JavaScript
fwdays

ПЕРЕВІРКА ТА ПОРІВНЯННЯ ОБ'ЄКТІВ ЗА ДОПОМОГОЮ RUNTIME ТИПІВ (IO-TS)

Олександр Сугак

Senior Software Engineer

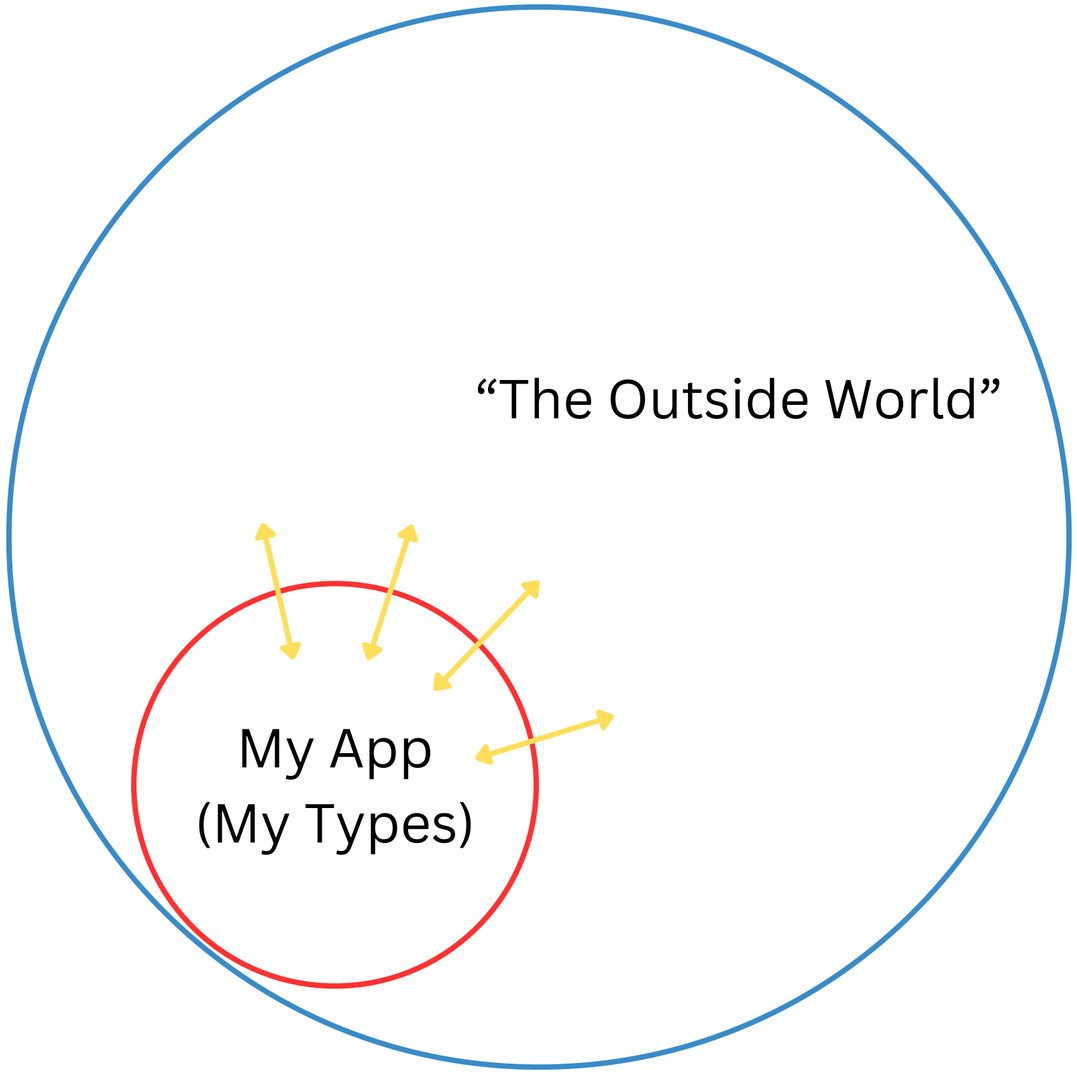
Знайомство

- Олександр Сугак
- > 15 років в IT
- Тім(тех)лід, архітектор, senior software engineer
- Ex Grammarly, now Scalable Capital
- ютуб: www.youtube.com/@AleksandrSugak
- інтеракивні курси: codereel.io
- twitter.com/alsugak

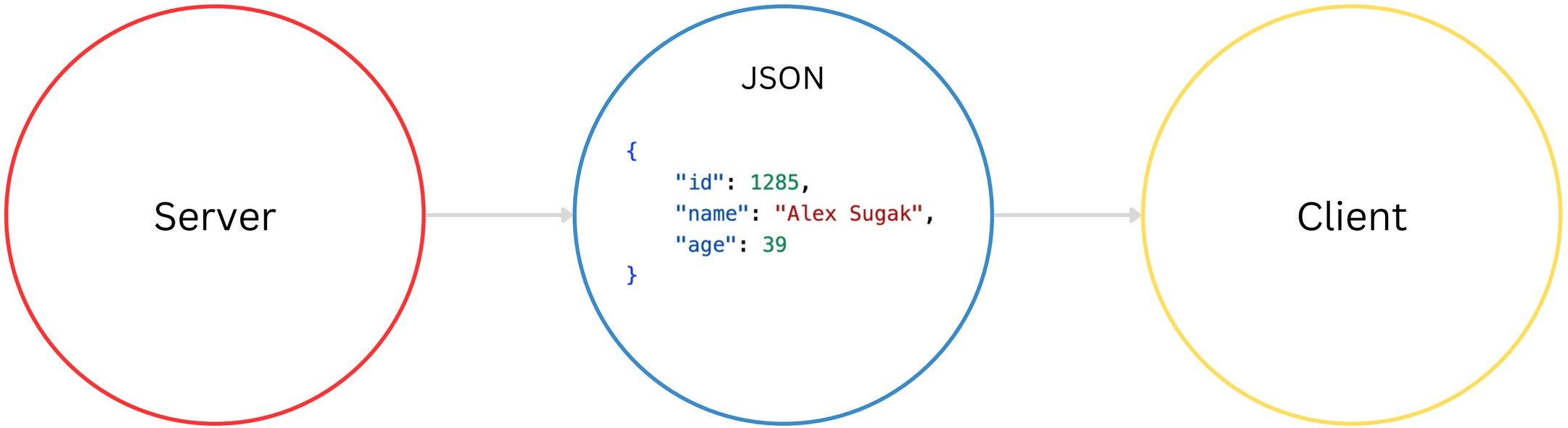
Background

- Used many static languages to build systems in various domains
- C#, F#, Java, Scala, TypeScript
- Always the same challenge: parse, validate, transform, compare raw data

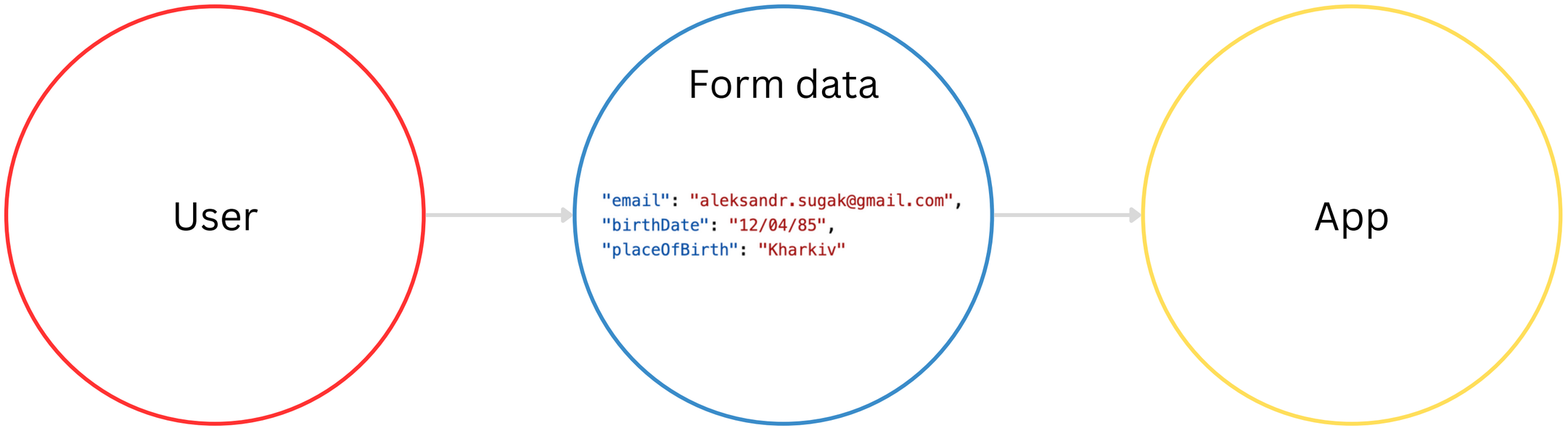
Types vs Otside World



Background



Background


User
Form data
"email": "aleksandr.sugak@gmail.com",
"birthDate": "12/04/85",
"placeOfBirth": "Kharkiv"
App

Examples

- Grammarly: back-end server sending text suggestions
- HTTO JSON APIs: encoding request payload to match server spec
- ECommerce: 3rd party service sending product updates via queue

Parsing, Validation + Comparison

“External Schema”

- JSON Schema, Swagger, GraphQL etc.
- TS: Code (types) generation using 3rd party libs
- Information duplication: schema + TS types
- Merge conflicts in generated code
- Hard to refactor code after schema changes
- “closed” code generation logic
- **as any as GeneratedType**

Runtime types

- Defined inside the app code as values/functions
- Available at runtime (not erased by TS compiler)
- Seen it before:

```ts
// TS type guards
function isNumber(x: unknown): x is number {
  return typeof x === 'number'
}

['a', 3, 'b', 7]
  .filter(isNumber)
  .map((n: number) => n / 2)
```

```ts
// Yup for forms validation
import * as Yup from 'yup'

const SignupSchema = Yup.object().shape({
  firstName: Yup.string()
    .min(2, 'Too Short!')
    .required('Required'),
  lastName: Yup.string()
    .min(2, 'Too Short!')
    .required('Required'),
  email: Yup.string()
    .email('Invalid email')
    .required('Required'),
})
```

IO-TS

- Runtime types encoding from Giulio Canti (author of fp-ts)
- TS native, static types inferred from runtime types
- Heavy use of fp-ts abstractions but can be used on its own
- Runtime types encoding, compositions, decoding and more
- **github.com/gcanti/io-ts**

IO-TS: Primitives

```ts
declare const nl: null

declare const s: string

declare const n: number
```

```ts
import * as t from 'io-ts'

const ioNl = t.null

const ioS = t.string

const ioN = t.number
```

IO-TS: Composition

```ts
declare const obj: {
  name: string
  age: number
}

declare const p:
  { a: string } &
  { b: number }


declare const u: 'Foo' | 'Bar'
```

```ts
const ioObj = t.type({
  name: t.string,
  age: t.number,
})

const ioP = t.intersection([
  t.type({ a: t.string }),
  t.type({ b: t.number })
])

const ioU = t.union([
  t.literal('Foo'),
  t.literal('Bar')]
)
```

IO-TS: Inferring Types

```ts
const ioObj = t.type({
  name: t.string,
  age: t.number,
})

const ioP = t.intersection([
  t.type({ a: t.string }),
  t.type({ b: t.number })
])

const ioU = t.union([
  t.literal('Foo'),
  t.literal('Bar')]
)
```

```ts
type ioObjT = t.TypeOf<typeof ioObj>
// type ioObjT = {
//   name: string;
//   age: number;
// }

type ioPT = t.TypeOf<typeof ioP>
// type ioPT = {
//   a: string;
// } & {
//   b: number;
// }

type ioUT = t.TypeOf<typeof ioU>
// type ioUT = "Foo" | "Bar"
```

IO-TS: Decoding

```ts
import * as t from 'io-ts'
import { PathReporter } from 'io-ts/PathReporter'
import * as E from 'fp-ts/Either'

const UserT = t.type({
  name: t.string,
  age: t.number,
})
type User = t.TypeOf<typeof UserT>

function decodeUser(data: unknown): User {
  const decoded = UserT.decode(data)
  if (E.isLeft(decoded)) {
    throw Error(`Could not parse data:
      ${PathReporter.report(decoded).join('\n')}`,
    )
  }
  return decoded.right
}
```

```ts
const correctData: unknown =
  { name: "Alex", age: 39 }
console.log(decodeUser(correctData))
// > { name: 'Alex', age: 39 }

const wrongData: unknown =
  { name: "Alex", age: "too old" }
console.log(decodeUser(wrongData))
// > Error: Could not parse data:
//   Invalid value "too old" supplied to :
//   { name: string, age: number }/age: number
```

IO-TS: much more

- Decoding/Encoding/Equality and more.
- Advanced compositions (e.g. recursive types)
- Data transformations (.refine, .parse)
- Branded types (E.g. PositiveInteger, ValidEmail, etc.)
- Custom error reporters
- Custom types
- Custom interpreters (!)

IO-TS: Custom types

```ts
import * as t from 'io-ts'
import * as E from 'fp-ts/Either'

// represents a Date from an ISO string
const DateFromString = new t.Type<Date, string, unknown>(
  'DateFromString',
  (u): u is Date => u instanceof Date, // guard
  (u, c) => // validate
    E.Chain.chain(t.string.validate(u, c), (s) => {
      const d = new Date(s)
      return isNaN(d.getTime()) ? t.failure(u, c) : t.success(d)
    }),
  (a) => a.toISOString() // encode
)
```

IO-TS: Custom types

```ts
const UserT = t.type({
  name: t.string,
  // new type can be used together with built in types
  registeredAt: DateFromString
})

type User = t.TypeOf<typeof UserT>
// type User = {
//   name: string;
//   registeredAt: Date;
// }
```

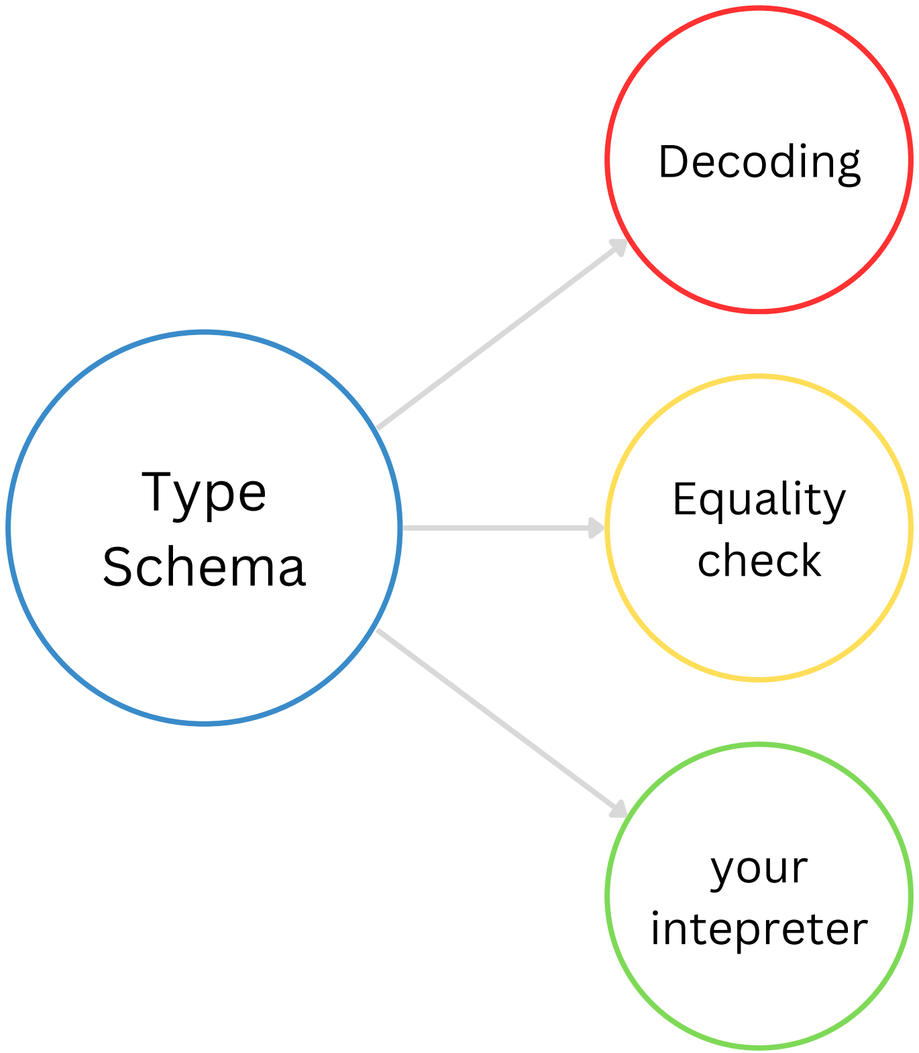
IO-TS: Custom types

```
console.log(UserT.decode({
  name: "Alex",
  registeredAt: '2024-05-20T23:00:00.000Z'
}))
// right: {
//   name: 'Alex',
//   registeredAt: new Date('2024-05-20T23:00:00.000Z')
// }

console.log(UserT.decode({
  name: "Alex",
  registeredAt: 'foo'
}))
// left: [ ...errors ]
```

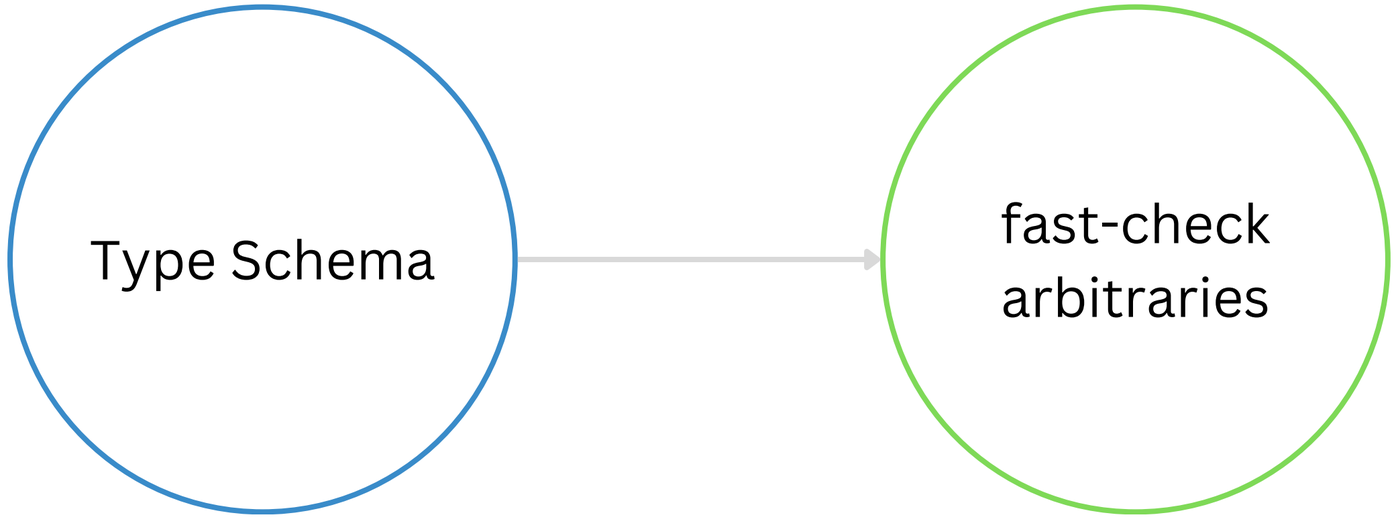
IO-TS: Schema and Schemable

```ts
import * as S from 'io-ts/Schema'

// same as type, but interpretable separately
const UserT = S.make((S) =>
  S.struct({
    name: S.string,
    age: S.number,
  }),
)
type User = S.TypeOf<typeof UserT>

import * as D from 'io-ts/Decoder'
import * as Eq from 'io-ts/Eq'

export const decodeUser = S.interpreter(D.Schemable)(UserT)
export const eqUser = S.interpreter(Eq.Schemable)(UserT)
```

IO-TS: Schema and Schemable



IO-TS: Schema and Schemable



Fast-check

```
fc.record(
  {
    id: fc.uuidV(4),
    age: fc.nat(99),
  },
  { requiredKeys: [] },
);
// Note: Both id and age will be optional values
// Examples of generated values:
// • {"id":"00000004-27f6-48bb-8000-000a69064200","age":3}
// • {"id":"ffffffee-ffef-4fff-8000-0015f69788ee","age":21}
// • {"age":34}
// • {"id":"2db92e09-3fdc-49e6-8000-001b00000007","age":5}
// • {"id":"00000006-0007-4000-8397-86ea00000004"}
// • …
```

https://fast-check.dev/

IO-TS: Schema and Schemable

```ts
// this function has a problem!
const findOldestAge = (users: User[]): number => {
  const oldestUser = users.reduce(
    (u, acc) => u.age > acc.age ? u : acc,
    users[0])
  return oldestUser.age
}
```

IO-TS: Schema and Schemable

```ts
import * as Arb from './arbitrary' // 160 LoC
import fc from 'fast-check'

const userArb = S.interpreter(Arb.Schemable)(UserT)

// verify that findOldestAge never throws
fc.assert(
  fc.property(fc.array(userArb), (randomUsers: User[]) => {
    findOldestAge(randomUsers)
  }),
)

// Error: Property failed after 10 tests
// { seed: 55396323, path: "9", endOnFailure: true }
// Counterexample: [[]]
// Shrunk 0 time(s)
// Got TypeError: Cannot read properties of undefined (reading 'age')
//     at ...
```

IO-TS: Comparing with Zod

- io-ts and fp-ts can be hard for beginners
- Zod is more developer-friendly
- both are good but to io-ts to me feels more thought-through
- custom interpreters > 3rd party libraries

```ts
import { z } from "zod";

const User = z.object({
  username: z.string(),
});

User.parse({ username: "Ludwig" });

// extract the inferred type
type User = z.infer<typeof User>;
// { username: string }
```

https://zod.dev/

Some conclusions

- Runtime types offer IO type-safety without info duplication
- IO-TS is a good library offerring sound foundation for your app IO logic
- IO-TS can be a good way to start “doing FP” in your project

“Unit tests for beginners” course

Leave your email to get notification when course is ready:
codereel.io/courses/unit-tests-intro