

ADDRESS BOOK APPLICATION

REQUIREMENTS

You have been asked to develop an address book that allows a user to store (between successive runs of the program) the name and phone numbers of their friends, with the following functionality:

- To be able to display the list of friends and their corresponding phone numbers sorted by their name.
- Given another address book that may or may not contain the same friends, display the list of friends that are unique to each address book (the union of all the relative complements). For example given:

Book1 = { "Bob", "Mary", "Jane" }

Book2 = { "Mary", "John", "Jane" }

The friends that are unique to each address book are:

Book1 \ Book2 = { "Bob", "John" }

Assumptions

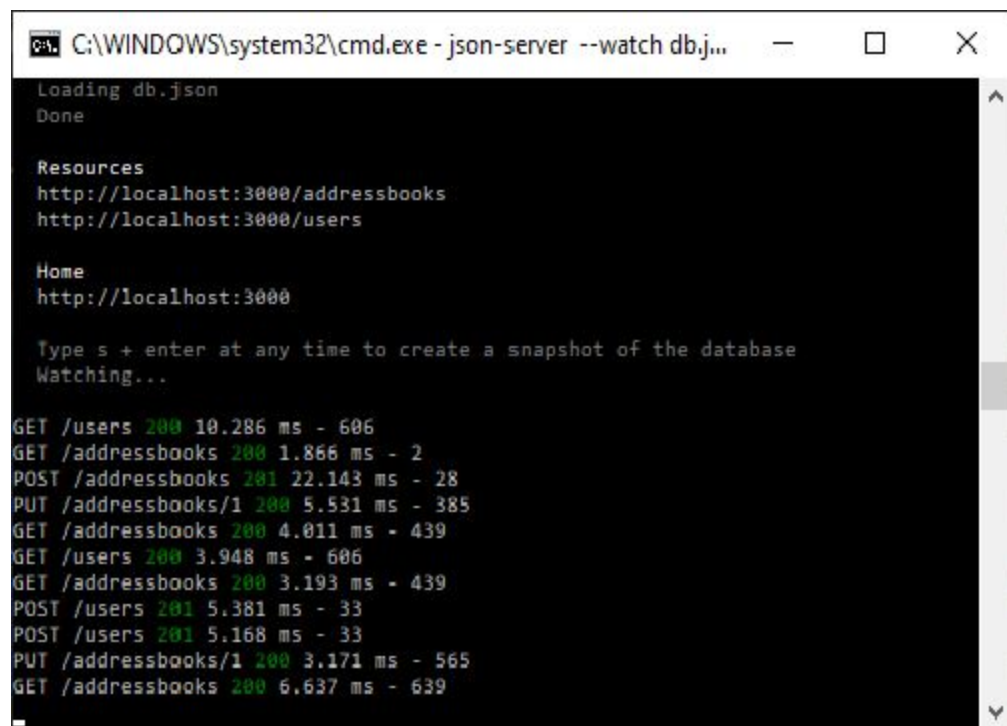
1. I have assumed that there is only ONE Address Book persisted, and that subsequent imports will add to this existing one
2. I have assumed that there cannot be duplicate names in a Book
3. I wanted to add multiple phone numbers for a name, but this was not possible in the allocated time.

SETUP

1. The AddressBook application has been written using node js v.12.18.2 and typescript version 3.9.5 on Windows10.
2. It is command line based, but has been designed in a loosely coupled way to allow the business logic to be unaware if the architecture of the application.
3. nodejs can be installed via details at <https://nodejs.org/en/download/>.
4. It is recommended to install npm - <https://www.npmjs.com/package/npm>
5. typescript can be installed from the command line using
6. "npm install -g typescript".
7. Check the versions of each with the following
 - a. npm -v
 - b. node -v
 - c. tsc -v
8. Also install json-server
 - a. npm install -g json-server

START REST API SERVICE

1. From the root-folder of this project at the command line, type the following
 - a. "json-server --watch db.json"



```
C:\WINDOWS\system32\cmd.exe - json-server --watch db.j...
Loading db.json
Done

Resources
http://localhost:3000/addressbooks
http://localhost:3000/users

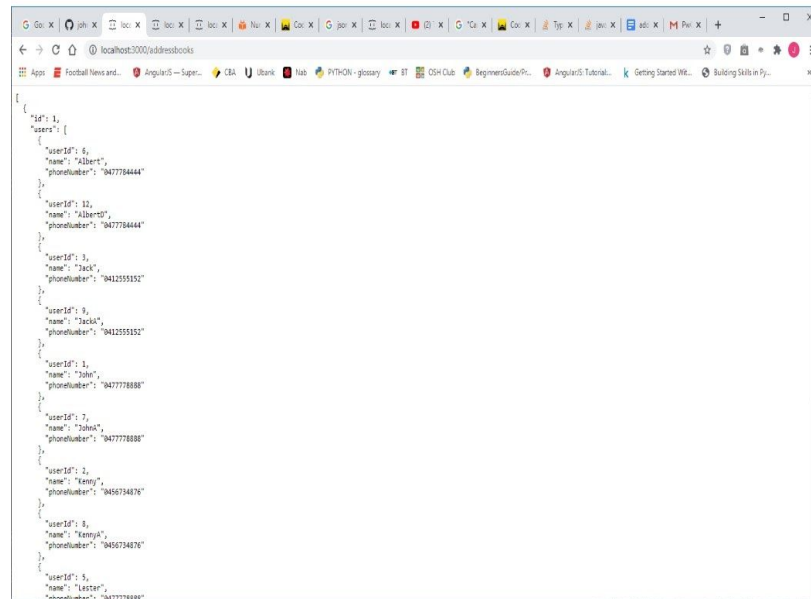
Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
Watching...

GET /users 200 10.286 ms - 606
GET /addressbooks 200 1.866 ms - 2
POST /addressbooks 201 22.143 ms - 28
PUT /addressbooks/1 200 5.531 ms - 385
GET /addressbooks 200 4.011 ms - 439
GET /users 200 3.948 ms - 606
GET /addressbooks 200 3.193 ms - 439
POST /users 201 5.381 ms - 33
POST /users 201 5.168 ms - 33
PUT /addressbooks/1 200 3.171 ms - 565
GET /addressbooks 200 6.637 ms - 639
```

- b. This will start the json-server REST api service locally on localhost:3000. This service uses a file called "db.json" which you will find in the root-folder.
NOTE: DON'T MANUALLY change db.json when the server is running, it can crash the application. USE THE BROWSER to see updates (see details below)

- d. From the browser go to **localhost:3000/addressbooks** and **localhost:3000/users** to see the data as it updates (more on the updates later)

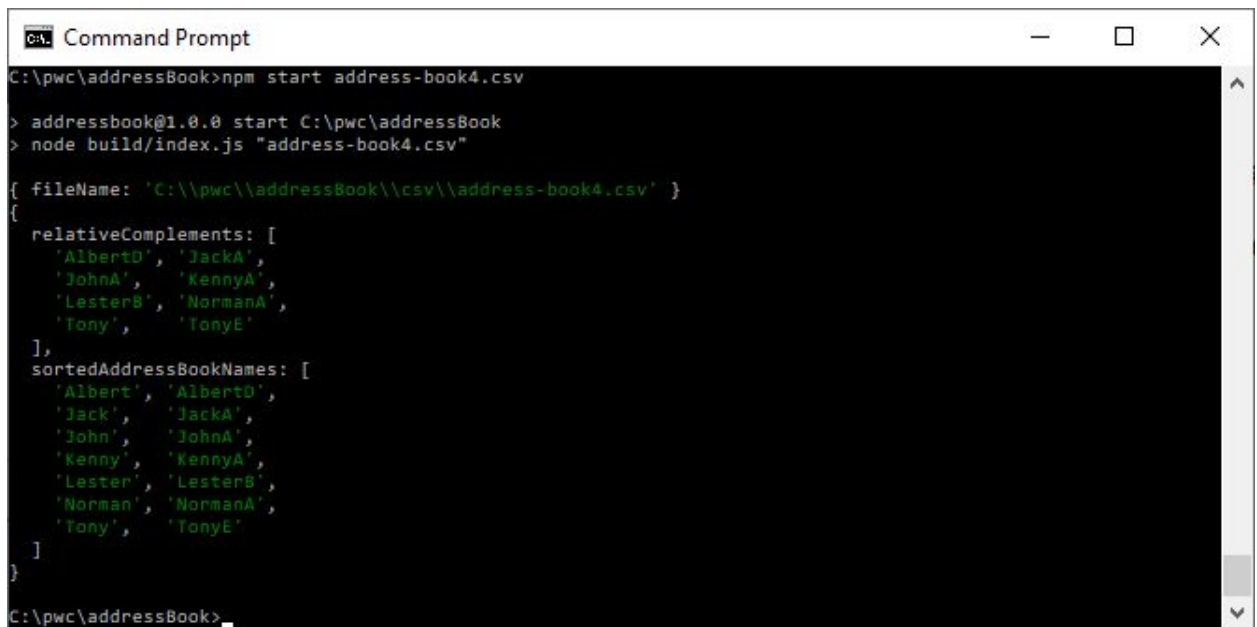


```
{
  "id": 1,
  "users": [
    {
      "userId": 6,
      "name": "Albert",
      "phoneNumber": "0477784444"
    },
    {
      "userId": 10,
      "name": "Albert",
      "phoneNumber": "0477784444"
    },
    {
      "userId": 3,
      "name": "Jack",
      "phoneNumber": "0412555353"
    },
    {
      "userId": 8,
      "name": "Jack",
      "phoneNumber": "0412555353"
    },
    {
      "userId": 1,
      "name": "John",
      "phoneNumber": "0477778888"
    },
    {
      "userId": 7,
      "name": "John",
      "phoneNumber": "0477778888"
    },
    {
      "userId": 2,
      "name": "Kenya",
      "phoneNumber": "0456734376"
    },
    {
      "userId": 4,
      "name": "Kenya",
      "phoneNumber": "0456734376"
    },
    {
      "userId": 5,
      "name": "Lester",
      "phoneNumber": "0477778888"
    }
  ]
}
```

2. The AddressBook application makes simple rest api calls to persist the data, so ensure the json-server is running when importing address books. I chose this mechanism as it provides a quick and highly visible way of seeing the updates while also providing a quite rich model that would have been difficult to support in the allowable development time.

RUNNING THE SERVICE

1. To run the service, go to the command line at the root folder and type
 - a. **"npm install"** ... this might take a few minutes, upon completion type
 - b. **"npm run build"**
2. Start the service with
 - a. **"npm start"**
 - b. This will take the contents of a file already at **"csv/address-book.csv"** and import these contents into the system.
3. Remember you can see the results via the **localhost:3000/addressbooks** and **localhost:3000/users** endpoints.
4. In the console, after an import is complete, you will see details similar to that displayed below



```
Command Prompt
C:\pwc\addressBook>npm start address-book4.csv

> addressbook@1.0.0 start C:\pwc\addressBook
> node build/index.js "address-book4.csv"

{ fileName: 'C:\\pwc\\addressBook\\csv\\address-book4.csv' }
{
  relativeComplements: [
    'AlbertD', 'JackA',
    'JohnA', 'KennyA',
    'LesterB', 'NormanA',
    'Tony', 'TonyE'
  ],
  sortedAddressBookNames: [
    'Albert', 'AlbertD',
    'Jack', 'JackA',
    'John', 'JohnA',
    'Kenny', 'KennyA',
    'Lester', 'LesterB',
    'Norman', 'NormanA',
    'Tony', 'TonyE'
  ]
}
```

This information shows the list of names unique to the existing AddressBook and the one just imported, i.e. the "Relative Complement". It also shows the contents of the AddressBook in sorted name order AFTER the has been completed -

5. Alternatively, you will see the same information at **localhost:3000/addressbooks**
6. To provide your own file, create a file as per the format shown in **address-book-csv** and store it in the same csv folder.
 - a. From the command line, type
 - i. **"npm start <file-name>"** where **<file-name>** is your file,
 - ii. e.g. **npm start myimport.csv**

7. Alternatively, use one of the other files in the **csv** folder ar.
 - a. From the command line, type
 - i. **"npm start address-book2-csv"**,
8. With each import, the view of the data will change at **localhost:3000/addressbooks**

MORE DETAIL ABOUT THE DESIGN

The entry point to this application is **/src/index.ts**. This module calls the **"execute"** method of an instance of an interface called Executable and supplies injects arguments into the method to allow a loose coupling of the functionality.

To that end, there are 4 modules used :-

controllers - which returns an instance of Executable as discussed.

persistence = where the AddressBook data is read and saved.

services - which deals with how the imported data is supplied (e.g. via a csv file)

model - which contains classes to hold the business logic of adding users to a AddressImport

Each module has an **interface** folder the details of which are exported to other modules.

For example **"src\persistencelindex.ts"** has the following logic to export the typescript classes which handle the interaction with the **json-server**. However. the instances of these classes are exported as their underlying interfaces (e.g. **UserSerializable** and **AddressBookSerializable**)

```
export const userSerializable: UserSerializable = new UserRESTApi();
```

```
export const addressBookSerializable: AddressBookSerializable = new AddressBookRESTApi();
```

The design has ensured that each module is loosely coupled. Communication between each module is via an interface supplied to a module by the controller. For example, only the persistence module knows that data is persisted in the json-server, and only the service module knows that the imported data is supplied via a csv file. This allows greater flexibility to use a more robust mechanism to, for example, persist data as a later enhancement (e.g. via a database instead of the local running REST API).

TESTING

1. I have provided test cases for the logic using jest.
 - a. The test cases can be requested by submitting either of the following from the command line
 - i. **npm run test** OR
 - ii. **npm run coverage**
 1. The coverage details can be seen at **./coverage/index.html**

AFTER THOUGHTS AND IMPROVEMENTS.

1. I am not sure about the REST API as a repository - it occasionally crashes and sometimes the db.json is locked.
 - a. You have to avoid using an editor to update the file. I found using POSTMAN to delete data was possible
 - b. I have used it previously with integration tests and found it very useful. A DB would possibly have been as good, but I was trying to focus on the functionality in the time I had and so decided to simplify the persistence.
2. Typescript is a much better choice than vanilla javascript, particularly in rapid dev and testing. The strong typing gives the developer so much immediate feedback at the time of coding.
3. The support for mocking an interface is not as straightforward and established as the equivalent in java..
4. I chose node.js simply because it is a faster development tool than java. Simple out of the box tools like "**nodemon**" and "**concurrently**" allow development with test cases running against the changes
5. The one issue I have is I still find **Mockito** and similar java test frameworks easier to use and better supported, however tools are improving and it is something I will look into more

IMPROVEMENTS

1. Requires environment based configuration
2. Security issues on the data access.
3. I have basic error handling at the top of the application.
4. There is no useful validation of the csv file contents.
5. Blank lines are not handled.
6. I consider the ***Execution.test.ts*** test case is too busy, I think that class should be broken up into more parts.
7. Introduce the facility to remove items from address book
8. Multiple Phone Numbers against one address book item.
9. Implement the logic behind a Web Service.
10. Persist the data in a fully functioning DB