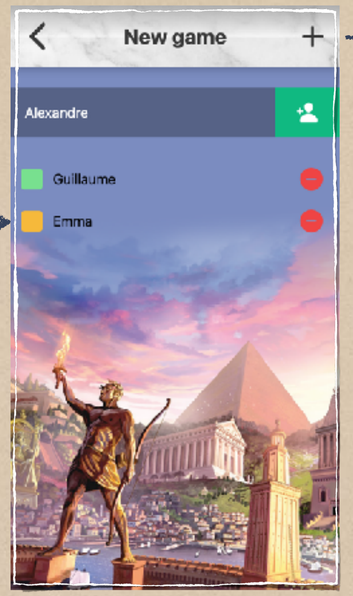
Case classes
Algebraic data types (now with a new syntax)
Immutable data structures
Extension methods
Union types
Opaque types
Rich ecosystem …

Let's dissect this PWA

/



/7wonders



/7wonders/new

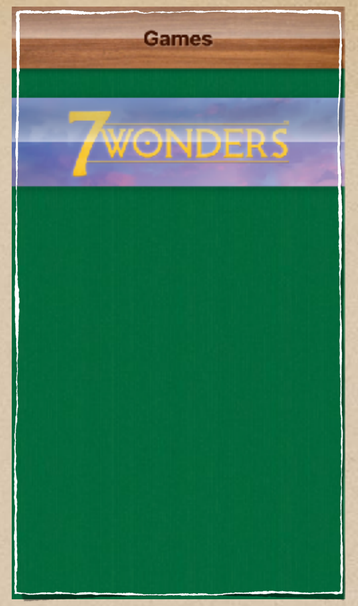
/7wonders/:gameId

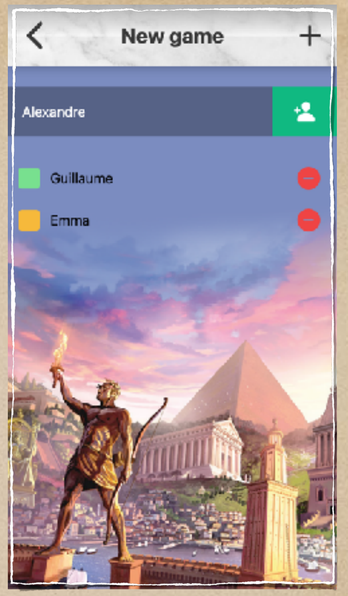
At this point we know the app will have 4 pages:

```
object Routes:
  opaque type Route = String
```

Additional type-safety
at no runtime cost!

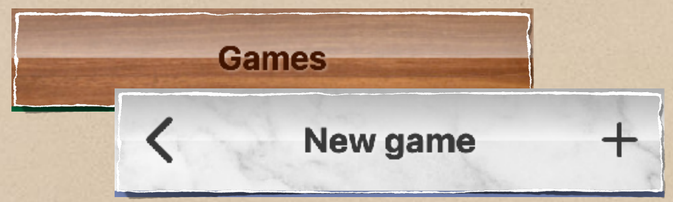
```
val home: Route = "/"
val sevenWondersLastGames: Route = "/7wonders"
val sevenWondersNewGame: Route = "/7wonders/new"
def sevenWondersGame(id: GameId): Route = s"/7wonders/$id"
```



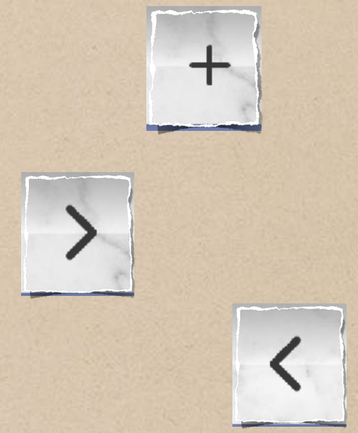


We can already identify reusable pieces

A page background
that comes in two variants:
“7 Wonders” and “green felt”



A navigation bar that also comes
in two variants…



… and holds exactly one
of three possible buttons on each side

Wait a minute!

It's a job for ADTs!

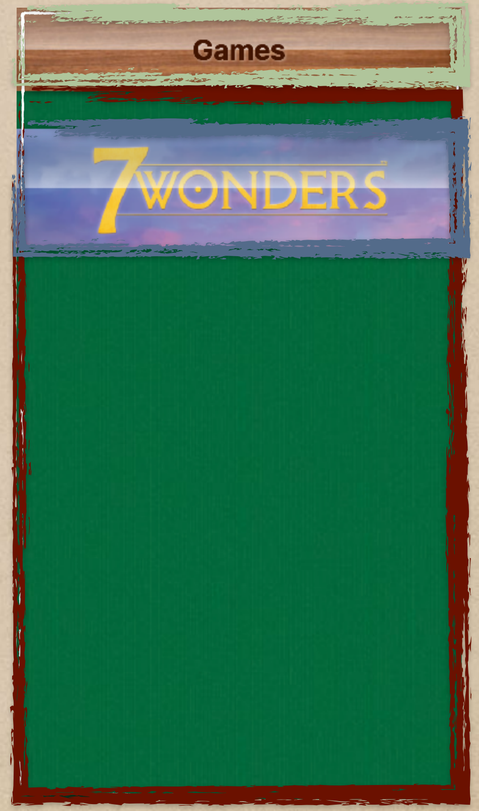
```
object Header:

  enum SideItem:
    case BackButton(action: Action)
    case NextButton(action: Action)
    case PlusButton(action: Action)

  type Action = Route | AsyncCallback[Unit] | Callback

  enum Style:
    case Wooden
    case Marble

  case class Props(
      title: String,
      rightSide: Option[SideItem] = None,
      leftSide: Option[SideItem] = None,
      style: Style = Style.Wooden
  )
```

```scala
object Style:
  protected val sheet: js.Dictionary[String] = js.native
  // Map every Style to a CSS class
  def apply(style: Style) = style match
    case Wooden ⇒ s"${sheet("header")} ${sheet("wooden")}"
    case Marble ⇒ s"${sheet("header")} ${sheet("marble")}"

// Map every action to a DOM element
private def renderAction(act: Action, node: VdomNode): VdomNode =
  act match
    case to: Route                ⇒ NextLink(to, node)
    case cb: Callback             ⇒ <.span(^.onClick ⟶ cb, node)
    case cb: AsyncCallback[Unit] ⇒ <.span(^.onClick ⟶ cb, node)

// Map every header item to a DOM element
private def renderSideItem(item: SideItem) =
  item match
    case SideItem.BackButton(action) ⇒
      renderAction(action, Icons.BackArrow(Icons.Props("3rem")))
    case SideItem.PlusButton(action) ⇒
      renderAction(action, Icons.Plus(Icons.Props("3rem")))
    case SideItem.NextButton(action) ⇒
      renderAction(action, Icons.ForwardArrow(Icons.Props("3rem")))
```

```scala
private val component =
  ScalaFnComponent[Props](props ⇒
    <.nav(
      ^.className := Style(props.style),
      <.div(props.leftSide.map(renderSideItem)),
      <.div(<.h1(^.className := "m-0 font-bold text-3xl", props.title)),
      <.div(props.rightSide.map(renderSideItem))
    )
  )
```

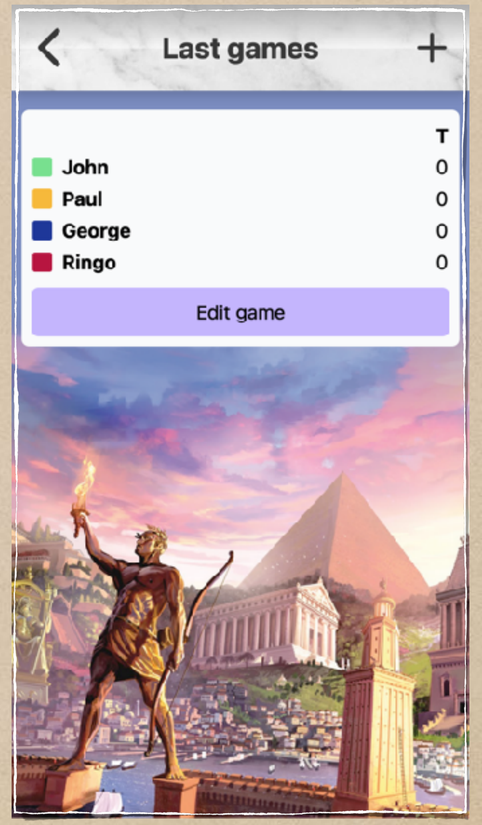
Once you have a few reusable components, building pages is like assembling LEGO bricks…

```
val Homepage = ScalaComponent
  .builder[Unit]
  .renderStatic(
    ReactFragment(
      PageBackground(PageBackground.GreenFelt),
      Header("Games"),
      GameButton(GameType.SevenWonders)
    )
  )
  .build
```

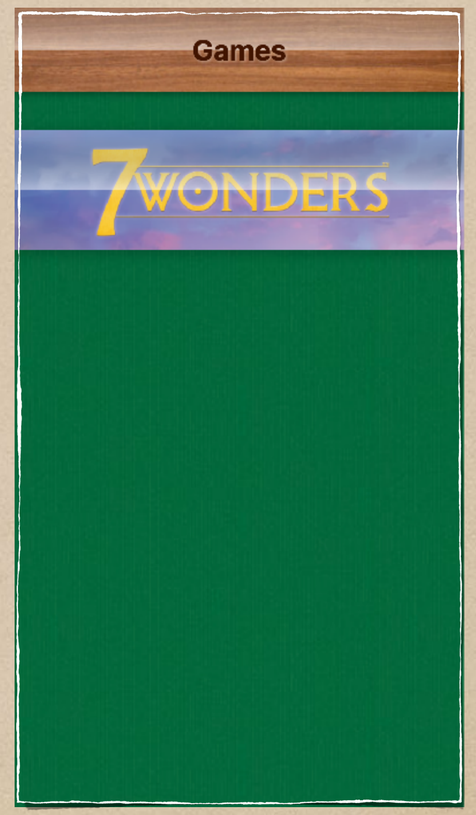


...except not all pages are so trivial

Where does this data come from ?



We need to hook into the page's lifecycle



We need to hook into the page's lifecycle

1. The user visits the "7 Wonders" page
   The page is loaded with an empty list of games
   as its initial state.

We need to hook into the page's lifecycle

1. The user visits the "7 Wonders" page
   The page is loaded with an empty list of games as its initial state.

2. ???

We need to hook into the page's lifecycle

1. The user visits the "7 Wonders" page
   The page is loaded with an empty list of games
   as its initial state.

2. ???

3. The page's state is set to a list of the most recent games

We need to hook into the page's lifecycle

1. The user visits the "7 Wonders" page
   The page is loaded with an empty list of games
   as its initial state.

2. ???

3. The page's state is set to a list of the most recent games

4. The view is updated to reflect the new state.
   (Our view remains a pure function from state to DOM elements)

We need to hook into the page's lifecycle

1. The user visits the "7 Wonders" page
   The page is loaded with an empty list of games
   as its initial state.

2. Games are fetched from some sort of persistent storage

3. The page's state is set to a list of the most recent games

4. The view is updated to reflect the new state.
   (Our view remains a pure function from state to DOM elements)

```scala
final case class State(games: List[Game])

final class Backend($ : BackendScope[Unit, State]):
  def renderGame(game: Game) = VdomElement = ???

  def render(state: State) =
    ReactFragment(
      PageBackground(PageBackground.SevenWonders),
      Header(
        "Last games",
        leftSide = Header.SideItem.BackButton(Routes.home).some,
        rightSide = Header.SideItem.PlusButton(Routes.sevenWondersNewGame).some,
        style = Header.Style.Marble
      ),
      state.games.toVdomArray(renderGame)
    )

end Backend

val component = ScalaComponent
  .builder[Unit]
  .initialState(State(games = Nil))
  .renderBackend[Backend]
  .build
```

Here, I can pass a Route or a Callback. Yay, union types!

```scala
final case class State(games: List[Game])

final class Backend($ : BackendScope[Unit, State]):
  def renderGame(game: Game) = VdomElement = ???

  def render(state: State) =
    ReactFragment(
      PageBackground(PageBackground.SevenWonders),
      Header(
        "Last games",
        leftSide = Header.SideItem.BackButton(Routes.home).some,
        rightSide = Header.SideItem.PlusButton(Routes.sevenWondersNewGame).some,
        style = Header.Style.Marble
      ),
      state.games.toVdomArray(renderGame)
    )

  // Fetch data here and update state afterwards
  val componentDidMount: AsyncCallback[Unit] = ???

end Backend

val component = ScalaComponent
  .builder[Unit]
  .initialState(State(games = Nil))
  .componentDidMount(_.backend.componentDidMount)
  .renderBackend[Backend]
  .build
```

A type-safe, declarative approach to data persistence

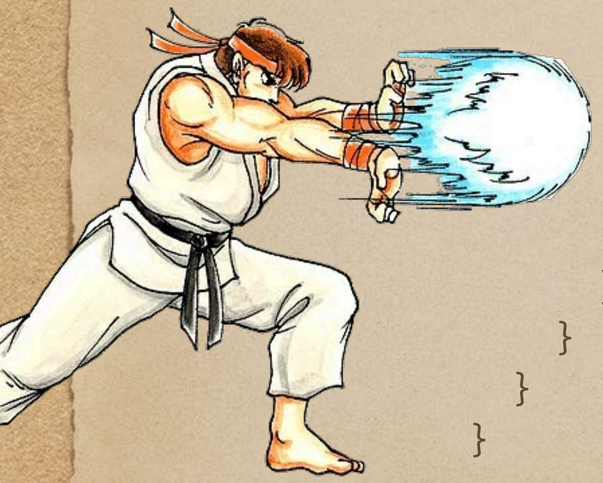
Or how IndexedDB goes from horrible to usable

What is IndexedDB?

- A transactional key-value store for client-side data
- Organised into object stores, that contains key-pairs ordered by keys
- Supports looking up values by their properties using indices
- Has an asynchronous API

IndexedDB transactions

```
let r = doSomething()
r.onsuccess = event ⇒ {
  let r = doSomething(event)
  r.onsuccess = event ⇒ {
    let r = doSomething(event)
    r.onsuccess = event ⇒ {
      let r = doSomething(event)
      r.onsuccess = event ⇒ {
        let r = doSomething(event)
        r.onsuccess = event ⇒ {
          let r = doSomething(event)
          // End transaction
        }
      }
    }
  }
}
```



IndexedDB's API is entirely callback-based:

- Operations return "requests". Reassign the "onsuccess" handler on every request to be notified of their completion.

A transaction's lifecycle is entirely implicit:

- Use callbacks to extend a transaction after a successful operation
- There is no explicit "commit" method: if you don't extend the transaction immediately, and the event loop finishes processing the current message, the transaction is implicitly closed.

IndexedDB Pitfalls: Error handling

This is actually the best-case scenario

```
let r = doSomething()
r.onsuccess = event ⇒ {
  let r = doSomething(event)
  r.onsuccess = event ⇒ {
    let r = doSomething(event)
    r.onsuccess = event ⇒ {
      let r = doSomething(event)
      r.onsuccess = event ⇒ {
        let r = doSomething(event)
        r.onsuccess = event ⇒ {
          let r = doSomething(event)
          // End transaction
        }
      }
    }
  }
}
```

If you also want to handle errors, you also have to reassign the “onerror” handler on every request:

```
let r = doSomething()
r.onsuccess = event ⇒ {}
r.onerror = event ⇒ {
  // Do something
}
```

If you really want to handle all scenarios, things get rapidly out of hand

IndexedDB Pitfalls: Error handling 2

Let's try to insert a value, and use an unsupported value as a key.
One would expect all errors to be reported in the "onerror" callback …

```
let openReq = indexedDB.open("MyDB3")

openReq.onsuccess = (event) ⇒ {
  let db = event.target.result
  let tx = db.transaction("users", "readwrite")

  let putReq = tx.objectStore("users").put("Value", {not: "a", valid: "key"})

  putReq.onerror = () ⇒ {
    console.error("Something wrong happened", e)
  }
}
```

Except this is never triggered

IndexedDB Pitfalls: Error handling 2

Let's try to insert a value, and use an unsupported value as a key.
One would expect all errors to be reported in the "onerror" callback …

```
let openReq = indexedDB.open("MyDB3")

openReq.onsuccess = (event) ⇒ {
  let db = event.target.result
  let tx = db.transaction("users", "readwrite")

  let putReq = tx.objectStore("users").put("Value", {not: "a", valid: "key"})

  putReq.onerror = () ⇒ {
    console.error("S                    ", e)
  }
}
```

```
Uncaught DOMException: Data provided to an operation does not meet requirements.
```

instead, this exception is thrown synchronously

IndexedDB Pitfalls: premature transaction closing

```
let tx = db.transaction("users", "readwrite")
 let newUser = { id: 1, fullName: "Paul" }
 let putReq = tx.objectStore("users").put(newUser, 1)
 putReq.onsuccess = async () ⇒ {
   // Suppose 'notifyBackend' returns a Promise
   await notifyBackend(newUser);

   let getReq = tx.objectStore("users").get(1)
   getReq.onsuccess = event ⇒ {
     // Never triggered
   }
   getReq.onerror = event ⇒ {
     // Never triggered
   }
 }
```

The lifecycle of transactions is closely related to the event loop.

Mixing asynchronous code with IndexedDB code may close your transaction prematurely

As a consequence, it would be potentially unsafe to manipulate IndexedDB using a Promise-based API

This would work:

```
const tx = db.transaction('keyval', 'readwrite');
const store = tx.objectStore('keyval');
const val = (await store.get('counter')) || 0;
await store.put(val + 1, 'counter');
await tx.done;
```

This wouldn't:

```
const tx = db.transaction('keyval', 'readwrite');
const store = tx.objectStore('keyval');
const val = (await store.get('counter')) || 0;
// This is where things go wrong:
const newVal = await fetch('/increment?val=' + val);
// And this throws an error:
await store.put(newVal, 'counter');
await tx.done;
```

Examples taken from jakearchibald/idb on Github, a Promise-based API for IndexedDB

IndexedDB Pitfalls: manual schema management

```
let openReq = indexedDB.open("MyDB", 3)
openReq.onupgradeneeded = e ⇒ {
  switch (e.oldVersion) {
    case 1:
      break;
    case 2:
      break;
  }
  db = e.target.result;
  db.createObjectStore("users");
}
```

- When you open a database, you need to provide a version
- Then, you get a single opportunity to create object stores and indices, by comparing the version you provided with the last seen version on the device.
- You cannot manipulate stores or indices outside of the `onupgradeneeded` callback

What I aspire to:

An interface for IndexedDB that minimises room for user error

- I took inspiration from tpolecat/doobie, “a pure functional JDBC layer for Scala and Cats” , to create a declarative IndexedDB layer
- gbogard/scalajs-idb is structured with 3 main layers

First layer: a straightforward facade to the JS API

- These "low level bindings" mimic the JS API
- They are callback-based and imperative
- They use (strict) nulls rather than options
- They use opaque types in some places, for object stores names for example

```scala
@js.native
@JSGlobal("indexedDB")
private[internal] val indexedDB: IDBFactory = js.native

@js.native
trait IDBFactory extends js.Object:
  def open(name: api.Database.Name, version: Int): IDBOpenDBRequest

@js.native
private[internal] trait IDBDatabase extends js.Object:
  val name: String = js.native
  val version: Int = js.native

  def close(): Unit = js.native

  def transaction(
      stores: js.Array[ObjectStore.Name],
      mode: api.Transaction.Mode.JS
  ): IDBTransaction = js.native

@js.native
private[internal] trait IDBRequest[Target, Result] extends js.Object:
  def result: Result | Null = js.native
  def error: DOMException | Null = js.native

  var onsuccess: js.Function1[DOMEvent[Target], Unit] = js.native
  var onerror: js.Function1[DOMEvent[Unit], Unit] = js.native

  @js.native
  trait Completed extends IDBRequest[Target, Result]:
    override def result: Result = js.native
```

Second layer: a Future-based API

- It uses Options instead of explicit nulls
- It still exposes you to premature closing of transactions if you mix IndexedDB futures with other Futures (API calls and whatnot)
- It requires type casts to add and retrieve values from/to a store
- Schema management is still imperative

```
val program: Future[String] =
  for
    dbRes ← Database.open(upgrade)(api.Database.Name("test"), 1)
    db = dbRes.database
    usersStoreName = api.ObjectStore.Name("users")
    transaction ← db.transactionFuture(
        Seq(usersStoreName),
        api.Transaction.Mode.ReadWrite
      )
    usersStore ← transaction.objectStoreFuture(usersStoreName)
    _ ← usersStore.putFuture("Paul", api.toKey(1).some)
    user ← usersStore.getFuture(api.toKey(1))
  yield user.asInstanceOf
```

Third layer: a Free-monad-based API

```
case class Champion(val name: String, val position: String) derives ObjectEncoder, Decoder

val champions = ObjectStore[Champion]("champions")

val schema = Schema().createObjectStore(championsStore)
```

Object stores are parametrised by the type of data you store in them.

Mappings between Scala types and JS types are provided by type classes.

No more `asInstanceOf`!

Third layer: a Free-monad-based API

```
case class Champion(val name: String, val position: String) derives ObjectEncoder, Decoder

val champions = ObjectStore[Champion]("champions")

val schema = Schema().createObjectStore(championsStore)
```

Scala 3's shiny new
type-class automatic derivation!

Instances of ObjectEncoder and Decoder are automatically-provided
for case classes and sum types
whose fields themselves can be encoded/decoded

Third layer: a Free-monad-based API

```
case class Champion(val name: String, val position: String) derives ObjectEncoder, Decoder

val champions = ObjectStore[Champion]("champions")

val schema = Schema().createObjectStore(championsStore)

Database
  .open[Future](Database.Name("LoL"), schema)
```

You must provide a Schema (a pure description) to open a database.

Database versions are managed automatically

Third layer: a Free-monad-based API

```scala
val transaction: Transaction[Boolean] =
  for
    key ← champions.put(illaoi, "favChampion".toKey)
    result ← champions.get(key)
  yield result == Some(illaoi)
```

Transactions are pure descriptions of IndexedDB programs,
they have no effect until you interpret them into another monad.

Because they're monads, you can use `map, flatMap, traverse` and more
useful combinators to build larger programs out of smaller ones.

Third layer: a Free-monad-based API

```
val transaction: Transaction[Boolean] =
  for
    key ← champions.put(illaoi, "favChampion".toKey)
    // Does not compile!
    _ ← TwitterAPI.send(Tweet("Hello Scalacon!"))
    result ← champions.get(key)
  yield result == Some(illaoi)
```

Because we now have a dedicated `Transaction[T]` type,
it is impossible to mix IndexedDB operations
with other asynchronous operations.
No more premature closing!

Third layer: a Free-monad-based API

```scala
val transaction: Transaction[Boolean] =
  for {
    key ← champions.put(illaoi, "favChampion".toKey)
    result ← champions.get(key)
  } yield result == Some(illaoi)

val res: Future[Boolean] =
  Database
    .open[Future](Database.Name("LoL"), schema)
    .rethrow
    // Run a read-write transaction against our champions store
    .flatMap(_.readWrite(NonEmptyList.of(champions.name))(transaction))
```

Transactions can be interpreted into an “effect monad”,

as long as you can provide the appropriate `Backend`.

sclajs-idb currently has backends for Future and Cats Effect’s IO

Wiring it all up

First, let's define our data types

```
import dev.guillaumebogard.idb.time.given
import dev.guillaumebogard.idb.api.*

final case class Game(
    id: GameId = GameId.fromNow(),
    createdAt: java.time.Instant = java.time.Instant.now(),
    state: GameState = GameState.Pending,
    players: Map[PlayerId, PlayerState] = Map.empty
) derives ObjectEncoder,
      Decoder
```

(Don't forget to derive Decoder and ObjectEncoder)

Then, a “repository” for our games

```
trait GamesRepository:

  def getGame(id: GameId): Future[Option[Game]]

  def listGames: Future[List[Game]]

  def getLastGame: Future[Option[Game]]

  def upsertGame(game: Game): Future[Game]

end GamesRepository
```

(You can use an abstract effect type, a.ka. tagless final, as well)

Let's open a database somewhere

```
import dev.guillaumebogard.idb.api
import scala.concurrent.ExecutionContext.Implicits.global

import scala.concurrent.Future

object ObjectStores:

  val sevenWondersGames =
    api.ObjectStore.withInlineKeys[Game]("games", api.KeyPath.Identifier("id"))

object Database:
  val schema = api.Schema().createObjectStore(ObjectStores.sevenWondersGames)

  val db: Future[api.Database[Future]] =
    api.Database.open[Future](
      api.Database.Name("boardgames"),
      schema
    )
```

(Here, I'm relying on Futures memoizing their values.
Not the prettiest approach, but it works)

And implement our games repository

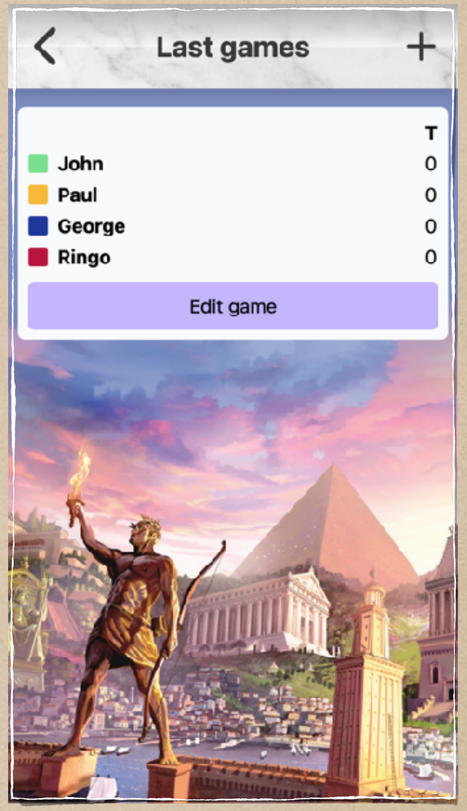
```scala
object GamesRepositoryImpl extends GamesRepository:
  import Database.{*, given}
  import dev.guillaumebogard.idb.api
  import dev.guillaumebogard.idb.api.*

  def getGame(id: GameId): Future[Option[Game]] =
    val tx = sevenWondersGames.get(id.toKey)
    db.flatMap(_.readOnly(NonEmptyList.of(sevenWondersGames.name))(tx))

  def listGames: Future[List[Game]] =
    val tx = sevenWondersGames.getAll().map(_.toList.sorted)
    db.flatMap(_.readOnly(NonEmptyList.of(sevenWondersGames.name))(tx))

  def getLastGame: Future[Option[Game]] =
    listGames.map(_.sorted.headOption)

  def upsertGame(game: Game): Future[Game] =
    db.flatMap(
      _.readWrite(NonEmptyList.of(sevenWondersGames.name))(
        sevenWondersGames.put(game) as game
      )
    )
```

Almost done! Let's pass that repo as prop to our page, and use it

```
final case class Props(repo: GamesRepository)
final case class State(games: List[Game])

final class Backend($ : BackendScope[Unit, State]):

  def render(state: State) =
    <.div("This is were we transform our games into UI elements")

  def componentDidMount(props: Props): AsyncCallback[Unit] =
    AsyncCallback
      .fromFuture(props.repo.listGames)
      .flatMap(lastGames ⇒ $.modState(_.copy(games = lastGames)).async)

end Backend

val component = ScalaComponent
  .builder[Unit]
  .initialState(State(games = Nil))
  .renderBackend[Backend]
  .componentDidMount($ ⇒ $.backend.componentDidMount($.props))
  .build
  // "fixing" the repository so the component can be used without passing any prop
  .cmapCtorProps[Unit](_ ⇒ Props(GamesRepositoryImpl))
```

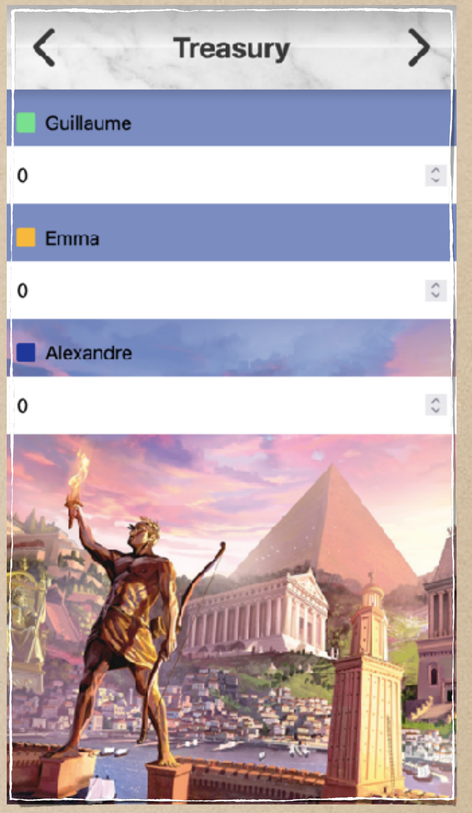
That's it!

```scala
final case class Props(repo: GamesRepository)
final case class State(games: List[Game])

final class Backend($ : BackendScope[Unit, State]):

  def render(state: State) =
    <.div("This is were we transform our games into UI elements")

  def componentDidMount(props: Props): AsyncCallback[Unit] =
    AsyncCallback
      .fromFuture(props.repo.listGames)
      .flatMap(lastGames ⇒ $.modState(_.copy(games = lastGames)).async)

end Backend

val component = ScalaComponent
  .builder[Unit]
  .initialState(State(games = Nil))
  .renderBackend[Backend]
  .componentDidMount($ ⇒ $.backend.componentDidMount($.props))
  .build
  // "fixing" the repository so the component can be used without passing any prop
  .cmapCtorProps[Unit](_ ⇒ Props(GamesRepositoryImpl))
```



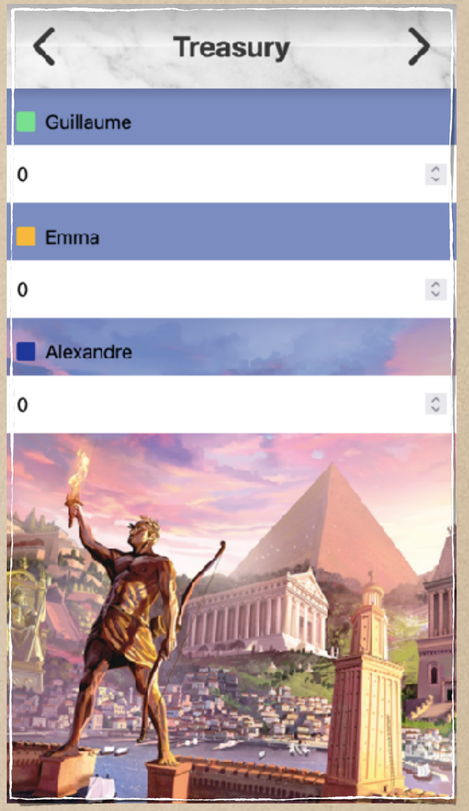
This page works similarly, except it fetches a single game instead of a list.
So I chose to model external entities using another ADT…

```scala
import cats.Monad

enum ExternalEntity[+T]:
  case Loading
  case NotFound
  case Loaded(data: T) extends ExternalEntity[T]

object ExternalEntity:
  extension [A](a: A) def loaded: ExternalEntity[A] = Loaded(a)

  extension [A](opt: Option[A])
    def loadedOrNotFound: ExternalEntity[A] = opt.fold(NotFound)(Loaded(_))

  // Implementation omitted but it works
  given Monad[ExternalEntity] = ???
```



...which lets me properly render loading types and missing entities

```
def render(gameEntity: State, props: Props): VdomNode = gameEntity match
  case ExternalEntity.Loaded(game) ⇒
    GameStepWizard(
      StepWizard.Props(
        GameWizardStep.Data(game, g ⇒ $.setState(g.loaded), finishGame),
        steps*
      )
    )
  case ExternalEntity.Loading ⇒
    PageBackground(PageBackground.SevenWonders)
  case ExternalEntity.NotFound ⇒
    ReactFragment(
      PageBackground(PageBackground.SevenWonders),
      <.div(^.className := "text-center p-4", "Game not found")
    )
```



And this is roughly how to build
a mobile-first web application using scalajs-react

Conclusion

Should you invest in this stack?

Statically-typed DOM
Re-usable components
Algebraic data types
Massive ecosystem (both React and Scala)
...

There is a lot to love, and a lot I haven't covered...

IDE and tooling on a daily basis

Slinky
Laminar
Rescript / ReasonML
Elm
Purescript…

There are also more than one way
to build type-safe web apps…

… so let me drop one hot take before we part …

Type-safe web applications are the future

Despite a few quirks,
I would choose this workflow any day over “vanilla javascript”.

Thank you! 🙏

Guillaume Bogard

Lead Scala dev. @Canal+

guillaume bogard.dev

scala-js / scala-js
japgolly / scalajs-react
gbogard / boardgames
scalajs-idb
scala3-nextjs-template