Workshop de TypeScript

Me presento

Pepe Fabra Valverde

Líder y Arquitecto de Front

Agenda del taller

Agenda del taller

1h 15m – Introducción de los conceptos, hasta intermedio

Agenda del taller

1h 15m – Introducción de los conceptos, hasta intermedio

30m aprox. – Descansito

Agenda del taller

1h 15m – Introducción de los conceptos, hasta intermedio

30m aprox. – Descansito

1h 15m – Intermedio hasta avanzado

Disclaimer

- La sesión se va a grabar
- Se compartirán las slides al terminar
- Pregunta sin miedo

TypeScript

Qué es TypeScript

JavaScript con tipos… o eso dicen

Qué es TypeScript

JavaScript con tipos… o eso dicen

Es un sistema de tipados que ofrece solución al problema de no saber con qué se está trabajando en JavaScript

- Tipos, interfaces, enums (hablaremos más adelante)

Orígenes de TypeScript

- Desarrollado por Microsoft en 2012

Orígenes de TypeScript

- Desarrollado por Microsoft en 2012
- Ya existían otras soluciones de tipado en JavaScript
  - CoffeeScript, Flow (Meta), etc.

Orígenes de TypeScript

- Desarrollado por Microsoft en 2012
- Ya existían otras soluciones de tipado en JavaScript
  - CoffeeScript, Flow (Meta), etc.
- Solución integral con un sistema más complejo
  - No solamente tipados básicos
  - Inferencias, genéricos, herencia completa

Sistemas de tipados

Sistemas de tipados

- Qué son
- Qué esperamos de ellos
- Otros lenguajes

Qué es un sistema de tipados

Qué esperamos de un sistema de tipados

Hindley–Milner

- https://stackoverflow.com/questions/399312/what-is-hindley-milner

Otros lenguajes

No todos los lenguajes son tipados, y los que lo son, no son iguales

Existen dos clases de lenguajes tipados

- Fuertemente tipados
  - Java, C#, Rust, Go
- Débilmente tipados
  - JavaScript, Python, PHP… TypeScript

Fuertemente tipado

Débilmente tipado

Modelo mental

Tiempos

- runtime
  - Cuando se esté ejecutando, lo ve el usuario
- buildtime/compiletime
  - Cuando se esté compilando, lo ve la pipeline y quien desarrolla
- devtime
  - Cuando se está desarrollando, lo ve quien desarrolla

En un proceso de desarrollo:

devtime -> buildtime -> runtime

JIT

Just In Time compiler

Algunos compiladores son JIT, lo compilan al vuelo

Nos permiten ver errores de compiletime en devtime

TypeScript

Es una herramienta/librería de devtime.

La única excepción son los enums que si que siguen un proceso en compiletime

TypeScript

Es una herramienta/librería de devtime.

La única excepción son los enums que si que siguen un proceso en compiletime

El paso que hay en compiletime de TypeScript es para eliminar código no nativo (tipados, genéricos)

Compilación y transpilación

- Compilar es convertir a lenguaje de máquina
- Transpilar es convertir de un lenguaje de alto nivel a otro de alto nivel
  - TypeScript se *transpila* a JavaScript

Compilación y transpilación

- Compilar es convertir a lenguaje de máquina
- Transpilar es convertir de un lenguaje de alto nivel a otro de alto nivel
  - TypeScript se *transpila* a JavaScript

Es más cómodo y universal utilizar compilar

Errores en TypeScript

Cuando hablamos de errores de tipado hemos de entender que son en devtime.

Los únicos errores que afectan a compiletime y runtime son los de sintáxis, si el código está mal escrito, TypeScript no puede detectar qué partes son suyas y tiene que eliminar

Es una ayuda

DARLE UNA PASADA QUE ESTÁ ESTO HECHO UN DESASTRE

- Desarrollo para desarrolladores
  - Devtooling es algo que no aprovechamos lo suficiente
- No perder tiempo de más en errores de TS
- Es documentación y guía para on-boarding

Merece más importancia, sigue siendo una pieza más del puzle

- No hay que engañarle, sino usarlo para que nos ayude

Primitivos

Primitivos de JavaScript

typeof

¿Qué devolverá?
typeof null

typeof null

- El tipo de null no existe en JavaScript
  - Aún que estemos en TypeScript, esto nos impactará
  - Type null existe, pero typeof null será “object”

typeof null

- El tipo de null no existe en JavaScript
  - Aún que estemos en TypeScript, esto nos impactará
  - Type null existe, pero typeof null será “object”
- Como null es un objeto que no apunta a nada, se entiende que es un *valor* que puede tener un objeto

typeof null

- El tipo de null no existe en JavaScript
  - Aún que estemos en TypeScript, esto nos impactará
  - Type null existe, pero typeof null será “object”
- Como null es un objeto que no apunta a nada, se entiende que es un *valor* que puede tener un objeto

- Null es el valor que debería estar pero no está
- Undefined es un valor que todavía no ha sido representado

Objetos

Objetos en JavaScript

- Qué es un objeto

Objetos en JavaScript

- Qué es un objeto
  - Es un prototipo, una cadena de prototipos más concretamente

Objetos en JavaScript

- Qué es un objeto
  - Es un prototipo, una cadena de prototipos más concretamente
- Sirve de estructura clave valor
  - Todo valor, o referencia, puede ser key de un objeto

Objetos en TypeScript

- Sólo string, number y Symbol pueden ser claves del objeto

Objetos en TypeScript

- Sólo string, number y Symbol pueden ser claves del objeto
- Existen dos notaciones principales
  - { [k: type]: value }
  - Record<TKey, TValue>

keyof

- Genera un tipo con todas las keys de un objeto como valores
  - En caso de ser strings, las guardará como literales

keyof

- Genera un tipo con todas las keys de un objeto como valores
  - En caso de ser strings, las guardará como literales
- Funciona con elementos que no sean objetos, pero no es recomendable
  - keyof devuelve todas las *properties* que se encuentran en un objeto
  - De un array devuelve todos sus métodos
  - De una string, las funciones de utilidades

k in {Type}

Podemos definir el tipo de una key con la keyword *in*

Lo que le decimos es que se encuentra en el universo proporcionado

Si el universo proporcionado sólo tiene un tipo, tomara ese tipo como único valor

Funciones

Funciones en TypeScript

Existen dos maneras de tipar funciones

- (…params: Parameters) => ReturnType
- nombreMetodo(…params: Parameters): ReturnType

Funciones en TypeScript

Existen dos maneras de tipar funciones

- (…params: Parameters) => ReturnType
- nombreMetodo(…params: Parameters): ReturnType

Nos vamos a centrar en la más conocida y usada

¿Cuántos argumentos le pasáis a las funciones?

Un único argumento

- A las funciones que generemos, idealmente, les pasaremos un único argumento
- Un objeto key: value que contenga lo que necesitemos

Nuestras funciones se verán algo como

- hasAccess({ userRoles, requiredRoles })
- capitalize(“string”)
- twoWays({ flag: true })
  - Para flags a veces prefiero reforzar el objeto por legibilidad

Objeto como argumento de función

- Tiene un pequeño impacto en el rendimiento
  - Como lo tienen las clases y crear instancias de helpers

Objeto como argumento de función

- Tiene un pequeño impacto en el rendimiento
  - Como lo tienen las clases y crear instancias de helpers
- Es más fácil de tipar, y asegurar de que esté bien tipado

Objeto como argumento de función

- Tiene un pequeño impacto en el rendimiento
  - Como lo tienen las clases y crear instancias de helpers
- Es más fácil de tipar, y asegurar de que esté bien tipado
- Reutilizar y componer tipos para otras funciones es más cómodo, y hasta semántico
  - El tipo de la función madre puede ser el resultado de lo que necesitan las funciones internas

Enums

Qué son los Enums

- Son enumeraciones
- Categorizaciones para un único valor
- Comúnmente representados como números

```
enum ValoracionTaller {
  pésimo = 0,
  decente = 1,
  increíble = 2,
}
```

Enums en TypeScript

Enums no oficiales

Target time: 5m

Demo

Experimentar con los diferentes enums

- Prueba a mostrarlos en consola

Enums en runtime

Los enums nativos de TypeScript se compilan a un objeto peculiar

Opcionales y defaults

Opcionales

Existen dos maneras de representar opcionales:

- Usar un interrogante
  - Valores que pueden *no* pasarse
- Usar “ | undefined”
  - Valores que *siempre* se pasarán, pero pueden no tener representación

Una opción es explícita pero forzosa, la otra puede ser imperceptible pero flexible

Defaults

Para una buena inferencia y genéricos, el default debería ser siempre *false* cuando hablemos flags

```
const Selection = ({ everything = false }) => </>
```

Aún así, defaults y genéricos no se llevan muy bien

Tipos especiales

Los internals de TypeScript

Hay muchos

Así que cubriremos los más usados y comúnes

void

No hay tipo, no hay nada

Una función puede devolver void, en más de una ocasión lo hará

Se pueden tipar props como void (y que no se le pasen nunca)

never

- Nunca se ha de llegar aquí
- Si una posibilidad llega a never, se devolverá error
- Usado para switches exhaustivos
  - Agotar todas las posibilidades de un enum/conjunto de literales en un switch

No pasar argumentos

Tenemos dos opciones para *prohibirlo*

- void
- never

{}

unknown

- No se sabe su tipo
- Pero se sabe que tiene tipo
  - No puede ser undefined
  - No puede ser null

any

- No usar any para salir del paso, salvo que sea estrictamente neceario

const

- Explicado en detalle más adelante
- Bastante común en soluciones type-safe, con literales y enums

as

Valor *as* {Type}

const

En TypeScript una keyword bastante mágica es *const*

Permite hacer una inferencia más cercana a la realidad. Son tipados más estrictos, pero porque su intención es facilitar el desarrollo.

Usados para devolver literales, objetos *narrowly typed*

as const

using

using

keyword bastante similar a *as*

Requiere de TypeScript >= v5

Cómo funciona

- Valor using {Type}
- Compara si el valor puede ser del {Type} proporcionado
  - De ser así, el tipo del valor pasa a ser el de {Type}

Caso de uso

Un tipo ambiguo

Se usa para poder especificar y comprobar que el valor puede ser y es del tipo que se espera

Diferencias con *as*

- *as* hace de paraguas, *using* intenta cercar el tipo lo máximo posible
- as puede usarse para mentir/engañar a TypeScript
  - A veces eso es precisamente lo que queremos

using

- Ejemplo de código

Operaciones

Tipo ambiguo

Props específicas

Con “|” podemos pedir ciertas props dependiendo de otra

Polimorfismo avanzado podríamos decirle

&

Literales

Literales

Los vamos a considerar tipos, aunque sean definidos en desarrollo

Literales abiertos

A veces podemos querer tipar literales, pero no restringir más opciones

La solución es:

“literal” | “otraOpción” | (string & {})

¿Quién sabría decirme por qué?

Porque

Prefijos y sufijos

Limitaciones de inputs

Comparación

Filosofía de comparación

De genérico a específico

Typescript funciona comprando de lo más genérico a lo más específico, no al revés.

El mínimo de especificidad se lo pones a la izquierda, si a la izquierda le pones algo muy específico ya no le va a gustar

Puede hacerse pasar por

El tipo de la derecha ¿dará error al hacerse pasar por el de la izquierda?

De más genérico a específico (moderadamente)

De más específico a genérico (suficientemente específico)

Serán ejemplos colaborativos, se agradece la participación

Puede hacerse pasar por

De más genérico a específico (moderadamente)

string

De más específico a genérico (suficientemente específico)

“literal”

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

string

De más específico a genérico (suficientemente específico)

✓ “literal”

string

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

✓ string

“literal”

De más específico a genérico (suficientemente específico)

✓ “literal”

✓ string

string

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

✓ string

× “literal”

“{number}-ex”

De más específico a genérico (suficientemente específico)

✓ “literal”

✓ string

× string

“tt-ex”

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

✓ string

× “literal”

× “{number}-ex”

number

De más específico a genérico (suficientemente específico)

✓ “literal”

✓ string

× string

× “tt-ex”

“0”

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

✓ string

× “literal”

× “{number}-ex”

× number

{}

De más específico a genérico (suficientemente específico)

✓ “literal”

✓ string

× string

× “tt-ex”

× “0”

{ foo: “bar” }

Puede hacerse pasar por

De más genérico a específico (moderadamente)

✓ string

✓ string

× “literal”

× “{number}-ex”

× number

✓ {}

unknown

De más específico a genérico (suficientemente específico)

✓ “literal”

✓ string

× string

× “tt-ex”

× “0”

✓ { foo: “bar” }

undefined

Puede hacerse pasar por

De más genérico a específico (moderadamente)

- ✓ string
- ✓ string
- × “literal”
- × “{number}-ex”
- × number
- ✓ {}
- × unknown

De más específico a genérico (suficientemente específico)

- ✓ “literal”
- ✓ string
- × string
- × “tt-ex”
- × “0”
- ✓ { foo: “bar” }
- × undefined

¿Alguna duda?

- ✓ string
- ✓ string
- ✗ “literal”
- ✗ “{number}-ex”
- ✗ number
- ✓ {}
- ✗ unknown

- ✓ “literal”
- ✓ string
- ✗ string
- ✗ “tt-ex”
- ✗ “0”
- ✓ { foo: “bar” }
- ✗ undefined

Type e Interface

Type

- Se comporta como *const*
- Herencia con union types

Interface

- Se comporta como *var*
- Usado en desarrollo de librerías
- Herencia

Diferencias

¿Cuándo usar cada uno?

Type vs Interface

- const
  - Sólo puede ser declarado una vez
- var
  - Puede declararse muchas veces

Type vs Interface

- const
  - Sólo puede ser declarado una vez
- Herencia con “|”

- var
  - Puede declararse muchas veces
- Herencia con extends y redeclaración

Type vs Interface

- const
  - Sólo puede ser declarado una vez
- Herencia con “|”
- **Declaración como variable**

- var
  - Puede declararse muchas veces
- Herencia con extends y redeclaración
- **Declaración como clase**

Type vs Interface

- const
  - Sólo puede ser declarado una vez
- Herencia con “|”
- Declaración como variable
- Mejor para aplicaciones

- var
  - Puede declararse muchas veces
- Herencia con extends y redeclaración
- Declaración como clase
- Mejor para librerías

Type vs Interface

- const
  - Sólo puede ser declarado una vez
- Herencia con “|”
- Declaración como variable
- Mejor para aplicaciones
- Herencia multiple con union

- var
  - Puede declararse muchas veces
- Herencia con extends y redeclaración
- Declaración como clase
- Mejor para librerías
- [A COMPROBAR] Una herencia a la vez

Utility Types

Pick

Omit

Exclude

ReturnType

Parameters

Array

- Para denotar arrays tenemos dos opciones, any[] o Array<any>

Corchetes -> más universal, puede llevar a error de formateo

- string | number[], si quisieses un array de ambos sería
  - (string | number)[]

Utility Type -> más legible, no tan universal ni semántico

- El ejemplo de antes sería Array<string | number>

tsconfig.json

Inversión de tiempo

Salvo desarrollo de librerías, la mayoría de aplicaciones deberían usar configuraciones muy similares

strict

Debería ir siempre a *true*

JavaScript con modo estricto lanzará errores cuando las cosas no funcionen

- El por defecto es no quejarse pero tampoco funcionar

noUncheckedIndexedAccess

Debería ir siempre a *true*

Recuerda que para acceder a un índice a mano, array[0], antes tendrás que comprobar si existe.

Por algún misterio desconocido, esta opción está por default a *false*

TotalTypescript

- https://www.totaltypescript.com/tsconfig-cheat-sheet
- https://github.com/total-typescript/tsconfig

Module declarations

.d.ts

Estructura del fichero

vite-env.d.ts

Target time: 5m

Demo

Cómo extender un tipo de una librería, readaptarlo o hacerlo más abierto

Extensiones y su conversión

- TypeScript .ts -> .js
- Common TypeScript .cts -> .cjs
  - Usado con node, para CLIs
- .mts -> .mjs
  - Exportación a módulos, no retrocompatible con todos los navegadores

Caso de uso

Tipar código antiguo

Preguntas

Descanso

Vuelta del descanso

¿Qué nos queda?

- Genéricos
- Inferencias
- JSDoc
- Utilidades
- Zod

¿Qué nos queda?

- Genéricos
- Inferencias
- JSDoc
- Utilidades
- Zod

Poco pero contundente

Genéricos

Genéricos

- Qué es un genérico
- Operador diamante
- Filosofía de los genéricos

Qué es un genérico

Cuándo se dan los casos de uso

- Un genérico no siempre hace falta
- Cuando los aprendemos, es común querer ponerlos en todos lados

Genéricos usando *as*

Filosofía de genéricos

- Cómo aplicarlos
- Cuándo
- Qué propósito cumplen

Las funciones reciben parámetros

- Algunos parámetros son valores
- Otros serán los tipos, a estos se les llama genéricos

Los tipos también pueden recibir genéricos

Estático vs Genérico (dinámico)

- inferencias

Tipados estáticos

- Idea general de cómo es la realidad

Tipados con genéricos

- Representan más cercanamente la realidad
- Se pueden adaptar, y se adaptan, a la realidad de cada caso de uso

Inferencias

La verdadera magia de TypeScript

Saber más – Inferencia

- Qué es una inferencia y qué es inferir
- De dónde viene
- En qué campos más se usa

Inferencias en TypeScript

Cuándo se infiere un valor

- Los tres casos, vídeo de Matt Pocock
- Función sin tipar

Tipado de retornos, ¿cuándo?

GenericBags

Para cuando tienes demasiados genéricos

A veces tendrás que pasar varios genéricos, en un orden específico, y luego vendrán los opcionales

- Usa un generic bag para que sea más fácil de consumir

Target time: 10m

Demo

GenericBags

- Cómo se crean y utilizan
- Cuándo usarlo

infer

Inferencia por uso

El objetivo es usar TypeScript para tener que usarlo lo menos posible

JSDoc

Cómo documentar código en JavaScript

Y tiparlo **nativamente**

Qué es JSDoc

```
/**
 *
 */
```

¿Se pueden documentar tipos en TypeScript?

Para saber más

https://tsdoc.org/

Casos de uso

- Desarrollo y mantenimiento de librerías
  - Svelte
- DDH (Ruby on Rails)
  - El Elon Musk de la programación
- Saltarte paso de build

@var {Type} name

@returns {Type}

@author y @source

Conceptos avanzados

Referencias a otros tipos

Genéricos

Inferencia

Target time: 10m

Demo

- Cómo funciona JSDoc, e integración con VSCode
- Cuánto de efectivo es
- ¿Es nativo? ¿Lo soportarán los navegadores?

Utilidades

Etiquetas de comentarios

- @ts-check
- @ts-nocheck
- @ts-ignore
- @ts-expect-error

Etiquetas de comentarios

- @ts-check
  - Comprueba TypeScript en ficheros de JavaScript
- @ts-nocheck
  - No comprueba TypeScript en el fichero
- @ts-ignore
  - No comprueba la siguiente línea
- @ts-expect-error
  - Esperará un error en la siguiente línea
  - Si no lo hay, dará un error (de ts, no de js)

Prettify

ObjectValues

Extensión de lookup

- Twoslash Query Comments
- // ^?
  - Donde sea que pongas el ^? te chivará su tipo

Retorno con keys dinámicas

- Artículo medium

Siempre podrás experimentar

En tsplay.dev

Zod

Validaciones en runtime

Qué es Zod

Validador y luego tipado

- Type-safe
- Realidad y dev-time iguales

Zod permite crear primero el validador (runtime) y de ese validador inferir el tipo (devtime)

¿Qué es type-safe?

type-safe

- Lo que representa el tipo será, seguramente, lo que represente el valor
  - Es decir, el tipo representa correctamente el valor en runtime
- Está bien tipado, es específico, no genérico
- Habilita el uso de tipos y los usa bien

Última demo

Target time: 15m

Demo

- Tipado recursivo de un objeto con anidaciones
- Concatenar keys en una única string (literal)

i18n type-safe

- Puedes tener tus lenguajes completamente type-safe
- Incluso para varios idiomas a la vez

Usando declaraciones de módulo y extendiendo cierta interfaz:

i18next.com/overview/typescript#create-a-declaration-file

Bonus

Referencias

TotalTypescript y Matt Pocock

Theo Browne (t3dotgg)

Creador de Contenido

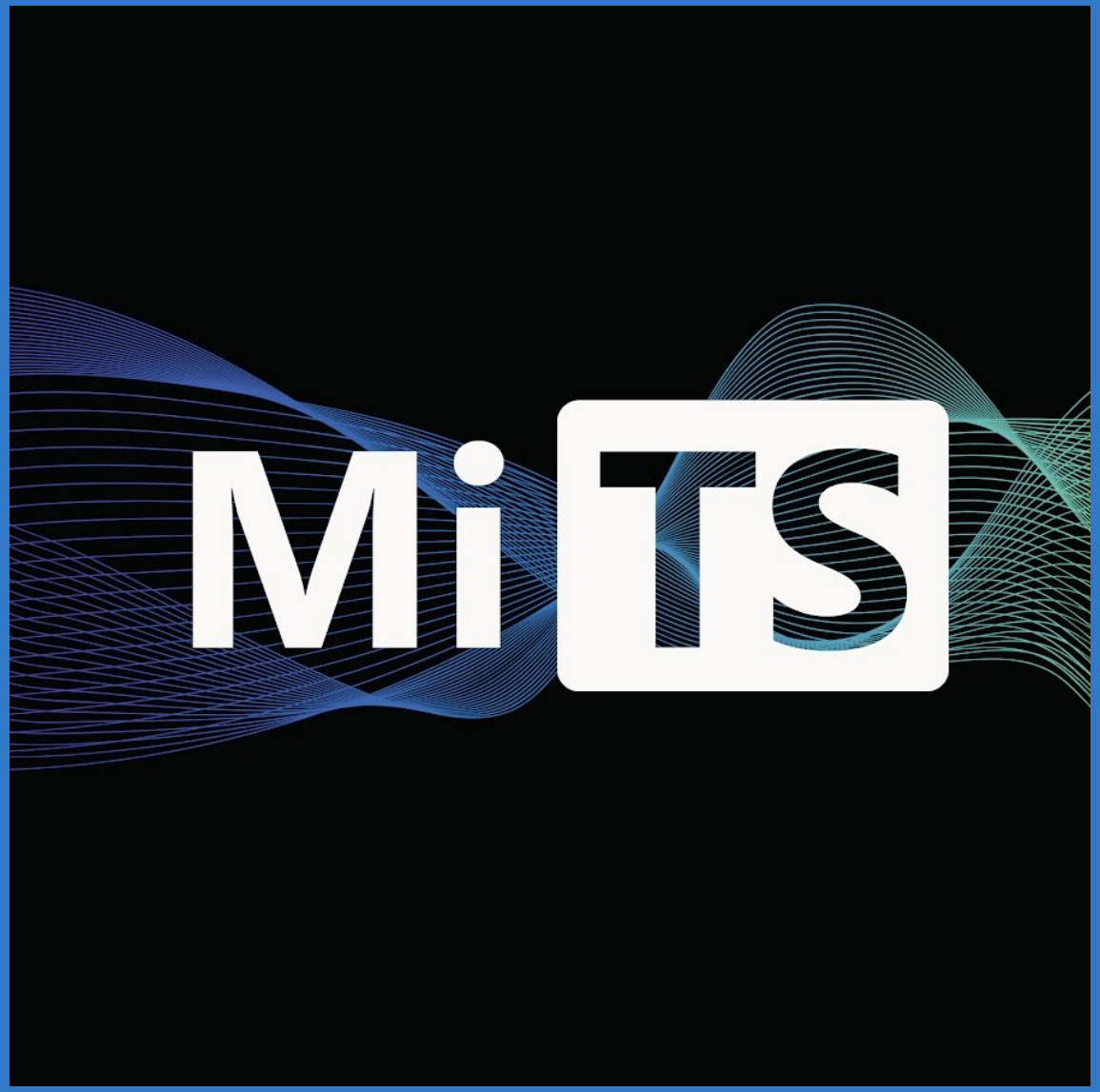
Ex-Twitch y Ex-Amazon

Tanner Linsley

Open-Source, creador de React Table, React Query y muchos más

Michigan TypeScript

Canal y comunidad



Librería de utilidades

ts-toolbelt

TypeChallenges

Retos increcendo de TypeScript

Leetcode pero de tipados

TypeHero

Advent of TypeScript

Advent of JS

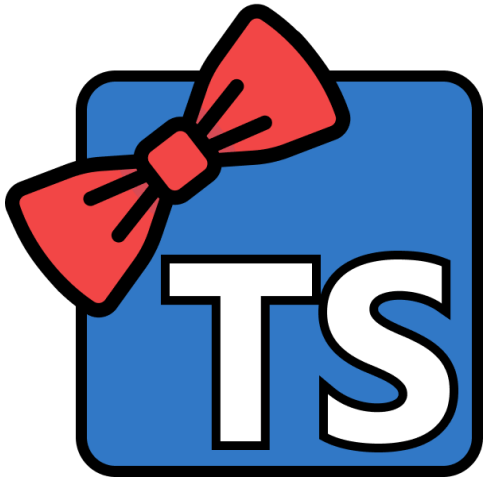
Advent of Code de midu.dev

Permite soluciones con TypeScript

Extensiones

Pretty TypeScript Errors

- Los errores de TypeScript no son los más fáciles de leer
- No sólo simplifica errores de TypeScript, sino todo tipo de errores en el Visual Studio Code



Error Lens

- Lee errores a nivel de línea
- Señala de una manera más clara las líneas erróneas (incluso con TypeScript)

TotalTypescript

- Te va dando pistas y tips de aprendizaje de TypeScript en el código

IDEs

Visual Studio Code

- Con extensiones
- De Microsoft

WebStorm

- Previo pago
- De JetBrains

Terminal

- Se pueden configurar plugins y LSPs, pero es más específico y no hay recomendaciones exactas *out of the box*

Algunas opciones podrían ser:

- NVim
- Vim
- Emacs

Conclusiones

QR de las slides

/talks-about/workshops/typescript/

Encuéntrame en

- LinkedIn - https://www.linkedin.com/in/jofaval/
- Github - https://github.com/jofaval

Preguntas

Workshop de TypeScript

Gracias por la atención