Update from 6
Administrator authored 6 months ago

ceb77bf3

README.md 10.4 KB

Day 01 - Go Boot camp

Comparing Incomparable

Contents

Chapter I

General rules

- Your programs should not quit unexpectedly (giving an error on a valid input). If this happens, your project will be considered non functional and will receive a 0 during the evaluation.
- We encourage you to create test programs for your project even though this work won't have to be submitted and won't be graded. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded.
- If your code is using external dependencies, it should use Go Modules for managing them

Chapter II

Rules of the day

- You should only turn in `*.go` files and (in case of external dependencies) `go.mod + go.sum`
- Your code for this task should be buildable with just `go build`

Chapter III

Intro

There are many popular data formats in the world of programming and Go in particular. But it's highly likely that you'll meet one of these to on the way - XML or JSON. Lots and lots of APIs are using JSON and/or XML to encode structured data.

And...sometimes even bakeries use them to store recipes. So the old famous bakery in Villariba was always using good old XML to store the list of cake recipes. If we take a look at the piece of that database, it looks kinda like this:

```
<recipes>
    <cake>
        <name>Red Velvet Strawberry Cake</name>
        <stovetime>40 min</stovetime>
        <ingredients>
            <item>
                <itemname>Flour</itemname>
                <itemcount>3</itemcount>
                <itemunit>cups</itemunit>
```

```
            </item>
            <item>
                <itemname>Vanilla extract</itemname>
                <itemcount>1.5</itemcount>
                <itemunit>tablespoons</itemunit>
            </item>
            <item>
                <itemname>Strawberries</itemname>
                <itemcount>7</itemcount>
                <itemunit></itemunit> <!-- itemunit may be empty  -->
            </item>
            <item>
                <itemname>Cinnamon</itemname>
                <itemcount>1</itemcount>
                <itemunit>pieces</itemunit>
            </item>
            <!-- Here can be more ingredients  -->
        </ingredients>
    </cake>
    <cake>
        <name>Blueberry Muffin Cake</name>
        <stovetime>30 min</stovetime>
        <ingredients>
            <item>
                <itemname>Baking powder</itemname>
                <itemcount>3</itemcount>
                <itemunit>teaspoons</itemunit>
            </item>
            <item>
                <itemname>Brown sugar</itemname>
                <itemcount>0.5</itemcount>
                <itemunit>cup</itemunit>
            </item>
            <item>
                <itemname>Blueberries</itemname>
                <itemcount>1</itemcount>
                <itemunit>cup</itemunit>
            </item>
            <!-- Here can be more ingredients  -->
        </ingredients>
    </cake>
    <!-- Here can be more cakes  -->
</recipes>
```

Life was great and simple until the owner of the bakery found out that in the next village, Villabajo, now lives a filthy impostor that managed to steal his recipes! To get away with his trickery, he even used a different data format to store it and hide from justice!

```
{
  "cake": [
    {
      "name": "Red Velvet Strawberry Cake",
      "time": "45 min",
      "ingredients": [
        {
          "ingredient_name": "Flour",
          "ingredient_count": "2",
          "ingredient_unit": "mugs"
        },
        {
          "ingredient_name": "Strawberries",
          "ingredient_count": "8"  // ingredient_unit is not even here!
        },
        {
          "ingredient_name": "Coffee Beans",
          "ingredient_count": "2.5",
          "ingredient_unit": "tablespoons"
        },
        {
          "ingredient_name": "Cinnamon",
          "ingredient_count": "1"
        }
      ]
```

```
    },
    {
      "name": "Moonshine Muffin",
      "time": "30 min",
      "ingredients": [
        {
          "ingredient_name": "Brown sugar",
          "ingredient_count": "1",
          "ingredient_unit": "mug"
        },
        {
          "ingredient_name": "Blueberries",
          "ingredient_count": "1",
          "ingredient_unit": "mug"
        }
      ]
    }
  ]
}
```

He couldn't help but notice that not only the thief stole his recipes, but he also changed some of them. Some ingredients were missing, counts changed, units renamed. So, he prepared for revenge!

Chapter IV

Exercise 00: Reading

First things first, he had to learn how to read the database. The owner already had a CLI, so he decided that reading the file should be straightforward, so both these should work (files can be distinguished by an extension, for simplicity):

```
~$ ./readDB -f original_database.xml ~$ ./readDB -f stolen_database.json
```

Not only that, he also decided that reading both files shouldn't be that difficult to do through the same interface, which he called `DBReader`. That means, reading different formats means that we have different *implementations* of the same interface `DBReader`, which should spit out the same object types as a result, whether it's reading from the original database or the stolen one. Right, his idea is to choose the appropriate implementation based on a file extension.

So, you'll need to help him with that. Think of which kinds of objects are there in these databases and how they can be represented in code. Then, write an interface `DBReader` and two implementations of it - one for reading JSON and one for reading XML. Both of them should return the object of the same type as a result.

To check that his idea works, make the code print JSON version of the database when it's reading from XML and vice versa. Both XML and JSON fields should be indented with 4 spaces ("pretty-printing").

Chapter V

Exercise 01: Assessing Damage

Okay, so now the owner decided to compare the databases. You've seen that the stolen database has modified versions of the same recipes, meaning there are several possible cases:

1. New cake is added or old one removed
2. Cooking time is different for the same cake
3. New ingredient is added or removed for the same cake. *Important:* the order of ingredients doesn't matter. Only the names are.
4. The count of units for the same ingredient has changed.
5. The unit itself for measuring the ingredient has changed.
6. Ingredient unit is missing or added

Quickly looking through the database, the owner also noticed that nobody bothered renaming the cakes or the ingredients (possibly, only added some new ones), so you may assume names are the same across both databases.

Your application should be runnable like this:

```
~$ ./compareDB --old original_database.xml --new stolen_database.json
```

It should work with both formats (JSON and XML) for original AND new database, reusing the code from Exercise 00.

The output should look like this (the same cases explained above):

```
ADDED cake "Moonshine Muffin"
REMOVED cake "Blueberry Muffin Cake"
CHANGED cooking time for cake "Red Velvet Strawberry Cake" - "45 min" instead of "40 min"
ADDED ingredient "Coffee beans" for cake  "Red Velvet Strawberry Cake"
```

```
REMOVED ingredient "Vanilla extract" for cake  "Red Velvet Strawberry Cake"
CHANGED unit for ingredient "Flour" for cake  "Red Velvet Strawberry Cake" - "mugs" instead of "cups"
CHANGED unit count for ingredient "Strawberries" for cake  "Red Velvet Strawberry Cake" - "8" instead of "7"
REMOVED unit "pieces" for ingredient "Cinnamon" for cake  "Red Velvet Strawberry Cake"
```

Chapter VI

Exercise 02: Afterparty

Digging through the database, the Villariba bakery owner suddenly realized - this guy is brilliant! Some recipes were improved a lot comparing to the old version, and these new ideas were really creative! He rushed into Villabajo and found the guy who, as he first thought, stole his most precious legacy.

...The same evening in the tavern two old bakers were hugging, drinking and laughing so hard that it was heard in both villages. They became best friends during the last couple of hours, and each of them was really happy to finally find the person who could blabber all night about cakes! Also turns out, the guy did't steal the database, he was just trying to guess by the taste and tried to improvise around a bit.

After this whole mess they both agreed to use your code, but asked you to do one last task for them. They were quite impressed by how you've managed to do the comparison between the databases, so they've also asked you to do the same thing with their server filesystem backups, so neither bakery would run into any technical issues in the future.

So, your program should take two filesystem dumps.

```
~$ ./compareFS --old snapshot1.txt --new snapshot2.txt
```

They are both plain text files, unsorted, and each of them includes a filepath on every like, like this:

```
/etc/stove/config.xml
/Users/baker/recipes/database.xml
/Users/baker/recipes/database_version3.yaml
/var/log/orders.log
/Users/baker/pokemon.avi
```

Your tool should output the very similar thing to a previous code (without CHANGED case though):

```
ADDED /etc/systemd/system/very_important/stash_location.jpg
REMOVED /var/log/browser_history.txt
```

There is one issue though - the files can be really big, so you can assume both of them won't fit into RAM on the same time. There are two possible ways to overcome this - either to compress the file in memory somehow, or just read one of them and then avoid reading the other.
Side note: this is actually a very popular interview question.