An 'exceptionally' talk

by Ivan Hofer

Svienna Meetup 2023/02

Who am I

- **Ivan Hofer**
- web developer
- passion for great DX
- Svelte and TypeScript =♡
- open source
  - typesafe-i18n (https://github.com/ivanhofer/typesafe-i18n)
  - Svelte(Kit) TypeScript Showcase (https://github.com/ivanhofer/sveltekit-typescript-showcase)

ivanhofer

What I will show you today

- **error and exception handling**
- not so much about Svelte 😢
- more about TypeScript 🚀
- write less error prone code
- concepts can be applied to any kind of applications
  * in the JavaScript ecosystem

What is an error?

- **unrecoverable**

- application can't function if condition is not met
- e.g.
  - missing environment variable on startup
  - DB migrations don't succeed

- application should crash

What is an exception?

- **recoverable**
- affects just a part of your application
  - temporary:
    e.g. external server is down
  - input-dependent
    e.g. passing incorrect data
- application should continue to work

error vs. exception

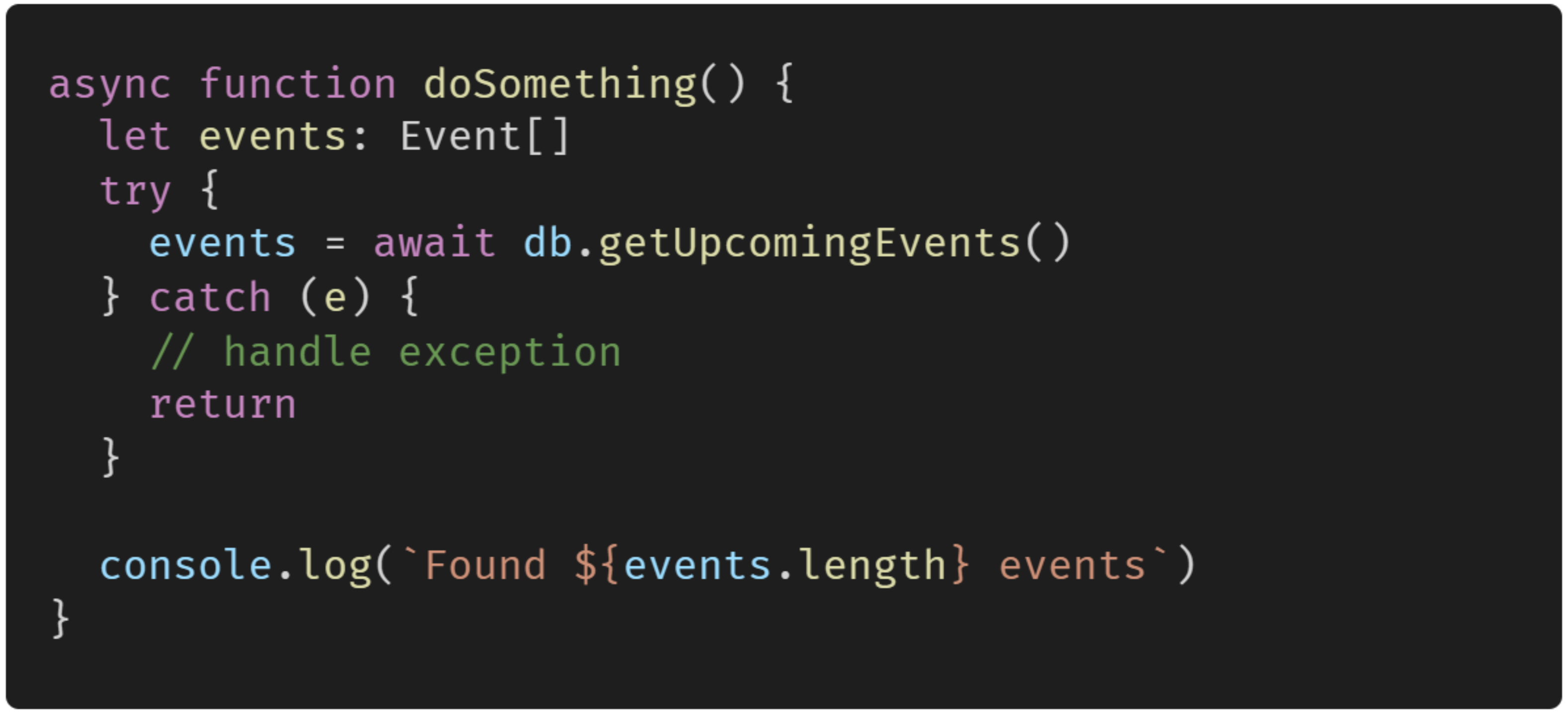
- blurry line
- not always clear
- you or your team have to decide
- case by base
- most things are probably exceptions

Problem

```
async function soSomething() {
  const events = await db.getUpcomingEvents()

  console.log(`Found ${events.length} events`)
}
```

- exceptions are not handled
- you need to know where to handle them

Problem



```
async function doSomething() {
  let events: Event[]
  try {
    events = await db.getUpcomingEvents()
  } catch (e) {
    // handle exception
    return
  }

  console.log(`Found ${events.length} events`)
}
```

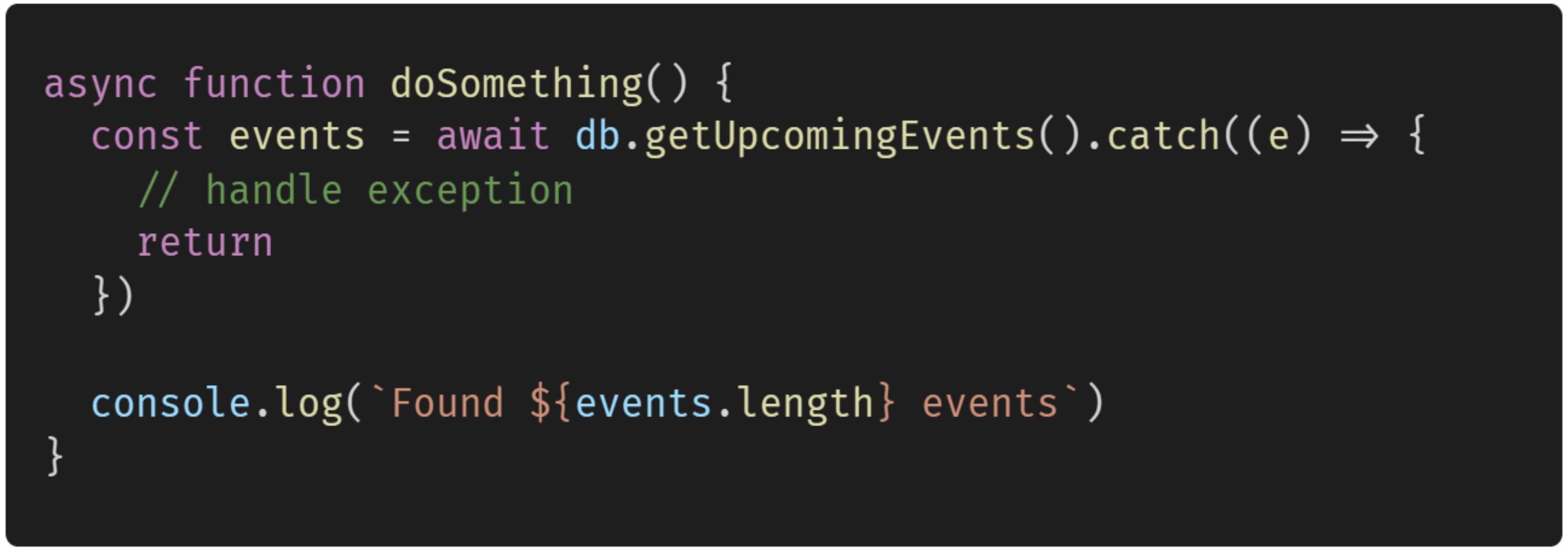
- indentation
  - less readable
- variable declaration
  - scoping issues
  - mutable
  - need to define types
- adds boilerplate code
  - min. 5 LOC per statement

Can you see the issue?

`try-catch` does not *(always*)* work with async function calls

**make sure to await them inside the `try` block*

Problem



```
async function doSomething() {
  const events = await db.getUpcomingEvents().catch((e) => {
    // handle exception
    return
  })

  console.log(`Found ${events.length} events`)
}
```

Can you see the issue?

`events` can be undefined

* TypeScript will tell you that in this specific case

Problem

```
async function doSomething() {
  const events = await db.getUpcomingEvents().catch((e) ⇒ {
    // handle exception
  })

  if (!events) return

  console.log(`Found ${events.length} events`)
}
```

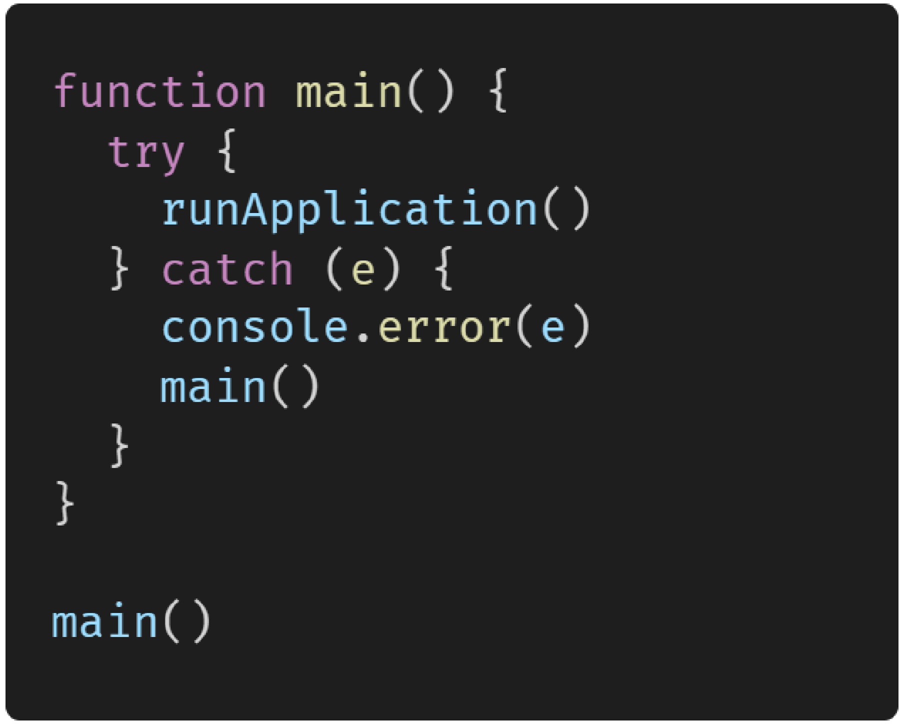
Finally!

Problem

- different approaches for sync and async code
- alter the structure of your code
- you need to know what functions could potentially throw
  - look into the implementation
  - multiple levels deep
  - but what if the implementation changes?
- TypeScript does not know
  - that something can result in an exception
  - where an exception can happen
  - what kind of exception can happen

How to deal with it?

- not handling it at all
- write more integration tests
- wrapping everything in a try-catch block
  - will not work with async code
- **return exceptions instead of throwing them**



```
function main() {
  try {
    runApplication()
  } catch (e) {
    console.error(e)
    main()
  }
}

main()
```

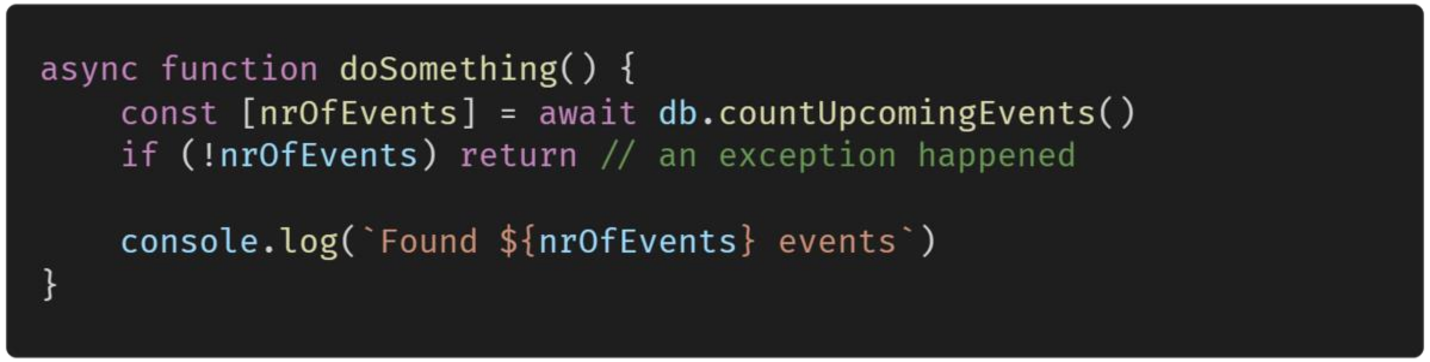
[data, exception] tuple

```
getUpcomingEvents: async () ⇒ {
  if (isMonday)
    return [undefined, new Error('Try again tomorrow')] as const

  const events = [{
    title: 'Svienna 2023/02',
    date: '2022-02-20',
  }]

  return [events, undefined] as const
}
```

```
async function doSomething() {
  const [events, exception] = await db.getUpcomingEvents()
  if (exception) return // an exception happened

  console.log(`Found ${events.length} events`)
}
```

```
type SuccessResult<T> =
    [T, undefined]

type ExceptionResult =
    [undefined, Error]

type Result<T> =
    | SuccessResult<T>
    | ExceptionResult
```

[data, exception] tuple

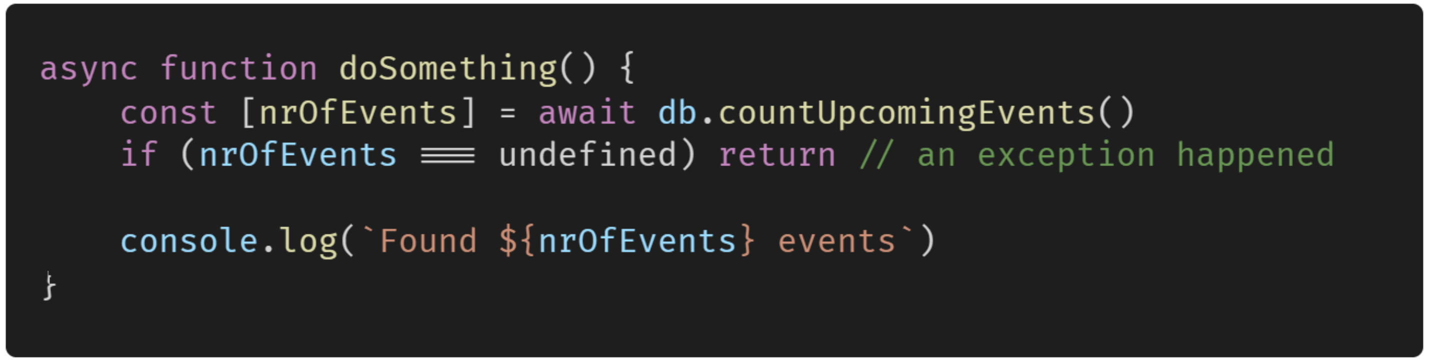


```
async function doSomething() {
    const [nrOfEvents] = await db.countUpcomingEvents()
    if (!nrOfEvents) return // an exception happened

    console.log(`Found ${nrOfEvents} events`)
}
```

don't become lazy!

Can you see the issue?

`nrOfEvents` could be `0` (falsy)



```
async function doSomething() {
    const [nrOfEvents] = await db.countUpcomingEvents()
    if (nrOfEvents === undefined) return // an exception happened

    console.log(`Found ${nrOfEvents} events`)
}
```

[data, exception] tuple

```
async function soSomething() {
    const [event] = await db.getUpcomingEvent()
    if (event === undefined) return // an exception happened

    console.log(`Upcoming event on ${event.date}`)
}
```

don't become lazy!

Can you see the issue?

`event` could be `undefined` even if no exception occurred

```
async function soSomething() {
    const [event, exception] = await db.getUpcomingEvent()
    if (exception !== undefined) return // an exception happened

    if (event)
        console.log(`Upcoming event on ${event.date}`)
    else
        console.log('No upcoming event')
}
```

{ data, exception } object

```
type SuccessResult<T> =
    { data: T; exception: never }

type ExceptionResult =
    { data: never; exception: Error }

type Result<T> =
    | SuccessResult<T>
    | ExceptionResult
```

```
getUpcomingEvents: async () ⇒ {
  if (isMonday)
    return { exception: new Error('Try again tomorrow') }

  const events = [{
    title: 'Svienna 2023/02',
    date: '2022-02-20',
  }]

  return { data: events }
}
```

```
async function doSomething() {
  const { data: events, exception } = await db.getUpcomingEvents()
  if (exception) return // an exception happened

  console.log(`Found ${events.length} events`)
}
```

Comparison

action	[data, exception] tuple	{ data, exception } object
returning data	≈	✓
auto inference of types	≈	✓
naming variables	✓	≈
enforced handling of exceptions	✕	✕

- some things work good
- some don't
- biggest issues:
    - no enforced handling of exceptions
    - not obvious how to handle them

Introducing: exceptionally

exception + a11y (accessibility)

- An exception handling approach that makes it easier to safely use unknown code. It can automatically inform you about potential exceptions and therefore being more accessible without proper knowledge of implementation details.

- enforces you to handle exceptions

Syntax

```
getUpcomingEvents: async () ⇒ {
  if (isMonday) return exception(new Error('Try again tomorrow'))

  const events = [{
    title: 'Svienna 2023/02',
    date: '2022-02-20',
  }]

  return success(events)
}
```

```
async function doSomething() {
  const result = await db.getUpcomingEvents()
  if (result.isException) return // an exception happened

  console.log(`Found ${result().length} events`)
}
```

DEMO

Fun Fact

Do you know how much the core of the library adds to your bundle size? Just as much as it takes a PC to save this paragraph on disk.

132 bytes (minimized and gzipped)

- just a few lines of code
- with some TypeScript magic

Implementation / runtime

```
const exceptionally = Symbol.for('exceptionally')

export const isExceptionallyResult = value => value?.exceptionally === exceptionally

const wrap = (success, data) => isExceptionallyResult(data)
    ? data
    : Object.assign(() => data,
        {
            exceptionally,
            isSuccess: success,
            isException: !success,
        },
    )

export const success = data => wrap(true, data)

export const exception = data => wrap(false, data)
```

Alternatives

- write your own solution
- @badrap/result (https://github.com/badrap/result)
- NeverThrow 🙅‍♀️ (https://github.com/supermacro/neverthrow)

Advantages

- code flow stays linear
- treat sync and async code the same
- full TypeScript support
  - auto-inferred types
  - don't need to look at the implementation
  - see if exceptions are unhandled
- huge help when refactoring

Disadvantages

- learning curve
- you need to wrap everything
  - can be solved with a proxy
- supports only code you own
  - unless you wrap the functionality

Shift left

- resolve issues as early as possible
  - in your brain
  - while writing the code $\boxed{\leftarrow}$ here
  - while manual testing
  - automatic tests (CI) $\boxed{\leftarrow}$ and here
  - staging environment
  - production system

- reduces costs

Conclusion

- will your code be bugsafe? **NO**
  but it will make you aware of potential bugs
- can you ignore writing tests? **NO**
  but you'll need to write fewer of them
- does it replace services like Sentry? **NO**
  using a reporting service as fallback is a great idea
- should you know implementation details of a function you use? **NO**
  TypeScript should tell you what all possible outcomes are

Links

- https://github.com/ivanhofer/exceptionally
  - try it
  - share your feedback

- https://github.com/ivanhofer

- i18n
  - https://github.com/pipeli18ne/RFC
  - Feedback wanted!