Delving deeper into TYPESCRIPT

Using utility types (and more)!

Bartek Polańczyk | 18/06/2024

Agenda

- TypeScript - the history
- Why, what, and where - the project we worked on
- The new architecture and the challenges
- Generic filters - how to obtain a proper complex type from just the keyword?
- Query types - how to transform the interface completely just with TS?
- A simple UI store - how to dispatch any action?
- Sindre Sorhus and Type Fest
- Did you know…?
- The summary

TypeScript

A bit of history

- If it were a child, it'd be in the 5th grade
- Created by Microsoft, the initial version - 0.8
- Created to deal with JS shortcomings (but still be a superset of JS)
- Initially, supported in MS Visual Studio only
- 1.0 released 2 years later, in 2014 (during Build)
  - Code made publicly available on GH
- Each stable release coming about 2 years apart
- Still no competition (Flow anyone?)

TypeScript

How did the library change over time

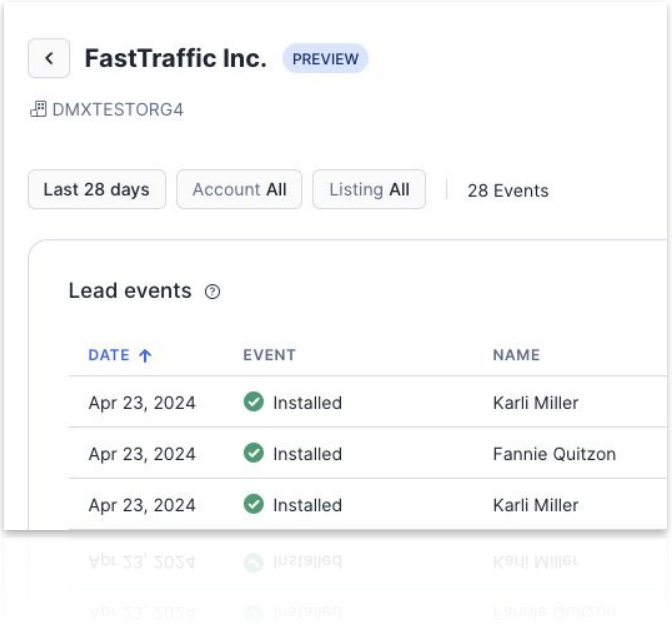
```
type TypeName<T> = T extends
string
 ? "string"
 : T extends number
 ? "number"
 : T extends boolean
 ? "boolean"
 : T extends undefined
 ? "undefined"
 : T extends Function
 ? "function"
 : "object";
```

- 12/04/2014 **(1.0)**
- 16/09/2015 **(1.6) JSX** support
- 22/09/2016 **(2.0)** null and undefined-aware types, control flow, discriminated unions, **this** (and in 2.8 - **infer** keyword)
- 30/07/2018 **(3.0)** mappable tuple and array types, spreading parameters (and **A LOT** of performance optimization)
- 19/11/2020 **(4.0)** improving template types, key remapping and **recursive conditional types**
  - In minor updates, **“satisfies”** operator, better async/await, ES modules and fixed type inference
- 16/03/2023 **(5.0)** ES decorators, better implicits, type narrowing, support for **groupBy** (just landed - 5.4!)

The new IA

Who are we? Why, what and where?

- Part of a new **Provider Experience** Snowflake domain
- Asking questions how can we help Snowflake providers better understand their listings (and revenue in the future)
- Building a new code from scratch
- Main goals of the new foundation:
  - Ease of use
  - Scalability
  - Modularity
- The **foundation** is (was?) TypeScript heavy, with a lot of generics



First, solve the problem. Then, write the code!

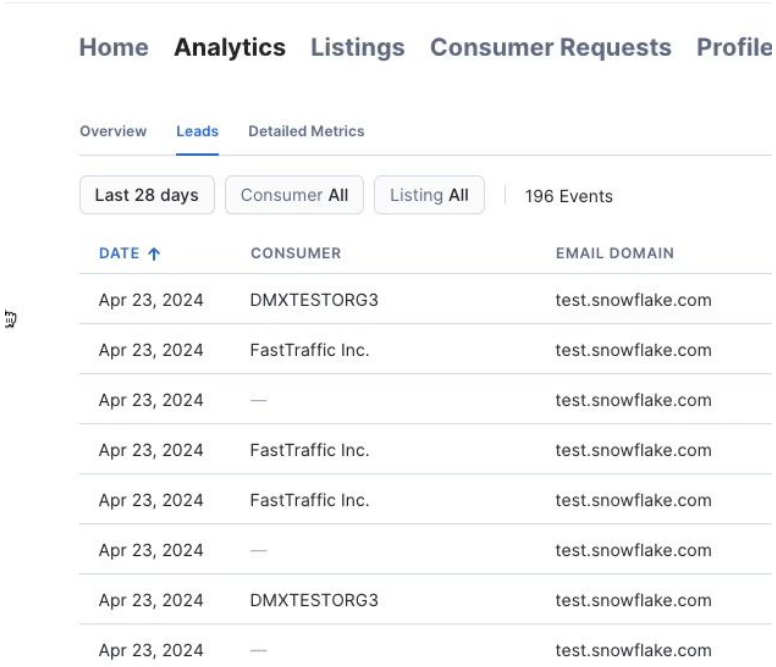
John Johnson

3 core problems

- **The filters**
  - A way of filtering the queries (and synchronizing with the URL)
  - TS areas: **generics, simple infer, mapped types, type casting**
- **The query engine**
  - Mapping SQL responses to TypeScript interfaces
  - Automatically creating camelCased interfaces from SQL field lists
  - TS areas: **complex type inference based on the shape, key format conversion**
- **The store**
  - Keeping the state of the UI in sync
  - Flux-like (for porting to Redux/RTKQ in the future)
  - TS areas: **extending generics, infer, mapped types, type inference from the code**

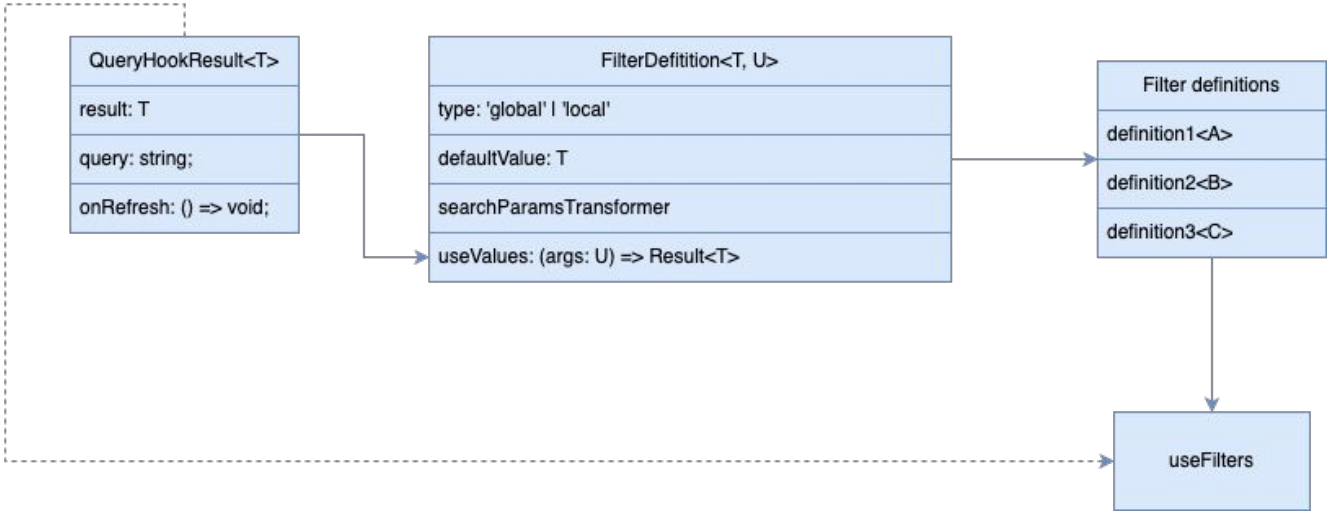
The filters

Getting just the right data

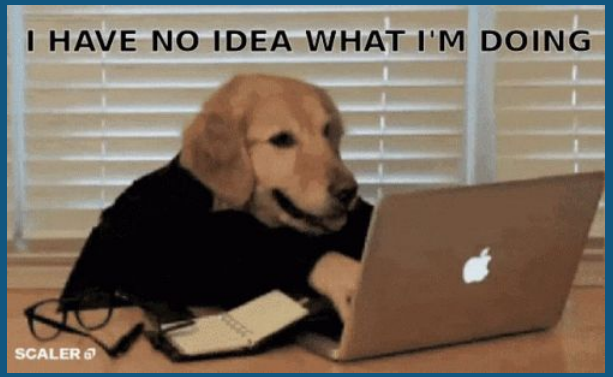


- What is a filter?
- How to define a filter?
- How to consume the filter definition?
- How to interact with filters?

How do filters work?



Coding time!



Filters - conclusions

- Sometimes “as” is a necessity (framework/library - OK-ish, UI code - not so much)
- TS still doesn’t play well with mapped types (intersection vs union for strict operation) - TypeScript/issues/30581
- TypeScript handles string-keyed types better than dynamically assuming the type from the context
- “Infer” helps to properly type cast generic shapes to a concrete type

The queries

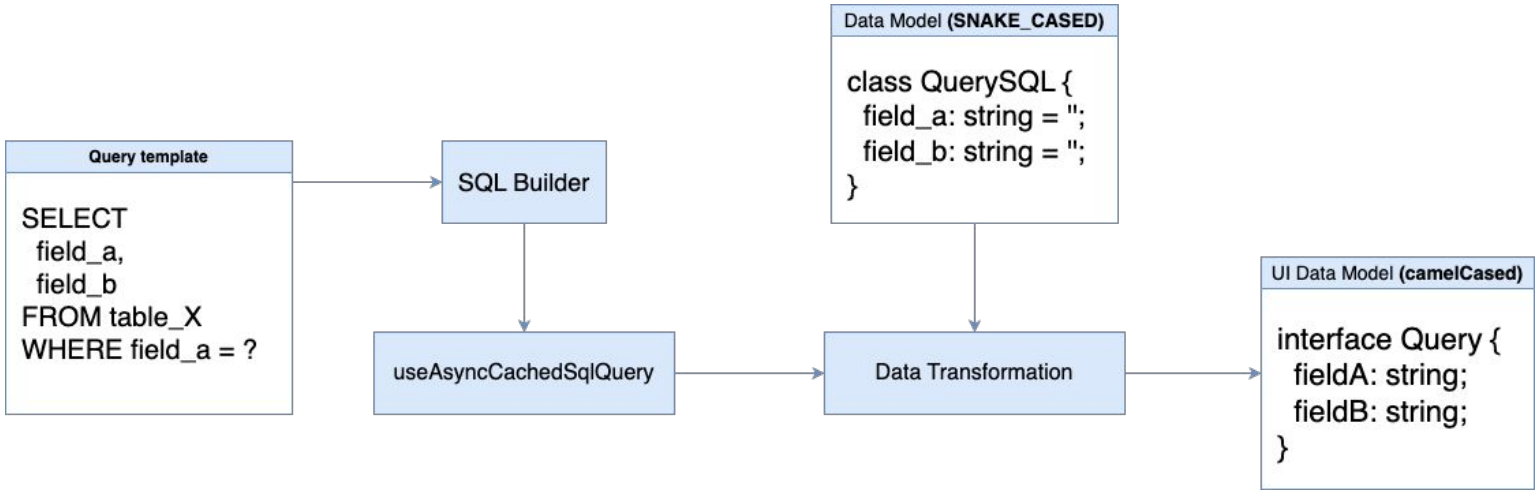
Being able to pass SQL results to the UI

```
1+ usages  Bartek Polanczyk +1
class LeadEventSQL {
  EVENT_DATE : Date  = new Date();
  EVENT_TYPE : string  = '';
  CONSUMER_ORGANIZATION : string  = '';
  CONSUMER_COMPANY_NAME : string  = '';
  CONSUMER_ACCOUNT_LOCATOR : string  = '';
  CONSUMER_EMAIL : string  = '';
  CONSUMER_FIRST_NAME : string  = '';
  CONSUMER_LAST_NAME : string  = '';
  LISTING_GLOBAL_NAME : string  = '';
  LISTING_DISPLAY_NAME : string  = '';
  ACCESS_TYPE : string  = '';
  SNOWFLAKE_REGION : string  = '';
}

export type LeadEvent = CamelCasedProperties<LeadEventSQL>;
```

- How to model SQL fields in the JS world?
- How to reduce the boilerplate to the max?
- How to generate UI-friendly interfaces out of SNAKE_CASEd information?

Transforming queries



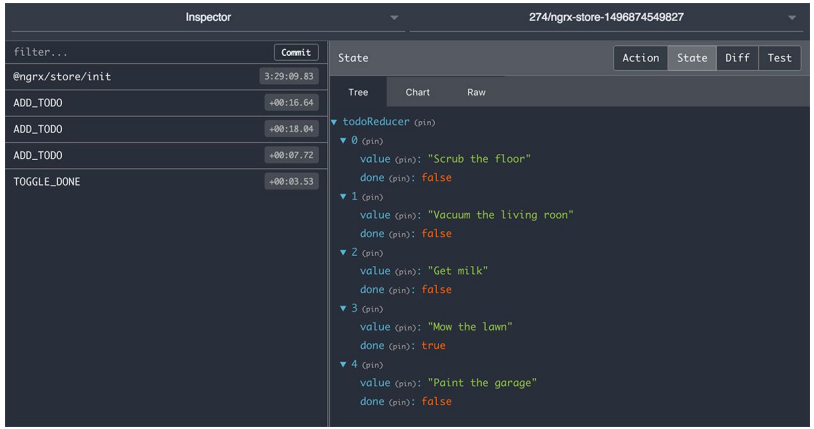
Coding time!

Queries - conclusions

- TypeScript is a powerful tool that enables devs to create powerful code
- Whole shapes of the objects can be “dynamically” transformed into each other, if properly defined
- There are TypeScript “quirks” that diverge from the standard ECMAScript implementation (looking at you, classes!) TypeScript/issues/34787
- Utility types facilitate such operations - they’re a solid basis for building complex transformations

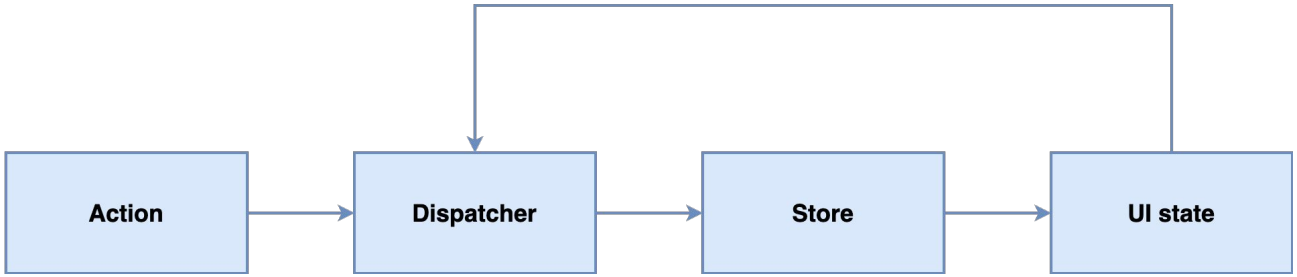
The store

Easily managing the UI state



- How to keep the state of the UI in sync?
- Flux-like (for porting to Redux/RTKQ in the future)
- Be able to define slices and actions in a strongly typed manner

Flux UI Store



Coding time!

UI store - conclusions

- “Infer” is a powerful tool that allows for getting the info about types extending a common base
- Building a custom flux-like store is not (very) complicated
- It’s hard to ditch type casting (“as”) in maps/reducers sometimes

Most useful built-in utility types (personal opinion)

Partial/Required/ Readonly

Used to change the access type to chosen properties of the object.

Pick/Omit/Exclude/ Extract

Change the shape of the object, returns new shape with different properties

Function utility types

Return different information about functions and classes (arguments, returned types, “this” context, instance types)

“New” types

“Awaited” - async/await

“NoInfer” - block “infer”

New String types - upper/lower/capitalize/ uncapitalize

Sindre Sorhus and his Type Fest

An **AMAZING** library of literally hundreds of utility types

Covers most of “strange” use cases (including some of the presented code!)

Saves **a lot of time** tinkering with TS!

https://github.com/sindresorhus/type-fest

API

Click the type names for complete docs.

Basic

- `Primitive` - Matches any primitive value.
- `Class` - Matches a `class`.
- `Constructor` - Matches a `class` constructor.
- `AbstractClass` - Matches an `abstract class`.
- `AbstractConstructor` - Matches an `abstract class` constructor.
- `TypedArray` - Matches any typed array, like `Uint8Array` or `Float64Array`.
- `ObservableLike` - Matches a value that is like an Observable.

Utilities

- `EmptyObject` - Represents a strictly empty plain object, the `{}` value.
- `NonEmptyObject` - Represents an object with at least 1 non-optional key.
- `UnknownRecord` - Represents an object with `unknown` value. You probably want this instead of `{}`.
- `UnknownArray` - Represents an array with `unknown` value.
- `Except` - Create a type from an object type without certain keys. This is a stricter version of `Omit`.
- `Writable` - Create a type that strips `readonly` from the given type. Inverse of `Readonly<T>`.
- `WritableDeep` - Create a deeply mutable version of an `object` / `ReadonlyMap` / `ReadonlySet` / `ReadonlyArray` type. The inverse of `ReadonlyDeep<T>`. Use `Writable<T>` if you only need one level deep.
- `Merge` - Merge two types into a new type. Keys of the second type overrides keys of the first type.
- `MergeDeep` - Merge two objects or two arrays/tuples recursively into a new type.
- `MergeExclusive` - Create a type that has mutually exclusive keys.
- `OverrideProperties` - Override only existing properties of the given type. Similar to `Merge`, but enforces that the original type has the properties you want to override.
- `RequireAtLeastOne` - Create a type that requires at least one of the given keys.
- `RequireExactlyOne` - Create a type that requires exactly a single key of the given keys and disallows more.
- `RequireAllOrNone` - Create a type that requires all of the given keys or none of the given keys.
- `RequireOneOrNone` - Create a type that requires exactly a single key of the given keys and disallows more, or none of the given keys.

Did you know?

TypeScript is Turing-complete!

```
type MyFunc<TArg> = {
  "true": TrueExpr<MyFunction, TArg>,
  "false": FalseExpr<MyFunc, TArg>
}[Test<MyFunc, TArg>];
```

```
type Operator = '+' | '-' |'*' | '/';

type Calc<A extends number, Op extends Operator, B extends number> = Op extends '+' ? (
    Add<A, B>
) : (
    Op extends '-' ? (
        Subtract<A, B>
    ) : (
        Op extends '*' ? (
            Multiply<A, B>
        ) : Divide<A, B>
    )
)

type FivePlusTwo = Calc<5, '+', 2>;
type FiveMinusThree = Calc<5, '-', 3>;
type FiveTimesFour = Calc<5, '*', 4>;
type EightByFour = Calc<8, '/', 4>;
```

- TypeScript may be considered Turing complete (since 2.2)
- Possible thanks to mapped types + recursion + index types
- The proof:
  - https://github.com/microsoft/TypeScript/issues/14833
- Another example - TypeScript calculator:
  - https://gist.github.com/MasterTuto/47ddcc240def44d32a4213b886a3993c

THANK YOU