

Introduction aux types tagués et raffinés: une approche par la sécurité



Qui suis je?



Nicolas François
Software Engineer - SRE
nfrancois@mediarithmics.com
@koisell

Motivations de cette présentation

Qu'est que la sécurité?

- Situation dans laquelle quelqu'un, quelque chose n'est exposé à aucun danger, à aucun risque. (larousse)
- Absence ou limitation des risques dans un domaine précis (larousse)
 - minimisation des risques == maximisation des opportunités ?
 - Approche similaire au développement, à la qualité, ... bref à l'ingénierie



Qu'est que la sécurité?

- Confidentialité: le fait de s'assurer que l'information n'est accessible qu'à ceux dont l'accès est autorisé (wikipedia)
- Intégrité: désigne l'état de données qui, lors de leur traitement, de leur conservation ou de leur transmission, ne subissent aucune altération ou destruction volontaire ou accidentelle. L'intégrité des données comprend quatre éléments : l'intégralité, la précision, l'exactitude/authenticité et la validité. (wikipedia)
- Disponibilité: l'aptitude d'un composant à être en état d'accomplir une fonction requise dans des conditions données, à un instant donné ou pendant un intervalle de temps donné. (wikipedia)

Une étude de 2005 a identifié que 75% des besoins fonctionnels incorporent des besoins de sécurité.

Le problème



- Comment donner à un type la bonne sémantique ?
- Comment ne pas ré-écrire des comportements qui existent (presque) déjà ?
- Comment conserver les mêmes performances à runtime (le fameux *no cost abstraction*)

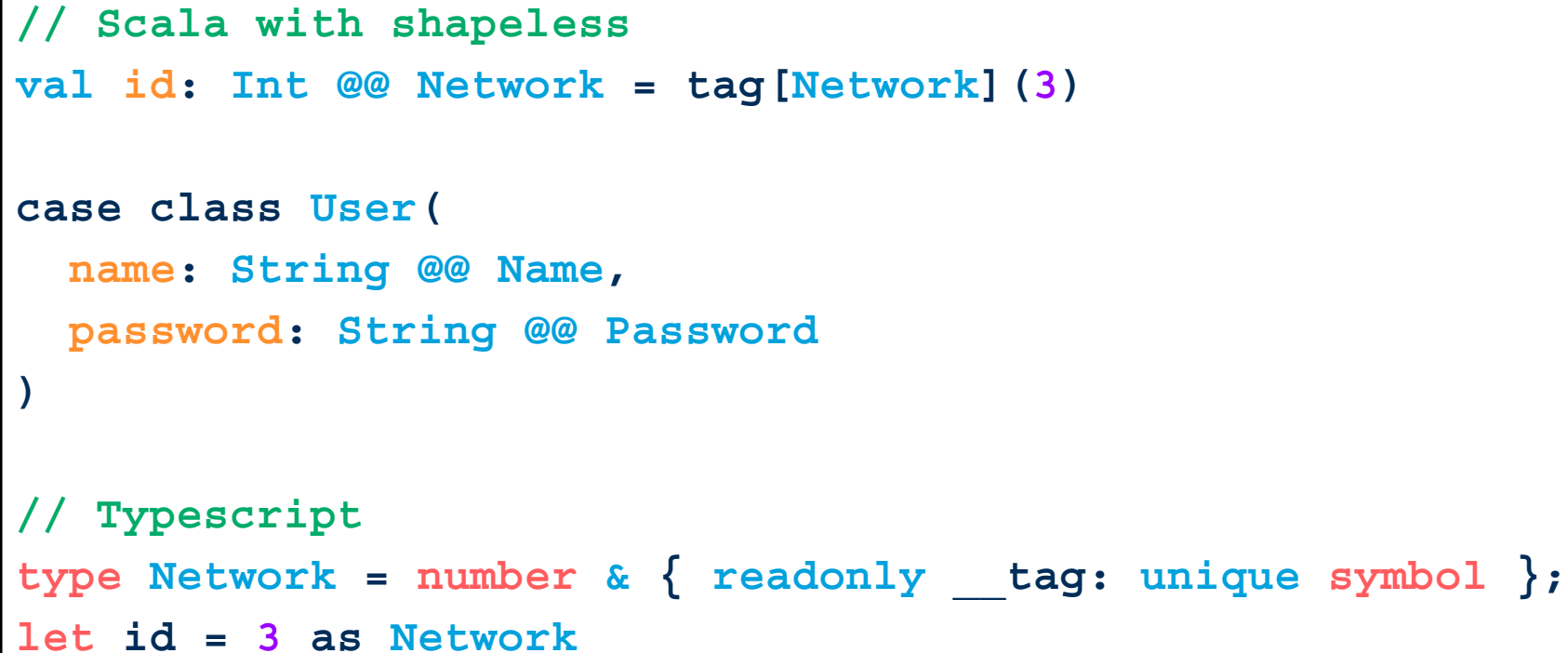
“All Models Are Wrong, Some Are Useful”

- George Box statistician

Type tagué



Type tagué



```
// Scala with shapeless
val id: Int @@ Network = tag[Network](3)

case class User(
  name: String @@ Name,
  password: String @@ Password
)

// Typescript
type Network = number & { readonly __tag: unique symbol };
let id = 3 as Network
```


Type tagué

- Existe uniquement à la compilation (on parle aussi de type fantôme)
- Couplage extrêmement faible
- Apporte peu de garanties. Tous repose sur le développeur.
- Aide grandement le refactoring, la lecture et le parcours du code
- Le tag est souvent un type *creux* (pas de fonctions, de constructeurs, ...)
- Disponible en Haskell, Scala, Swift, Typescript, ...

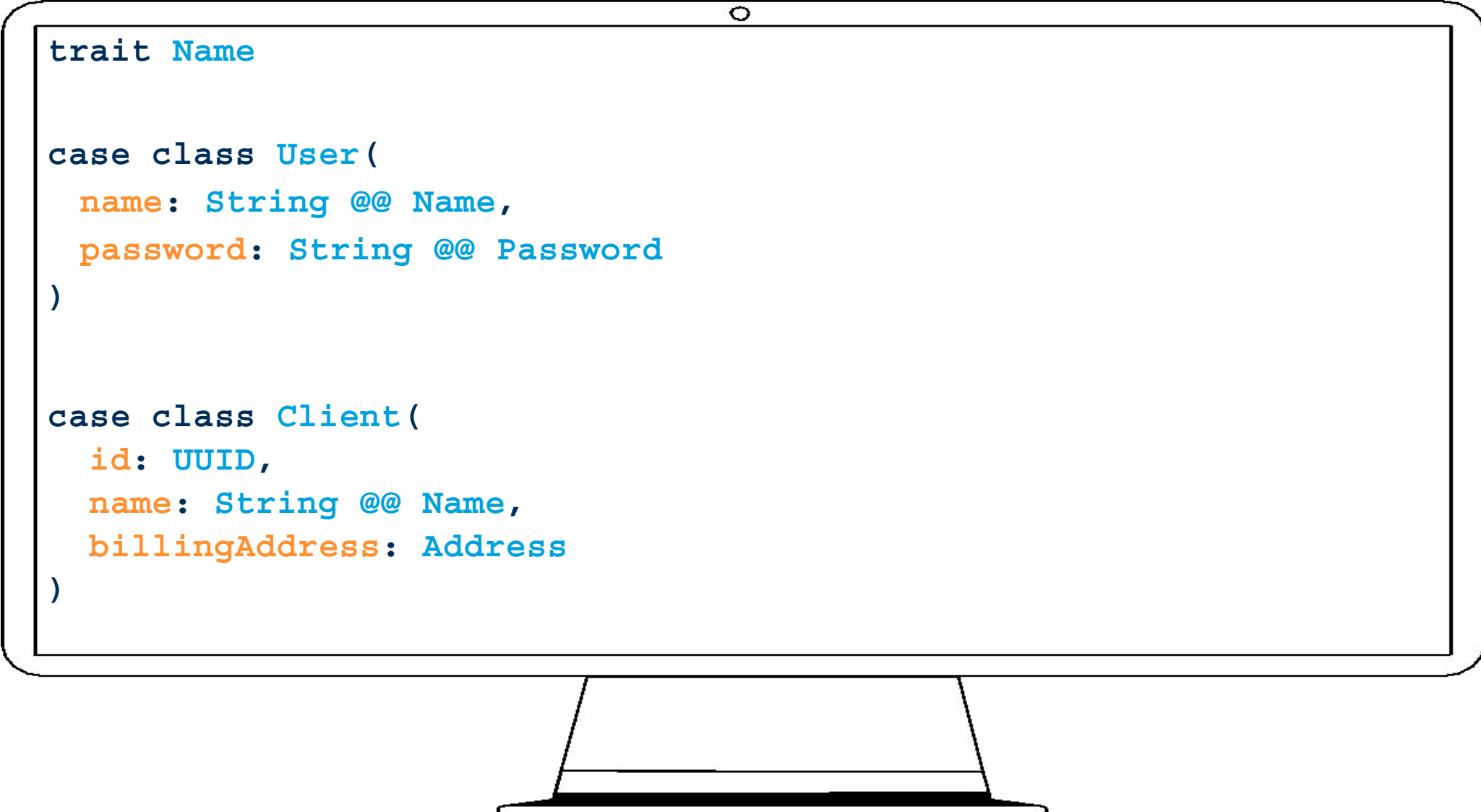
Type tagué

```
trait Version

case class Container(
  id: UUID @@ Container,
  image: String,
  name: String,
  version: Int @@ Version,
  network: Int @@ Network
)

case class Network(
  id: Int @@ Network,
  `type`: NetworkType
)
```

Type tagué

A computer monitor with a black bezel and a silver stand. The screen displays Scala code with syntax highlighting. The code defines a trait 'Name' and two case classes, 'User' and 'Client', both of which inherit from the 'Name' trait. The 'User' class has fields 'name' and 'password', while the 'Client' class has fields 'id', 'name', and 'billingAddress'.

```
trait Name

case class User(
  name: String @@ Name,
  password: String @@ Password
)

case class Client(
  id: UUID,
  name: String @@ Name,
  billingAddress: Address
)
```

Type raffiné



Type raffiné

```
import eu.timepit.refined._
import eu.timepit.refined.api.Refined
import eu.timepit.refined.auto._
import eu.timepit.refined.numeric._

val x: Int = 42
val compileTime1: Int Refined Positive = 5
val compileTime2: Int Refined Positive = -5 // Doesn't compile

val runTime1: Either[String, Int Refined Positive] =
  refineV[Positive](x) // Right(42)

val runTime2: Either[String, Int Refined Positive] =
  refineV[Positive](-x) // Left(Predicate failed: (-42 > 0).)
```

Type raffiné

- Aide la création de fonction totale (pour chaque entrée possible il y a un résultat):
 - On peut augmenter l'ensemble de sortie => Option
 - On peut diminuer l'ensemble d'entrée => Refined
- Les fonctions totales sont plus simples à tester
- La pose d'un tag à travers une fonction qui valide l'appartenance au sous-ensemble
- Permet de tester une précondition une seule et unique fois
- Les types raffinés n'existent qu'à la compilation. Cela peut poser des problèmes avec des outils reposants sur l'introspection comme Jackson.
- Les types raffinés peuvent poser des problèmes de migration si le type a imposé une contrainte différentes par le passé.

Type raffiné

```
import eu.timepit.refined.W
import eu.timepit.refined.api.{Refined, RefinedTypeOps}
import eu.timepit.refined.numeric.Greater

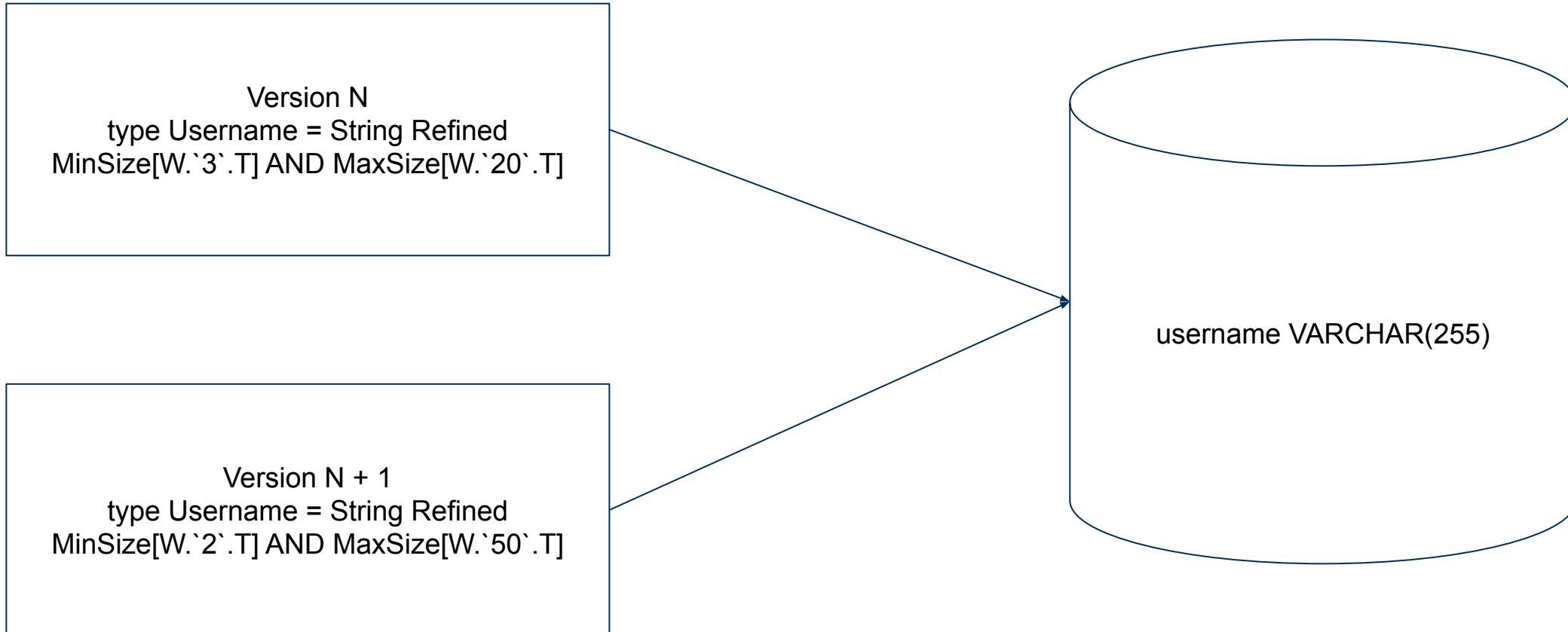
private type UnixNonRootUserAndGroupPredicate = Greater[W.`1`.T]

type UnixNonRootUserId = Int Refined UnixNonRootUserAndGroupPredicate
type UnixNonRootGroupId = Int Refined UnixNonRootUserAndGroupPredicate

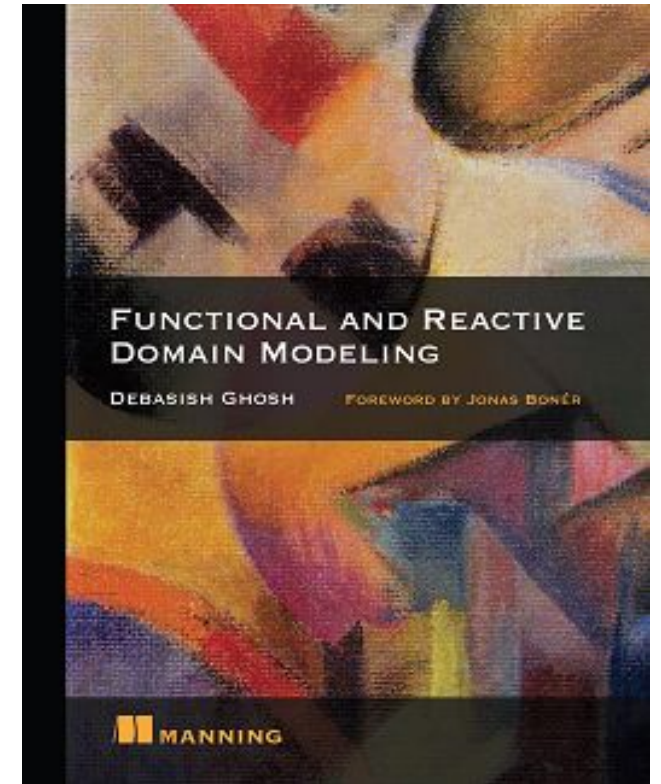
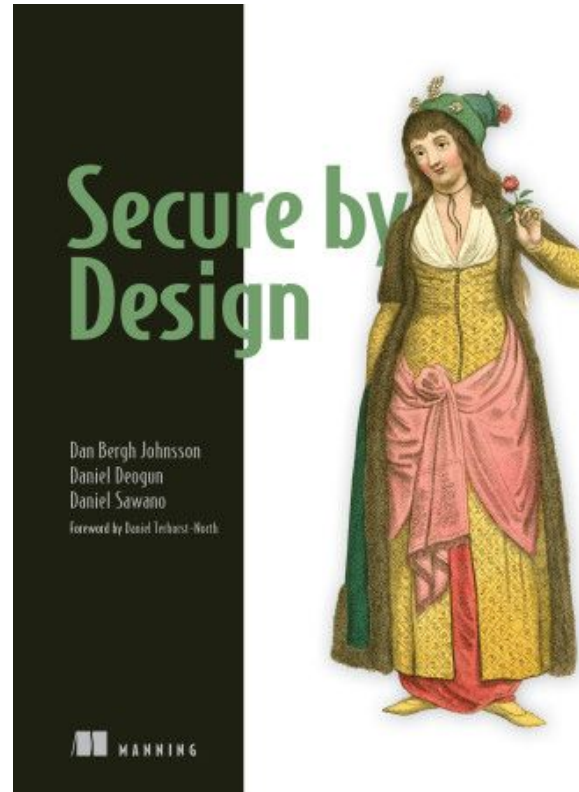
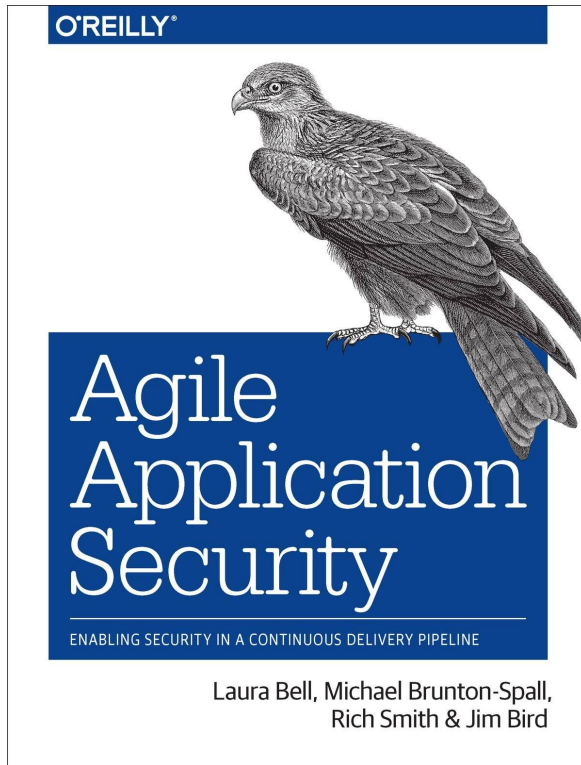
object UnixNonRootUserId extends RefinedTypeOps[UnixNonRootUserId, Int]
object UnixNonRootGroupId extends RefinedTypeOps[UnixNonRootGroupId, Int]

val userId = UnixNonRootUserId.unsafeFrom(4242 + i)
val groupId = UnixNonRootGroupId.unsafeFrom(4242 + i)
```

Type raffiné

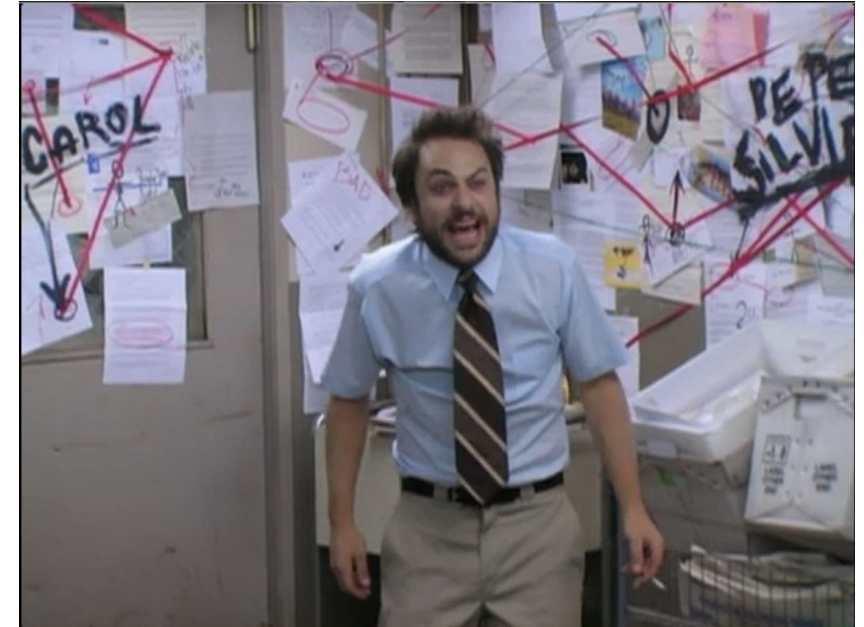


Recommended book

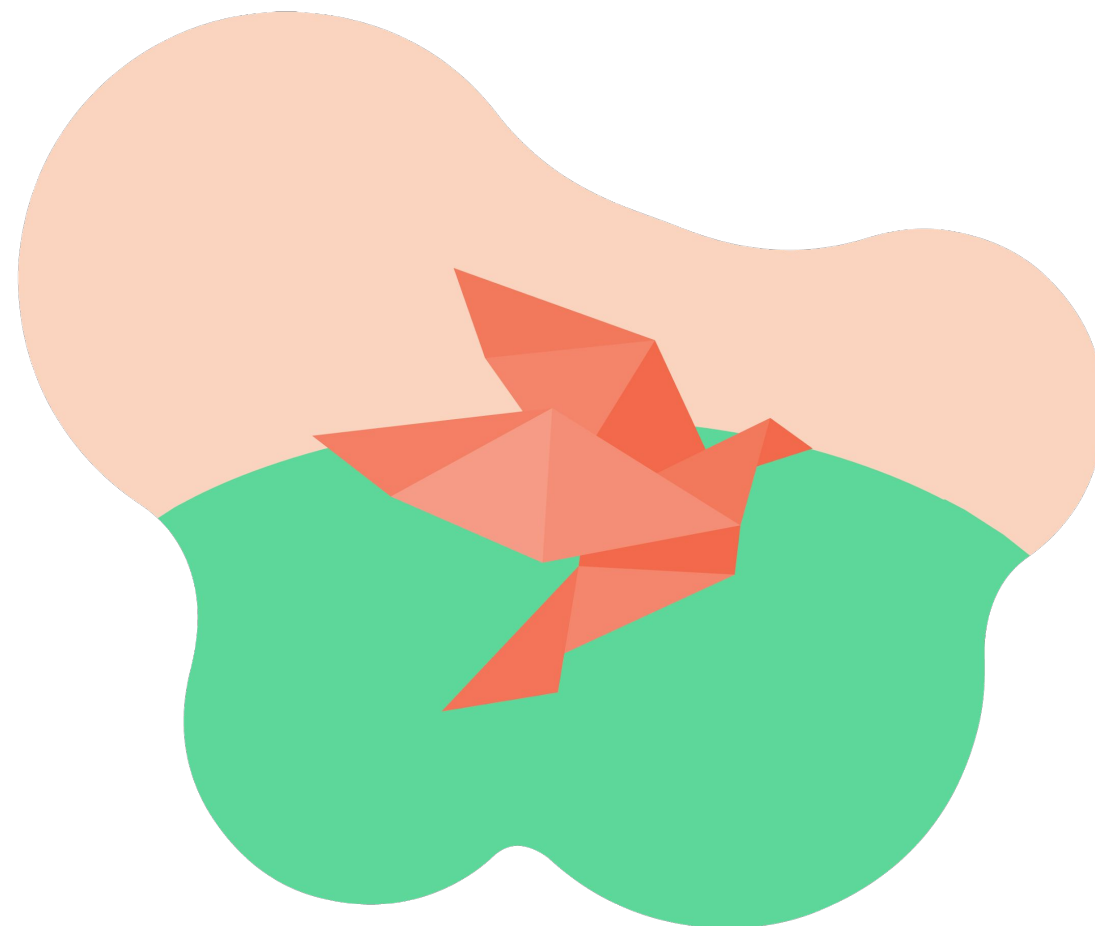


Source

- http://johnwilander.se/research_publications/paper_sreis2005_wilander_gustavsson.pdf
- <https://www.youtube.com/watch?v=tuxmPkLyhqw>
- <https://youtu.be/w7FuQiSi48w>
- <https://kubyshkin.name/posts/newtype-in-typescript/>
- <https://medium.com/iterators/to-tag-a-type-88dc344bb66c>
- Chapter 5.2: [The type astronaut's guide to Shapeless](#)



Question ?



Merci !



<https://github.com/MEDIARITHMICS/talks>

