UUID – Universally Unique ID

When any part of the information system can create unique id:s for new objects and events

30.10.2022

Haaga-Helia

Database can generate autoincrement id:s

- MariaDB/MySQL: AUTO_INCREMENT, Oracle: Sequence, MS SQL Server: Identity (1001,1) etc… etc…
- Works fine when it’s database who creates the id:s
- Challenges:
  - If database is a distributed database, how to make sure each item will get a unique id?
  - When database creates the id:s, how to get a correct id back to the frontend that initiated the creation of that new resource / object?
    - E.g. the Knex library returns an array of newly created database row primary key(s) if creation was successful
    - => Return that to the frontend too, e.g. in an object as JSON in body of the http response
  - What if we have a totally distributed architecture and there is no certain database? Or we have an IoT system where devices are e.g. under water and collect information and events, and they have access to the other parts of the system only occasionally?
- Then we start to have a case for system where _everyone_ needs to be able to create unique id:s …

UUID – Universally Unique ID

- *How everybody would be able to create id:s that are not clashing with ones created by others?*
  - *Let's create so 1. **long** and 2. so **random** character sequence from 3. **hex**ademical digits 0-9,a-f that two random generations would basically never yield the same value.*
- There were similar ideas from lot of companies (later e.g. Microsoft's GUID), which were in 2005 standardized to be a 128-bit number
  - That can be presented in text format like this: 123e4567-e89b-12d3-a456-426614174000.
  - That has 32 hexadecimal digits 0-f divided by four dashes
- Read more here: https://en.wikipedia.org/wiki/Universally_unique_identifier#History (extra reading)
- Technical explanation or standard here: https://www.rfc-editor.org/rfc/rfc4122.html (extra reading)
- Each programming language and environment has implementations for this. They allow the creation of valid UUIDs.
- For JavaScript you can start e.g. from the normal good source MDN by searching for e.g.:
  - ”MDN uuid”, ”MDN javascript uuid”

While testing manually, short version displayed

- For testing by human beings you can select some substring of that UUID to be displayed.
- E.g. if you take four hex digits from some part that is fully random, the chance for getting same sequence is
  - 1 / 16^4 = one out of 65 536
  - which is apparently enough for some initial manual testing
  - Still for human it’s rather easy to spot the values as different id:s
  - and thus connect the one created e.g. by one client and shown by another part of the system as the same id
- Similarly in git we can safely use just the start of the commit hash to refer to a certain commit in history:
  - **ca82a6d**ff817ec66f44342007202690a93763949 => **ca82a6d**

Extra: Random generation is hard for computers

- **Note:** True random generation is difficult for exact logical digital devices like computers.
- Implementing your own randomizer would be working fine if you are a PhD specialized in cryptography or related mathematics.
  - And even then it would probably fail to be as close to random as the ones we get from established libraries.
- So let's just use those pseudo-random generators from libraries.
  - Even they are faulty as computers are often so fast that random generator gets same system time based seed
    - Like we have seen before on 1st 2nd semester programming courses.
    - We might get numerous identical values in row or other kind of unwanted non-random patterns.
  - Basically we would need to create a single random generator, and continue to ask more randomized values from the same generator to get more real pseudo random values
  - (or make the system time change by adding so much delay that the used seed changes, but that's us 'doctoring' the randomization again …)
  - Of course if we are just creating some trivial test values for an app. We may accept non-fully random values.