## TYPAGE DE DONNÉES EXTERNES

Retour d'expériences



#### About me

Romain Deneau - @DeneauRomain

Senior Developer | .NET, Angular, 😍 TypeScript



Animateur communauté Craft





## **CONTEXTE**

- Données externes, non TypeScript :
  - Objets JSON en E/S d'une Web API
- Librairie d'infrastructure
  - S'interface avec l'API
  - httpClient Angular, \$ajax jQuery
- Types à spécifier à l'usage
  - httpClient.get<SomeDto>(url,
    args)



# CONTEXTE (2)

- Essais successifs sur différents projets
- Différentes stratégies utilisées pour le typage :
  - Typage côté API / côté client
  - Typage manuel / généré
  - Génération manuelle / automatisée



# TYPAGE GÉNÉRÉ CÔTÉ API

- Différents outils de génération C# →
   TS
  - TypeLITE
  - TypeWriter



## Typage généré avec TypeLITE

- http://type.litesolutions.net/
- Installation
  - NuGet dans projet C#
  - Ajout d'un fichier TypeLite.tt (template T4)
- Étapes de génération
  - Compilation de la solution .NET
  - Lancement du template T4 (design-time)
    - → Génération TypeLite.d.ts



#### TypeLITE: Bilan

- Typage complet (facades et dépendances) et correcte
- Reprise des namespaces C#
  - Pas de conflit de noms
  - X Longs → pas idiomatique TS/ES6
- X Génération manuelle
- X IDE compatible Template T4 (autre que VS)
- X Tous les types TS dans un seul fichier



## Typage généré avec TypeWriter

- http://frhagn.github.io/Typewriter/
- Installation:
  - Extension de Visual Studio
- Génération :
  - Écriture d'un Models.tst (TypeScript Template)
  - Compilation → Génération du/des fichiers TS



#### TypeWriter: Bilan

- Lancé à la compilation
  - X Uniquement dans Visual Studio
- Totalement configurable
  - ✓ Un fichier par type Modules ES6
  - Dépendance entre 2 types → import ES6
  - Génériques: Result<T> = { data: T[]
    }
- - Objets littéraux + Interfaces TypeScript
  - X Instances de classes ES6



## Autres générateurs côté API

- NSwag: Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript
- ToTypeScriptD: Generate TypeScript Definition files
   (\*.d.ts) from .NET assembly files.
- TypeScripter: class library for generating TypeScript definition files from .NET assemblies and types.
- Sûrement plein d'autres...



## Typage généré côté API - Bilan

- Typage complet, correcte, up-to-date
- Outils tiers et spécifiques C# → TS
- X Chaîne de build (API-SPA) + compliquée
- X Marche mieux voire qu'avec Visual Studio
- Reproducer driven
  - Typage au plus près de la source
  - X Producer concern → Consumer concern



## TYPAGE CÔTÉ CLIENT

- 🗗 Conditions
  - Disposer du contrat de l'API
  - Différents formats possibles
    - OpenAPI Spec, WSDL...
    - HTML, Wiki, Word...
- - API REST Pet Store avec Swagger
     UI



## Typage entièrement manuel



POST /pet : add a pet to the store

- Écriture manuelle des 3 types :
  - Pet et ses 2 types imbriqués Category et
     Tag
  - Démo



## Typage entièrement manuel: Bilan

- X Fastidieux 🔄
- X Trop sujet à erreur : typos...
- Ouvert à tout format du contrat
- Personnalisation des types
  - Ignorer un champ inutilisé...
- A Changement de version de l'API



## Typage semi-manuel "outillé"

Utiliser un convertisseur
JSON → TypeScript

- Extension VS code: JSON to TS
  - "Convert from clipboard" (Ctrl + Alt +
    v)
- En ligne: JSON 2 TS
  - Démo



## Typage semi-manuel "outillé": Bilan

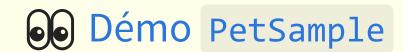
- Types générés = complets, correctes
- Ajustements nécessaires
  - Renommer RootObject → Pet
  - Enlever le namespace
  - Indiquer les champs enum (status)
- X Besoin d'exemples JSON
  - API REST → Swagger / capture résultats
- X 2 actions manuelles : Copy, Convert



# TYPAGE PAR SAMPLES (\*)



- Typage semi-manuel avec inférénce
  - JSON = object literal (array) = type implicite
  - Copie dans variable: const sample = ~JSON
    - → Type inféré par le compilateur TypeScript
- Capture du type avec type query typeof sample



## Comparaison avec le typage semimanuel outillé (convertisseur JSON-TS)

- Types <del>générés</del> inférés = sans erreur
- X Besoin d'exemples JSON
  - Samples dans le code → Autant s'en servir
    - Valeur par défaut d'un champ PetDto
    - Résultat d'un service bouchonné dans un TU
       (PetServiceStub: get = () => petSample;)
- X 2 → 1 action manuelle (copy<del>, convert</del>) 👍
- & 2 ajustements en moins 👍
  - RootObject, namespace



# TYPAGE SEMI-MANUEL AJUSTEMENTS

- Types imbriqués / enum : à définir et à mapper
- Champs optionnels (property) a?: T
- Option de compilation strictNullChecks
  - → Identifier les valeurs **nullables** (au sens large)
  - T | null, T | undefined
  - $\triangle$  Optionnel  $\Leftrightarrow \Rightarrow$  undefinedable



## Définition des types imbriqués

Dans Pet : Category et Tag

```
type PetSample = typeof petSample;
type CategorySample = typeof petSample.category;
type TagSample = typeof petSample.tags[0];
```



## Énumération status

- Type imbriqué non inférable
- Specs → status peut valoir:
  - 'available'
  - 'pending'
  - 'sold'
- Plusieurs manières de le modéliser...



#### Énumération status : union type

```
type PetStatus = 'available' | 'pending' | 'sold';
const petStatus: PetStatus = 'sold';
```

- **Concis**
- IntelliSense pour petStatus indique parfois
  - la structure: 'available' | 'pending'...
  - plutôt que le nom : PetStatus



### Énumération status : string enum

```
enum PetStatus {
    Available = 'available',
    Pending = 'pending',
    Sold = 'sold',
}
```

- X Verbeux
- Explicite à la lecture sur les valeurs permises
  - status = Status.Available
  - status = 'available'



## Mapper les types imbriqués

- Stratégies
  - Personnalisation inline du sample
  - Extension de types



#### Personnalisation du sample (1)

- Pattern: "key: value as CustomType"
  - 🗱 Type union, enum
    - o status: 'sold' as
      PetStatus



#### Personnalisation du sample (2)

- Pattern: "key: nullable(value)"
  - Champ nullable
    - nullable() permet de continuer d'inférer la valeur rendue nullable :

```
function nullable<T>(value: T): T | null {
   return value;
}
```



#### Personnalisation du sample - Bilan

- & Modifications directes du sample
  - Intrusives
  - Difficiles à identifier
  - Risque de les supprimer par erreur lors de l'obtention du sample de la version N+1
- X Champs optionnels

Stratégie non utilisée



#### **Extension des types**

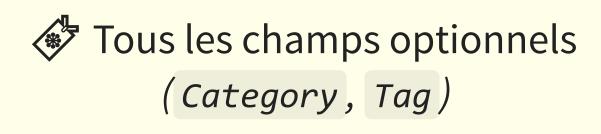
- 🖞 Principes
  - Préserver le sample
  - Créer type personnalisé depuis type du sample
- Différents "patterns"



#### Extension des types - *Utility Types*

- Tous les champs optionnels →
   Partial<T>
- Tous les champs readonly →
   Readonly<T>
- etc.
- Liste des *utility types* (aka helper types)

#### Extension des types - Pattern 1



Pattern au choix selon IntelliSense: type (structure) vs interface (nom)



#### Extension des types - "Surcharge"

```
Surcharge de champs (category, status, tags)
```

```
type PetSample = typeof petSample;
interface PetDtoBaseKO extends PetSample {
  category: CategoryDto, // X Incompatible
  tags : TagDto[], // X Incompatible
  status : PetStatus, // Avec string enum / union
}
```



Surcharge → Retrait puis Redéfinition des champs



#### **Extension des types - Pattern 2**

- Utility type maison Extends<T, U>
  - $\dot{A}$  la  $\$.extends({}, t, u) \iff {...t, ...u}$
  - Avec U champs à redéfinir (ou à ajouter) dans

```
type Extends<T, U> = Omit<T, keyof T & keyof U> & U;

type PetDtoBase = Extends<typeof petSample, {
    category: CategoryDto,
    tags : TagDto[],
    status : PetStatus,
}>;
```



#### **Extension des types - Pattern 3**

```
Mixte champs requis / optionnels

(Pet → name, photoUrls)
```

Utility type maison PartialExcept<T,</li>

```
K...>
```

- Tous les champs optionnels
- Sauf ceux de clés dans K

#### Extension des types - Démo

- Patterns combinés
- IntelliSense: erreurs, typages, tests



#### Extension des types - Bilan

- Sample préservé
- Juste quelques patterns à combiner
- DRY (pas de duplication)
- Type safety / sample version N+1
- <u>M</u> Utility Types
  - A minima les connaître
  - Au besoin en créer → Advanced

**Types** 

## BILAN GÉNÉRAL DU TYPAGE

- Côté API → côté client
- (Semi-) manuel → "par samples étendus"
  - Semi-manuel!
  - Alternative : générateur côté client ?
- Autre problème : typage statique suffit-il ?



# Générateur côté client : Swagger Codegen

- Génère modèle + services
- Choix du langage et de la librairie
  - TypeScript + Angular → Code propre
  - TypeScript + Fetch → Code fourre-tout
- Générateur en ligne



#### DTO générés (extrait)

```
// ./model/category.ts
export interface Category { id?: number; name?: string;
// ./model/pet.ts
import { Category } from './category';
export interface Pet {
    id?: number;
    category?: Category;
    name: string;
```



#### Services Angular générés (extrait)

```
// ./api/pet.service.ts
@Injectable()
export class PetService {
  constructor(protected httpClient: HttpClient...) { ... }
  public addPet(body: Pet...) {
    return this.httpClient.post<any>(
      `${this.basePath}/pet`,
      body, ...);
```



## Limite du typage statique

- Types TypeScript = garanties compiletime
  - Effacés à la compilation en JavaScript
- Garanties au runtime
  - API untrusty → types inattendus
  - X Philosophie TypeScript
  - /⊋ Librairies de validation



#### Librairies de validation

- Données externes ⇔ Types TypeScript attendus
- 2 stratégies opposées
  - Transpilation
  - Runtime Types



#### Librairies de validation à la transpilation

Nom	$\stackrel{\wedge}{\Sigma}$		wiiii.
typescript-is	219	0.12.0	05/2019
ts-runtime	241	0.2.0	10/2018

♠ Pas encore production ready



#### Librairies de Runtime Types

- Objet de validation d'un type → Type
   TypeScript
- Syntaxe rappelle celle des types TypeScript
- A coder manuellement depuis contrat de l'API

Nom	$\stackrel{\wedge}{\boxtimes}$		<b>*****</b>	
io-ts	1731	1.8.6	05/2019	decode/encode
runtypes	562	3.2.0	04/2019	constraint, msg



## Perspectives 😂

- Contrat d'API exploitable
- Génération côté client combinant :
  - Runtime Types
  - + Types TypeScript sous-jacents



# QUESTIONS



# **居 Références - Typage**

- Type queries & typeof ~ Marius Schulz, Mar
   2016
- TS 2.1: Mapped Types ~ Marius Schulz, Jan 2017
- Interface vs Type alias ~ Martin Hochel, Mar 2018
- TS Type Inference Guide ~ Tomasz Ducin, Jan 2019



## Références - Runtime type checking

- Typescript and validations at runtime boundaries
  - ~ Gaurab Paul, Mar 2018
- TypeScript—Make types "real", the type guards
  - ~ Charly Poly, Nov 2018



### **MERCI**

@DeneauRomain



