Type Guards & User-Defined Type Predicates

Safe API Handling with TypeScript

Why Do We Need Type Guards?

- APIs often return unions: success or error
- TypeScript can’t know the shape at runtime
- We need a way to safely access properties
- Type guards let us narrow types at runtime
- User-defined predicates unlock full type safety

The Problem: Unsafe Access

- What happens if we access properties without checking the type?
- TypeScript will warn us, but runtime errors can still happen

```
type Pizza = {
  id: number
  name: string
  toppings: string[]
}

type ErrorResponse = {
  error: string
  code: number
}

function getPizzaApiResponse(): Pizza | ErrorResponse {
  return Math.random() > 0.5
    ? { id: 1, name: 'Margherita', toppings: ['cheese', 'tomato'] }
    : { error: 'Not found', code: 404 }
}

const response = getPizzaApiResponse()

console.log('Pizza name:', response.name)
```

The Solution: User-Defined Type Guards

- Write a function that checks the type at runtime
- Use a type predicate: `response is Pizza`
- TypeScript will narrow the type for us!

```ts
type Pizza = {
  id: number
  name: string
  toppings: string[]
}

type ErrorResponse = {
  error: string
  code: number
}

function isPizza(response: Pizza | ErrorResponse): response is Pizza {
  return Object.keys(response).includes('name')
}
// @ts-expect-error Example
function getPizzaApiResponse(): Pizza | ErrorResponse { /* Same as before */ }

const response = getPizzaApiResponse()

if (isPizza(response)) {
  console.log('Pizza name:', response.name)
}
else {
  console.log('API error:', response.error)
}
```

Why Does This Matter?

- Type guards make your code safer and more expressive
- User-defined predicates unlock advanced type narrowing
- Essential for robust API handling

Bonus: Discriminated Unions

- Add a `type` property to your API responses
- TypeScript can narrow types automatically
- Combine with type guards for exhaustive checks

```ts
type PizzaDiscriminated = {
  result: 'success'
  id: number
  name: string
  toppings: string[]
}

type ErrorResponseDiscriminated = {
  result: 'error'
  error: string
  code: number
}

type PizzaApiResponse = PizzaDiscriminated | ErrorResponseDiscriminated
// @ts-expect-error Example
function getPizzaApiResponseDiscriminated(): PizzaApiResponse { /* Same as before */ }

const response = getPizzaApiResponseDiscriminated()
switch (response.result) {
  case 'success':
    console.log('Pizza name:', response.name)
    break
  case 'error':
    console.log('API error:', response.error)
    break
}
```

Bonus-Bonus: Exhaustive Checks

- But what if we add another possibility for `result`?
- Exhhaustive checks have entered the room...

```ts
type PizzaDiscriminated = {
  result: 'success'
  id: number
  name: string
  toppings: string[]
}

type ErrorResponseDiscriminated = {
  result: 'error'
  error: string
  code: number
}

type ProgressResponseDiscriminated = {
  result: 'progress'
  progress: number
}

type PizzaApiResponse = PizzaDiscriminated | ErrorResponseDiscriminated
// @ts-expect-error Example
function getPizzaApiResponseDiscriminated(): PizzaApiResponse { /* Same as before */ }

const response = getPizzaApiResponseDiscriminated()

switch (response.result) {
  case 'success':
    console.log('Pizza name:', response.name)
    break
  case 'error':
```

Questions?

Ever used type guards or user-defined predicates? Any API horror stories?

Thank You!

TypeScript's type system is full of little surprises—keep exploring!