As you might now know, our voting system works on a basis of having unique UUID that maps on to voters and candidate which provides many benefits but issues as well. One of the issue that arises from having these long strings is the abstractness and lack of ease in inputting and verifying correctness of the information when it comes to a series of disjointed number and letters. This is especially true for less tech savvy (like elderlies) people and although QR codes helps in this manner immensly, one can not rely on this relatively advance concept for all people. To remedy this issue, a solution came to mind: NUCLEAR CODES! Or to put it simply, having a series of small often used words for which one can more easily write into a form with easily apparent error checking.
For example: Fox and Foz, which becomes more apparent than Tfj and Trj.
Both these example have a single letter difference with 1 key away from the keyboard, but Fox becomes readily apparent.

The code:
one solution to be able to use, what for now I will call fact, (I am not in the mood to do the scholarly research), is to be have a way to use many word that would map 1 to 1 onto a set number of bits for which when the translated bits are appended together acheives the same result as the UUID in a qr code.
For example a complete example with only 4 bit Ids:
Here is the map:
Box: 00
Fox: 01
Bat: 10
Bag: 11

As we can see every words is maped onto 2 bits and there for we can say that for a number of bits n we need n/2 words or 1 word allow for 2 bits of information since every word is unique and has unique mapping. Therefore to get a 6 bit Id ex: “100100” a form of 3 words needs to be filled:

Word 1: Bat	Word 2: Fox	Word 3: Box
10	01	00

This is what the plan is. This requires 4 parts to do:
1: Finding the correct number of words when related to bits per word with the best compromise
2: Finding the words to use
3: Making the translation func from a series of words to a UUID
4: Making an api to expose this func using json body
5: Liam can very generously provide us with a pretty frontend:)

1:
As previously seen, words can be mapped onto bits. The issue is that to get a full UUID, one needs 128 bits and using only 2 bits per words, as above, means people would need to enter 64 Box and fox which is unrealistic. On the other hand, say we have 32 bits per word, we can say that one would only need 4 words to achieve a full 128 bits! Unfortunately, since every words needs to be unique that would also require 2^32 words or about 4,294,967,296 words! This a couple of order of magnitude too big for the whole of the english language at only 500,000 and those include the words nobody has ever seen and 30 char long words. Therefore to get a feasibility of this project we need to be able to know what is the maximum number of words that are common enough and short if possible and we need the largest 2^n. 2^n since multiple bits per words and all words are unique, then the more bits a word represent the

more words you need to ensure uniqueness of the words. For ex: if Bat: 0000 then we would need 2^4 words to reach word uniqueness.

2.
Very much in conjunction with point 1, we need a bank of words that are easy to spell and known by as many people as we can. These should be english only. There are many ways to go about it, wether it be downloading books and taking the top 200 most used words, using LLM like chat gpt which has gotten a big framework of words or google trends to see what people have searched for. A dictionary where you run code to find all the words of length <n. These should be a short and common as possible and creativity is a given. The plural of words should be discouraged as it is more easily mixed up with singular. A way to improve would be to have a possibility of multiple words per form line for example have a box contain |Cat| or |Cat Bat| or |Bat| being all different, but this will need to be said to the person doing step 3 since this will involve splitting strings on space and handling appropriatly to allow it to be as human error resilient as possible.

3.
Go is a pretty simple language, but making something that will run quickly in an O(1) fashion meaning maps and if possible, which it should, in memory manipulation. This will receive a slice of strings always the same size, but should still be verified and return a UUID. It is not this functions job to verify if the UUID is valid, only check for that the words exists and that the number of is correct.

4.
Making an API endpoint to allow the frontend to retrieve the UUID with a GET request in the json body. It should unmarshal the response into the correct form for the above function description and send any relevant errors or a success and the UUID.