

Human (o) IDs

Designing IDs for machines *and* humans

Jakub Wojnowski

Senior Software Engineer @ SiriusXM

- 11 years in Software Engineering
- 7 years in ~~Tibet~~ Scala
- Occasional Open Source contributor
- A connoisseur of bad movies and bad jokes



Let's automate parking!

1. A customer drives onto a platform, steps out of a car and leaves
2. The car gets moved automagically into a free location underground
3. The customer retrieves the car by entering the licence plate number and paying the fee



How do we know which platform to retrieve?

- DW5T386 — licence plate number
- 50 - level (vertical position)
- C - column (horizontal position)



How do we know which platform to retrieve?

- DW5T386 — licence plate number
- 50C - storage space



aff5172c-9330-4657-ade4-a3169e95008c

a446364e-c05a-457b-90c0-de4287866793



aff5172c-9330-4657-ade4-a3169e95008c

a446364e-c05a-457b-90c0-de4287866793

Something is *no yes* 🤔

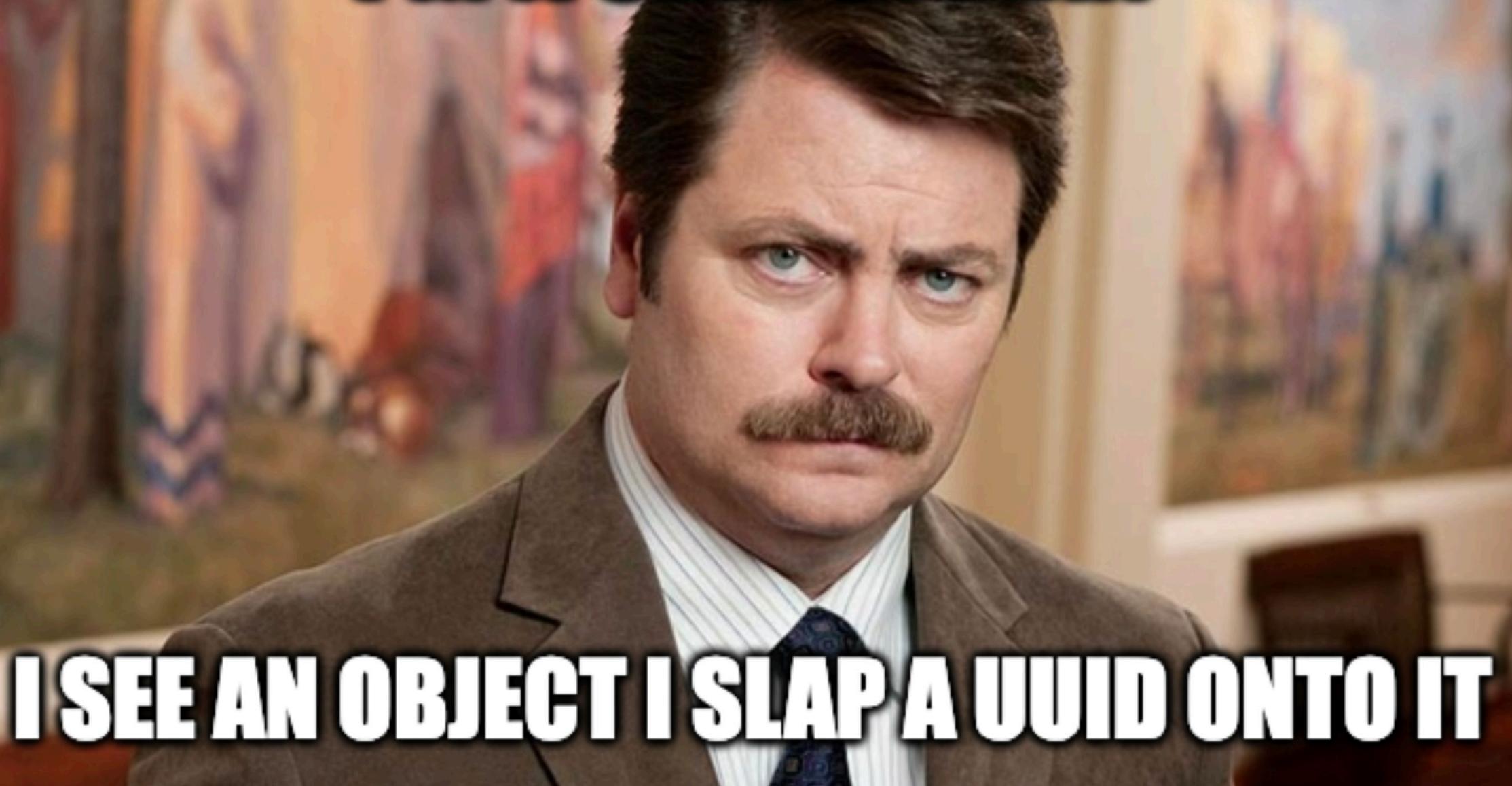
You-you-eye-dosis

Compulsive usage of UUIDs as object identifiers, without consideration of the business, operational, or technical context of a given object.

Severe cases include writing and publishing binding, company-wide documents mandating the usage of UUIDs as primary identifier for all objects, regardless of business, operational or technical context of a given object. [1]

[1] <https://handbook.ocado.tech/#/sw-development/hallmarks?id=r1>

I'M A SIMPLE MAN



I SEE AN OBJECT I SLAP A UUID ONTO IT

ICD-10

International Classification of Diseases

- J00.1 Common cold 😊
- I21.9 Heart attack 💔
- F[10.129] Cats Effect 🍻

ICD-10

International Classification of Diseases - *rematch*

- W58.02 Struck by alligator 🐊
- W58.02XD Struck by alligator - subsequent encounter 🐊🐊
- V97.33 Sucked into jet engine 🤖✈️
- V97.33XD Sucked into jet engine - subsequent encounter ✈️✈️

F21.37XD

You-you-eye-dosis

Compulsive usage of UUIDs as object identifiers, without consideration of the business, operational, or technical context of a given object.

Severe cases include writing and publishing binding, company-wide documents mandating the usage of UUIDs as primary identifier for all objects, regardless of business, operational or technical context of a given object. [1]

[1] <https://handbook.ocado.tech/#/sw-development/hallmarks?id=r1>

How about natural IDs?

50C

DW5T386

F21.37XD

us-east-2

- human-friendly
- machine-friendly
- URL-friendly
- alligator-friendly? 

Why use natural IDs?

Let's assume that:

1. Majority of Software Engineers are humans.
2. Majority of ~~resources~~ people supporting software are humans as well.
3. Majority of customers of the software are also humans.

3:00 a.m. production incident simulation

50C

DW5T386

F21.37XD

us-east-2

3:00 a.m. production incident simulation

ebda7e03-eb06-4653-85f4-88a0f509aba4

78a03aa9-67c3-4a5a-be2e-4b380275b204

c4d55307-d927-443e-91d9-5e01a439384d

b9346281-486e-486b-aac2-1c37456d4385

When can we directly use natural IDs?

- immutable
- machine- and URL-friendly
 - or easily, deterministically adaptable, e.g. `wroclaw_dluga` or `plwr03`
- safe for public sharing
 - non-PII (GDPR)
- relatively few records (<100k with a good system, ideally much less)

OK, what if we can't use natural IDs?

Random, globally unique IDs

- Random generation (more-or-less)
- Independent generation
 - Offline/Local
 - Horizontally-scalable
 - Collision resistant
- Unguessable

UUIDs?

UUIDs

30c4aeb8-047b-4898-b454-
e7f1230aa3a9

Pros

- All the pros of globally unique IDs
- *Very* popular
- Native support (languages, libraries, databases)
- Can be represented with two longs (128 bits)

JA

You

What does UUID stand for?



ChatGPT

Here are a few possible expansions of "UUID":

1. Unusually Ugly Internet Duckling
2. Unexpectedly Unstable Icecream Dispenser
3. Universally Underestimated Incompetence Detector
4. Unbelievably Uninteresting Indie Drama
5. Utterly Useless In Dance
6. Ultimately Unimportant Intrusive Device
7. Universally Undercooked Italian Dumplings
8. Unusually Upbeat Iguana Disco
9. Underwhelmingly Unclear Instructional Diagram
10. Utterly Unavoidable Indecision Dilemma

UUIDs

30c4aeb8-047b-4898-b454-e7f1230aa3a9

Pros

- All the pros of globally unique IDs
- Native support (languages, libraries, databases)
- *Very* popular
- Can be represented with just two longs (128 bits)

Cons

- Limited alphabet (16 chars)
 - human readability
 - length vs entropy
 - 2Long
- Dashes
 - Selecting is a nightmare
 - Full-text search might be tricky (Elasticsearch, Kibana)

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```



Ah shit, here we go again.

Selecting example

```
select * from account where account_id = '4cc99bf1-044c-4c67-95f7-bd6d6774e638';
```

What would a *good* globally unique ID be?

Globally unique ID wishlist

Must haves*

- Reasonably unique (~UUID)
- Larger alphabet (>30 chars)
- Easily-selectable
 - no dashes
 - actually, only A-Za-z0-9_
 - single-case letters
- URL-friendly

Nice-to-haves*

- no plaintext timestamp or machine ID
- no lookalikes (0 vs O, 1 vs l vs capital I)
- no obscenities

* Personal opinion, open for discussion, often depends on the context

Globally unique ID examples

Name	Example
UUID	0ac3bd4d-faeb-4417-8746-d376ebc8341d
UUID Readable	Loren Chariot Addy the Titbit of Cholame questioned Cele Garth Alda and 16 windy frogs
ULID	01HX4F1C7HSRGY4WPWY6RZTN5W
Nano ID	-_UaP3i0A15SSkf5ePsog , Z5Te0yXXVZn3kwIqMbj0
(Truly) Random Alphanum	Dupa1234
Cuid2	qg625h9s1hfoxwv0ip8e5jzg

Let's assume we can't use these

50C

F21.37XD

DW5T386

us-east-2

And we don't want these

ebda7e03-eb06-4653-85f4-88a0f509aba4

78a03aa9-67c3-4a5a-be2e-4b380275b204

c4d55307-d927-443e-91d9-5e01a439384d

b9346281-486e-486b-aac2-1c37456d4385

But are these better?

j rk7sxiq1y6ph4r6sgn1a6i4

kxhqzn24pvhw5oy179inbwoc

pkijda0r99u3yumon7osfu3d

a7muawrcmsm84xtb8qz5kd8b

How can we make globally unique IDs more human-friendly?

Prefixes!

alig_y49vmd6ceu7p3x6wyj7ht7kc

cus_ko88jpiayeamxql8hwkrbj2t

car_ryx12rfb9klwawa{jg6dhhnc}

Benefits of prefixed IDs

`cus_mb3fwgn5p3dhntbpfrvosjg2`

- instantly recognisable by humans
 - harder to mix up different IDs during an investigation/f-up
- easy to validate (e.g. API level)
- quick search (e.g. in dev tools, admin panels)



Before

j rk7sxiq1y6ph4r6sgn1a6i4

kxhqzn24pvhw5oy179inbwoc

pki jda0r99u3yumon7osfu3d

a7muawrcmsm84xtb8qz5kd8b

A photograph of Drake, the Canadian rapper, wearing a bright orange puffer jacket over a yellow t-shirt with a black Jordan logo. He is pointing his right index finger directly at the camera with a neutral expression.

After

loc_jrk7sxiq1y6ph4r6sgn1a6i4

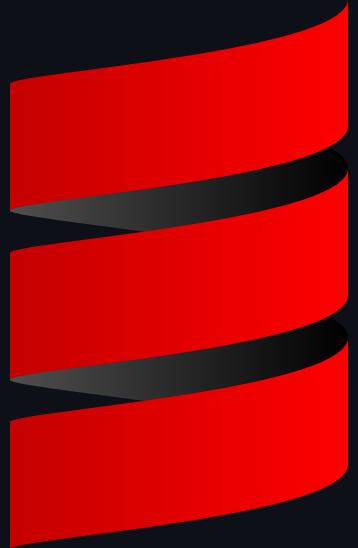
icd10_kxhqzn24pvhw5oy179inbwoc

car_pkijda0r99u3yumon7osfu3d

region_h1lohraf2bbure5rmxjpx4mn

**That's a lot on IDs, how about
some Scala code?**

(Finally!)



Let's write our own prefixed IDs in Scala

Requirements

1. String prefixes (3-5 letters)
2. Input validation
3. Random generation
4. Easy access to a clean ID

Naïve approach

```
final case class PrefixedId(prefix: String, id: Cuid2)

object PrefixedId {
  def parse(rawId: String, expectedPrefix: String): Either[String, PrefixedId] =
    ...
  def random[F[_]](prefix: String): F[PrefixedId] =
    ...
}
```

(Still) naïve approach

```
final case class PrefixedId private (prefix: String, id: Cuid2)

object PrefixedId {
  def apply(prefix: String, id: Cuid2): Either[String, PrefixedId] = ...

  def parse(rawId: String, expectedPrefix: String): Either[String, PrefixedId] =
    ...

  def random[F[_]](prefix: String): F[PrefixedId] =
    ...
}
```

(A bit less) naïve approach

```
sealed abstract case class PrefixedId(prefix: String, id: Cuid2)

object PrefixedId {
  def parse(rawId: String, expectedPrefix: String): Either[String, PrefixedId] = {
    val (prefix, id) = ...
    new PrefixedId(prefix, id) {}
  }
}
```

Feeding the alligator

```
def feed(  
    alligatorId: PrefixedId,  
    ratId: PrefixedId  
) = ...  
  
...  
val someRatId: PrefixedId = ...  
val someAlligatorId: PrefixedId = ...  
  
feed(someRatId, someAlligatorId) // compiles
```

Feeding the alligator

```
def feed(  
    alligatorId: PrefixedId,  
    ratId: PrefixedId  
) = ...  
  
...  
val someRatId: PrefixedId = ...  
val someAlligatorId: PrefixedId = ...  
  
feed(someRatId, someAlligatorId) // compiles
```

Easy to mix up the *dinner* and the *diner*

(**cough** W58.02XD **cough**)

Let's write our own prefixed IDs in Scala

Requirements

1. String prefixes (3-5 letters)
2. Input validation
3. Easy access to a clean ID
4. Random generation
5. Type safety

Type-safe approach

```
// Scala 2.x
final case class AlligatorId(value: PrefixedId) extends AnyVal

object AlligatorId {
    def parse(raw: String): Either[String, AlligatorId] =
        PrefixedId.parse(raw, "alig")
}
```

```
// Scala 3.x
opaque type AlligatorId = PrefixedId

object AlligatorId:
    def parse(raw: String): Either[String, AlligatorId] =
        PrefixedId.parse(raw, "alig")
```

Refined types

```
type AlligatorId = String Refined StartsWith["alig_"]
```

```
type AlligatorId = String Refined MatchesRegex["^alig_[a-z][a-z0-9-_]{24}$"]
```

Wait, is that even a valid type? 🤔

```
StartsWith["alig_"]
```

Introduction to literal types

All primitives (`Byte` , `Short` , `Int` , `Long` , `Double` , `Float` , `Char`) plus `String`

```
val anInt: 42 = 42
val aBoolean: false = false
val aChar: 'x' = 'x'
val aDouble: 3.14 = 3.14
val aString: "foo" = "foo"

val kacEffect: F[10.129] = F.success(10.129) // 🍺
```

```
val just42: 42 = 42 // compiles
// val thisDoesntCompile: 42 = 41 // doesn't compile
```

```
val int: Int = just42 // widening the type
val double: Double = just42 * 3.14 // 131.88
```

Literal types on a quest for better IDs

```
case class PrefixedId[P <: String](prefix: P, id: Cuid2)  
PrefixedId["pfx"]("pfx", "ep88jpiayeamxql8hwkrbj2t")
```

Literal types

Duplicated "pfx" :

```
PrefixId["pfx"]("pfx", "ep88jpiayeamxql8hwkrbj2t")
```

Solution?

Macros? 🍺

ValueOf[A] to the rescue

```
val actualValue: Int = valueOf[42] // 42
```

ValueOf[A] to the rescue 🚑

```
case class PrefixedId[P <: String: ValueOf](id: Cuid2) {  
  
  def renderWithPrefix: String = {  
    val prefix: String = valueOf[P]  
  
    s"${prefix}_${id}"  
  }  
}  
  
object PrefixedId {  
  def parse[P <: String](rawId: String)(implicit valueOfPrefix: ValueOf[P]) = {  
    val expectedPrefix: String = valueOfPrefix.value  
    ...  
  }  
}
```

Type-safe approach with literal types

```
type RatId = PrefixId["rat"]
type AlligatorId = PrefixId["alig"]

val someRat: RatId = ...
val someAlligator: AlligatorId = someRat // compilation fails
```

AUDYCJA ZAWIERA LOKOWANIE PRODUKTU

Say hi to [jwojnowski/humanoid](#) 

```
// type definition
type AlligatorId = PrefixedCuid2["alig"]
val AlligatorId = PrefixedCuid20ps["alig"]

// usage
val myFavouriteAlligatorId: Either[String, AlligatorId] =
  AlligatorId.parseRequirePrefix("alig_duniua2fm1yh7lilk18xfd77")

val someRandomAlligatorId: F[AlligatorId] =
  AlligatorId.random[F]
```

AUDYCJA ZAWIERA LOKOWANIE PRODUKTU

Say hi to [jwojnowski/humanoid](#) 

```
// type definition
type AlligatorId = PrefixedId["alig", String]
val AlligatorId = PrefixedIdOps["alig", String]

// usage
val validatedId: Either[String, AlligatorId] =
  AlligatorId.parseRequirePrefix("alig_1234")

val prefixedIdFromId: AlligatorId =
  AlligatorId.fromId("2345")
```

AUDYCJA ZAWIERA LOKOWANIE PRODUKTU

Say hi to **jwojnowski/humanoid** 

- Out of the box support for:
 - Cuid2
 - UUID (*if you must*)
- Integrations with:
 - Circe
 - Tapir
- Adaptable to any ID, random or not

Take-home message

~~Take-home message~~

Why do programmers prefer dark mode? 

~~Take-home message~~

Why do programmers prefer dark mode? 

Because the light attracts bugs.   

The *real* take-home message

1. Consider using human-readable, natural identifiers
2. If you *really* can't, use prefixed, human-friendly identifiers
 - Cuid2 *wink wink*
3. Use literal types for enhanced type safety

Finding me 👋

GitHub: [jwojnowski](#)

LinkedIn: [jakub-wojnowski](#)

Slides 

[jwojnowski/wsug-talk-humanoids](#)

Libraries 

Scuid 🦑 (Scala Cuid2): [jwojnowski/scuid](#)

Humanoid: [jwojnowski/humanoid](#)



Thank you!

References

1. <https://dev.to/stripe/designing-apis-for-humans-object-ids-3o5a>
2. <https://scala-lang.org/files/archive/spec/3.4/03-types.html#literal-types>
3. <https://github.com/paralleldrive/cuid2>
4. <https://github.com/Debdut/uuid-readable>
5. <https://magazyn.koleo.pl/zasada-numeracji-miejsc-w-wagonie/>
6. <https://www.icd10data.com/>

Bonus: Built-in database IDs?

e.g. auto-increment/sequential/Mongo Object ID

Security says: nope.

- Insecure Direct Object Reference
- Too much information (e.g. time in Mongo Object IDs)

Bonus:

Why not use both natural IDs and UUIDs under-the-hood?

1. Majority of Software Engineers are humans.
2. Majority of *resources* supporting (failing) software are humans as well.
3. Majority of customers of the software are also humans.
4. When systems break, translation:
 - i. takes time,
 - ii. is error-prone (on the human-human level) <!-- ask me how I know that-->
 - iii. is error-prone (mixing the IDs up by an engineer on support),
 - iv. can lead to multiple standards ([XKCD #927](#))

Bonus: Best practices on choosing prefixes

1. short (ideally, 3-5 chars)
2. distinct
3. as future-proof as possible
4. separator that doesn't break double-click-to-select

Bonus: Interaction time!

Natural	UUID
50C	46174ce4-41c0-4ce9-b3a0-c7e910c27d21
DW5T386	78a03aa9-67c3-4a5a-be2e-4b380275b204
F21.37XD	ebda7e03-eb06-4653-85f4-88a0f509aba4
plwr03	c4d55307-d927-443e-91d9-5e01a439384d
us-east-2	b9346281-486e-486b-aac2-1c37456d4385

Bonus: Which one is the licence plate?

Natural	UUID
🔍	b9346281-486e-486b-aac2-1c37456d4385
👀	46174ce4-41c0-4ce9-b3a0-c7e910c27d21
✂️	c4d55307-d927-443e-91d9-5e01a439384d
🏀	78a03aa9-67c3-4a5a-be2e-4b380275b204
🎩	ebda7e03-eb06-4653-85f4-88a0f509aba4

And the winner is...



78a03aa9-67c3-4a5a-be2e-4b380275b204

Bonus: Existing natural ID system to avoid



Bonus: Hungarian notation

```
val bForce          = true // ya know, b is for Boolean  
val iNumberOfChunks = 42 // ... and i is for Integer
```

