Observability-driven hexagonal architecture in TypeScript

Antti Pitkänen @ TampereJS 4.4.2024

Agenda

- whoami
- Why observability?
- How observability?
- How to write easily observable hexagonal TypeScript code?
  - Tagged unions
  - Hexagonal architecture
  - Demo

- Disclaimer: these are my opinions
- Not within the scope:
  - Real world working example
  - More thorough explanations of tagged unions or hexagonal architecture
  - How to use monitoring approach of your choice to monitor the software

whoami

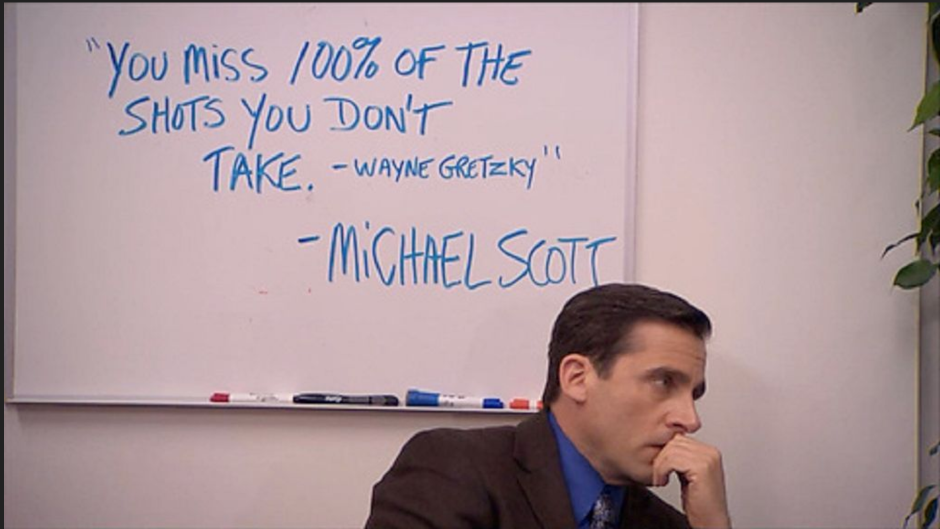
- Antti Pitkänen
- Former TampereJS organizer!
- Staff Software Engineer @ Swappie

Observability

Observability

“The ability to tell from the outside whether your system is working correctly, and when it’s not, what exactly is going wrong”

-Me



Observability

Example: ecommerce site

You want to be able to *observe* whether you customers are able to…

- View your products
- Enter the checkout
- Successfully buy your products

Observability

Unhelpful signals 😕👎

- Error making payment!!!
- No errors, all good?

Observability

Unhelpful signals 😕👎

- Error making payment!!!
- No errors, all good?

Helpful signals 😌👍

- 68% of payments were authorized during the past hour (against a baseline of 71%)
- 90% of failures are due to expected reasons (e.g. wrong CVV, not enough funds, card declined, fraud suspected…), while 10% are due to unexpected errors
- The order creation success rate is going down fast
- Our calls to the 3rd party payment API started failing due to credentials being rejected
- Orders are flowing through at a normal rate

How to make your application observable?

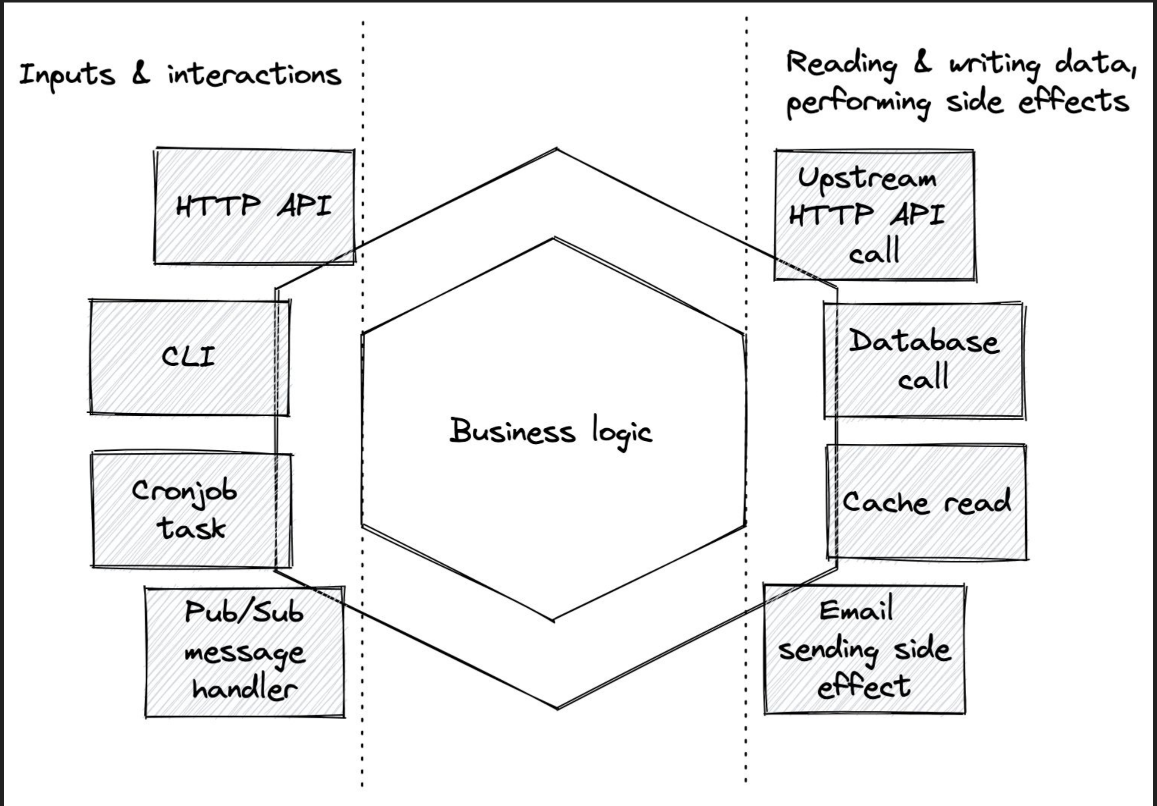
- Provide enough specific context
  - E.g. if a payment failed, which payment was it? Which payment method? Which order? What was the attempted amount?
  - => Enable precise debugging of the situation
- Report not only the errors, but the overall context
  - Is one payment failing out of a million, or 10%, or are all payments failing?
  - => Should we be worried?
- Separate the errors (and positive outcomes) into meaningful categories to understand what is happening
  - Expected failures (not enough balance, card expired, withdrawal limit exceeded…)
  - Unexpected failures (upstream API downtime, programming errors, bad requests…)
  - Successful payments
  - => Understand the development of the situation over time

“But my observability tool can do all that for me?”

“Can’t I just look at the HTTP status codes?”

Hexagonal architecture

Hexagonal architecture



Hexagonal architecture

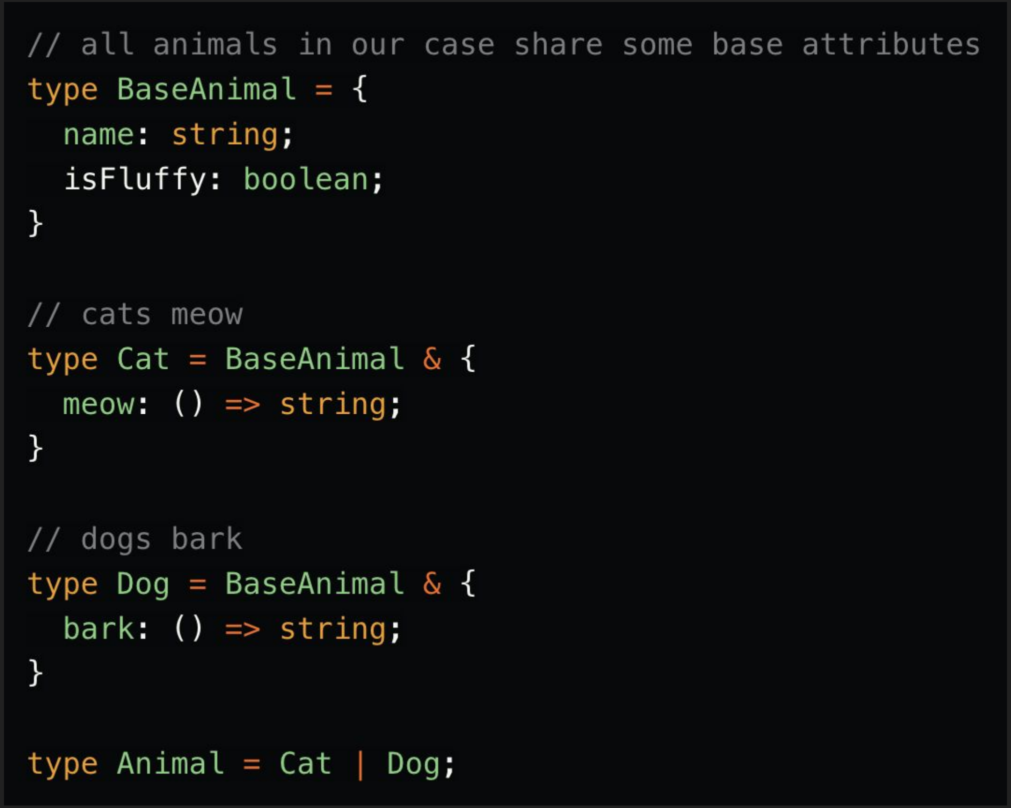
- Separation into three layers: views, services and connectors
- Each layer is an error boundary returning explicit values (not throwing errors)
- Model the data you need for the business logic (services) carefully
- Write code interfaces first, implementations second, implementation details can change without the interface needing to change
- Inject the dependencies for decoupled logic and easy testing

Observable TypeScript

Tagged unions / discriminated unions

- TypeScript’s best feature for modelling data
- A way to say a type is “either A or B” (...or C or D or…)
- …and a runtime mechanism for type safe operations!
- No need to throw Errors for the control flow
- I have a blog posts about this on https://dev.to/anttispitkanen 😉

Tagged unions: example problem



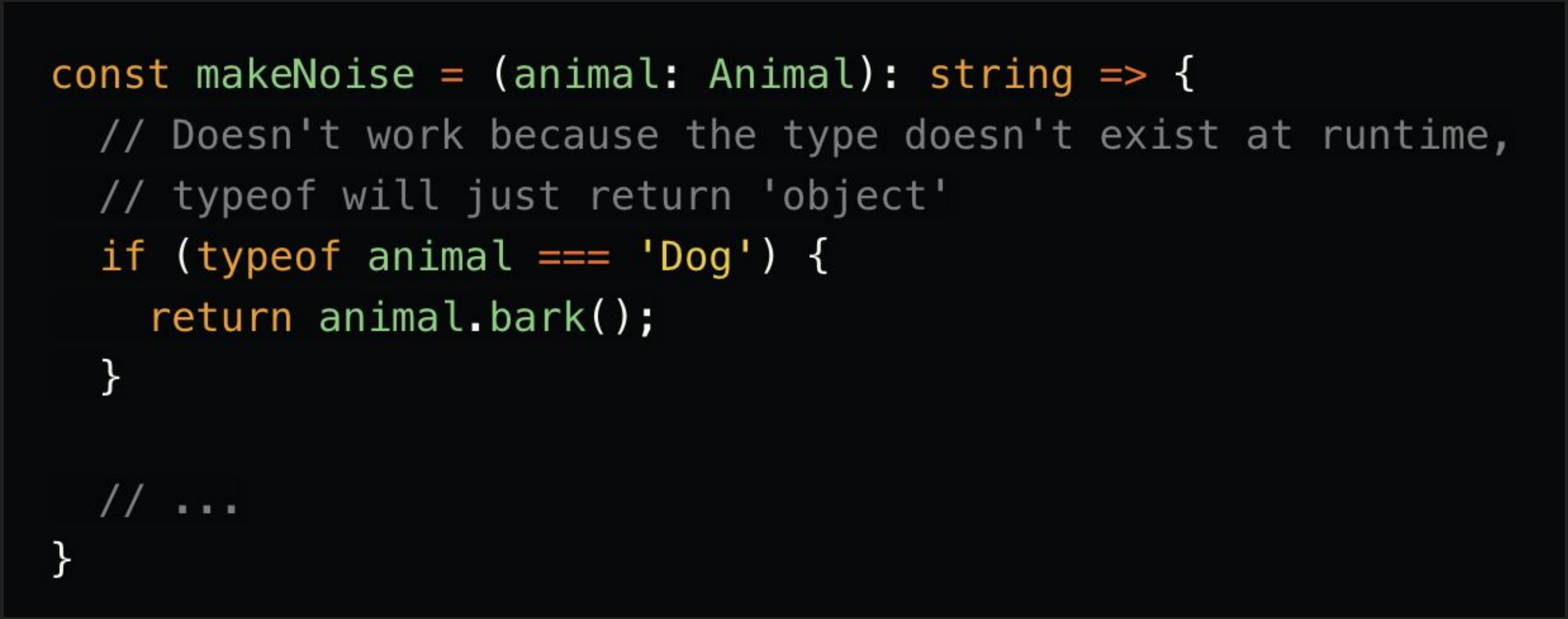
```
// all animals in our case share some base attributes
type BaseAnimal = {
  name: string;
  isFluffy: boolean;
}

// cats meow
type Cat = BaseAnimal & {
  meow: () => string;
}

// dogs bark
type Dog = BaseAnimal & {
  bark: () => string;
}

type Animal = Cat | Dog;
```

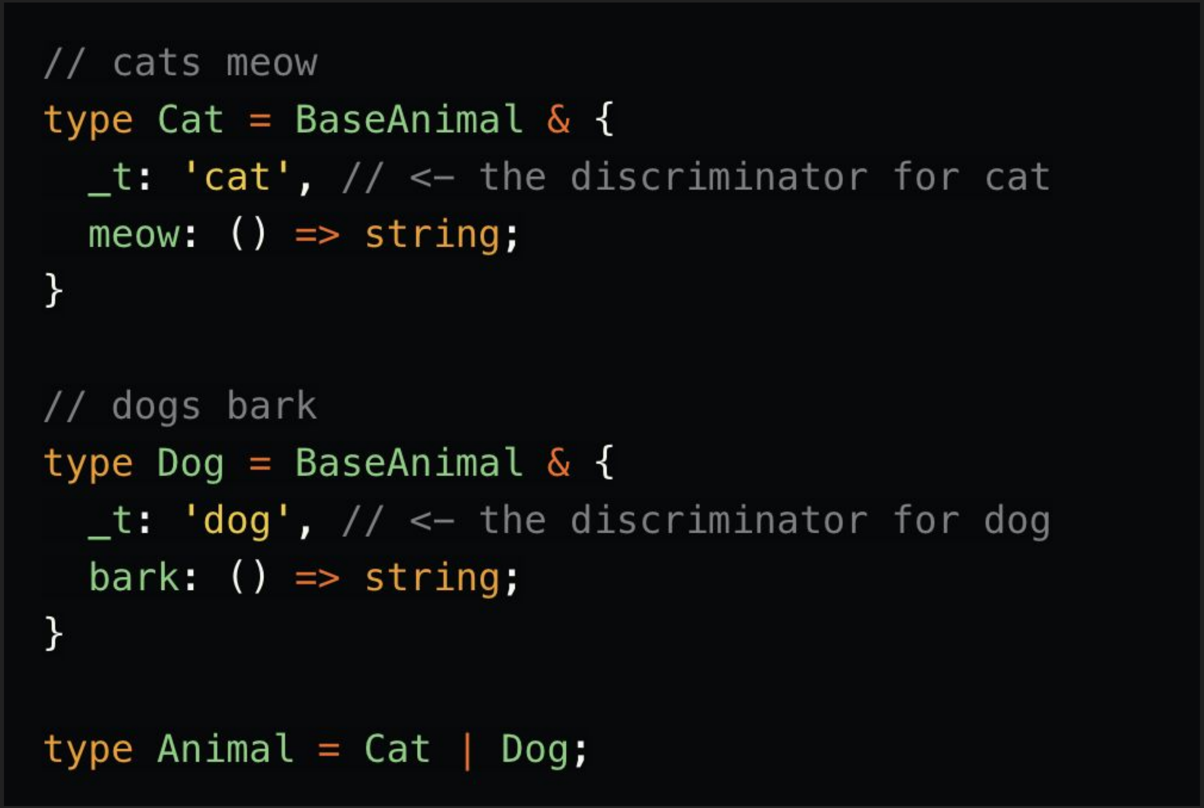
Tagged unions: example problem



```
const makeNoise = (animal: Animal): string => {
  // Doesn't work because the type doesn't exist at runtime,
  // typeof will just return 'object'
  if (typeof animal === 'Dog') {
    return animal.bark();
  }

  // ...
}
```

Tagged unions: example solution

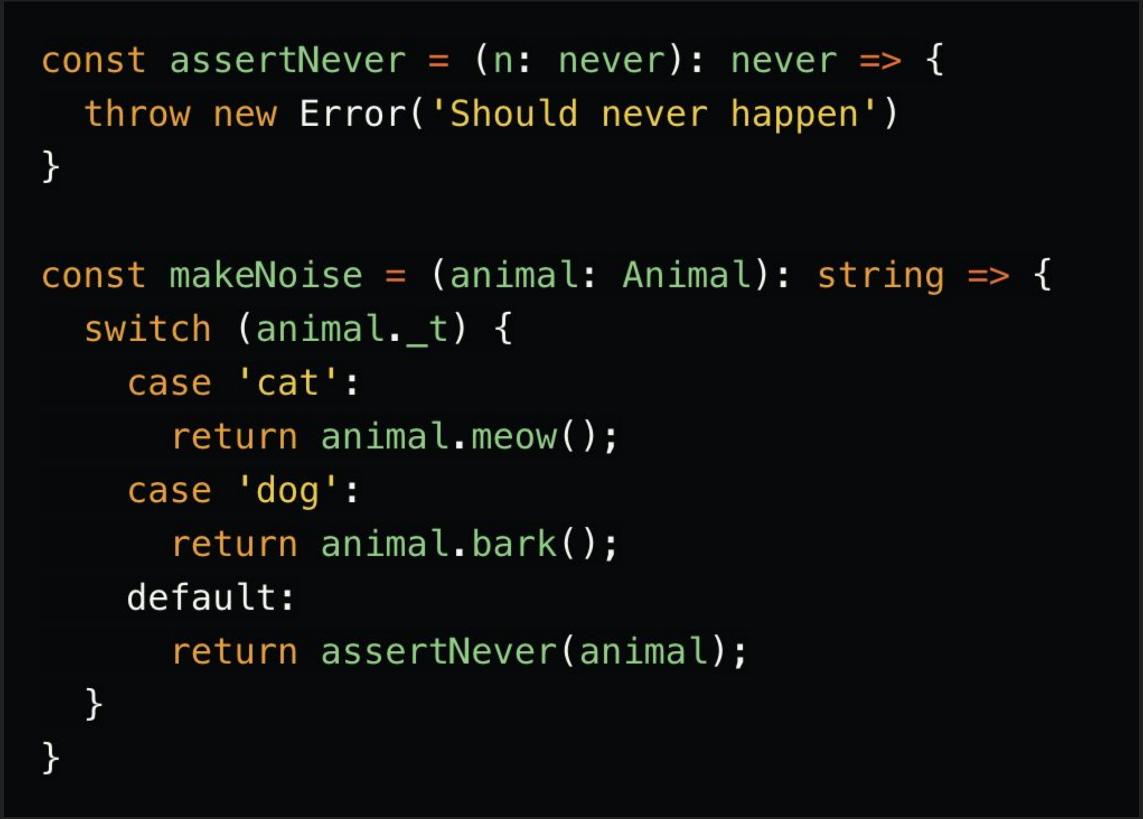


```
// cats meow
type Cat = BaseAnimal & {
  _t: 'cat', // <- the discriminator for cat
  meow: () => string;
}

// dogs bark
type Dog = BaseAnimal & {
  _t: 'dog', // <- the discriminator for dog
  bark: () => string;
}

type Animal = Cat | Dog;
```

Tagged unions: example solution



```
const assertNever = (n: never): never => {
  throw new Error('Should never happen')
}

const makeNoise = (animal: Animal): string => {
  switch (animal._t) {
    case 'cat':
      return animal.meow();
    case 'dog':
      return animal.bark();
    default:
      return assertNever(animal);
  }
}
```

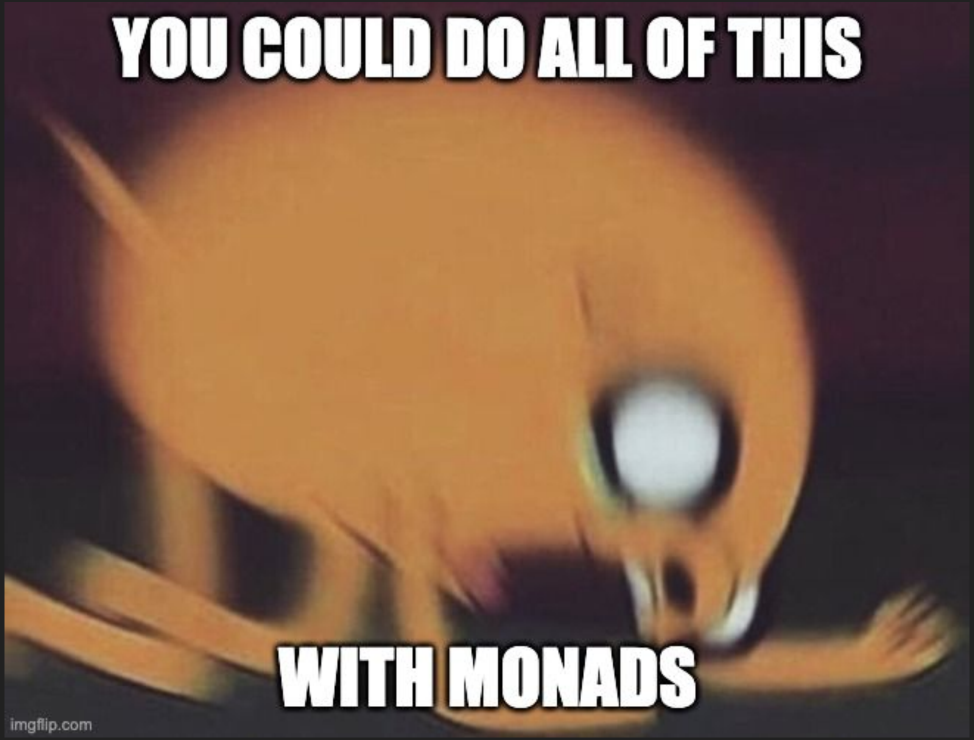
Observability-driven hexagonal TypeScript

- Think about what kind of (success/error) situations your application can get into, and model those into outcomes using tagged unions
  - The different kinds of errors can also be used to branch the logic, e.g. certain failures might be retryable in the code
- Return the outcomes and observability related metadata from the business logic to the topmost layer, and report them there
- For each operation it’s good to also report an “init” metric, because sometimes you might not get an outcome at all
  - Then the success rate becomes “count of successful events / count of initiated events”

Demo
Time!

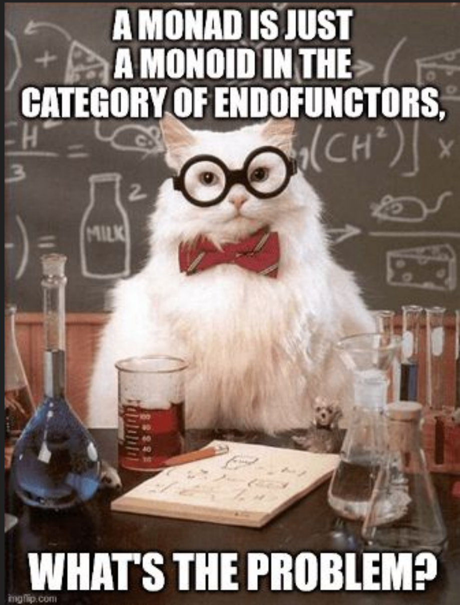
“Why not just throw and catch Errors?”

Bonus



Bonus: monads

- Modelling the success vs failure states with Either monads
- Collecting the purely observability based data with Writer monads
- Writing the “right based” business logic safely and simply, separating the concern of error handling from the happy path
- fp-ts is nice, but can have a high learning curve for those not familiar with the concepts
  - Task monad for a safer async abstraction
  - Mechanism for forcing the error boundaries



German ⇄ English

dankmemes

thank you memes