HOW TO MAKE TYPESCRIPT WORK FOR YOU

THE BASICS

THE BASICS

- TS compiles to JS

THE BASICS

- TS compiles to JS
- Types have *no effect* at runtime!

THE BASICS

- TS compiles to JS
- Types have *no effect* at runtime!
- Type annotations are just assumptions

TS'S HELP IS ONLY AS GOOD AS YOUR ASSUMPTIONS!

HOW TO AVOID ASSUMPTIONS

- NEVER use any
- Avoid type casting
- Verify I/O data at runtime

ONCE ASSUMPTIONS ARE GONE...

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?
  - Yes → add runtime checks!

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?
  - Yes → add runtime checks!
  - No → Improve types

NEVER USE **any**

NEVER USE any

```
type User = {
  name: string
  email: string
}

const getUser = async () => {
  const response = await fetch('api/user-data')
  return await response.json()
}
```

NEVER USE any

```
type User = {
  name: string
  email: string
}

const getUser = async (): Promise<User> => {
  const response = await fetch('api/user-data')
  return await response.json()
}
```

NEVER USE any

```
type User = {
  name: string
  email: string
}

const getUser = async (): Promise<User> => {
  const response = await fetch('api/user-data')
  return await response.json()
}
```

NEVER USE any

```
type User = {
  name: string
  email: string
}

const getUser = async (): Promise<User> => {
  const response = await fetch('api/user-data')
  return (await response.json()) as User
}
```

AVOID TYPE CASTING

AVOID TYPE CASTING

```ts
// command.ts
export type Command = {
  name: string
  alias?: string
}

// other stuff
```

```ts
// addCommand.ts

export default {
  mane: 'add',
}
```

AVOID TYPE CASTING

```ts
// command.ts
export type Command = {
  name: string
  alias?: string
}

// other stuff
```

```ts
// addCommand.ts

export default {
  name: 'add',
} as Command
```

AVOID TYPE CASTING

```ts
// command.ts
export type Command = {
  name: string
  alias?: string
}

// other stuff
```

```ts
// addCommand.ts

export default {
  name: 'add',
  altName: 'a',
} as Command
```

AVOID TYPE CASTING

```ts
type Something = {
  onlyProperty: number
}

const foo: Something = {
  onlyProperty: 42,
  additional: 'nope',
}
// Object literal may only
// specify known properties
```

```ts
// addCommand.ts

export default {
  name: 'add',
  altName: 'a',
} as Command
```

AVOID TYPE CASTING

```ts
type Something = {
  onlyProperty: number
}

const foo: Something = {
  onlyProperty: 42,
  additional: 'nope',
}
// Object literal may only
// specify known properties
```

```ts
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
}

export default add
```

AVOID TYPE CASTING

```ts
// addCommand.ts

export default {
  name: 'add',
  alias: 'a',
} satisfies Command
```

```ts
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
}

export default add
```

AVOID TYPE CASTING

```ts
// addCommand.ts

export default {
  name: 'add',
  alias: 'a',
} satisfies Command
/*
 * => {
 *   name: string
 *   alias: string
 * }
 */
```

```ts
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
}

export default add
/*
 * => {
 *   name: string
 *   alias?: string
 * }
 */
```

AVOID TYPE CASTING

```ts
// addCommand.ts

export default {
  name: 'add',
  alias: 'a',
} satisfies Command
/*
 * => {
 *   name: string
 *   alias: string
 * }
 */
```

```ts
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
}

export default add
/*
 * => {
 *   name: string
 *   alias?: string
 * }
 */
```

AVOID TYPE CASTING

```ts
// addCommand.ts

export default {
  name: 'add',
  alias: 'a',
} as const satisfies Command
/*
 * => {
 *   name: "add"
 *   alias: "a"
 * }
 */
```

```ts
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
} as const

export default add
/*
 * => {
 *   name: string
 *   alias?: string
 * }
 */
```

VERIFY I/O DATA AT RUNTIME

VERIFY I/O DATA AT RUNTIME

```
1  const verifyUser = (input: unknown): User => {
2    if (typeof input !== 'object' || input === null) {
3      throw new Error('user needs to be an object')
4    }
5
6    const result: User = {
7      name: '',
8      email: '',
9    }
10
11   if (
12     !('name' in input) ||
13     typeof input.name !== 'string'
14   ) {
```

VERIFY I/O DATA AT RUNTIME

```
1  const verifyUser = (input: unknown): User => {
2    if (typeof input !== 'object' || input === null) {
3      throw new Error('user needs to be an object')
4    }
5
6    const result: User = {
7      name: '',
8      email: '',
9    }
10
11   if (
12     !('name' in input) ||
13     typeof input.name !== 'string'
14   ) {
```

VERIFY I/O DATA AT RUNTIME

```
  const result: User = {
    name: '',
    email: '',
  }

  if (
    !('name' in input) ||
    typeof input.name !== 'string'
  ) {
    throw new Error('missing name of type string')
  } else {
    result.name = input.name
  }

```

VERIFY I/O DATA AT RUNTIME

```
    result.name = input.name
  }

  if (
    !('email' in input) ||
    typeof input.email !== 'string'
  ) {
    throw new Error('missing name of type string')
  } else {
    result.email = input.email
  }

  return result
}
```

VERIFY I/O DATA AT RUNTIME

```
1  import { z } from 'zod'
2
3  const userSchema = z.object({
4    name: z.string(),
5    email: z.string().email(),
6  })
7  type User = z.infer<typeof userSchema>
8
9  const getUser = async () => {
10   const response = await fetch('api/user-data')
11   const fromJson = await response.json()
12   return userSchema.parse(fromJson)
13 }
14
```

VERIFY I/O DATA AT RUNTIME

```zod
import { z } from 'zod'

const userSchema = z.object({
  name: z.string(),
  email: z.string().email(),
})
type User = z.infer<typeof userSchema>

const getUser = async () => {
  const response = await fetch('api/user-data')
  const fromJson = await response.json()
  return userSchema.parse(fromJson)
}

```

VERIFY I/O DATA AT RUNTIME

```
4    name: z.string(),
5    email: z.string().email(),
6  })
7  type User = z.infer<typeof userSchema>
8
9  const getUser = async () => {
10   const response = await fetch('api/user-data')
11   const fromJson = await response.json()
12   return userSchema.parse(fromJson)
13 }
14
15 const parseResult = userSchema.safeParse({
16   name: 1,
17   email: 'not an email!',
18 })
```

VERIFY I/O DATA AT RUNTIME

```
10    const response = await fetch('api/user-data')
11    const fromJson = await response.json()
12    return userSchema.parse(fromJson)
13  }
14
15  const parseResult = userSchema.safeParse({
16    name: 1,
17    email: 'not an email!',
18  })
19
20  parseResult.data // <- User | undefined
21  parseResult.error // <- ZodError | undefined
22
23  if (parseResult.success) {
```

VERIFY I/O DATA AT RUNTIME

```
14
15 const parseResult = userSchema.safeParse({
16   name: 1,
17   email: 'not an email!',
18 })
19
20 parseResult.data // <- User | undefined
21 parseResult.error // <- ZodError | undefined
22
23 if (parseResult.success) {
24   parseResult.data // <- User
25   parseResult.error // <- undefined
26 } else {
27   parseResult.data // <- undefined
```

VERIFY I/O DATA AT RUNTIME

```
19
20 parseResult.data // <- User | undefined
21 parseResult.error // <- ZodError | undefined
22
23 if (parseResult.success) {
24   parseResult.data // <- User
25   parseResult.error // <- undefined
26 } else {
27   parseResult.data // <- undefined
28   parseResult.error // <- ZodError
29
30   parseResult.error.issues
31   /*
32       [
```

VERIFY I/O DATA AT RUNTIME

```
28  parseResult.error // <- ZodError
29
30  parseResult.error.issues
31  /*
32      [
33        {
34          code: "invalid_type",
35          expected: "string",
36          received: "number",
37          path: [ "name" ],
38          message: "Expected string, received number"
39        },
40        {
41          validation: "email",
```

VERIFY I/O DATA AT RUNTIME

```
37            path: [ "name" ],
38            message: "Expected string, received number"
39          },
40          {
41            validation: "email",
42            code: "invalid_string",
43            message: "Invalid email",
44            path: [ "email" ]
45          }
46        ]
47        */
48
49    const errorsByProperty = parseResult.error.format()
50    errorsByProperty.name?._errors
```

VERIFY I/O DATA AT RUNTIME

```
41            validation: "email",
42            code: "invalid_string",
43            message: "Invalid email",
44            path: [ "email" ]
45          }
46        ]
47        */
48
49    const errorsByProperty = parseResult.error.format()
50    errorsByProperty.name?._errors
51    // -> ["Expected string, received number"]
52    errorsByProperty.email?._errors
53    // -> ["Invalid email"]
54  }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
// Attention: this may only ever be either CSV,
// one of the standard options or the config form data
type UserInput = {
  csv?: string
  providedStandardOptionId?: string

  // the rest are input values from the config form
  start?: Date
  end?: Date
  values?: number[]
}
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
13 let userInput: UserInput = {}
14
15 const setCsv = (csv: string) => {
16   userInput = { csv }
17 }
18
19 const setConfigData = (start: Date, end: Date) => {
20   userInput = { start, end }
21 }
22
23 const setStandardOption = (optionId: string) => {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
26
27 const getInputType = (userInput: UserInput): string => {
28   if (userInput.csv !== undefined) {
29     return 'csv'
30   } else if (
31     userInput.providedStandardOptionId !== undefined
32   ) {
33     return 'standardOption'
34   } else {
35     return 'configForm'
36   }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
14
15 const setCsv = (csv: string) => {
16   userInput = { csv }
17 }
18
19 const setConfigData = (start: Date, end: Date) => {
20   userInput = { start, end }
21 }
22
23 const setStandardOption = (optionId: string) => {
24   userInput = { providedStandardOptionId: optionId }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
34    } else {
35      return 'configForm'
36    }
37  }
38
39  // Problem: this type-checks
40  const everything: UserInput = {
41    csv: '',
42    providedStandardOptionId: '',
43    start: new Date(),
44    end: new Date(),
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
type UserInput =
  | {
      csv: string
    }
  | {
      providedStandardOptionId: string
    }
  | {
      start: Date
      end: Date
      values: number[]
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
14 let userInput: UserInput | undefined
15
16 const setCsv = (csv: string) => {
17   userInput = { csv }
18 }
19
20 const setConfigData = (
21   start: Date,
22   end: Date,
23   values: number[],
24  ) => {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
30 }
31
32 const getInputType = (userInput: UserInput): string => {
33   if ('csv' in userInput) {
34     return 'csv'
35   } else if ('providedStandardOptionId' in userInput) {
36     return 'standardOption'
37   } else {
38     return 'configForm'
39   }
40 }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
37    } else {
38      return 'configForm'
39    }
40  }
41
42  // Problem: this type-checks
43  const everything: UserInput = {
44    csv: '',
45    providedStandardOptionId: '',
46    start: new Date(),
47    end: new Date(),
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
type UserInput =
  | {
      type: 'csv'
      csv: string
    }
  | {
      type: 'standardOption'
      providedStandardOptionId: string
    }
  | {
      type: 'configForm'
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
17 let userInput: UserInput | undefined
18
19 const setCsv = (csv: string) => {
20   userInput = { type: 'csv', csv }
21 }
22
23 const setConfigData = (
24   start: Date,
25   end: Date,
26   values: number[],
27  ) => {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
33      type: 'standardOption',
34      providedStandardOptionId: optionId,
35    }
36  }
37
38  const getInputType = (userInput: UserInput): string => {
39    return userInput.type
40  }
41
42  // This doesn't type-check -> problem solved
43  // @ts-expect-error
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
38 const getInputType = (userInput: UserInput): string => {
39   return userInput.type
40 }
41
42 // This doesn't type-check -> problem solved
43 // @ts-expect-error
44 const everything: UserInput = {
45   csv: '',
46   providedStandardOptionId: '',
47   start: new Date(),
48   end: new Date(),
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
11        case 'default':
12          values = [1, 2, 3]
13          break
14        case 'special':
15          values = [42, 42, 42, 42]
16          break
17        default:
18          values = []
19      }
20    }
21
```

DEALING WITH DISCRIMINATED UNIONS

```
const sumValues = (userInput: UserInput): number => {
  let values: number[] = []
  if (userInput.type === 'configForm') {
    values = userInput.values
  } else if (userInput.type === 'csv') {
    values = userInput.csv
      .split(',')
      .map((n) => parseInt(n))
  } else {
    switch (userInput.providedStandardOptionId) {
      case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
const sumValues = (userInput: UserInput): number => {
  let values: number[] = []
  if (userInput.type === 'configForm') {
    values = userInput.values
  } else if (userInput.type === 'csv') {
    values = userInput.csv
      .split(',')
      .map((n) => parseInt(n))
  } else {
    switch (userInput.providedStandardOptionId) {
      case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
8         .map((n) => parseInt(n))
9     } else {
10      switch (userInput.providedStandardOptionId) {
11        case 'default':
12          values = [1, 2, 3]
13          break
14        case 'special':
15          values = [42, 42, 42, 42]
16          break
17        default:
18          values = []
```

DEALING WITH DISCRIMINATED UNIONS

```
const sumValues = (userInput: UserInput): number => {
  let values: number[] = []
  if (userInput.type === 'configForm') {
    values = userInput.values
  } else if (userInput.type === 'csv') {
    values = userInput.csv
      .split(',')
      .map((n) => parseInt(n))
  } else {
    switch (userInput.providedStandardOptionId) {
      case 'default':
```

PATTERN MATCHING

```ts
import { UserInput } from './userInput'
import { P, match } from 'ts-pattern'

const sumValues = (userInput: UserInput): number => {
  const values = match(userInput)
    .with(
      { type: 'configForm' },
      (userInput) => userInput.values,
    )
    .with({ type: 'csv' }, ({ csv }) =>
      csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```ts
import { UserInput } from './userInput'
import { P, match } from 'ts-pattern'

const sumValues = (userInput: UserInput): number => {
  const values = match(userInput)
    .with(
      { type: 'configForm' },
      (userInput) => userInput.values,
    )
    .with({ type: 'csv' }, ({ csv }) =>
      csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
 5   const values = match(userInput)
 6     .with(
 7       { type: 'configForm' },
 8       (userInput) => userInput.values,
 9     )
10     .with({ type: 'csv' }, ({ csv }) =>
11       csv.split(',').map((n) => parseInt(n)),
12     )
13     .with(
14       {
15         type: 'standardOption',
```

PATTERN MATCHING

```
13      .with(
14        {
15          type: 'standardOption',
16          providedStandardOptionId: P.select(),
17        },
18        (optionId) => {
19          switch (optionId) {
20            case 'default':
21              return [1, 2, 3]
22            case 'special':
23              return [42, 42, 42, 42]
```

PATTERN MATCHING

```
20            case 'default':
21              return [1, 2, 3]
22            case 'special':
23              return [42, 42, 42, 42]
24            default:
25              return []
26          }
27        },
28      )
29      .exhaustive()
30
```

PATTERN MATCHING

```ts
import { UserInput } from './userInput'
import { P, match } from 'ts-pattern'

const sumValues = (userInput: UserInput): number => {
  const values = match(userInput)
    .with(
      { type: 'configForm' },
      (userInput) => userInput.values,
    )
    .with({ type: 'csv' }, ({ csv }) =>
      csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
13      .with(
14        {
15          type: 'standardOption',
16          providedStandardOptionId: P.select(),
17        },
18        (optionId) => {
19          switch (optionId) {
20            case 'default':
21              return [1, 2, 3]
22            case 'special':
23              return [42, 42, 42, 42]
```

PATTERN MATCHING

```
 1 import { UserInput } from './userInput'
 2 import { match } from 'ts-pattern'
 3
 4 const sumValues = (userInput: UserInput): number => {
 5   const values = match(userInput)
 6     .with(
 7       { type: 'configForm' },
 8       (userInput) => userInput.values,
 9     )
10     .with({ type: 'csv' }, ({ csv }) =>
11       csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
17        },
18        () => [1, 2, 3],
19      )
20      .with(
21        {
22          type: 'standardOption',
23          providedStandardOptionId: 'special',
24        },
25        () => [42, 42, 42, 42],
26      )
27      .otherwise(() => [])
```

PATTERN MATCHING

```
18          () => [1, 2, 3],
19        )
20        .with(
21          {
22            type: 'standardOption',
23            providedStandardOptionId: 'special',
24          },
25          () => [42, 42, 42, 42],
26        )
27        .otherwise(() => [])
28
```

PATTERN MATCHING

```ts
import { UserInput } from './userInput'
import { P, match } from 'ts-pattern'

const sumValues = (userInput: UserInput): number => {
  const values = match(userInput)
    .with(
      { type: 'configForm' },
      (userInput) => userInput.values,
    )
    .with({ type: 'csv' }, ({ csv }) =>
      csv.split(',').map((n) => parseInt(n)),
```

UTILITY TYPES

UTILITY TYPES

```
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = {
  name?: User['name']
  age?: User['age']
  foodPreference?: User['foodPreference']
}
```

UTILITY TYPES

```
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
```

UTILITY TYPES

```ts
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
type ComplicatedUser = Required<UserUpdate>

type WithoutFood = Omit<User, 'foodPreference'>
type WithoutFood2 = Pick<User, 'name' | 'age'>

type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```ts
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
type ComplicatedUser = Required<UserUpdate>

type WithoutFood = Omit<User, 'foodPreference'>
type WithoutFood2 = Pick<User, 'name' | 'age'>

type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```ts
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
type ComplicatedUser = Required<UserUpdate>

type WithoutFood = Omit<User, 'foodPreference'>
type WithoutFood2 = Pick<User, 'name' | 'age'>

type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```ts
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
type ComplicatedUser = Required<UserUpdate>

type WithoutFood = Omit<User, 'foodPreference'>
type WithoutFood2 = Pick<User, 'name' | 'age'>

type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```ts
type User = {
  name: string
  age: number
  foodPreference: 'none' | 'vegetarian' | 'vegan'
}

type UserUpdate = Partial<User>
type ComplicatedUser = Required<UserUpdate>

type WithoutFood = Omit<User, 'foodPreference'>
type WithoutFood2 = Pick<User, 'name' | 'age'>

type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```
const magicFunction = (a: number, b: string) => {
  return {
    foo: 42,
    bar: {
      biz: 'baz',
    },
  }
}

type Result = ReturnType<typeof magicFunction>
type Params = Parameters<typeof magicFunction>
// => [a: number, B; string]
```

UTILITY TYPES

```ts
const magicFunction = (a: number, b: string) => {
  return {
    foo: 42,
    bar: {
      biz: 'baz',
    },
  }
}

type Result = ReturnType<typeof magicFunction>
type Params = Parameters<typeof magicFunction>
// => [a: number, B; string]
```

UTILITY TYPES

```
const magicFunction = (a: number, b: string) => {
  return {
    foo: 42,
    bar: {
      biz: 'baz',
    },
  }
}

type Result = ReturnType<typeof magicFunction>
type Params = Parameters<typeof magicFunction>
// => [a: number, B; string]
```

type hero BETA

Explore Tracks Advent of TypeScript New

Login

LIVE DEMO 🤞

TAKE-AWAYS

- Never use any
- Types should always match reality
  - Make illegal values unrepresentable
  - Don't assume (= cast), parse / validate!
  - If a check happens at runtime, let TypeScript know about it

TAKE-AWAYS

- Never use any
- Types should always match reality
  - Make illegal values unrepresentable
  - Don't assume (= cast), parse / validate!
  - If a check happens at runtime, let TypeScript know about it
- Compiler errors are bugs you don't have to catch - appreciate them!

THANK YOU!

Michael Kuckuk

@LBBO@ruhr.social

inovex