TYPESCRIPT TYPAGE DE DONNÉES EXTERNES

Retour d'expériences



About me

Romain Deneau - @DeneauRomain

Senior Developer | .NET, Angular, 😍 TypeScript



Animateur communauté Craft





CONTEXTE

- Données externes
 - Objets JSON en E/S de serveurs C# (ASP.NET Views, Controllers / Web API)
 - Front / Back gérés par l'équipe
- Interfaçage avec librairie d'infrastructure
 - $jQuery \rightarrow \$.ajax()$
 - Angular → httpClient
- 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

👉 Puissance du typage en TS 🦠



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
 - 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
 - ② 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
- <u>∧</u> Outils tiers et spécifiques C# → TS
- X Chaînes de build Front-Back + compliquées
- X Marche mieux/que avec Visual Studio
- Producer driven
 - Typage au plus près de la source
 - X Producer concern → Consumer concern



TYPAGE CÔTÉ CLIENT

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



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



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...
- <u>API</u> Changement de version de l'API



Typage avec convertisseur JSON→TS

- Typage semi-manuel avec ces convertisseurs :
 - Extension VS code: JSON to TS
 - "Convert from clipboard" (Ctrl + Alt + V)
 - En ligne: JSON 2 TS





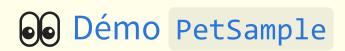
Typage avec convertisseur JSON→TS: Bilan

- Types générés = complets, correctes
- & Ajustements nécessaires
 - Renommer RootObject → Pet
 - Enlever le namespace
 - Autres... → on en reparle juste après
- X Besoin d'exemples JSON
 - API REST → Swagger / capture résultats
- X 2 actions manuelles : Copy, Convert



TYPAGE PAR SAMPLES \heartsuit

- Typage semi-manuel avec inférence
 - 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 / typage précédent (avec convertisseur JSON→TS)

- Types générés 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 ≥ → 1 action manuelle (copy, convert) 👍
- & 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

• category, tag définissable par inférence

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

 status à la main → type union ou enum



Appliquer les ajustements dans les types inférés du *sample*

- 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

```
Tous les champs optionnels (Category, Tag)
```

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

```
type TagDto = Partial<typeof petSample.tags[0]>;
// ~~~~> type TagDto = { id?: number, name?: string }
interface TagDto extends Partial<typeof petSample.tags[0]> {}
// ~~~~> interface TagDto
```



Extension des types - Surcharge

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

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

Surcharge → Retrait puis Redéfinition des champs



Extension des types - Pattern 2

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

T

```
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

- Utility type maison PartialExcept<T,K...>
 - Tous les champs optionnels
 - Sauf ceux de clés dans ĸ



Extension des types - Démo

- Patterns combinés
- IntelliSense: erreurs, typages





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)



Limite du typage statique

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



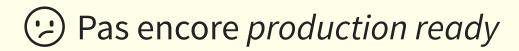
Librairies de validation

- 2 stratégies opposées
 - Transpilation
 - Runtime Types



Librairies de validation à la transpilation

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





Librairies de Runtime Types

- Syntaxe rappelle celle des types TypeScript
- A coder manuellement depuis contrat de l'API

Nom	$\stackrel{\wedge}{\boxtimes}$			
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
- Alternative : REST → gRPC



QUESTIONS



居 Ressources - 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



居 Ressources - Runtime type checking

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





@DeneauRomain

https://github.com/rdeneau/talks/ typescript-typing-external-data/



